This is to certify that the

thesis entitled

Hardware-software Partitioning in Co-design of Embedded Systems

presented by

Habeel Ahmad

has been accepted towards fulfillment
of the requirements for

Master's degree in Electrical Eng

Major professor

Date 8/20/98

DATE DUE	DATE DUE	DATE DUE

HARDWARE-SOFTWARE PARTITIONING IN CO-DESIGN OF EMBEDDED SYSTEMS

By

Habeel Ahmad

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Electrical Engineering

1998

ABSTRACT

HARDWARE-SOFTWARE PARTITIONING IN CO-DESIGN OF EMBEDDED SYSTEMS

By

Habeel Ahmad

The rapid advancements in science and technology and especially in the field of computers have made an impact on every aspect of our daily life. Among the latest trends is to make devices known as embedded systems that consist of one or more programmable components. Hardware-software co-design is a methodology that provides rules and techniques for embedded system design.

This work has focused on one of the aspects of co-design called hardware-software partitioning. A variety of co-design frameworks were studied with a view to explore the partitioning algorithms. POLIS, which currently supports manual partitioning, was chosen as a candidate for implementation of automatic partitioning. Two partitioning algorithms, namely, Group Migration and Simulated Annealing were implemented in C^{++} for this purpose. The existing design flow in POLIS was altered to generate the performance estimates for both software and hardware implementations of system modules. The algorithms used these estimates to find the best partition that satisfied the constraints.

The results produced by the partitioning algorithms are in agreement with the results published in the POLIS documentation. The work can be extended to integrate the partitioning algorithm with POLIS design flow and to include a rapid prototyping environment for verification.

I dedicate this thesis to my parents,

Aziz Ahmad and Amna Aziz

ACKNOWLEDGEMENTS

I start in the name of Allah Almighty, the most beneficent and the most merciful, whose everlasting blessings made the completion of this thesis possible.

Next, I wish to express my sincere gratitude to my advisor Dr Diane T. Rover for her commitment, encouragement and guidance that helped me conclude this study. Appreciation is also extended to Dr. Micheal Shanblatt and Dr. Bruce E. Kim for their valuable suggestions and also for serving as members of my guidance committee. My thanks are also due to the secretarial staff of EE Department for their support at all times.

I am grateful to Mr. Bassam Tabbara at UC Berkeley for his prompt response and guidance during my struggle with POLIS and PTOLEMY.

I would like to express my gratitude to all of my valuable friends, especially Mr. Abdul Naeem Khan, Dr. Pervaiz Akhtar, Mr. Fida M. Khan and Dr. Dale Joachim for their understanding, encouragement and social support during my stay at East Lansing. My special thanks to Mr Aman-ullah Ateequi and his family for their caring.

Finally, I wish to acknowledge the support and sacrifice of my mother, wife and sister back home during my stay at MSU that enabled me to fully concentrate on my studies. I am especially grateful to my loving wife, Najm us Saher, who always remained a source of inspiration and encouragement during the course of studies. I also acknowledge her patience and courage for single handedly running all the affairs in looking after our children, Rafiah, Sa'ad, Saleha and Sa'adiah during my long absence from home.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Electronics has made inroads in all fields of science and technology. Digital electronics is one of the most prolific fields in the present day world. It provides the building blocks for the wonder of the 20^{th} century, the computer. Digital electronics is the enabling technology for the design of high-performance systems in which reliability and dependability are key issues, in such applications as space exploration and medical instrumentation. The continued miniaturization of digital circuits has made it possible to design and manufacture programmable components such as microprocessors and microcontrollers.

With the increasing role of computers in our daily life, one may envision a world where everything would assume an electronic dimension; *i.e.*, either it would be controlled by a computer or may contain one or more programmable components. This demands reconsideration of the role of computers in our daily life. For now, a user buys a computer as a platform to run a variety of programs that make it perform widely different tasks. With the advances in technology and increased market competition, the prices of computer hardware have gone so low that the highest cost a user has to pay today is that of software. In this scenario, it is logical to think of building special-purpose devices with optimized functionality for a dedicated application. The manufacturers of electronics systems have already adopted this approach. Out of the millions of microprocessors manufactured each year, only 20% are being used in the general-purpose

computers, whereas the rest of them are used in digital systems for dedicated applications known as embedded systems.

1.1 Embedded Systems

Embedded systems are digital systems, which perform specific functions. They are normally categorized as hardware-software systems. Embedded systems are defined as a collection of programmable parts and dedicated hardware components that are continuously interacting with the environment. By virtue of the requirement of continuous interaction with the environment they are also termed as reactive systems [1]. The software runs on microcontrollers or Digital Signal Processors (DSPs) and dedicated hardware is implemented in Application Specific Integrated Circuits (ASICs) or Field Programmable Gate Arrays (FPGAs). Generally, software is used for features and flexibility, while hardware is used for performance. Design of embedded systems can be subject to many different types of constraints, including timing, size, weight, power-consumption, reliability and cost. Rapid advancement in the field of computer aided design (CAD) during the recent past has opened numerous vistas of research and development in the area of embedded system design. The research efforts are now being directed towards the development of such techniques, which not only result in meeting the requisite performance criteria but also reduce time-to-market and cost [2]. Following are some examples of embedded systems:

- Consumer Electronics: medical instruments, cameras, compact disc players, VCRs, microwave ovens and washing machines.

- Telecommunications: networking and communication systems such as satellites and cellular phones.
- Automotive: engine controllers, anti-lock brakes and dashboard controllers.
- Defense and Aviation Electronics: airborne radio and radar, fire control, navigation and guidance, and cryptographic systems.
- Plant and Process Control: Remote controlled toys, robots and plant monitors.

Figure 1.1 shows an example of the architecture of a typical embedded system. It consists of programmable components such as microcontroller and DSP and dedicated hardware such as ASIC and standard logic components. The microcontroller runs the application program under the control of real-time operating system (RTOS). The interface among the various components is implemented with the system bus.

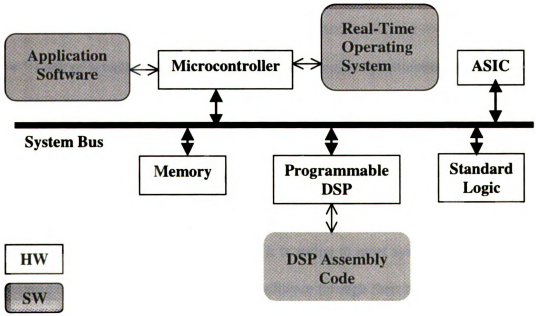

Figure 1.1: Architecture of a typical embedded system.

Current practices of embedded system design tend to follow different paths for the design of hardware and software. A specification, often incomplete and written in non-formal language, is developed and sent to the hardware and software engineers. It involves multiple, subsequent hardware/software development steps in which a prototype is designed through refinement of specifications. Hardware-software partition is decided in advance and is adhered to as much as possible, because any changes in this partition may necessitate extensive redesign. Designers often strive to make everything fit in software, and off-load only some parts of the design to hardware to meet timing constraints. The problems with this design strategy are:

- Lack of a unified hardware-software representation, which leads to difficulties in verifying the whole system and, therefore, to incompatibilities across the hardware-software boundary.
- Definition of hardware-software partitions in the early design stages, which leads to sub-optimal designs.
- Lack of a well-defined design flow, which makes specification revision difficult, and directly impacts time-to-market.

1.2 Hardware-Software Co-design

Hardware-Software co-design is considered to be a design methodology that can avoid the above-mentioned disadvantages. It is trying to meet system-level objectives by exploiting the synergism of hardware and software through their concurrent design. It can be viewed as a management discipline, which offers the possibility to develop large complex system products. Hardware-Software codesign is a complex process that

involves transforming a high-level system specification to an implemented hardware-software system that meets the specification constraints.

Hardware-software co-design is predominant in the development of such systems where hardware and software modules closely interact to solve a certain task [3]. As discussed earlier, hardware-software systems are not new and they have continued to be designed using conventional approaches, however, methodologies that concurrently apply to both domains are now emerging. The growing interest in hardware-software co-design can be attributed to following developments or compulsions:

- Advances in enabling technologies such as system level specification and simulation environments, prototyping techniques, formal methods for design and verification, high-level synthesis and the emergence of CAD frameworks have opened new venues for hardware-software co-design.
- The increasing diversity and complexity of embedded systems demands advanced design methods for the development of both hardware and software.
- The market competition has made it imperative to decrease the cost of design and test of hardware-software systems. More than ever, optimization of cost and performance and a significant reduction in time-to-market are vital issues in the development of embedded systems.

1.3 Co-design Methodology

Co-design as practiced today relies heavily on techniques and methods that have been successfully applied in the past. New contributions are being made in the areas of design automation tools, tool interface, hardware-software partitioning techniques, and enhanced

framework technologies. Currently, two approaches are followed by hardware-software system designers [3]. The conventional approach is the traditional approach where generic framework techniques are employed to facilitate tool encapsulation and integration, and management support is provided for coordinated and cooperative design. Figure 1.2 depicts a typical series of steps in this co-design methodology. This approach has been the focus of research in the past. The model-based approach is the focus of more recent research and favors late partitioning during the design process. Figure 1.3 illustrates the model-based approach. A brief description of both approaches follows.

1.4 Conventional Approach

1.4.1 Specification

The first step in the conventional approach is description of specification. The objectives, requirements and constraints supplied by the user are often incomplete and lack clarity. This step helps in removal of inconsistencies and location of missing information, which results in formulation of system specifications. The traditional approach of system specification as an informal natural-language description has proved to be inadequate over the past [2][3]. It is not possible to automate the co-design process by using natural language descriptions. The designers have therefore evolved an executable-specification approach to overcome this limitation. In this approach the system's functionality is first captured with an executable language and then the functional objects are derived and partitioned. Since the specifications are machine readable, it is possible to develop tools to automate co-design. The specifications could

be verified using simulation, thereby eliminating errors early in the design and preventing costly changes in the subsequent stages of the design.

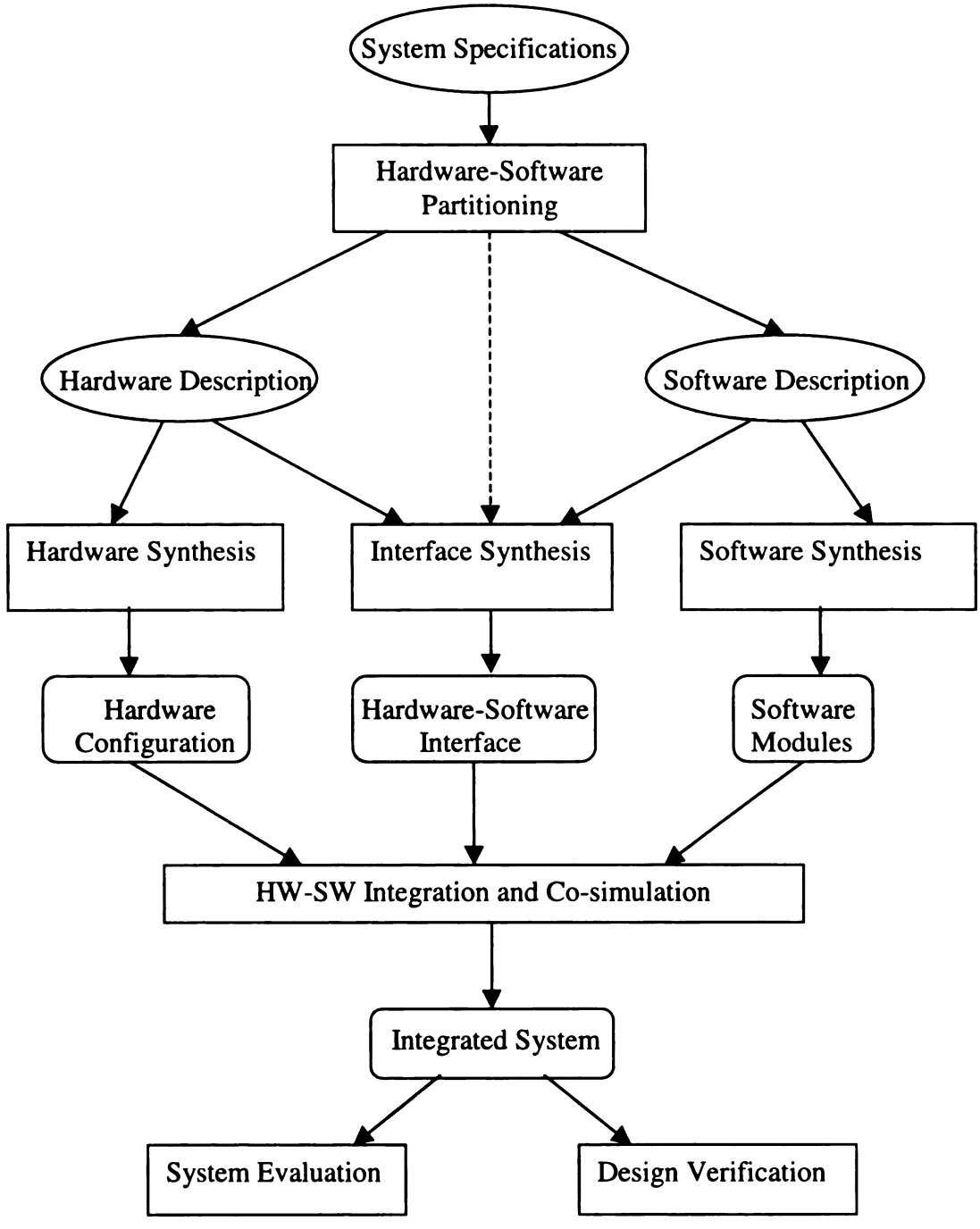

Figure 1.2: Conventional co-design methodology.

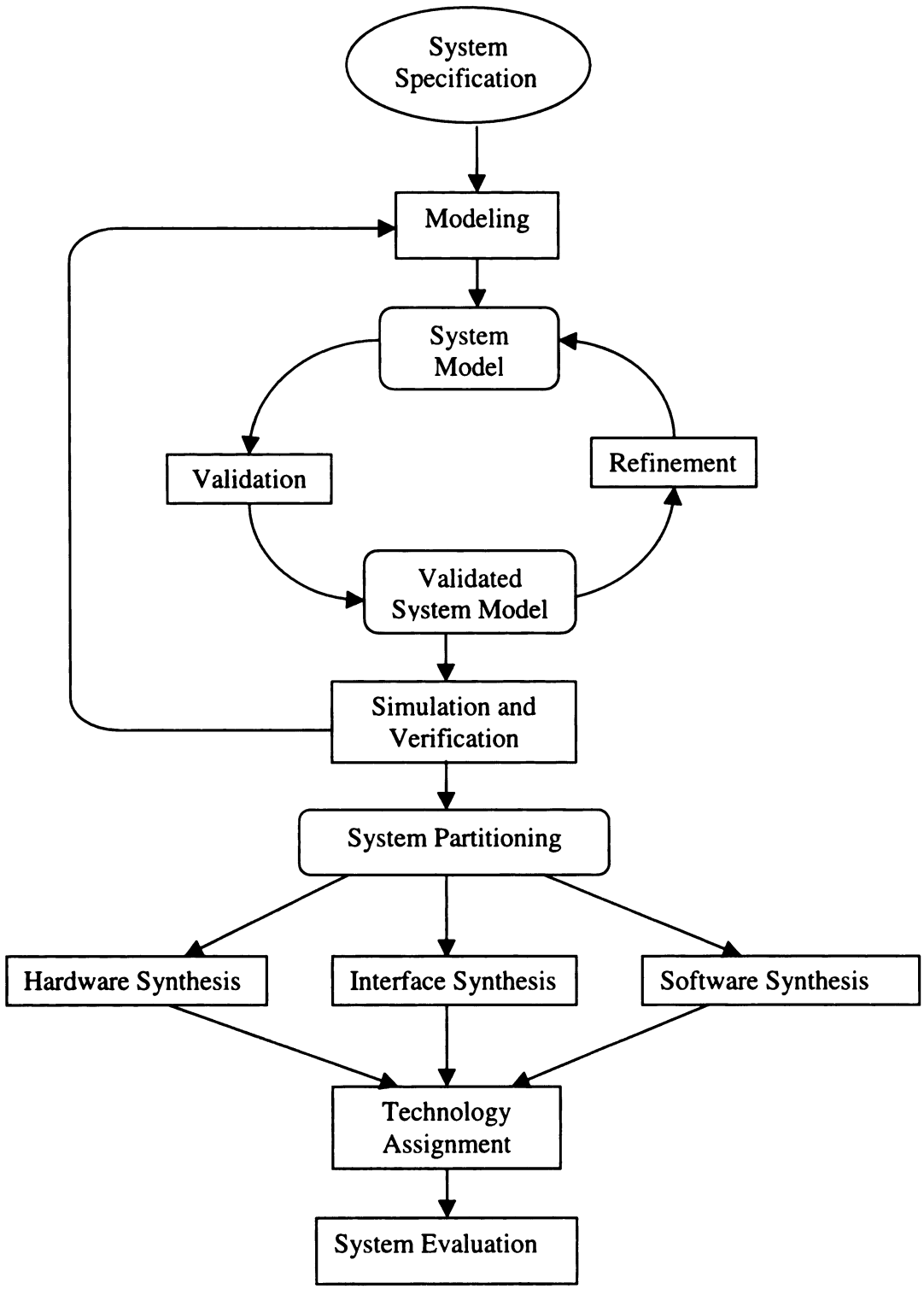

Figure 1.3: Model-based co-design methodology.

1.4.2 Partitioning

The next step is partitioning, where the designer decides to realize various components of the design in hardware and software. Partitioning is usually done manually. Partitioning may be done at various levels of abstraction or at different stages of the design. Early partitioning is preferred by the industry because of requirements of preplanning the development cycle. This however restricts late changes in the specifications. Late partitioning may result in better performance optimization and allows user change requests at later stages of the design.

1.4.3 Synthesis

Synthesis is final realization of a system in hardware and software. It also provides some feedback to the partitioning process based on the technology requirements or the specific application of the system. Tools for software synthesis have already been available in the past because of efforts put in by the software designers in development of compilers. Hardware synthesis tools are also appearing which have made rapid prototyping of hardware possible.

1.4.3.1 Hardware Synthesis

The hardware platform on which the software is to execute and the dedicated hardware that shares the functionality are synthesized during this phase, using the results of partitioning. Hardware synthesis involves technology binding by translation (mapping) of hardware descriptions such as VHDL, HardwareC, BLIF, *etc.* into gate level netlists.

1.4.3.2 Software Synthesis

Software synthesis involves translation of functionality into a program for a particular processor. The software description is generated in terms of a high level language such as C/C^{++}. A compiler is then used to translate this description into an executable code for the selected processor. The software synthesis may also include synthesis of the operating system, some times called real-time operating system (RTOS) due to the real-time reactive nature of embedded systems [1].

1.4.3.3 Interface Synthesis

In order to provide the signal/data exchange capability an interface between the hardware and software is often required. Interface synthesis provides a means of hardware and software synchronization. Typically signal exchange (hardware), semaphore (software) or interrupt driven schemes are employed in this phase [1]. Implementations range from custom logic to dynamically configurable logic devices. A central scheduler in software may also be incorporated to control the hardware processes.

1.4.4 Hardware-Software Integration and Co-simulation

The integration step involves co-simulation of hardware and software on a heterogeneous simulator. The results of simulation provide an assessment of the performance of design in terms of meeting the specifications and satisfaction of constraints. The verification step ensures that the designed system performs according to specifications.

1.5 Model-based Approach

Modeling is the process of conceptualizing and refining the given specifications. There exist a wide variety of potential formalizations of a design, but often a relation between a set of inputs and outputs characterizes the behavior of a system. This relation could be informal, may even be expressed in natural language, but the result of such an informal specification can easily be an expensive and unnecessary redesign. Formal modeling of a design consists of the following components:

- A functional specification, given as a set of explicit or implicit relations, which consists of inputs, outputs and internal state information.
- A set of performance indices that evaluate the quality of the design (cost, reliability, size, speed, power consumption, etc.) given as a set of equations, which include at least inputs and outputs.
- A set of constraints on these performance indices, specified as a set of inequalities.
- A set of properties that must be satisfied, given as a set of relations over inputs, outputs and states that can be checked against the functional specification.

1.5.1 Validation

After transforming a specification into a formal model, the whole model has to be checked on its functional correctness. This step is known as validation. It includes functional validation of subtasks and of the whole system, including formal verification and functional simulation. Functional validation and simulation concerning functionality and constraints such as hardware size, costs, timing and reusability are also performed.

1.5.2 Partitioning and Implementation

After validation, the designer can start to be more concrete by partitioning the model. The decision of what to do in hardware and what in software is important. If the best partitioning is found and validated successfully, a prototype of the hardware and the necessary machine code is generated for testing the system in actual environment.

1.6 Partitioning Problem

One phase of the co-design process, the partitioning of specification into components and binding them to hardware/software resources, is the focus of current research. It is the central problem where a system designer decides which components of the system will be implemented in hardware and which will be realized in software. Partitioning requires an effective means of exploring the design space through evaluation of candidate solutions, considering the interaction of multiple constraints [2]. The decision to put a particular component in hardware or software has to be based on an evaluation of the metrics of interest for the entire system. This evaluation can either be done in the physical domain by actual implementation, *e.g.*, by synthesizing the hardware to a gate netlist on which accurate metrics for area and performance can be obtained; or it can be done in the model domain, which is less accurate but much faster. Co-design is an iterative process that requires repeated partitioning and evaluation for design space exploration, and thus the speed of this process is a critical issue. In practice, it means that the model domain is the right choice for efficient estimation and evaluation.

The hardware-software partitioning problem involves two key metrics: performance and hardware size. Performance is normally improved by moving objects to hardware, while the hardware size is obviously improved by moving objects out of hardware. This tradeoff has led to the development of specialized algorithms for hardware-software partitioning. Depending on the underlying theoretical model, the level of abstraction, and the integration strategy, several performance and size estimation methods are available [5]. Among the possible partitioning schemes derived mostly from related areas, such as VLSI design, the deterministic, statistical, benchmarking and profiling techniques are most popular. Deterministic estimation requires a fully specified model, with all data dependencies removed and all costs of components known. This method leads to very good partitions but fails whenever data items are unavailable. Statistical estimation based on the analysis of similar systems and certain design parameters is then required.

1.7 Scope of This Work

This thesis examines the hardware-software co-design framework called POLIS to explore the possibility of performing automatic partitioning. POLIS is a software system which takes as input the system specifications in terms of an executable code written in a high-level language called Esterél [6] and provides the output in terms of C^{++} code for software and VHDL code or technology-mapped gate-level netlists for hardware. The partitioning process in POLIS is manual. In this thesis we investigated automatic partitioning algorithms that may be integrated within the POLIS framework and report the results of several case studies.

In the next chapter the related work in the area of co-design, with an emphasis on the partitioning issues are discussed. Various techniques and algorithms have been explored and their strengths and weaknesses are highlighted. Chapter 3 presents an in-depth analysis of the design flow in POLIS and the partitioning methodology based on simulations using the Ptolemy simulation platform. In Chapter 4, partitioning issues and algorithms for automatic partitioning are discussed. Application of automatic partitioning algorithms in POLIS and their implementation in C^{++} are presented in Chapter 5. The results obtained after the application of selected algorithms to automate the partitioning process in POLIS and their validity are also discussed in this chapter. The last chapter draws some conclusions and sets forth the goals for future work in this direction.

CHAPTER 2

RELATED WORK

Hardware-software co-design with its promise of bringing order to the chaotic world of embedded system design, has attracted a lot of attention in recent years. Several research groups have addressed the problem of co-design and in particular that of hardware-software partitioning. Two research projects can be called the pioneers in the field of hardware-software partitioning: Vulcan [7][8][9] and COSYMA [10][11]. Others have also joined in the effort, and numerous co-design frameworks employing automatic or manual partitioning techniques are now being developed at various universities and research institutions. A brief description of some of these frameworks and environments is presented in this chapter. Where applicable, the partitioning strategies employed in each environment are also identified. Chapter 4 describes the partitioning in detail. Table 2.1 lists the co-design environments examined in this thesis along with their important attributes.

	System	Input	Partitioning	Algorithm
1.	Vulcan	HardwareC	Automatic	Iterative
2.	COSYMA	C, C^x	Automatic	Simulated Annealing
3.	LYCOS	C, VHDL	Automatic	PACE(Fine grain partitioning)
4.	COMET	VHDL	Automatic	Scoreboard (Ratio-cut)
5.	PTOLEMY	C, ptl-code	Manual	Iterative
6.	SpecSyn	SpecChart	Automatic	Clustering
7.	TOSCA	C, VHDL	Automatic	Clustering
8.	POLIS	Esterél	Manual	None
9.	Chinook	Verilog	Manual	None

Table 2.1: List of co-design frameworks examined in this thesis.

2.1 Vulcan

The Vulcan system was developed at Stanford University [7][8]. The target architecture consists of single CPU and one or more ASICs, connected to the CPU via a communication bus. It starts with an all-hardware solution specified in HardwareC. The input to Vulcan consists of two components: system's functionality and design constraints. The specification is translated into an internal graph-based representation on which the partitioning is performed. The partitioning algorithm uses an iterative approach to move operations from hardware to software under a timing constraint. It tries to keep the communication cost minimum by keeping the neighboring vertices together in either software or hardware. The Vulcan system attempts to reduce the size of hardware by moving the functionality to software using a CPU. The system can handle multiple processes as hardware and software can run in parallel.

2.2 COSYMA

COSYMA is an acronym for **COSY**nthesis of e**M**bedded **A**rchitectures. It is a hardware-software codesign system designed at Technical University of Braunsheweig, Sweden [10][11]. COSYMA has a graphical user interface which runs under X-windows environment. The target architecture for COSYMA consists of one CPU and one ASIC, where ASIC is used as a co-processor to speed up the execution.

The input to COSYMA comprises a high level description of the algorithm in C^x and a constraint file. C^x is a C dialect with some restrictions and modifications to ANSI C. The objective of COSYMA is to partition the algorithmic description into hardware-software parts such that the user supplied constraints are met while keeping the hardware

cost minimum. The system provides the user with information to fine-tune or remove the emerging bottlenecks from the algorithmic description.

Input to COSYMA can be of three types. The input file can be in ANSI C, pure C^x or a C^x program with parallel extensions to encode concurrent interacting processes. The user can specify rate constraints, inter/intra-process constraints and communication between processes. The input description is translated to a syntax graph. The run-time analysis of the program is then performed. For the parallel extensions, a scheduler orders the processes for a single processor environment with respect to their timing and communication constraints. COSYMA uses simulated annealing algorithm for automatic partitioning. The algorithm starts with an all-software solution and moves the chunks of software code to hardware until the timing constraints are met. The software part is compiled by a C compiler and the hardware part is handed over to the high level synthesis. Finally a co-simulation is performed using the compiled software part and the timing information of the hardware part supplied by the high level synthesis.

2.3 LYCOS: The Lyngby Co-Synthesis System

LYCOS is an experimental co-synthesis environment being developed at Technical University of Denmark [12]. LYCOS targets an architecture consisting of a single CPU and a single dedicated hardware component (ASIC, FPGA etc.) communicating through memory mapped I/O. The areas of application include DSP, embedded systems, software execution acceleration and hardware emulation and prototyping.

LYCOS is built as a suite of tools centered around an implementation independent model of computation based on communicating Control Data Flow Graphs (CDFGs). The

design process starts with an input specification described in either C or VHDL translated to the computational model, Quenya. The CDFGs are divided into chunks of computation called Basic Scheduling Blocks (BSBs) [13], that may be moved between hardware and software. An automatic partitioning algorithm called PACE [14] is used to partition the CDFG by moving BSBs from software to hardware to achieve the best speed-up while keeping the total hardware area less than or equal to the area available on a particular ASIC or FPGA. PACE is based on a fine-grain partitioning [15] approach where adjacent blocks sharing the same variables are moved to hardware so as to reduce communication between hardware and software as well as increase hardware utilization.

2.4 COMET

The main goal of the COMET project [16] at University of Cincinnati is to transform a high-level system specification into application-specific electronic signal processing modules using a hardware-software co-synthesis process and to produce working hardware within a two-week time period. The input to and output from the COMET system are in VHDL. The design framework maintains a database of modules with a variety of functionality.

The system specification is divided into modules, matched to component specifications, and then allocated to either hardware or software synthesis processes. The Co-synthesis process is iterative during which alternate bindings are used to satisfy constraints such as performance and area requirements. The Co-synthesis tool issues requests to the design database using qualifications on design properties, and the query processor determines the set of design objects that match the request. In other words, a

query is a module description, and any modules in the database that have at least the desired functionality (possibly additional functionality) are returned. The co-synthesis tool analyzes candidate solutions and determines the best assignment of resources to hardware and software using an iterative binding approach, called Scoreboard algorithm. The hardware and software specifications are processed by hardware and software synthesis tools, then integrated to form a system that satisfies the initial specifications. The end result of these transformations is an application-specific hardware design that can be fabricated along with the embedded software that will be executed on the manufactured hardware.

2.5 PTOLEMY

The PTOLEMY [17] [19] software is a system-level design framework developed at University of California, Berkeley. The objectives of the PTOLEMY project include most aspects of designing signal processing and communications systems, ranging from designing and simulating algorithms to synthesizing hardware and software, parallelizing algorithms, and prototyping real-time systems. PTOLEMY allows the interaction of diverse models of computation by using the object-oriented principles of polymorphism and information hiding. For example, using PTOLEMY, a high-level data-flow model of a signal processing system can be connected to a hardware simulator that in turn may be connected to a discrete-event model of a communication network.

PTOLEMY is still in a state of evolution. Recent enhancements of the software have been in the field of data-flow modeling of algorithms, synthesis of embedded software from such data-flow models, animation and visualization, multidimensional signal

processing, managing complexity by means of higher-order functions, hardware-software partitioning, and VHDL code generation.

PTOLEMY has been used for a broad range of applications including signal processing, telecommunications, parallel processing, wireless communications, network design, radio astronomy, real time systems, and hardware-software co-design. The main emphasis in PTOLEMY is on co-simulation of system modules targeted for different implementations. A hardware-software partitioning algorithm [18] has also been implemented in the framework of PTOLEMY. The input to this algorithm is a system level description. The partitioning goal is to minimize hardware area given a global execution time constraint.

2.6 SpecSyn

SpecSyn [3] is another Co-design system, which incorporates automatic hardware-software partitioning. It extends the scope of system design from the previously discussed systems in three key ways:

- It includes two additional component types such as memories and buses,
- It allows inclusion of functionality in terms of variables and communication channels that together with behaviors comprise an executable specification, and
- The numbers and types of physical system components can be changed as an integral part of system design.

In SpecSyn the input specification is produced in the visual language SpecChart which is based on Statecharts [21]. This is translated into an intermediate system representation called SLIF [22], on which the system analysis and partitioning is

performed. SpecSyn supports several partitioning algorithms [3] & [23]. [23] Presents a combined approach where clustering is used to reduce the number of code blocks to be considered and a greedy algorithm is used to obtain the partition. The interesting aspect of this approach is that it is able to reach regions in the design space, which lie between the regions obtained by fast greedy algorithms and those obtained by the more costly simulated annealing algorithms.

2.7 TOSCA

In TOSCA [20] the internal representation is based on concurrent hierarchical finite state machines (FSM) which are generated from either standard languages such as C or VHDL or from higher-level languages such as SpecChart [2]. Hardware-software partitioning is done automatically by a clustering algorithm, which tries to cluster FSMs based on some closeness criteria. The target architecture for TOSCA is a single standard processor and one or more coprocessors embedded on a single chip.

2.8 Other Frameworks

A number of researchers have focused on algorithmic aspects rather than complete systems. Janstch et al. [24], [25] present a dynamic programming algorithm to solve the partitioning problem of optimizing an existing C program for speed, given a hardware area constraint. The algorithm is derived from the Knapsack Stuffing algorithm [26] and solves (with exponential memory requirements) the partitioning problem for a partitioning model in which blocks can include other blocks and blocks in general therefore cannot be moved to/from hardware independently of each other. A full loop

block for example includes the loop body and loop test blocks but all three are considered simultaneously in their model. Another approach using a high level language as input is presented by Barros et al. [4]. The partitioning algorithm is a two-stage clustering algorithm which selects groups of code based on similarity measures obtained from classification of assignments in the input specification, which is described in UNITY [27].

Co-design systems in which hardware-software partitioning is obtained with user interaction were also investigated. Among these are POLIS [1], PARTIF [28] and CASTLE [29]. POLIS is the focus of our research and its features are discussed in detail in Chapter 3. PARTIF is an interactive partitioning tool, which allows the designer to explore different partitions by applying a small set of transformation and decomposition rules. These rules are applied to a system representation consisting of hierarchical concurrent FSMs. The CASTLE system is another co-design framework where the input specs are given in a standard language, which can be Verilog, VHDL or C/C++. This input specification is translated into a common representation based on control data flow graphs called SIR, which provides the backbone for all tools. As for POLIS and PARTIF, the partitioning is done manually, but in CASTLE it is based on mappings from a hardware library which is used to specify complex components including microprocessors.

In the Chinook [30] system the emphasis is on module interface and synchronization. The system is used for real-time reactive controllers initially specified in Verilog. Chinook does not provide automatic hardware-software partitioning, but leaves it to the designer, nor does it provide code generation tools for the target processors, but uses

standard C compilers. However, Chinook does synthesize the hardware and software needed for inter- process communication which is a difficult task as different components may not initially fit very well together.

This chapter has shown that there are two main directions for hardware-software partitioning methods:

- Automatic partitioning, which almost always means a restricted target architecture i.e. one processor and one or more ASICs/FPGAs. The emphasis in automatic partitioning is on transferring the functionality from one domain to the other so as to achieve the best performance at a minimum cost.
- Manual partitioning, which typically allows for more advanced architectures involves detailed analysis of design tradeoffs. The complexity of the system and design constraints demand exploration of a variety of architectural choices to meet the system specifications. Also, with advanced target architectures, synchronization and communication between different components becomes much more difficult and very important.

With this review, we may now proceed to explore the design methodology adopted by POLIS.

CHAPTER 3

POLIS

3.1 Introduction

POLIS is a software system developed at the University of California Berkeley for hardware-software co-design of control-dominated embedded systems [1]. The main aspect of POLIS that distinguishes it from other co-design methods is the use of a formal model of computation. This model, called Co-design Finite State Machine (CFSM) [1], is based on the Extended Finite State Machine (EFSM) [1] model operating on a set of finite-valued (enumerated or integer sub-range) variables by using arithmetic, relational and Boolean operators as well as user-defined functions. EFSM model is similar to the FSM model, but the transition relation may also depend on a set of internal variables. POLIS offers a flexible environment for design analysis and verification.

The design can be analyzed at the behavioral level either with formal tools such as model checking or by co-simulation in a heterogeneous simulation environment offered by PTOLEMY, another tool for co-design. The user can select an architecture for evaluation of design tradeoffs with respect to constraints during the simulation phase. Hardware-software partitioning is based on the results of simulation. Software synthesis is fully automated including generation of a custom scheduler and hardware synthesis also involves limited user interaction. POLIS has a path towards an emulation board including Xilinx FPGAs [1], the microprocessor of choice, and A/D and D/A interfaces.

3.1.1 Model of Computation (CFSM)

A Co-design Finite State Machine (CFSM), as does a classical finite state machine, transforms a set of inputs into a set of outputs with only a finite number of internal states. The difference between the two models is that the synchronous communication model of classical concurrent FSMs is replaced in the CFSM model by a finite, non-zero, unbounded reaction time. This model of computation can also be described as Globally Asynchronous, Locally Synchronous. Each element of a network of CFSMs describes a component of the system to be modeled.

The CFSM specification is a priori unbiased towards a hardware or software implementation. While both perform the same computation for each CFSM transition, hardware and software exhibit different delay characteristics. A synchronous hardware implementation of CFSM can execute a transition in one clock cycle, while a software implementation will require more than one clock cycle. CFSM is also a synthesizable and verifiable model, because many existing theories and tools for the FSM model can be easily adapted for CFSM. Each transition of a CFSM is an atomic operation. All the analysis and synthesis steps ensure that:

- A consistent snapshot of the system-state is taken just before the transition is executed.
- The transition is executed, thus updating the internal state and output of the CFSM.
- The result of the transition is propagated to the other CFSMs and to the environment.

The interaction between CFSMs is asynchronous, in order to support neutral specification of hardware and software components by means of a single CFSM network. This means that:

- The execution time for a CFSM transition is unknown but assumed to be non-zero in order to avoid the problem of zero-delay feedback loops. The synthesis procedure refines this initial specification, by adding more precise timing information, as more design choices are made (*e.g.* partitioning, processor selection, compilation, *etc.*). The designer, during the analysis steps, may on the other hand add constraints on this timing information that synthesis process should satisfy. The overall design philosophy of POLIS is to provide the designer with such tools that help in meeting these constraints, rather than provide a quick-fix solution.
- Communication between CFSMs is not by means of shared variables (as in the classical composition of finite state machines), but by means of events. An event is a semi-synchronizing communication primitive that is both powerful enough to represent practical design specifications and efficiently implementable in hardware, software, and between the two domains.

CFSM's behavior and the CFSM network topology are represented using an intermediate language called SHIFT, for **S**oftware **H**ardware **I**nterchange **F**orma**T**. SHIFT is however, not meant to be used as a specification language. The FSM semantics of each CFSM ensures that any of the following graphical or textual languages can be used to specify individual behaviors:

- Reactive synchronous languages, such as StateCharts, Esterél, Lustre and Signal;

- The so-called synthesizable subset of hardware description languages such as VHDL and Verilog; and
- System specification languages with FSM semantics such as SDL [2].

The interconnection between the CFSMs, on the other hand, can be specified (due to the distinctive asynchronous interconnection semantics) within the POLIS environment, using either a textual netlist auxiliary language or a graphical editor VEM [17] that is part of the PTOLEMY co-simulation environment.

Events are emitted by CFSMs and/or by the environment over a set of carriers called signals. One or more CFSMs can detect the emission of each event (the actual delay depends on several implementation-related factors, such as partitioning, scheduling policy and so on). Each detecting CFSM has its own copy of the event, and each emission can be detected at most once by each receiving CFSM.

Signals can carry control information, data information, or both. Events occurring on pure control signals, such as reset input, can be used only to trigger a transition of a CFSM. Once the receiver CFSM has detected an event, it can no longer be detected again until its sender CFSM re-emits it. Values carried by data signals, such as keyboard input or a data sample, can be used as inputs to and output from the CFSM data path. Signals carrying only control information are often called pure signals, while signals carrying only data information are often called pure values. Each CFSM transition has a pre-condition of a set of:

- Input event presence or absence conditions (only for signals with a control part)
- Boolean functions of some relational operations over the values of its input signals.

The post-condition is the conjunction of a set of:

- Output event presence or absence conditions (presence implies emission, absence implies no action), and
- Values assigned to output data signals.

Note that no buffering is provided by the POLIS communication mechanism, apart from the event and value information. This means that events can be overwritten, if the sending end is faster than the receiving end. This overwriting, also called "losing" may or may not be a problem depending both on the application and the type of the event. The designer can make sure that "critical" events are never lost either by:

- Providing an explicit handshaking mechanism, built by using a set of signals, between the CFSMs, or
- Using synthesis directives, such as partitioning choices or scheduling techniques, that ensure that no such loss can ever occur. For example, this can be achieved by:
  a) implementing the receiving CFSM in hardware
  b) implementing both CFSMs in software and using a round-robin scheduler that executes both at the same rate.

3.2 Design Flow

3.2.1 Overview

An overview of the design flow in the POLIS system is depicted in Figure 3.1. The detailed composition of POLIS is shown in Figure 3.2. The input specifications are described in Esterél language. The Esterél code is first translated to SHIFT code. This format is used by POLIS to generate S-Graphs similar to Control Data Flow Graph

(CDFG) for software synthesis and BLIF (Berkeley Logic Interchange Format) code for hardware synthesis. In the architecture selection step, a processor is chosen. POLIS supports Motorola 68HC11 and 68832, and MIPS R3000 processors. POLIS has powerful software and hardware cost estimation capabilities, which can guide the user in selection of the best architecture.

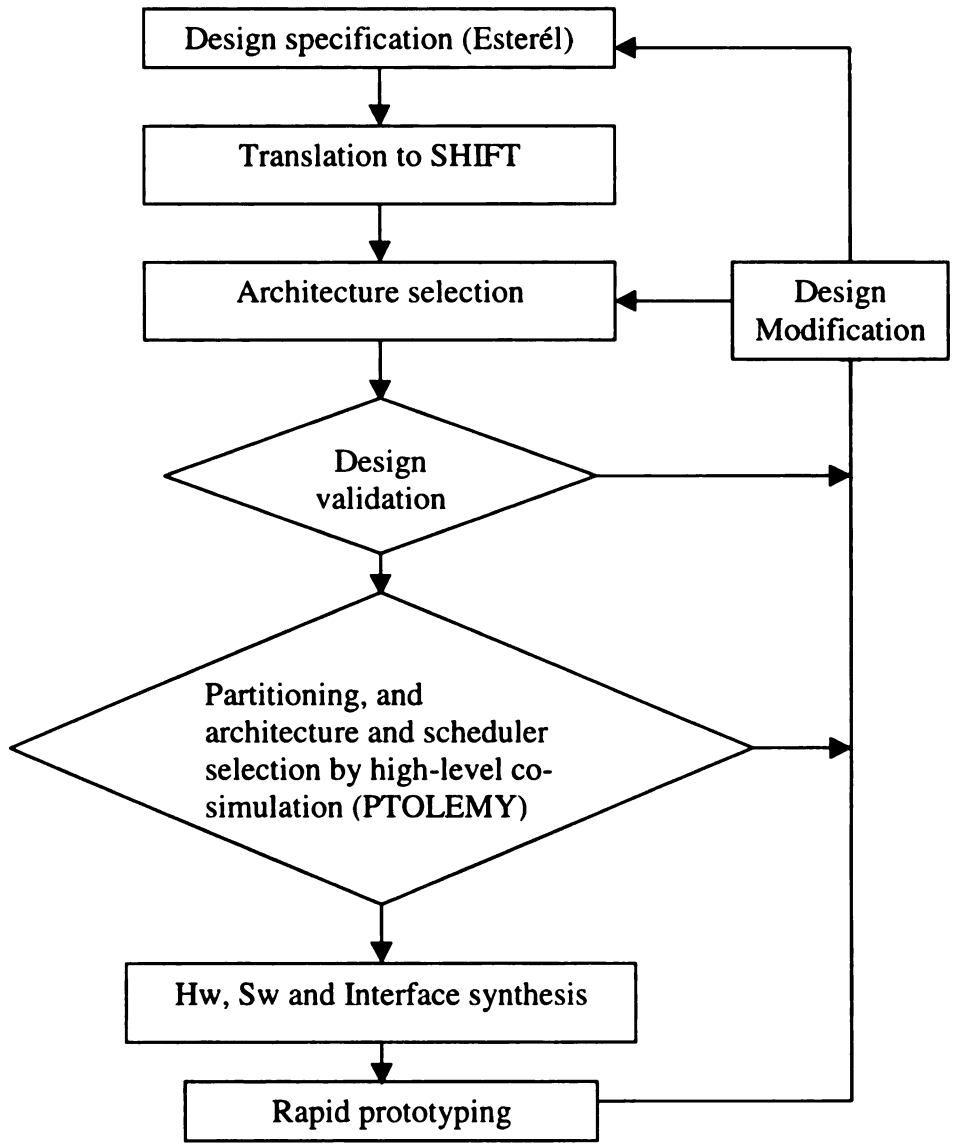

Figure 3.1: Overview of the design flow in POLIS [courtesy POLIS group].

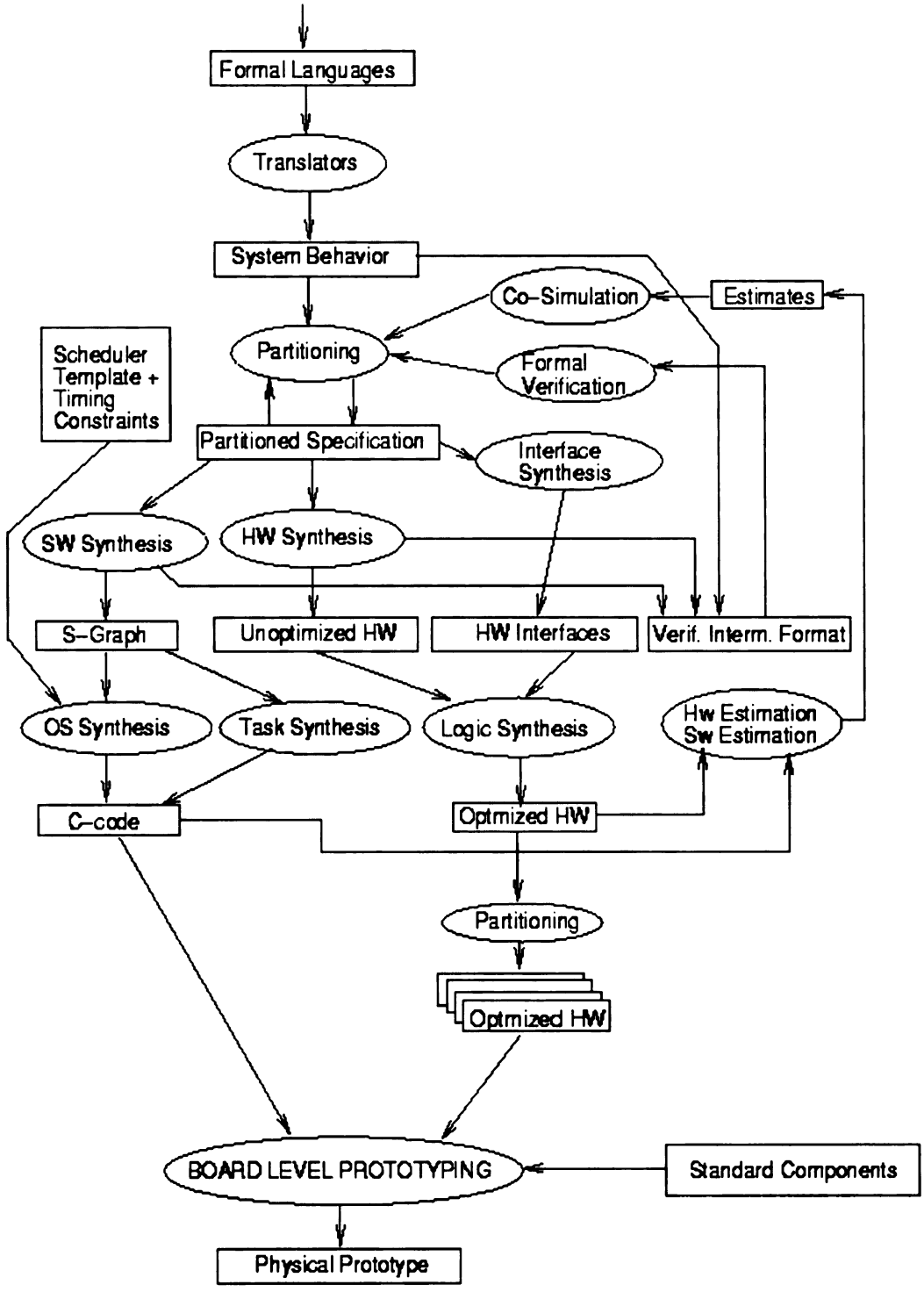

Figure 3.2: The POLIS design flow [courtesy POLIS group].

The S-Graph is further translated to C code and PTOLEMY code for use by PTOLEMY for simulation. The cost estimates are also passed to PTOLEMY for use in simulation. The user can interactively define the system architecture within PTOLEMY as well by changing the implementation of each CFSM to software or hardware. The software and hardware models are executed in the same simulation environment, and the simulation code is the same that will run on the target processor. The simulation can be done at two levels of abstraction: functional and timing. In functional simulation, timing is ignored and only the functional correctness is checked. In timing-based simulation, the timing is approximated using cycle count estimations for software and using a cycle-based simulation for hardware.

After starting the simulation, input events are generated and then the overflow file is checked for any missed deadlines. The processor clock speed can be adjusted to simulate different versions of a processor. If the highest speed processor is also unable to meet the deadlines, then the critical CFSM is transferred to hardware by changing its implementation. Depending upon the results of co-simulation, the final partitioning is decided. A complete SHIFT netlist describing the CFSM-network topology and implementation choice for each CFSM is then created. This SHIFT file is passed back to POLIS for synthesis of hardware (BLIF) and software (C code). The final software also contains a RTOS for the target processor, and the hardware contains the interface circuitry required for communication between hardware and software. There is a path available in POLIS for rapid prototyping of final design for functional validation or in-system testing. Results of this step may then be fed back for the improvement in design.

3.2.2 High Level Language Translation

In POLIS, designers write their specifications in a high level language (*e.g.*, Esterél, graphical FSMs, subsets of Verilog or VHDL) that can be directly translated into CFSMs. Any high level language with precise semantics based on extended FSMs can be used to model individual CFSMs. Currently, however, Esterél is supported directly. The Esterél programs are translated to SHIFT format using the command *strl2shift*. The Esterél compiler first compiles Esterél code and then another tool *oct2shift* translates the output to SHIFT.

3.2.3 Formal Verification

The formal specification and synthesis methodology embedded within POLIS makes it possible to interface directly with existing formal verification algorithms that are based on FSMs. POLIS includes a translator from the CFSM to the FSM formalism which can be fed directly to verification systems (*e.g.* VIS [31]). In addition to uncovering bugs in a design, it also uses formal verification to guide the synthesis process. Since the abstract CFSM model covers the behavior of all possible hardware-software implementations at once, it is possible to refine the specification based on the results of formal verification. Formal verification tools of today still have problems with complexity. A methodology has been developed that incorporates a set of rules specific to POLIS and CFSMs so that it is now possible to verify larger designs.

3.2.3 System Co-simulation

System level hardware-software co-simulation provides feedback on the design choices. These design choices include hardware-software partitioning, CPU selection, and scheduler selection. Fast co-simulation, in the order of millions of clock cycles per second (on a workstation) is possible due to the software synthesis and performance estimation techniques available. The purpose of high-level co-simulation in POLIS is to provide the designer with a flexible environment where architectural tradeoffs can be explored. POLIS currently utilizes PTOLEMY as a simulation engine, but it is not limited to PTOLEMY. VHDL code including all the co-simulation information such as code size and delays *etc.* can also be generated by POLIS.

3.2.4 Partitioning and Architecture Selection

Making system-level design decisions such as hardware-software partitioning, target architecture selection and scheduler selection is not a trivial task. These decisions are based heavily on design experience and are very difficult to automate. In POLIS the designer is provided with an environment to quickly evaluate any such decisions through various feedback mechanisms from either formal verification or system co-simulation. This feedback is however limited to the results of simulation in terms of signal arrival times and missed deadlines. The key advantage of CFSM specification is that it is implementation-independent. The designer can interactively explore the implementation options using the same user interface as co-simulation.

3.2.5 Hardware Synthesis

A CFSM sub-network chosen for hardware implementation is implemented and optimized using logic synthesis techniques from SIS [32]. Each CFSM, interpreted as a Register-Transfer Level (RTL) specification, can be mapped into BLIF, XNF (XILINX Netlist Format), VHDL or Verilog.

3.2.6 Software Synthesis

A CFSM sub-network chosen for software implementation is mapped into a software structure that includes a procedure for each CFSM, together with a simple real-time operating system (RTOS).

The reactive behavior of CFSM is synthesized in a two-step process:

- Implement and optimize the desired behavior in a high-level, processor-independent representation similar to a CDFG,
- Translate the CDFG into portable C code and use any available compiler to implement and optimize it in a specific microcontroller dependent instruction set.

A timing estimator quickly analyzes the program and reports code size and speed characteristics. The algorithm uses a formula, with parameters obtained from benchmark programs, to compute the delay of each node in the CDFG for various microcontroller architectures (characterization data for MIPS R3000 and Motorola 68HC11 and 68332 are already available). The precision of the estimator, with respect to true cycle counting, is currently on the order of ± 20 %. An application-specific operating system, consisting

of a scheduler (*e.g.* Rate-Monotonic and Deadline-Monotonic) and I/O drivers, is generated for each partitioned design.

3.2.7 Interfacing Implementation Domains

Interfaces between different implementation domains (hardware-software) are automatically synthesized within POLIS. These interfaces come in the form of cooperating circuits and software procedures (I/O drivers) embedded in the synthesized implementation. Communication can be through I/O ports available on the micro-controller, or general memory-mapped I/O.

3.2.8 Rapid Prototyping

A rapid prototyping environment is also available in POLIS based on APTIX architecture [1] [33] system.

3.3 Design Example

In order to elaborate the various design steps in POLIS, it is convenient to work through an example. A seat-belt alarm example given in POLIS user's manual [33] is highlighted in this work to explain the design methodology. Figure 3.3 shows the transition diagram for the system. The specification of the system is stated as follows:

> When the ignition key is turned on, wait for five seconds for the belt to be fastened. If the belt is not fastened within five seconds, turn the alarm on for five seconds. If the belt is fastened or the ignition is turned off, then don't turn the alarm on.

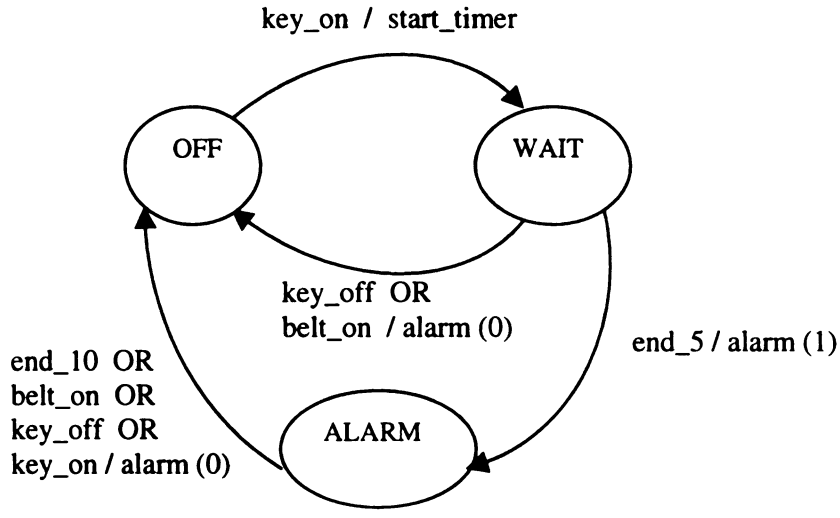

Figure 3.3: Transition diagram of seat belt controller [courtesy POLIS group].

3.3.1 Specification

The initial specification of the system is written in Esterél language. The system is divided into two modules; belt_control and timer, and functionality of each module is then described in terms of an Esterél program. The belt_control program is shown below:

```
module belt_control:
input reset, key_on, key_off, belt_on, end_5, end_10;
output alarm: boolean, start_timer;
loop
   abort
       emit alarm(false);
       every key_on do
         abort
            emit start_timer;
            await end_5;
            emit alarm(true);
            await end_10;
         when [key_off or belt_on];
         emit alarm(false);
      end
   when reset
end
.
```

The first line of each Esterél program gives a name to the module being described. The next two lines declare the input and output signals of the module. All input signals here are control signals meaning they do not have any values associated. One of the output signals *alarm* has a value of type Boolean. The **loop** statement starts an infinite loop. The **abort** statement instantaneously kills its body whenever *reset* signal is received. The **emit** *alarm* (false) signal has an associated Boolean value to turn the alarm off. The **every** *key_on* **do** statement executes its body every time *key_on* is present. The **abort** statement instantaneously kills its body whenever *key_off* or *belt_on* signals are present. The **emit** *start_timer* signal is an output to be sent to timer module. The **await** *end_5* halts the execution until *end_5* signal is received. The **emit** *alarm* (true) signal is an output signal with a value (true) to turn the alarm on. The **await** *end_10* again halts the execution until *end_10* signal is received, which means that five seconds have elapsed since the alarm was turned on. The **emit** *alarm* (false) signal then turns the alarm off and the system goes back to its initial or OFF state.

Following is a listing of timer module:

```
module timer:
constant count_5, count_10 : integer;
input msec, start_timer;
output end_5, end_10;
every start_timer do
     await count_5 msec;
     emit end_5;
     await count_10 msec;
     emit end_10;
end
.
```

The second line declares a **constant**, whose actual value will be defined later in the design. The timer starts counting whenever it receives the *start_timer* signal. The **await**

const_5 msec statement counts *const_5* transitions in which signal *msec* has an event, and then yields control to the **emit** statement that emits the corresponding output signals.

3.3.2 Estimation

The next step is to read the design into POLIS. The POLIS environment is invoked by giving the command **polis** at the command prompt. The POLIS interpreter is then used to enter POLIS commands. The SHIFT file is read by giving the command **read_shift** *filename*. To read the belt_control.shift file, the following command is used:

read_shift -a belt_control.shift ;

where **-a** means that no auxiliary file is used

In the next step an implementation choice for the module is specified, using **set_impl –s/h**, where **-s** means software and **-h** means hardware. Initially software implementation is selected unless a module is specified for implementation in hardware only. Following command sets the implementation to software:

set_impl -s

The module is assigned to the selected partition by the command **partition**. The selection of target microprocessor is done by using the command **set arch** *processor*. We will select Motorola 68HC11 as our target processor.

partition

set arch 68hc11

The estimation tools are run to get an idea of the size and speed of the resulting software. The command is:

read_cost_param

The SHIFT format is now translated to S-Graph representation by **build_sg** command. The graph is then optimized internally by POLIS and translated to C code using:

sg-to-c -d ptolemy

The PTOLEMY code is also generated for use in simulation by the command:

write_pl –d ptolemy

3.3.3 Co-simulation

Now we exit from the POLIS program and use the Makefiles, available in POLIS distribution, to create enough information to simulate the entire design in PTOLEMY. Subsequently the modules are instantiated in PTOLEMY environment as *stars* and their interconnections are described as *galaxies*. After setting various parameters we run the simulation to verify that the design behaves as expected. The design can be improved by selecting different implementation of the modules or selecting different architectures. This is a manual process therefore the quality of design depends upon the user's experience rather than the design choices available. Once the design has been verified using this mixed simulation, a real implementation is created. We again go back to POLIS and read the final design into it. An auxiliary file is also generated to indicate the interconnections between the modules. Each module is designated for implementation in either hardware or software. The synthesis process performs different operations for hardware and software.

3.3.4 Hardware Synthesis

The SHIFT files are translated to BLIF files. The command for this is:

net_to_blif

This representation is then optimized using SIS, which is embedded in POLIS. Finally the netlist is written out either as an interconnection of gates and flip-flops or as an XNF file for the Xilinx FPGAs. The complete set of commands for hardware implementation is as follows:

read_shift *filename*.shift
propagate_const
set_impl -h
set arch 68hc11
partition
net_to_blif
print_stats
source script.rugged
write_blif *filename*.blif
print_stats

For ASIC:
read_library *library*.genlib
map -W
write_blif -n *filename*.gate

For FPGA (XILINX 3000 family):
read_blif *filename*.blif
xl_merge -l -o temp.merge
write_xnf -M -m -d xilinx temp.merge *filename* _3000.xnf

For VHDL (Outside POLIS):

blif2vst *library*.genlib *filename*.gate **–o** *filename*.vhd

Alternatively the designer can also use following make utilities for hardware synthesis but some commands do not perform the required operations due to certain limitations:

make hw_opt

make xnf3000

3.3.5 Software Synthesis

The steps for creating the S-Graph are repeated with the modules for software synthesis. Finally the C code for the software modules and the operating system is generated. The command for generation of operating system is:

gen_os -d belt_part_sg

The set of commands for software synthesis is as follows:

read_shift *filename*.shift
propagate_const
set_impl -s
set arch 68hc11
partition
read_cost_param
print_cost -c
build_sg
print_cost -s
sg_to_c -d *sg_directory*
gen_os -d sg_directory

Alternatively the whole sequence can be executed by the single command:

make sw

The following command generates a VHDL simulation model of the software:

sg_to_vhdl

Alternatively the following command can be used to generate a VHDL file:

make beh_vhdl

3.3.6 Implementation

The output of POLIS can now be used to implement the system. The C code is compiled for the target processor and loaded onto a prototyping board. The hardware netlist is downloaded onto a FPGA using the Xilinx software tools.

CHAPTER 4

SYSTEM PARTITIONING

Partitioning is one of the central problems in the design of embedded systems. Hardware-software partitioning is a way of deciding that which part or task of the system is to be implemented in hardware and which in software. Finding the best partition to implement a system's functionality is a critical and challenging task. To achieve this goal a set of system components is to be selected and the system's functionality is to be allocated amongst them. The partition must lead to an implementation that satisfies a set of design constraints, such as cost, performance, size and power consumption.

Partitioning of a system specification onto a target-architecture consisting of a single CPU and a single ASIC has been investigated by a number of research groups [1, 8, 11, 12]. This architecture is of interest in situations where the performance requirements can not be met by general-purpose microprocessor or where a complete hardware solution is too costly. Different approaches have been adopted by different researchers for partitioning of a hardware-software system based on the specific application areas, such as embedded systems, DSP, software execution acceleration and hardware emulation and prototyping. One of the important issues during partitioning is the way the communication between hardware and software is taken into account. Gupta and De Micheli [8] have presented a partitioning approach which starts from an all-hardware solution. Their partitioning algorithm takes communications into account and is able to reduce communication when neighboring vertices are placed together in either software or hardware. The algorithm presented by Henkle, Ernst et al. [11] is based on simulated annealing algorithm which moves chunks of software code to hardware until timing

constraints are met. The algorithm accounts for communication and only variables that need to be transferred are actually taken into account. Kalavade and Lee [18] have presented an algorithm that also takes communications into account by attributing a fixed communication time to each pair of blocks. The COMET [16] project employs the scoreboard algorithm, which involves a three-step evaluation process for selecting the best node to move based on user supplied constraints.

4.1 Partitioning Approaches

The partitioning problem is of two types: homogenous or heterogeneous. The homogenous partitioning attempts to partition a system into a minimal number of parts that are completely implemented in either hardware or software. In case of hardware the goal is to reduce the size, whereas software implementations try to achieve a speed up in overall execution time. The focus of heterogeneous partitioning problem is to partition the system into hardware and software implementations. The problem of partitioning into hardware and software is many times more complex as compared to partitioning for implementation into purely hardware or software.

There are two different approaches to system partitioning: structural and functional. In the structural approach, the system is implemented with structure first and then partitioned. In the functional approach, the partitioning is done prior to implementation.

4.1.1 Structural Partitioning

Structural approach is very popular in hardware design because of its straightforward methods for obtaining size and pin estimates. Another factor for its popularity is the ease

with which it allows mapping of structural partitioning problem to a graph-partitioning problem, for which an extensive body of formal theory, algorithms and tools exist. It has produced good results in the past due to relatively small sizes of system components. Structural partitioning suffers from three main drawbacks.

- It is difficult to make decisions to trade off size and performance while implementing the structure because subsequent partitioning steps may nullify them. It may lead to increased size of hardware or inter-chip communication.
- With the increasing size of systems the number of objects tend to grow. This leads to poor results from partitioning algorithms. It also makes it more difficult to interact during the partitioning process.
- The structural approach is limited to hardware design. It does not support allocation of the functionality to software.

4.1.2 Functional Partitioning

In this approach, a system's functionality is first divided into non-divisible parts called functional objects. These objects are then partitioned among system components, which are implemented either as hardware or software. The functional approach is more suited to hardware-software co-design due to several advantages over structural approach.

One of the key advantages is that it is possible to make performance/design tradeoffs during the subsequent structural implementation stage with full knowledge of the partition. The performance estimates obtained during the structural implementation stage are accurate because of the complete knowledge of all data transfers between system

components. The second advantage is reduction in number of objects to be partitioned, since there are fewer functional objects than Register-Transfer level (RTL) structural objects. It is easier for the designer to interact with the partitioning algorithm while dealing with fewer objects. Functional approach naturally provides a means of hardware-software partitioning, since the partitioned objects are functional. Each object can either be implemented in hardware or software depending upon the system's requirements.

4.2 Partitioning Issues

It is easier to understand and compare various partitioning techniques if the partitioning problem is divided into seven essential issues as depicted in Figure 4.1 and described below.

4.2.1 Specification and Levels of Abstraction

Partitioning techniques greatly depend upon the specification language used and the abstraction level of the conceptual model. The language alone may not efficiently describe the functionality in terms of specifications, since the same language can be used to represent many different conceptual models. It is, therefore, important to define the input at an appropriate level of abstraction. At low level of abstraction, the system is defined as a large number of low-complexity objects. At higher levels the system consists of small number of high-complexity objects.

The highest abstraction level is the task level. Here the conceptual model does not define the actual computations but the data transfers between the tasks and other characteristics such as size or delay are defined. A data flow graph of tasks and

behavioral level description are examples of task level abstraction. At the next lower level, the system is described by a control data flow graph (CDFG), which includes arithmetic and control operations. It is the most-used level in DSP applications and a variety of partitioning algorithms is based on this representation. The next level involves FSMs for system description. They may be simple FSMs or complex ones like CFSMs in POLIS. At RTL the input represents a set of register transfers for each operation. The lowest level of abstraction is structural interconnection of physical components commonly known as netlist.

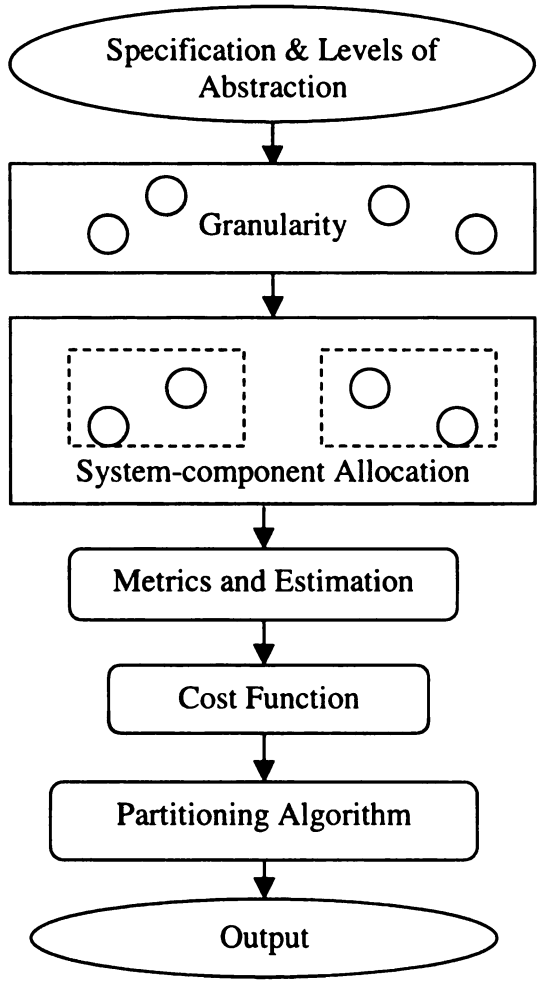

Figure 4.1: Essential partitioning issues [2].

It is possible to define a single input specification with multiple parts at various levels of abstraction depending upon the intermediate implementations during the design.

4.2.2 Granularity

The extent of decomposition of the input specifications decides the granularity of the functional objects. A large number of objects indicates fine granularity with small amount of specification allocated to each object; whereas coarse granularity implies a small number of objects, each with a lot of functionality assigned. The granularity greatly depends upon the level of abstraction. The task level has the most flexibility for decomposition, where each task could further be decomposed into procedures and statement blocks. Fine granularity is better suited for partitioning optimization but it has certain drawbacks, such as more computation time, difficulty in recognizing fine-grained objects and difficulty in estimation.

4.2.3 System-Component Allocation

A partitioning algorithm needs to know the types of system components to which the functional objects may be mapped. These include programmable devices such as processors with associated memories, I/O devices and buses as well as dedicated hardware such as ASICs, FPGAs and standard logic components. Selection of these components is greatly influenced by the specifications and constraints of the design.

4.2.4 Metrics and Estimation

Metrics are the attributes of a partition that determine its goodness. These include monetary cost, execution time, program size, hardware area, power consumption, communication bandwidth, testability and reliability.

Some techniques adopt an incremental approach towards partitioning, where objects are grouped one at a time. A new type of metric is required for this technique that could predict the benefit of grouping any two objects. Such metrics are called closeness metrics. These metrics help in reducing communication between hardware and software by grouping functions sharing same variables in software or data paths in hardware.

Computation of metrics is another challenge that needs to be addressed during the co-design process. There are two options available for computing the metrics. Either the system is actually created, resulting in accurate metric values or a rough implementation is created resulting in estimated values. The second option is also known as estimation. An estimation process must posses speed and accuracy. These are two conflicting requirements because speed requires less amount of detail in the implementation, whereas the accuracy requires detailed implementation. The lower the level of abstraction the better are the estimates.

4.2.5 Cost Function

A cost function defines the goodness of a partition in terms of various metrics. Different metrics of a system have conflicting requirements and it is very important to assess their combined effect as a single value. The cost function is a linear weighted-sum expression of the products of metrics and their associated weights, where weight

indicates each metric's relative importance to the goodness of partition. For example the following cost function is a weighted-sum expression of four metric values; hardware-area (A), program-size (S), program-execution-time (T) and hardware-delay (D) are weighted by constants w_1, w_2, w_3 and w_4 respectively, and then summed:

$$Cost\_Func = w_1 \cdot A + w_2 \cdot S + w_3 \cdot T + w_4 \cdot D$$

selecting a larger value for w_1 than w_2, w_3 and w_4 makes the hardware-area more important metric.

It is more common to incorporate the constraints into the cost function to help make constraints driven design decisions. The partitions that meet the constraints are considered better than those that do not. A generalized cost function, incorporating the constraints can be written as follows:

$$Cost\_Func = \sum_{i=1}^{n} w_i \cdot (m_i - c_i)$$

where i, is the index of metric, n is the total number of metrics, w_i is the weight for the ith metric, m_i is the ith metric and c_i is the constraint on ith metric.

It is also important to take care of the units of various metrics included in the cost function. It would not be a good idea to compare units of area with the units of delay. To overcome this problem normalization of metric's units is carried out. It is achieved by dividing the difference of the metric and constraint by the constraint. A cost function employing this technique is given below:

$$Cost\_Func = \sum_{i=1}^{n} w_i \cdot (m_i - c_i)/c_i$$

While a cost function combines metrics to evaluate a partition, a closeness function combines closeness metrics to indicate the desirability of grouping the objects, before a complete partition is formed.

4.2.6 Partitioning Algorithm

For a given set of functional objects and a set of system components, it is the goal of a partitioning algorithm to find a partition with lowest cost as computed by the cost function. With a relatively large number of functional objects it would take a large amount of computations to evaluate all possible partitions. The algorithm must therefore be capable of choosing a subset of all possible partitions, evaluate each of them and find the best partition that meets the constraints. The partitioning problem can be defined as follows [2]:

Given a set of Objects $O = \{o_1, o_2, \ldots.o_n\}$, determine a partition

$P = \{p_1, p_2, \ldots.p_m\}$ such that $p_1 \cup p_2 \cup \ldots p_m = O$,

$p_i \cap p_j = \phi$ for all i, j, $i \neq j$, and the cost determined by an objective function

Cost_Fuct (P) is minimal.

4.2.6.1 Constructive/Iterative Algorithms

Partitioning algorithms can be generally classified as constructive or iterative. Constructive algorithms group objects into a complete partition. These algorithms use closeness metrics to group objects to achieve a good partition. The Iterative algorithms repeatedly modify a partition for improvements. They use an objective cost function to

evaluate the partition, which yields better results as compared to closeness functions used by the constructive algorithms.

4.2.6.2 Greedy/Hill-climbing Algorithms

There are two types of partitioning algorithms being used for hardware-software co-design: greedy and hill-climbing. Greedy algorithms start with an initial partition and move objects to the opposite group as long as the cost is improved. They cannot escape a local minimum. The concept of local minimums is depicted in Figure 4.2. Point A is a local minimum, whereas point B is the global minimum. Hill-climbing algorithms have the capability of escaping local minimum. This is achieved by accepting cost increasing moves during the iterations as shown in Figure 4.2.

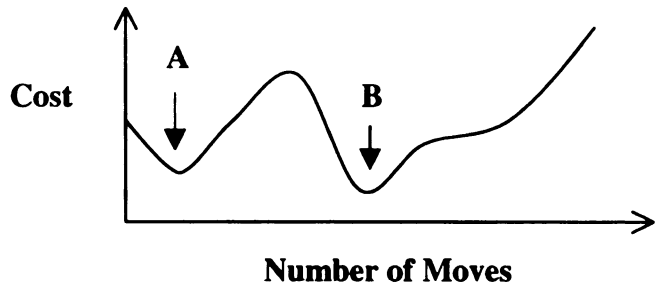

Figure 4.2: Escaping local minimum in iterative partitioning [2].

4.2.7 Output

The output of a partitioning algorithm must be comprehendible by the user who has to perform the next step of design. The output may be a list of objects indicating which object is to be implemented in hardware and which in software. It may be a new version of specifications that contains information about the structural implementation of system components. The output may be machine-readable so that a synthesis tool could use it to map the functions onto hardware and software components.

4.3 Basic Partitioning Algorithms

Most of the partitioning algorithms adopted by hardware-software co-design researchers are commonly used in hardware partitioning. Iterative techniques such as simulated annealing (SA), Kernighan-Lin (KL), Fiduccia-Mattheyses (FM) and genetic algorithms (GA) are some examples. Hardware partitioning provides a means of breaking a system into smaller, more manageable parts based primarily on the number of communication channels between them. Many partitioning approaches have been developed which incorporate an ancillary goal of balancing the relative sizes of the parts. However, during hardware-software partitioning, this goal is not relevant because the system has to be partitioned into two parts only, *i.e.*, hardware and software. Given an initial partition of the system into two parts, iterative techniques move one or more objects between the partitions in an effort to minimize objective function cost. Figure 4.3 shows a classification of partitioning algorithms.

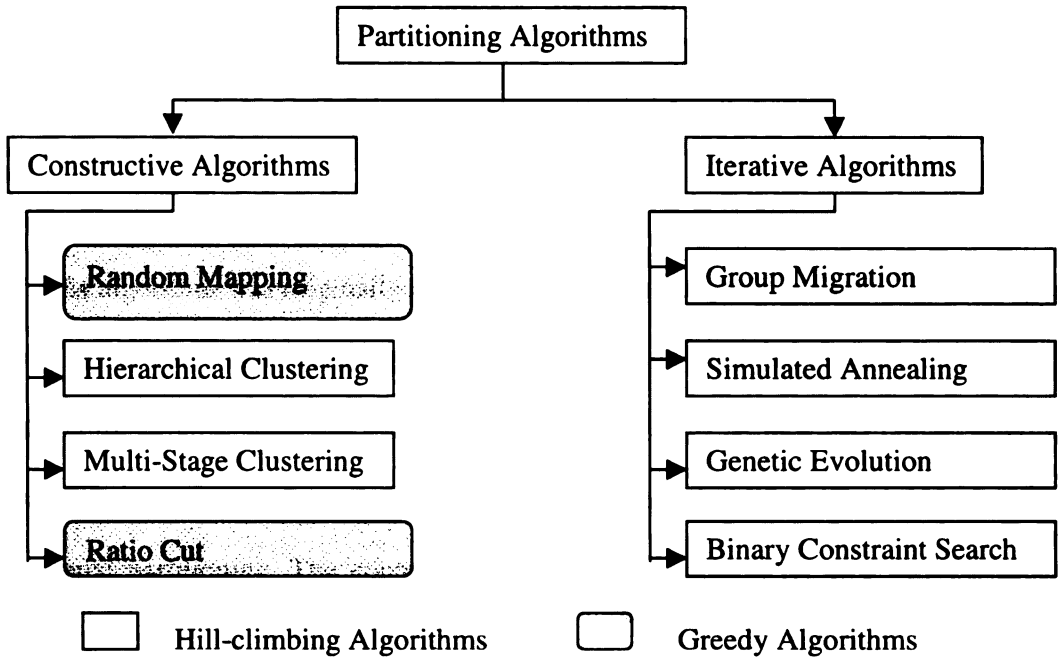

Figure 4.3: Classification of automatic partitioning algorithms.

4.3.1 Vulcan

The Vulcan [7] system uses two types of greedy algorithms. Vulcan I, deals with hardware partitioning only, while Vulcan II attempts to reduce hardware cost by partitioning the functionality between hardware and software. Both algorithms adopt an iterative approach to find the best partition.

4.3.2 Ratio-Cut

The ratio-cut algorithm is a constructive algorithm that was originally developed for structural partitioning. This algorithm is very effective when large number of objects have to be moved between the partitions. The algorithm groups objects till the time no

object is considered suitable to be merged. Scoreboard algorithm [16] is a version of ratio-cut method that is used for hardware-software partitioning in the COMET system.

4.3.3 Group Migration (Kernighan-Lin)

This algorithm was originally proposed by Kernighan and Lin [34] in 1970 for graph partitioning to improve two-way partitions. It has been modified over time by Fiduccia and Mattheyses [35] and later by Krishnamurthy [36] to require less computation and obtain better results. This algorithm yields excellent results for both structural and functional partitioning. The control strategy of the algorithm for two-way partitioning may be stated as follows:

> For each object, determine the decrease in cost if the object is moved to the other group and then move the object if it generates the greatest decrease or smallest increase in cost. In order to prevent an infinite loop caused by moving the same object back and forth between partitions, each object can be moved once only. After all the objects have been moved once, the lowest cost partition is selected as the initial partition and another iteration is done.

A Group migration algorithm with two way partitioning is listed in figure 4.4. The algorithm starts with an initial two-way partition $P_{init} = \{P_1, P_2\}$, where P_1 may be termed as hardware partition and P_2 as software partition.

```
P = P_init                          //  Start with an initial partition
Loop                                //  Iterative Loop
    P_prev = P                      //  Previous Partition is equal to new Partition
    Cost_prev = Cost_Func(P)        //  Previous Cost is equal to the cost of new Partition
    Cost_best_P = ∞                 //  Best Partition Cost is equal to infinity
    Loop for each Obj_i             //  Initialize status of each Object
        Obj_i.moved = false         //  Moved flag of each object is set to false
    End Loop
    Loop for i=1, n                 //  Create a sequence of n moves
        Cost_best_move = ∞          //  Best_move Cost is equal to infinity
        Loop for each Obj_i.moved == false
            P = Move(P, Obj_i)      //  Create new partition by moving Obj_i to opposite side
            Cost = Cost_Func (P)    //  Calculate Cost of new Partition
            If Cost < Cost_best_move then
                Cost_best_move = Cost   //  Best_move Cost is equal to new Cost
                Obj_best_move = Obj_i   //  Best_move Object is now Obj_i
            End if
        End Loop
        P = Move(P, Obj_best_move)      //  Create new partition
        Obj_best_move.moved = true      //  Set moved flag to true
        If Cost_best_move < Cost_best_P then
            P_best_P = P                    //  Best Partition is P
            Cost_best_P = Cost_best_move    //  Best Partition Cost is Best_move Cost
        End if
    End Loop
    If Cost_best_P < Cost_prev then
        P = P_best_P                //  P is the Best Partition of this iteration
    Else return P_prev              //  Otherwise return the previous Partition
End Loop
```

Figure 4.4: Group migration algorithm with two-way partitioning [2].

After initialization of certain variables and setting the flag, *moved*, of each object as false, a sequence of n iterations is generated. During each iteration, for each object with a false flag, a function Move(P,Obj_i) is called that moves object (Obj_i) to the opposite side and returns a new partition. The cost of each new partition is calculated and the minimum cost ($Cost_{best\_move}$) found during these moves is saved along with the object (Obj_{best_move}) that generated the minimum cost partition. Once all moveable objects have been moved,

all possible partitions would have been created and their costs would have been compared. The object (Obj $_{best\_move}$) giving the minimum cost partition is now permanently moved to the opposite side and its flag is set to true so that it will not be moved during the next iteration. A new partition P is thus, created by this operation and its cost (Cost $_{best\_move}$) is now compared with the best partition cost (Cost $_{best\_P}$) saved earlier. If the cost is lower than the previous cost then the current partition P is saved as the best partition (P $_{best\_P}$) along with its cost (Cost $_{best\_P}$). During the next iteration one less object would be moved and all possible partitions would again be compared. After n iterations the best partition found so far, is returned by the algorithm. The algorithm is iterated by the outer loop till no improvement in cost is generated.

The whole process may be repeated with a certain number of new initial partitions and the results may be saved to find the best partition. During the experiments with this algorithm it has been observed that the number of outer loops in the algorithm is indeed less than five, which is in conformity with the earlier observation [34].

4.3.4 Simulated Annealing

This algorithm is similar to group migration in that it also accepts cost increasing moves in order to escape local minimums. But it also allows multiple moves of an object, while limiting the complexity by decreasing the tolerance for accepting cost increasing moves. The algorithm is intended to model the annealing process in physics where a material is melted and its minimal energy stage is achieved by lowering the temperature slowly enough that equilibrium is reached at each temperature.

The algorithm starts with an initial partition and an initial simulated temperature. While the temperature is being slowly decreased, for each temperature, random moves are generated. The move that improves the cost is accepted, otherwise the move may still be accepted but such acceptance at lower temperatures becomes less likely. Figure 4.5 lists the simulated annealing algorithm.

```
Temp = Initial temperature          // Assign a high value to Temp
Cost = Cost_Func(P)                 // Calculate cost of initial partition
Loop while not Frozen               // Start of outer loop, continues until Temp > freezing
    Loop while not Equilibrium      // Start of inner loop, continues till equilibrium
        P_tent = Move_Random(P)     // Create tentative partition
        Cost_tent = Cost_Func(P_tent) // Calculate cost of partition
        Cost_delta = Cost_tent – Cost  // Calculate ΔCost
        If (Accept(Cost_delta, Temp) > Random(0, 1) then
            P = P_tent              // Tentative partition is the new partition
            Cost = Cost_tent        // Cost of new partition
        End if
    End Loop
    Temp = α × Temp                 // Lower the teperature
End Loop
```

Figure 4.5 Simulated annealing algorithm [2].

A variable *Temp*, is initialized with a high value and the cost of current partition is calculated. Then a sequence of random moves is generated inside a two-level loop. The moves are accepted until an equilibrium state is reached, that is when for a certain number of moves the cost is not improved. At this point the temperature is lowered by multiplying the old temperature with a variable α, where $0<\alpha<1$; and the inner loop is repeated until the equilibrium is again reached. The outer loop continues until the temperature has reached a certain smallest value specified as freezing point, which means the termination of process. The Accept function determines whether to accept a move

based on the cost improvement and current temperature. This function is defined in [37] as:

$$Accept(\Delta Cost, temp) = min(1, e^{-\frac{\Delta Cost}{temp}})$$

When ΔCost is negative, meaning the tentative partition is better than the current one, the function returns '1'. Otherwise it returns a value in the range of [0,1]. The function Random (0,1) returns a value in the range of [0,1].

It has been theoretically shown that simulated annealing algorithm can climb out of a local minimum and find the optimal solution if the process reaches equilibrium state at each temperature and if the temperature is lowered infinitely slowly. This approach requires infinite iterations at an infinite number of temperatures, which is not practicable. Several heuristic approaches have been developed to control the process. These heuristics define the equilibrium-state and describe how to lower the temperature.

4.3.5 Genetic Evolution

In contrast to the group migration and simulated annealing algorithms, where the best partition is saved in each iteration, this class of algorithms saves a set of partitions between iterations. In such algorithms the set of partitions is referred to as a generation. Genetic algorithms create a new generation by imitating three evolutionary processes found in nature, namely selection, crossover and mutation.

4.3.6 Binary Constraint-Search

This algorithm involves finding the first zero cost solution in the sequence of constrained partitions. The first zero-cost solution is the minimal constrained solution. The algorithm performs a binary search through the range of possible constraints, applying partitioning and then the cost function as each constraint is visited.

4.3.7 Integer Linear Programming

Linear program formulation is another technique applied in partitioning. It consists of a set of variables, a set of linear inequalities that constraint the values of the variables and a single linear function of the variables which serves as an objective function. The goal is to choose values for the variables to satisfy all inequalities and minimize the objective function.

CHAPTER 5

APPLICATION OF PARTITIONING ALGORTIHMS IN POLIS

5.1 Background

Hardware-software partitioning is one of the main challenges in co-design of embedded systems, as it has a crucial impact on the cost and performance of the resulting product. Several approaches have been presented for system partitioning. They differ in the initial specification, the level of abstraction, granularity of the partition, the degree of automation of the partitioning process, cost function and the partitioning algorithm. The system is automatically partitioned in [3, 8, 11, 12, 16, 19, 27], while the partitioning process is manual in [1, 17, 28, 30].

For small systems with a very well understood structure and functionality, the number of realistic alternatives in hardware-software partitioning is small. Thus, a designer, based on his experience can easily assign functionality to hardware and software domains using an ad-hoc approach. This design can then be refined through simulation and/or in-system evaluation. The main problems that the designer has to deal with in this approach are hardware/software synthesis and co-simulation.

For a large system with a large number of interacting components implementing a complex functionality, the choice of partitions is extremely large. It is not possible to evaluate their impact on the cost and performance of the system without the support of a design tool. A computer-aided partitioning scheme is therefore, essential to support the evaluation of different partitioning schemes based on adequate cost functions and estimations.

5.2 Partitioning in POLIS

As discussed in Chapter 3, POLIS does not include any hardware-software partitioning algorithms. It only offers a flexible environment that supports manual partitioning. This environment supports evaluation of tradeoffs via simulation rather than mathematical analysis. Instead of using separate simulation models for hardware and software, POLIS uses the same model for both types of components.

POLIS has certain peculiarities that distinguish it from its contemporary frameworks. The POLIS system is based on a CFSM model of computation. The input specifications of the system are described as a network of communicating CFSMs at the behavioral level. Each CFSM may represent a function or a group of functions depending upon the granularity of the objects. Each CFSM can be implemented either as hardware or software. Based on the architecture selection, the software and hardware cost estimates are generated. The CFSM specifications after translation to C code are passed to PTOLEMY for simulation. The partitioning is based on the results of simulation.

In order to automate the partitioning process, the POLIS design flow of Figure 3.1 had to be altered. We decided to replace the co-simulation tool (PTOLEMY) and change the normal design flow by generating the estimates for both hardware and software implementations of each CFSM, using them in an automatic partitioning algorithm and feeding the results back into POLIS for synthesis and implementation. The modified design flow is shown in Figure 5.1. The new design flow incorporates an interactive automatic partitioning process. The estimates generated by POLIS are tabulated in a text file for use by the automatic partitioning algorithm. The user adds the constraint data to this file and performs various iterations of the partitioning algorithm to analyze the

effects of the changes in constraints and architecture selection. As a result of this analysis the best hardware-software partition satisfying the design constraints is selected. An auxiliary file indicating the choice of implementation for each CFSM is created and used in POLIS for final synthesis and implementation.

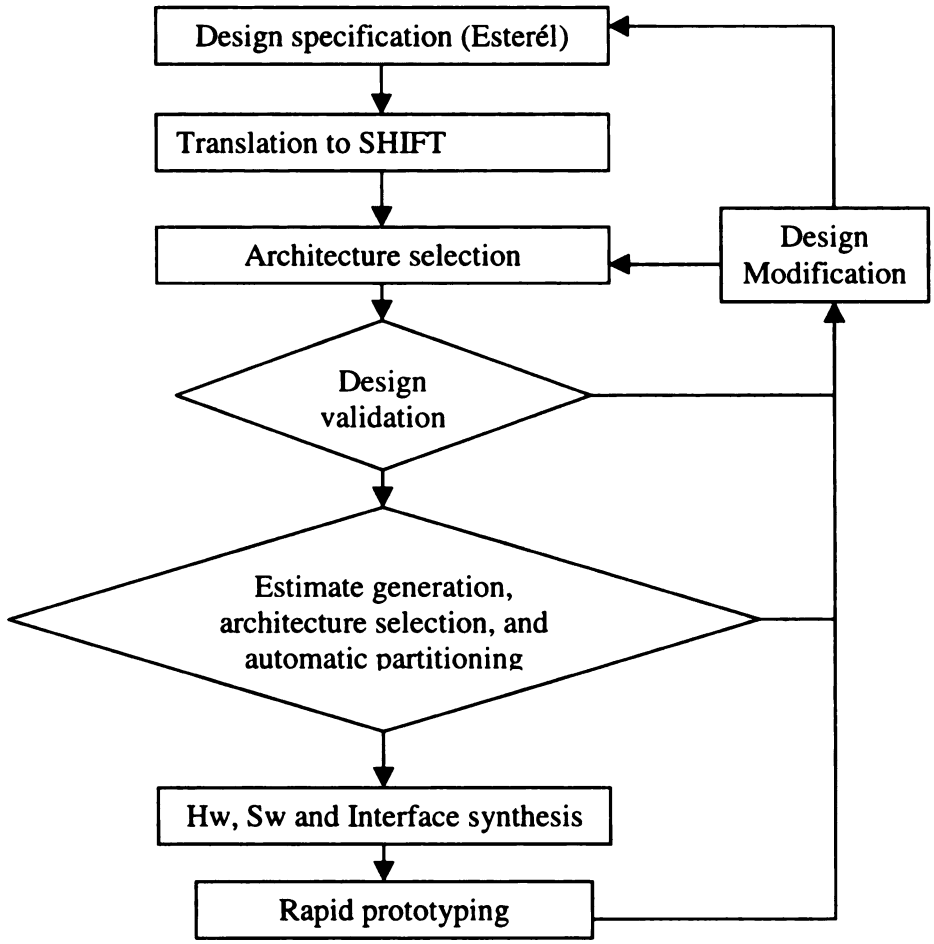

Figure 5.1: Modified design flow in POLIS.

5.3 Generation of Estimates

POLIS generates software estimates for 68HC11 and R3000 processors in terms of minimum and maximum execution times in clock cycles and code size in bytes for each

CFSM. SIS, a system for sequential circuit synthesis, has been embedded in POLIS to get estimates for technology dependent hardware implementation. For this purpose a technology library has to be generated on which to map the BLIF file. The hardware estimates include area and maximum and minimum delays of the CFSM. This information is calculated on the basis of technology library passed to SIS along with the BLIF file. The SIS only provides information on total area of gates and the total delays. This information does not include any estimate of total chip area, which would require placement and routing as well as calculation of interconnection delays.

POLIS is also capable of generating XILINX Netlist Format (XNF) files for a given family of FPGAs. Mapping the design onto a specific FPGA family generates this information. Presently it is possible to generate XNF files for Xilinx 3000 family only. The XNF file has to be passed to XILINX design manager software, which performs map and route functions for that particular FPGA chip. The results of this operation include CLB, IOB counts as well as delays on each signal. This information is more reliable because the hardware is actually mapped onto the FPGA during this process. But it takes more time and involves porting of code from one platform to the other and tabulation of results for use by the partitioning algorithm. The existing algorithm used in POLIS for mapping the BLIF file onto a FPGA is not reliable as, it sometimes fails to generate the XNF file when the number of variables in a Boolean expression is more than the maximum number of inputs allowed for a CLB.

The following script-files were written to generate estimates for software and hardware implementation of each CFSM:

Script file for generation of software estimates:

read_shift *filename*.shift
propagate_const
set_impl -s
set arch 68hc11
partition
read_cost_param
print_cost -c
build_sg
print_cost -s

Script file for generation of hardware estimates:

read_shift *filename*.shift
propagate_const
set_impl -h
set arch 68hc11
partition
net_to_blif
print_stats
source script.rugged
print_stats
write_blif *filename*.blif
rlib *libraryname.genlib*
map -W
print_map_stats
// for mapping onto Xilinx 3000 family
read_blif *filename*.blif
xl_merge -l -o temp.merge
write_xnf -M -m temp.merge *path/filename* _3000.xnf

These script files were used to read the individual CFSM SHIFT files into POLIS and perform various steps for estimate generation. The estimates were tabulated in an input file for the partitioning algorithm. The results of partitioning were then fed to POLIS for further processing as usual.

It would be advantageous to have estimates of the cost of communication between the CFSMs as well. That would indicate the amount of information being exchanged between

any two CFSMs. If the two CFSMs are closely associated due to sharing of data, then implementing them in different domains may affect the overall performance of the system. The partitioning algorithm would try to move such CFSMs together from one domain to another based on the value of this index. The cost function would include an additional metric, *i.e.*, communication cost. If the two CFSMs having a large value of communication cost with respect to each other are in the same domain, then their communication cost estimate would be zero, indicating no communication overhead in terms of hardware-software interface. The cost function would add only those communication costs, for which the two CFSMs involved are not in the same domain.

5.4 Selection of Algorithms for Automatic Partitioning in POLIS

The selection of algorithms is based on various factors, including abstraction level, granularity of objects, metric values and estimates, and design constraints. The target architecture in POLIS consists of a single processor and one or more ASICs connected to the processor. The objective is to transfer only the essential functionality to hardware to improve performance. This results in lowering the overall complexity and cost of the system.

The choice of group migration (GM) and simulated annealing (SA) algorithms was made on the basis of their hill-climbing nature and iterative approach. Both algorithms are well-suited to the level of abstraction in POLIS as well as the granularity of the objects. The algorithms try to find a minimum-cost partition based on performance estimates and constraints. The cost function developed for use in the partitioning algorithms includes four metrics: hardware area (A), software code size (S), hardware

delay (D), and software execution time (T). The constraints on these metrics were also incorporated in the cost function to find the best partition that satisfies the constraints.

The cost function is as follows:

$$Cost\_Func = w_A \cdot f(A, C_A) + w_S \cdot f(S, C_S) + w_D \cdot f(D, C_D) + w_T \cdot f(T, C_T)$$

where

$$f(x, y) = \frac{x - y}{y}$$

and

C_A is the constraint on hardware area,
C_S is the constraint on software code size,
C_D is the constraint on hardware delay, and
C_T is the constraint on software execution time

The methodology of the GM and SA algorithms has already been discussed in chapter 4. Two C^{++} programs were developed to implement these algorithms. The source codes of programs for GA and SA algorithm are listed at Appendix A and Appendix B respectively.

5.4.1 Assumptions

For the purpose of completing our investigation into partitioning, we have assumed that the estimates generated by POLIS are sufficiently accurate. Any estimation process can not be expected to generate accurate data [2]. The estimates give a rough idea of the size and timing characteristics of the modules, based on which a partitioning algorithm could partition the system. This partition may not be the best for final implementation but nevertheless it provides a starting point. Co-design is an iterative process, therefore, as the design progresses, the metrics are refined and more accurate information becomes available for use by the partitioning algorithm to improve the design.

5.5 Features of C++ Code

5.5.1 Group Migration (GM) Algorithm

The C++ code implemented for the group migration algorithm is capable of reading an input file containing estimates of various metrics for each object's implementation in hardware and software. The input file also contains the constraints on each metric as well as the associated weights. The number of metrics to be used in the cost function and the number of objects are constant for a particular project. The variables and functions used in the program are explained in appendix A. The flowchart of the program is shown in Figure A.1.

After reading the input file, the program prompts the user to input the processor clock frequency (**clk**). Based on the input data the program generates an initial partition **P_in**. This is done by first defining an all-software partition except for those objects which are already assigned to hardware. A loop is started with counter **y**=0, which continues until **y**≥**nn**, *i.e.*, number of objects in the system. The initial partition is passed to a function **Gen_part()**, which generates a random partition **P**. A do loop is started that continues while **bestpart_cost** is less then **prev_cost**. A counter **z** keeps track of the number of iterations of this loop. The cost of partition **P** is calculated using **Cost_fct()** and saved as **prev_cost**. The variable **bestpart_cost** is assigned a high value **infi**. Flag **moved** of all objects is set to false except for those, which are already locked in hardware or software and current partition is saved as **prev_P**. The algorithm starts a loop with counter **j**=0, which continues until **j**≥**nn**. The variable **bestmove_cost** is assigned the high value **infi**. Another loop is started with counter **k**=0 that continues until **k**≥**nn**. Counter **k** is used as the index for accessing a particular object from the array for **nn** objects and to point to its

corresponding position in the partition array. If the flag **moved** of the indexed object is false and it is not locked then a trial partition is generated by moving this object to the other side using **Move()** function. If the object was originally mapped to software then it is moved to hardware and vice versa, and the **cost** of the trial partition is calculated. If the **cost** is less then the **bestmove_cost** then the new **cost** is saved as the **bestmove_cost** and the index of current object is also saved as **bestmove_obj**. At the end of this loop all moveable objects have been moved once and the cost of best move *i.e.*, **bestmove_cost** and the index of corresponding object *i.e.*, **bestmove_obj** have been saved. Now this object is actually moved to form a new partition **P**, and the **moved**-flag of this object is set '**true**' to indicate that the object has been moved during the current iteration.

If the **bestmove_cost** is less then the **bestpart_cost**, then the current partition **P** is saved as **bestcost_P** and **bestmove_cost** is saved as the **bestpart_cost**. The **bestpart_cost** is written to output file for plotting and analysis. The algorithm repeats the above process until all objects have been moved to the other side *i.e.*, **j≥nn**. If the resulting **bestpart_cost** is less then the **prev_cost,** then the best partition **bestcost_P** is saved as the new partition **P** otherwise the previous partition **prev_P** is saved as **P**. The algorithm iterates with **P** as the new partition and continues the above process while **bestpart_cost** is less than **prev_cost**. When the above condition fails the algorithm stops and the best partition **best_P** found so far and the **best_cost** are saved. The algorithm is repeated with randomly formed initial partitions, until **y≥nn** condition is reached during which a global minimum is usually found.

The results are saved in an output file for use in final implementation. The user can change the clock frequency of the processor at the start of the execution of program to

select a particular speed of software execution. Higher clock speed reduces the size of hardware and helps in assessing the effect of higher speed processor on overall system performance. The code is portable and runs equally well on PCs and Sun workstations. The algorithm was applied to two hypothetical cases with five and 50 objects, respectively, and was found to converge to the lowest cost partition in both cases. The complexity of the algorithm is $O(n^3)$.

5.5.2 Simulated Annealing (SA) Algorithm.

The C^{++} code implemented for SA algorithm reads an input file with metric, constraint and the weight values. The number of metrics to be used in the cost function and the number of objects are constant for a particular project. The variables and functions used in the program are explained in appendix B. The flowchart of the program is shown in Figure B.1.

After reading the input file the program prompts the user to input the processor clock frequency (**clk**) in MHz. Based on the input data the program generates an initial partition **P_in**. This is done by first defining an all-software partition except for those objects which are already assigned to hardware. The program then generates a random partition **P** using **Random_gen()** function. The temperature is initialized by assigning **temp = init**. The initial value **init** and the lowest value **freez** are two important factors affecting the speed of convergence or number of iterations performed by the algorithm. The **cost** of initial partition is calculated using **Cost_fct()** and saved as **best_cost** for comparison with the cost of new partitions later in the program. A loop is started with a counter **y**=0, that continues while the **temp** is greater than the **freez** value. Two more counters are

initialized, *i.e.*, **n**=1 and **z**=0. Another loop is started that continues while **n** is less than **n_eq.** During this loop the current partition **P** is assigned to tentative partition **P_ten.** An object is randomly moved in **P_ten** from hardware to software or vice versa using **Random_move()** function.. After each move the cost **cost_ten** of the tentative partition **P_ten** is calculated and the difference **delta_cost** of the tentative cost and original cost is determined. A **random** number in the range [0,1] is also generated for use in **Accept()** function. The values of current temperature **temp** and **d_cost** are passed to the **Accept()** function, and the returned value from the function is compared with the **random** number earlier generated. If the value of **Accept()** function is greater than the **random** number, then the tentative partition is accepted, *i.e.*, **P=P_ten**. However, if the partition is not accepted or the tentative cost **cost_ten** is greater then **cost** then the counter **n** is incremented by one meaning that there is no improvement in cost. The equilibrium state is reached if for a certain number of moves **n_eq** the tentative cost is found to be greater than the previous cost or the value of **Accept()** function is less then the random number. In either case the value of **cost_ten** is assigned to **cost**. The counter **z** is incremented to keep track of the iterations carried out in reaching the equilibrium. Once the counter **n** has incremented to the value **n_eq** , it means that equilibrium is reached and the loop is collapsed. The terminal value **n_eq** was determined after extensive experimentation and consultation of results of earlier applications of simulated annealing algorithm [38][39].

Upon reaching equilibrium, the temperature is lowered to a new value by multiplying the old value with the constant **alpha**, selection of which also affects the convergence of algorithm. The value of **alpha** depends upon the number of objects comprising the system being partitioned. If the **cost** of the new partition is less than the **best_cost**, saved

earlier then the new value is saved as the **best_cost** and the partition is saved as **best_P**. The counter **y** is then incremented and the algorithm repeats with the new value of temperature and the process continues until freezing point is reached, *i.e.*, **temp** becomes less then or equal to **freez**. At that point the algorithm should have converged to the lowest cost partition.

The program generates two output files, one containing a list of cost saved at each equilibrium state, for plotting and analysis. The second file contains the results of partitioning with allocation of objects to hardware and software mentioned against each. The program is capable of analyzing the effect of different clock speeds on the execution speed of objects mapped to software.

5.6 Case Studies

5.6.1 Hypothetical Cases

Both algorithms were tested using two sets of hypothetical data and were found to converge to a minimum cost partition. The results for various selections of cost estimates and constraints in case of GM algorithm are shown in Appendix C. The results of experimentation with various values of initial temperature **init,** final temperature **freez**, terminal value **n_eq** of counter **n,** and multiplier **alpha** for SA algorithm, are shown in appendix D. It was observed that GM algorithm always finds the lowest cost partition, but requires a large number of iterations. The SA algorithm is faster and takes fewer iterations but may not always find the lowest cost partition in the first attempt.

5.6.1.1 Small Size Example

A system consisting of a set of five objects was taken as an example to analyze the behavior of the algorithms. The hardware and software estimates and constraints for these objects were selected in such a way to force various selections by the algorithms. Appendices C and D lists this information. Both programs found the same minimum cost partition for the system with the same values of estimates assigned to modules. The resultant partitioning choices are also shown in appendices C and D, along with the plots of cost vs. iterations are given in Figures C.1 and C.2 for group migration and Figures D.1 to D.6 for SA algorithms respectively. It is quite evident that both algorithms did escape local minimums and SA algorithm did converge to best partition within a limited number of iterations.

5.6.1.2 Large Size Example

A set of 50 objects was taken as a large example for further analysis of the software. Different values of the hardware and software estimates and constraints were selected to force various selections by the algorithms. Appendix C and D list this information. Both programs were found to successfully find the minimum cost partition of the system. The partitioning choices corresponding to different values of weights are shown in Tables C.3 to C.5, and Tables D.7 and D.8 for GM and SA algorithms respectively. The plots of cost vs. iterations are also shown in Figures C.2 and C.3 for GM algorithm and Figures D.7 and D.8 for SA algorithm respectively.

5.6.2 Real World Example: Dashboard Controller

A real world example has been included in the POLIS distribution for experimentation by the new users. The dashboard controller consists of five subsystems, namely, tachometer, speedometer and odometer, fuel gauge, temperature gauge and seat-belt alarm system. The dashboard controller interacts with various parts of the automobile and controls display of information on various dials and indicators. For the purpose of design implementation, the functionality is divided amongst three sub-systems each consisting of certain number of modules. This example also makes use of some pre-built modules representing processor peripherals and on-board hardware support such as counter/timer. A list of modules in each subsystem, identified as net and the complete dac_demo.aux file are given at appendix E. The modules' interaction is defined in terms of various galaxies in PTOLEMY for simulation. The complete system is simulated in PTOLEMY by generating various inputs at different rates to simulate the varying engine r.p.m. and vehicle speed. The output of the controller is also required to be at a certain rate to update various indicators. Based on the results of co-simulation the system is partitioned between hardware and software.

5.6.2.1 Design Constraints

The input to odometer and speedometer is wheel-pulses. The wheel-pulses are received from the wheel at a rate of 4 pulses per revolution. Assuming a standard tire circumference of 0.66 m, the maximum vehicle speed of 260 Km/h would generate an input frequency of 438 Hz for the speedometer and odometer. The engine speed is in the range of 0 to 8000 r.p.m. One pulse is generated for each engine revolution, therefore it

corresponds to a maximum frequency of 134 Hz. The outputs for odometer and tachometer must be produced at a rate of at least 100 Hz and with a maximum jitter of 100 microseconds to drive the gauge coils.

The system should be capable of receiving the above inputs and generation of outputs at a rate equal to or more than the frequency required. This can be translated as hardware delay and software execution time constraints. We selected the hardware delay constraint to be 10 microseconds to force mapping of maximum modules in software. The software execution time constraint was set at 10 milliseconds corresponding to requirement of updating the meters at 100Hz. Initial hardware area constraint was selected as 10^6 units and the software size was restricted to 8 K Bytes.

In order to perform automatic partitioning of this system the software and hardware estimates of all CFSMs were required. POLIS successfully generated the software estimates for all modules of the dashboard, but while generating hardware estimates numerous problems were encountered. POLIS is still in development phase, and it does not support hardware synthesis of all arithmetic or scientific functions. Some of the modules used in the example perform division and some require sinusoidal functions. For each function used in the design a software and hardware library file is required in POLIS. Unfortunately the hwlib.blif file does not have these functions included. At present, these functions are implementable in software only. It was beyond the scope of this work to write the libraries for these functions as the codesign group at UC Berkeley is already in the phase of enhancing the capabilities of POLIS and version 0.4 is due to be placed on the Internet later this year. However in order to prove the validity of partitioning algorithm those modules were permanently mapped to software during the

partitioning process and the results were then compared with the recommended partition in [1].

5.6.3 Results Obtained

Table 5.1 shows the estimates, constraints and weights data for the dashboard example that was used as input to the partitioning algorithms. These estimates are based on the selected architecture consisting of a 68HC11 microcontroller and a sample ASIC technology library included in SIS software distribution. The units of hardware area are assumed to be in micro-meters sq., the software code size in bytes, software execution delay in clock cycles and hardware delay in nanoseconds. A detailed analysis of dashboard example with different values of constraints and clock speeds to explore the partitioning choices is presented at appendix E.

The results of the automatic partitioning using GM and SA are shown in Table 5.2 and Table 5.3 for processor clocks of 1 MHz and 4 MHz, respectively. The choices mentioned in [1] are similar to the partitions obtained by our algorithms. Figures 5.2 and 5.4 show the plots of cost vs. iterations to indicate the hill-climbing nature of the group migration algorithm. Figures 5.3 and 5.5 show the same plots for SA algorithm. Note that the algorithm indeed converges to minimum cost partition within a limited number of iterations.

Modules	HW Area (μ meters)	SW Size (Bytes)	HW Delay (n sec)	SW Delay (clock cycles)	Locked
DEBOUNCE	1286208	267	101	218	-
ODO_COUNT_PULSES	2148320	405	152	301	-
ALARM_COMPARE	33872	228	7	216	-
HW_LATCH	20416	-	4	-	H
BELT	-	720	-	501	S
MEASURE_PERIOD	839376	441	60	238	-
ENGINE_CROSS_DISPLAY	-	3139	-	4034	S
SPEED_COUNT_PULSES	450080	211	37	176	-
SPEED_CROSS_DISPLAY	-	4807	-	7392	S
oc_prog_rel	218544	532	23	691	-
Pwm	461216	458	47	853	-
Pwm	461216	458	47	853	-
Frc	586960	260	62	486	-
oc_self	170288	235	17	482	-
Ic	152656	91	15	161	-
Pwmfrc	575360	260	51	486	-
Constraints	1000000	8000	1000	10000	
Weights	0.25	0.25	0.25	0.25	

Table 5.1: Estimates for the modules of dashboard example.

The best cost found, after 1742 iterations of GM algorithm is 0.146883, whereas SA algorithm took 688 iterations to find the best cost.

Partitioned System — Processor clock = 1 MHz

Modules	HW Area	SW Size	HW Delay	SW Delay	Implement
DEBOUNCE	-	267	-	218	SW
DEBOUNCE	-	267	-	218	SW
DEBOUNCE	-	267	-	218	SW
ODO_COUNT_PULSES	-	405	-	301	SW
ALARM_COMPARE	33872	-	7	-	HW
ALARM_COMPARE	33872	-	7	-	HW
HW_LATCH	20416	-	4	-	HW
HW_LATCH	20416	-	4	-	HW
HW_LATCH	20416	-	4	-	HW
BELT	-	720	-	501	SW
oc_prog_rel	218544	-	23	-	HW
MEASURE_PERIOD	839376	-	60	-	HW
ENGINE_CROSS_DISPLAY	-	3139	-	4034	SW
pwm	461216	-	47	-	HW
pwm	461216	-	47	-	HW
ic	152656	-	15	-	HW
oc_prog_rel	218544	-	23	-	HW
SPEED_COUNT_PULSES	-	211	-	176	SW
SPEED_CROSS_DISPLAY	-	4807	-	7392	SW
oc_prog_rel	218544	-	23	-	HW
pwm	461216	-	47	-	HW
pwm	461216	-	47	-	HW
frc	586960	-	62	-	HW
oc_self	170288	-	17	-	HW
ic	152656	-	15	-	HW
pwmfrc	575360	-	51	-	HW
Totals	5106784	10083	503	13058	
Constraints	5000000	8000	1000	10000	
Difference	106784	2083	-497	3058	

Table 5.2: Best-cost partition using GM and SA algorithms (1 MHz Clock).

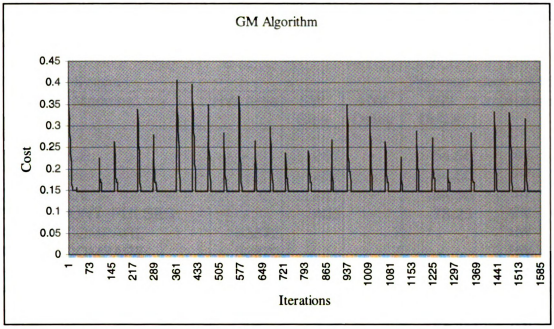

Figure 5.2: Plot of cost vs. iterations for GM algorithm.

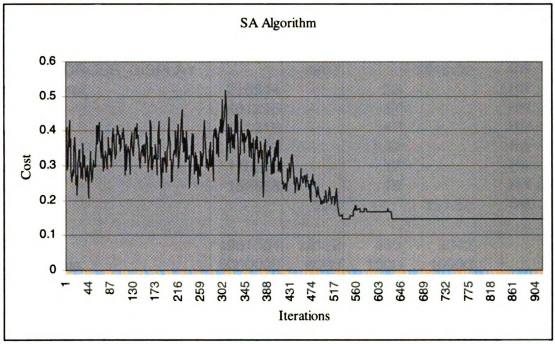

Figure 5.3: Plot of costs vs. iterations for SA algorithm.

The best cost found, after 2028 iterations of GM algorithm is 0.066625, whereas SA algorithm took 688 iterations to find the best cost.

Partitioned System — Processor clock = 4 MHz

Modules	HW Area	SW Size	HW Delay	SW Delay	Implement
DEBOUNCE	-	267	-	54.50	SW
DEBOUNCE	-	267	-	54.50	SW
DEBOUNCE	-	267	-	54.50	SW
ODO_COUNT_PULSES	-	405	-	75.25	SW
ALARM_COMPARE	33872	-	7	-	HW
ALARM_COMPARE	33872	-	7	-	HW
HW_LATCH	20416	-	4	-	HW
HW_LATCH	20416	-	4	-	HW
HW_LATCH	20416	-	4	-	HW
BELT	-	720	-	125.25	SW
oc_prog_rel	218544	-	23	-	HW
MEASURE_PERIOD	839376	-	60	-	HW
ENGINE_CROSS_DISPLAY	-	3139	-	1008.50	SW
pwm	461216	-	47	-	HW
pwm	461216	-	47	-	HW
ic	152656	-	15	-	HW
oc_prog_rel	218544	-	23	-	HW
SPEED_COUNT_PULSES	450080	-	37	-	HW
SPEED_CROSS_DISPLAY	-	4807	-	1848.00	SW
oc_prog_rel	218544	-	23	-	HW
pwm	461216	-	47	-	HW
pwm	461216	-	47	-	HW
frc	586960	-	62	-	HW
oc_self	170288	-	17	-	HW
ic	152656	-	15	-	HW
pwmfrc	-	260	-	121.50	SW
Totals	4981504	10132	489	3342	
Constraints	5000000	8000	1000	10000	
Difference	-18496	2132	-511	-6658	

Table 5.3: Partitioning found using GM and SA algorithms (4 MHz Clock).

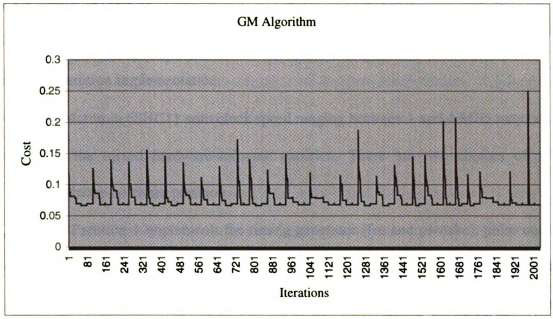

Figure 5.4: Plot of cost vs. iterations for GM algorithm.

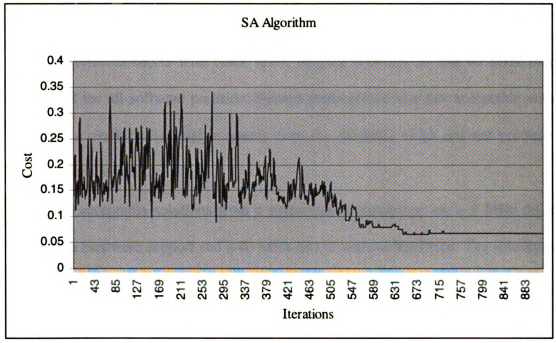

Figure 5.5: Plot of costs vs. iterations for SA algorithm.

5.7 Analysis of Results

Following is a summary of simulation results presented in [1] for possible hardware-software partition implementation:

- A Motorola 68HC11 with clock speed ranging between 1 and 4 MHz may be used for the mixed hardware-software partition. There are two choices of mixed hardware-software partition:
  (a) Partition 1 implements the timing generator (frc and pwmfrc), pulse counters (speed_count_pulses and ic) and PWM (pwm) generators in hardware. System performance becomes degraded at high engine r.p.m. and vehicle speed with 1 MHz clock. However, the system performed well at all speeds and engine r.p.m. values with the 4 MHz clock .
  (b) Partition 2 implements timing generator and pulse counter in hardware. System performance was degraded at low engine r.p.m. and vehicle speed.
- A Motorola 68332 with clock speed ranging between 20 and 40 MHz may be used for all software partition. System performance was not acceptable with 20 MHz clock and at higher speeds even the 40 MHz clock did not produce the desired results.
- The acceptable solutions from a performance standpoint are a 4 MHz 68HC11 with hardware support and a 40 MHz 68332 without hardware. The solution with 68HC11 would most likely be the best choice due to cost reasons.

The results of automatic partitioning shown in Tables 5.2 and 5.3 indicate that the frc, pwmfrc, speed_count_pulses, pwm oc_prog_rel and oc_self are allocated to hardware partition as discussed above.

5.8 Limitations of POLIS

POLIS is still in its development phase. Though it has an elaborate set of tools to perform all steps of co-design in a smooth and straightforward manner, it falls short of many requirements at the same time. Following are some of the problem areas identified during the learning and experimentation phases:

- Presently POLIS supports specifications in Esterél language, which is well suited for specification of control-dominated embedded systems. It is however, cumbersome and at times tedious to describe mathematical relationships and data dependent loops in Esterél. POLIS and Esterél have different semantics for concurrent composition of modules. The concurrent modules execute synchronously in Esterél, implying that computation takes no time and the results of composition are independent of the physical implementation. The execution of concurrent modules is asynchronous in POLIS, depending on the choice of scheduling mechanism.
- POLIS offers some support for user defined data types, limited to software implementation and to pure assignment *i.e.* copying through interfaces in case of hardware implementation.
- Support for cost estimation and synthesis of arithmetic functions for hardware implementation is limited. During the course of experimentation, it was learnt that POLIS does not support implementation of division or trigonometric functions in hardware. These functions are only implementable in software at present. It was not possible to determine the hardware-cost estimates for those objects that used either of these functions. The developers at UC Berkeley were consulted and we

learned that presently it is not possible to generate the cost estimates for these functions due to non-availability of requisite libraries. The best way to deal with this problem would be to either use PTOLEMY to perform co-simulation to decide the partitioning or write VHDL code for these functions to perform hardware synthesis to determine the estimates. But still hardware synthesis of CFSMs containing these functions would not be possible in POLIS.

CHAPTER 6

CONCLUSION AND FUTURE DIRECTIONS

6.1 Summary

With the increasing role of computers in our daily life and availability of enabling technology, a paradigm shift, from general-purpose computer design to embedded system design is occurring. Co-design of an embedded system involves many challenging tasks. This work has focused on one of them known as hardware-software partitioning. The partitioning problem in hardware-software co-design involves various issues. The choice of a hardware-software partition greatly influences the cost and performance of the final product. It is therefore imperative that a co-design framework must be capable of exploring the design space with a view to finding the best system partitioning.

POLIS provides an elaborate set of design tools integrated in its design flow to assist a user from specification to implementation of an embedded system. The partitioning process is however, manual and requires detailed analysis of the simulation results for partitioning. An attempt was made to explore the possibility of automating this process by application of automatic partitioning algorithms.

The current-state-of-the-art in hardware-software co-design has two distinct flavors: one favoring early or manual partitioning, and the other involving late or automatic partitioning. POLIS does not fit exactly in either of the above; the partitioning process is manual but it takes place later in the design process. Based on the granularity of CFSMs and the level of abstraction, Group Migration and Simulated Annealing algorithms were selected for automatic partitioning in POLIS. The estimates generated by POLIS were

assumed to provide a fair idea of the metrics. The algorithms were successfully applied for automatic partitioning and found to correspond to the published results. However, due to the following problems or limitations of POLIS, the automatic partitioning algorithms could not be fully integrated in its existing design flow:

- Lack of support for implementation of certain mathematical functions in hardware. All functions distributed amongst various CFSMs have to be converted to C code/BLIF description for software/hardware implementation. A set of library files is provided in POLIS to translate the SHIFT code to C or BLIF code for this purpose. The set of hardware-libraries is, however, incomplete and does not allow implementation of division and trigonometric functions in hardware. Due to this limitation hardware estimates for such mathematical functions can not be generated in the present POLIS version.
- Generation of hardware estimates involves mapping the BLIF files onto a target technology using either a technology library for ASICs or generating the XNF files and porting them to Xilinx software for mapping onto FPGAs. The BLIF to XNF conversion is also under development and does not provide full capability to map any design onto any FPGA. The present POLIS version supports mapping of design onto X3000 family.
- The auxiliary file describes the interconnections and implementation choices of CFSMs. This file is normally created with the help of schematics described in PTOLEMY. There is no provision of automatic incorporation of partitioning data into this file. The results of partitioning have to be manually entered as implementation choices for CFSMs.

6.2 Validity of Results

The results obtained were compared with the published data on POLIS. They were found to generally agree with the recommended partition based on manual partitioning as mentioned at section 5.7 above. It may however be pointed out that the generation of estimates should not be a lengthy and costly process because the idea of automatic partitioning can only be fruitful if all steps prior to partitioning are also automatic. The present arrangement of generating the estimates in POLIS or Xilinx software involving porting them from one to another software tool on a different platform defeats the purpose of speeding up the partitioning process. It is also desirable to have a database of certain standard modules/CFSMs with their software and hardware estimates already calculated for use in the partitioning process.

6.3 Recommendations and Future Directions

Based on the study carried out during this work following recommendations are made:

- Improvements in estimate generation may be made either by employing better cost-estimation algorithms or generation of a database for various standardized modules.
- An attempt may be made to explore the possibility of using Binary Decision Diagram (BDD)/CDFG for partitioning. The s-graph nodes may be treated as the nodes of a CDFG and as in LYCOS and COSYMA these nodes may be moved across hardware and software boundaries. The CFSM model of POLIS may require some modifications for this purpose.

6.3.1 Integrating the Partitioning Algorithm in Polis Design Flow

In order to fully integrate the automatic partitioning algorithms in POLIS the following are recommended:

- The estimates generated by POLIS and constraints supplied by the user should be automatically appended to a file readable by the partitioning programs.
- The design flow in POLIS should be modified such that the synthesis tools could read the results of partitioning. The hardware and software synthesis could then become automated.

6.3.2 Rapid Prototyping Platform

The validity of partitioning results can not be confirmed until the hardware and software modules are implemented as a prototype. The POLIS design group at Berkeley uses a rapid prototyping platform based on APTIX architecture. At MSU it would be advantageous to establish a facility based on Xilinx Foundation Express software that supports loading of BIT files onto the Xilinx chips mounted on an emulator board. The synthesized C code can also be compiled to generate the binary code that can be downloaded onto the PROM of the emulator. This would provide a means of testing the functionality of system hardware and software and feedback to the partitioning algorithm in terms of improved metric and constraints.

APPENDICES

APPENDIX A

GROUP MIGRATION (KERNIGHAN-LIN) ALGORITHM IMPLEMENTED IN C++

// Group Migration Algorithm
//
// Global variables
// infi is a high value used for initializing some of the variables
// np is the number of metrics to be considered for partitioning
// nn is the number of Modules or Objects in the system
// kk[] is an array of weights used in the cost function.
// float, area, hdelay and sdelay are the constraints on four metrics
// proc_clk is the processor clock frequency in MHz (Default value = 1 MHz)
// Objects are the modules/CFSMs in the system. The objects are defined as data structure containing name, metric values (hw_area, sw_size, hw_delay, sw_delay), moved and locked flags, and partition assignment information (pp)
//
// Functions
// Cost_fct() is the implementation of cost function discussed at paragraph 5.4. It accepts a partition and calculates its cost using individual object's metrics and constraints corresponding to partition assignment and returns the cost as a floating-point value. It calls Func() to calculate the normalized difference depicted at paragraph 5.4
// Move() accepts the index of object to be moved and the current partition information. The object is moved from one domain to other depending on its present assignment
// Gen_part() is used to generate random partitions. A partition is sent as an argument that is randomly changed by this function, the random() function uses present time as the seed for random number generation.
// Pr_part() prints the contents of partition array passed as an argument.
// Part_file() writes the contents of partition array to the fstream passed as an argument
// Final_part writes the final partition information to the fstream passed as an argument
//
// Local variables in main()
// ii, n, y and z are counters used in variuos loops
// bestmove_obj is the index number of object that when moved creates the lowest cost partition
// Partitions (P[nn], bestcost_P[nn], PP[nn], prev_P[nn], best_P[nn], P_in[nn]) are arrays containing strings of 'H', 'S' and 'X' indicating allocation of objects to HW, SW and HW-SW in case of a mixed implementation respectively.
// inputfile stores the name of input file to be read by the program
// clk is the processor clock frequency entered by the user.
// cost is the value of Cost_fct calculated for each trial partition
// prev_cost is the cost found in the previous iteration of the algorithm
// bestpart_cost is the cost of best partition
// bestmove_cost is the lowest cost after each trial partition

```
//  best_cost  is the best cost, determined after all of the iterations
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

#define infi 999.
#define np 4
#define nn 26
float kk[np], area, size, hdelay, sdelay, proc_clk=1.0;

struct Object
{
    public:
        char name[20];
        float hw_area, sw_size, hw_delay, sw_delay;
        bool moved, locked;;
        char pp;
};

Object objt[nn];

float Cost_fct(char []);
float Func(float, float);
void Move(int, char []);
void Gen_part(char []);
void Pr_part(char []);
void Part_file(char [], fstream);
void Final_part(char Pt[], fstream);

void main()
{
    int ii, n, bestmove_obj, y, z;
    char P[nn], bestcost_P[nn], PP[nn], prev_P[nn], best_P[nn], P_in[nn];
    char inputfile[15];
    float clk, cost, prev_cost, bestpart_cost, bestmove_cost, best_cost=infi;

    cout << "Enter the name of input file to use: ";
    cin >> inputfile;

    fstream infile(inputfile, ios::in);
    fstream outfile("output.dat", ios::out);
    fstream outfile1("out1.dat", ios::out);
```

```
    if(outfile.fail()|| outfile1.fail()) {
        cout << "Could not open output file" <<endl;
    }

// Read file code
    if(!infile.fail()) {
        for(int counter=0; counter<nn; counter++) {
            infile >> objt[counter].name
                >> objt[counter].hw_area
                >> objt[counter].sw_size
                >> objt[counter].hw_delay
                >> objt[counter].sw_delay
                >> objt[counter].pp;
            if ((objt[counter].pp=='H')||(objt[counter].pp=='S')) {
                objt[counter].locked=true;
            }
            else if (objt[counter].pp=='X') {
                objt[counter].locked=true;
            }
            else {objt[counter].locked=false;}

        }
        infile >> area >> size >> hdelay >> sdelay;
        for (counter=0; counter<np; counter++) { infile >> kk[counter]; }
    }else
        cout << "Could not open input file" <<endl;

//  Write input data to output file
    outfile << "The following data is read" << endl;
    outfile << "----------------------------------------------------------------" << endl;
    outfile << " Module                    HW Area   SW Size    HW Delay  SW Delay
Locked" << endl;
    outfile << "----------------------------------------------------------------" << endl;
    for (ii=0; ii<nn; ii++) {
        outfile << setiosflags(ios::left) << setw(20) << objt[ii].name <<
resetiosflags(ios::left);
        outfile << setw(10) << objt[ii].hw_area;
        outfile << setw(10) << objt[ii].sw_size;
        outfile << setw(10) << objt[ii].hw_delay;
        outfile << setw(10) << objt[ii].sw_delay;
        outfile << setw(10) << objt[ii].pp << endl;
    }
    outfile << "----------------------------------------------------------------" << endl;
    outfile << endl << "Constraints            " << setw(10) << area << setw(10) << size
<< setw(10)
            << setw(10) << hdelay << setw(10) << sdelay << endl;
```

```
    outfile << endl << "Weights         ";
    for (ii=0; ii<np; ii++) { outfile << kk[ii] << ", "; }
    outfile << endl;

    cout << "Enter Processor Clock in MHz, Press '0' for default: ";
    cin >> clk;
    if(clk>0) {
        proc_clk=clk;
    }
//  Seed the random number
    srand(time(0));

    outfile1 << "Constraints " << area << " " << size
             << " " << hdelay << " " << sdelay << endl;

    outfile1 << "Weights ";
    for (ii=0; ii<np; ii++) { outfile1 << kk[ii] << " "; }
    outfile1 << endl;

//  Create initial partition
    for (ii=0; ii<nn; ii++) {
        if (objt[ii].pp=='H') {
            P_in[ii]='H';
        }
        else if (objt[ii].pp=='X') {
            P_in[ii]='X';
        }
        else P_in[ii]='S';

    }

    y=0, n=0;
    while (y < nn)
    {
        Gen_part(P_in);

//      Initial Partiton
        for (ii=0; ii<nn; ii++) {
            P[ii]=P_in[ii];
        }

        z=0;
        do
        {
            prev_cost=Cost_fct(P);
            bestpart_cost=infi;
```

```
            for (ii=0; ii<nn; ii++) {
                if ( !objt[ii].locked) {
                    objt[ii].moved=false;
                }
                prev_P[ii]=P[ii];
            }
            for (int j=0; j<nn; j++) {
                bestmove_cost=infi;
                for (int k=0; k<nn; k++) {
                    if (!objt[k].moved && !objt[k].locked) {
                        for (ii=0; ii<nn; ii++) {
                            PP[ii]=P[ii];
                        }
                        Move(k, PP);
                        cost=Cost_fct(PP);
                        if (cost < bestmove_cost) {
                            bestmove_cost=cost;
                            bestmove_obj=k;
                        }
                    }
                }
// Move the bestmove_object to make new partiton
                Move(bestmove_obj, P);
                objt[bestmove_obj].moved=true;
// Save the best partition during the sequence
                if (bestmove_cost < bestpart_cost) {
                    for (ii=0; ii<nn; ii++) {
                        bestcost_P[ii]=P[ii];
                    }
                    bestpart_cost=bestmove_cost;
                }
                n++;
                outfile1 << bestpart_cost << endl;    // Output the best cost in each outer
iteration

            }
// Update P if a better cost was found, else exit
            if (bestpart_cost < prev_cost) {
                for (ii=0; ii<nn; ii++) {
                    P[ii]=bestcost_P[ii];
                }
            }
            else {
                for (ii=0; ii<nn; ii++) {
                    P[ii]=prev_P[ii];
                }
```

```
            }
            z++;

        } while ( bestpart_cost < prev_cost );
        if ( Cost_fct(P) < best_cost ) {
            best_cost = Cost_fct(P);
            for (ii=0; ii<nn; ii++) {
                best_P[ii]=P[ii];
            }
        }
        y ++;
    }

    outfile << "\n\nThe best cost found after " << n << " iterations is "
        << (Cost_fct(best_P)) << endl << "The best cost partition is : {";
        Part_file(best_P, outfile);
        Final_part(best_P, outfile);
}


float Cost_fct(char PX[])
{
    float sz=0, ha=0, sd=0, hd=0;
    float est_cost, x;
    for (int jj=0; jj<nn; jj++) {
        switch (PX[jj]) {
            case 'S': {
                sz += objt[jj].sw_size;
                sd += objt[jj].sw_delay;
                break;
            }
            case 'H': {
                ha += objt[jj].hw_area;
                hd += objt[jj].hw_delay;
            break;
            }
            case 'X': {
                sz += objt[jj].sw_size;
                sd += objt[jj].sw_delay;
                ha += objt[jj].hw_area;
                hd += objt[jj].hw_delay;
                break;
            }
        cout << "No valid case, Press anykey to continue" << endl;
        cin >> x;
        break;
```

```
        }
    }
    sd=sd/proc_clk;
    est_cost = kk[0]*Func(sz, size) + kk[1]*Func(sd, sdelay) + kk[2]*Func(ha, area) +
kk[3]*Func(hd, hdelay);
    return est_cost;
}

float Func(float val, float constr)  // Estimate the normalized cost of a parameter for an
Object
{
    float temp;
    temp=(val-constr)/constr;
    if (temp <= 0) {temp=0;}
    return temp;
}

void Move(int ob, char Pt[])
{
    switch (Pt[ob]) {
        case 'S': {
            Pt[ob]='H';
            objt[ob].pp='H';
            cout << "Object moved to HW" << endl;
            break;
        }
        case 'H': {
            Pt[ob]='S';
            objt[ob].pp='S';
            cout << "Object moved to SW" << endl;
            break;
        }
        case 'X': {
            cout << "Object cannot be moved" << endl;
            break;
        }
        cout << "Un-identified partition code " << Pt[ob] << endl;
        break;
    }
}

void Pr_part(char Pt[])
{
    for (int pj=0; pj<nn; pj++) {
        cout << Pt[pj];
    }
```

```
    cout << "}" << endl;
}

void Part_file(char Pt[], fstream outf)
{
    for (int pj=0; pj<nn; pj++) {
        outf << Pt[pj];
    }
    outf << "}" << endl;
}

void Gen_part(char Pr[])
{
    int k, m;
    for (m=0; m<nn; m ++) {
        if (!objt[m].locked) {
            k=rand() % 1000;
            if (k < 500) { Pr[m]='S';}
            else { Pr[m]='H';}
        }
    }
    cout << "Generated Partition : {";
    Pr_part(Pr);

}

void Final_part(char Pt[], fstream outf2)
{
    float ha, hd, sz, sd;
    outf2 << endl << "\nPartitioned System, Processor clock = " << proc_clk << " MHz"
<< endl;
    outf2 << "-----------------------------------------------------------------" << endl;
    outf2 << " Module                    HW Area   SW Size   HW Delay  SW Delay
Implement" << endl;
    outf2 << "                  (micro m sq) (Bytes)    (ns)  (micro sec)   in" << endl;
    outf2 << "-----------------------------------------------------------------" << endl;
    for (int ii=0; ii<nn; ii++) {
        outf2   <<   setiosflags(ios::left)   <<   setw(20)   <<   objt[ii].name   <<
resetiosflags(ios::left);

        switch (Pt[ii]) {
        case 'H': {
            outf2 << setw(10) << objt[ii].hw_area << setw(10)
                  << "   - " << setw(10) << objt[ii].hw_delay
                  << setw(10) << "   - " << setw(10) << "HW" << endl;
            ha += objt[ii].hw_area;
```

```
            hd += objt[ii].hw_delay;
            break;
            }
        case 'S': {
            outf2 << setw(10) << "    - " << setw(10) << objt[ii].sw_size << setw(10)
                << "    - " << setw(10) << (objt[ii].sw_delay/proc_clk) <<  setw(10) <<
"SW" << endl;
            sz += objt[ii].sw_size;
            sd += objt[ii].sw_delay;
            break;
            }
        cout << "Un-identified partiton code " << Pt[ii] << endl;
        break;
        }
    }
    outf2 << "-----------------------------------------------------------" << endl;
    outf2 << "Totals                " << setw(10) << ha << setw(10) << sz << setw(10) <<
hd << setw(11)
        << (sd/proc_clk) << endl;
    outf2 << "Constraints           " << setw(10) << area << setw(10) << size << setw(10)
        << hdelay << setw(10) << sdelay << endl;
    outf2 << "-----------------------------------------------------------" << endl;
    outf2 << "Difference            " << setw(10) << ha-area << setw(10) << sz-size <<
setw(10)
        << hd-hdelay << setw(10) << (sd/proc_clk)-sdelay << endl;

}
```

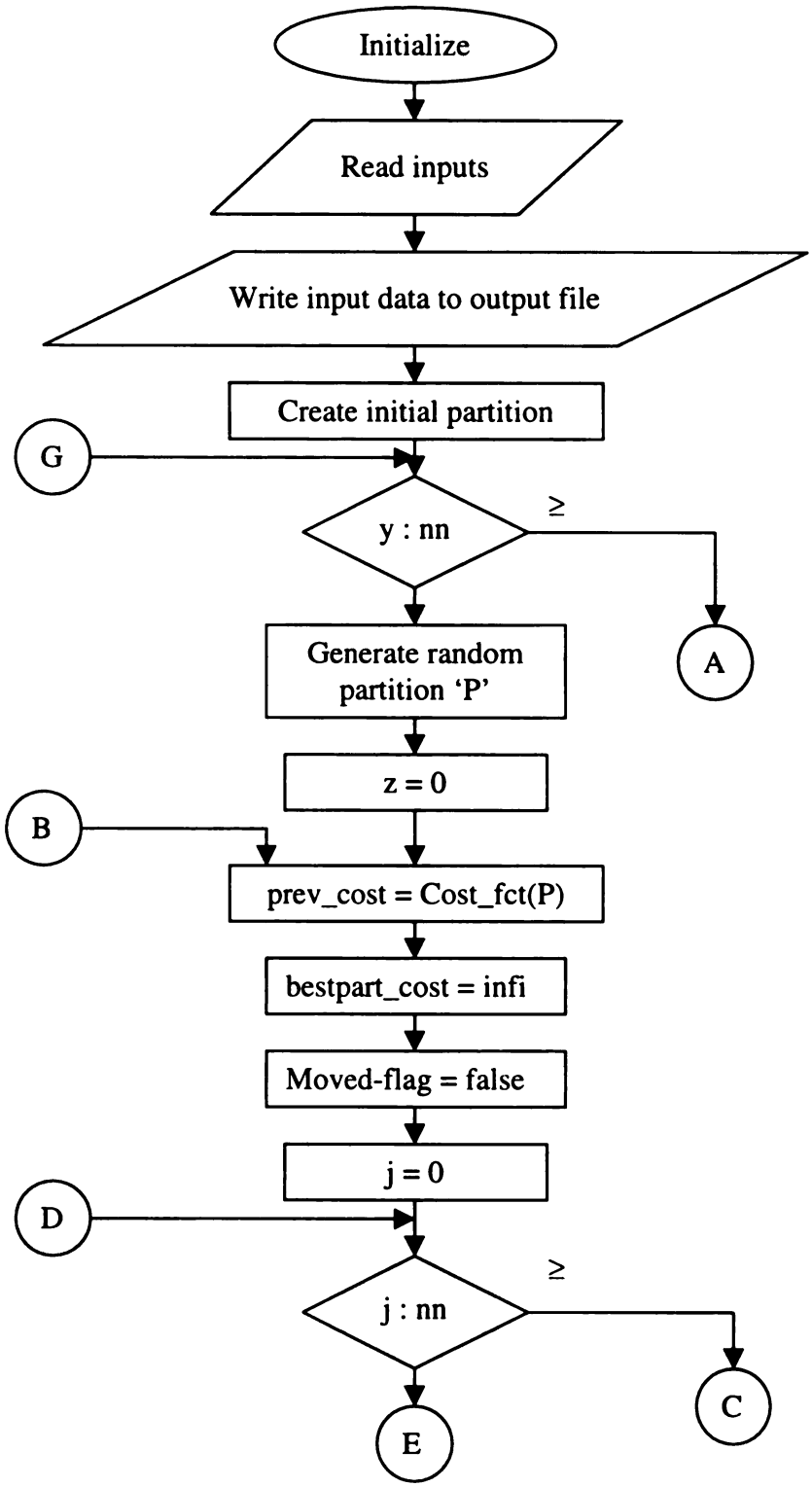

Figure A.1: Flow chart of GM algorithm program.

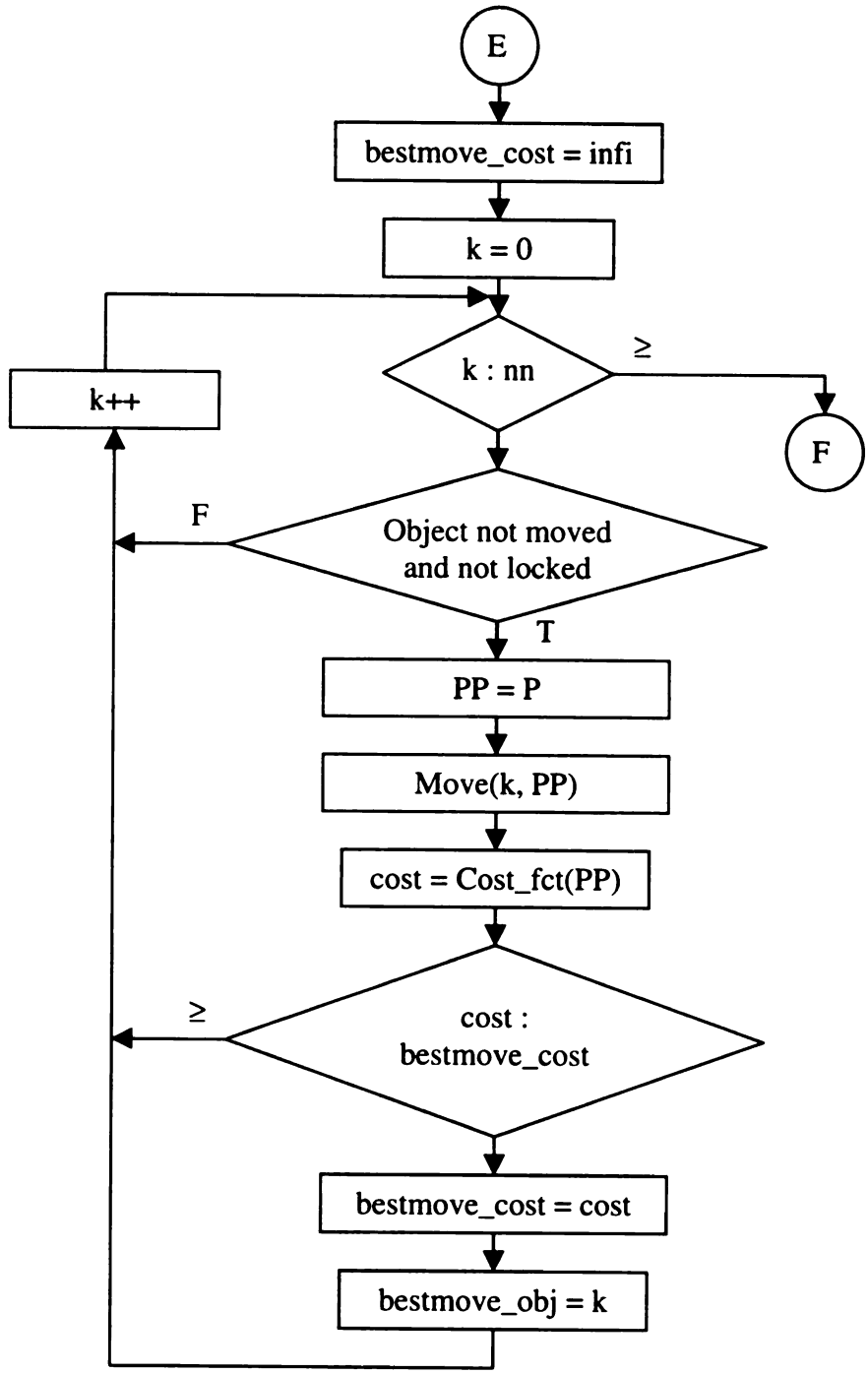

Figure A.1: (Continued).

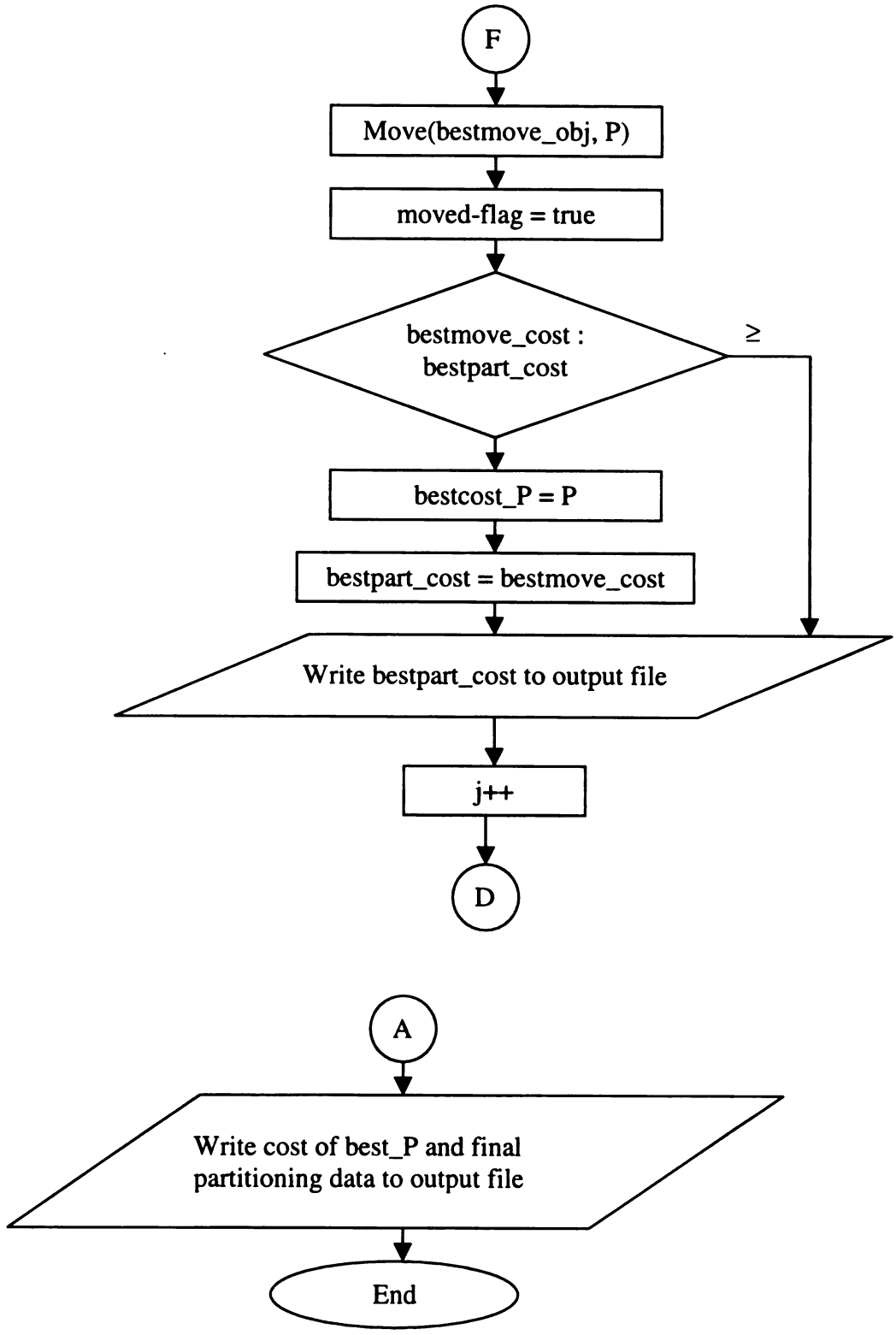

Figure A.1: (Continued).

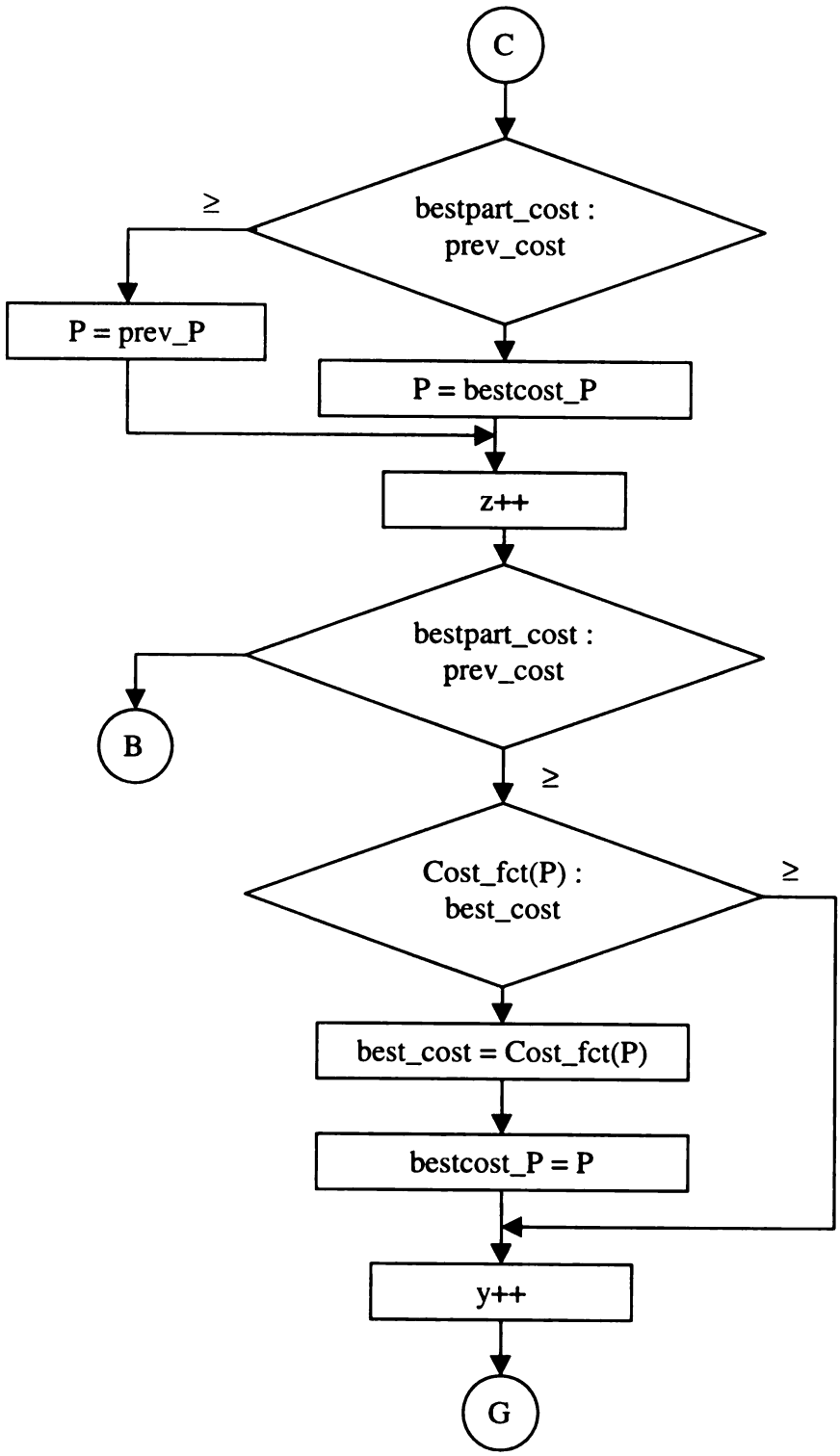

Figure A.1: (Continued).

APPENDIX B

SIMULATED ANNEALING ALGORITHM IMPLEMENTED IN C++

// Simulated Annealing Algorithm
// Objt are the modules/CFSMs in the system
// Partition is an array of objects allocated to SW/HW
//
// Global variables
// init is a value used for initializing the temperature variable: temp
// np is the number of metrics to be considered for partitioning
// nn is the number of modules or objects in the system
// alpha is the multiplier used for lowering the temperature
// freez is the lowest value of temperature at which to stop the algorithm
// n_eq is the maximum value of counter n that indicates the state of equilibrium
// kk[] is an array of weights used in the cost function
// temp stores the value of current temperature
// area, size, hdelay and sdelay are the constraints on four metrics
// proc_clk is the processor clock frequency in MHz (Default value = 1 MHz)
// Objects are the modules/CFSMs in the system. The objects are defined as data structure containing name, metric values (hw_area, sw_size, hw_delay, sw_delay), moved and locked flags, and partition assignment information (pp)
//
// Functions
// Cost_fct() is the implementation of cost function discussed at paragraph 5.4. It accepts a partition and calculates its cost using individual object's metrics and constraints corresponding to partition assignment and returns the cost as a floating-point value. It calls Func() to calculate the normalized difference depicted at paragraph 5.4
// Random_move() accepts a partition array as an argument and moves an object randomly from one domain to other only if that object is not locked
// Gen_part() is used to generate random partitions. A partition is received as an argument that is randomly changed by this function, It makes use of the random() function that uses present time as the seed for random number generation.
// Pr_part() prints the contents of partition array passed as an argument.
// Part_file() writes the contents of partition array to the fstream passed as an argument
// Accept() receives d_cost and temp1 values as arguments. It calculates the exponent of the ratio of -d_cost and temp values and compares it with 1. If the exponent is less then 1 then it returns the exponent otherwise it returns 1
// Final_part writes the final partition information to the fstream passed as an argument
//
// Local variables in main()
// ii, n, y and z are counters used in various loops
// Partitions (P[nn], P_ten[nn], best_P[nn], P_in[nn]) are arrays containing strings of 'H', 'S' and 'X' indicating allocation of objects to HW, SW and HW-SW in case of a mixed implementation respectively.

```
//  clk is the processor clock frequency (in MHz) entered by the user.
//  cost is the value of Cost_fct calculated for initial partition
//  cost_ten is the cost of tentative partition
//  best_cost is the best cost found after reaching equilibrium
//  delta_cost is the difference of costs between initial partition and tentative partition
//  random is a random number in the range [0,1] generated using rand() function

#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

#define init_temp 1.0
#define np 4
#define nn 26
#define alpha 0.99
#define freez 0.001
#define n_eq 3
float kk[np], temp, area, size, hdelay, sdelay, proc_clk=1.0;

struct Object
{
    public:
        char name[20];
        float hw_area, sw_size, hw_delay, sw_delay;
        bool locked;
        char pp;
};

Object objt[nn];

float Cost_fct(char []);
float Func(float, float);
void Random_move(char []);
void Gen_part(char Pr[]);
void Pr_part(char []);
void Part_file(char [], fstream);
float Accept(float, float);
void Final_part(char Pt[], fstream);

void main()
{
    int ii, z, y, n;
    char P[nn], P_ten[nn], best_P[nn], P_in[nn];
```

```
    float clk, cost, cost_ten, best_cost, delta_cost, random;

    fstream infile("input00.dat", ios::in);      // input file
    fstream outfile("output.dat", ios::out);     // output file
    fstream outfile1("out1.dat", ios::out);      // output file for graphs

    if(outfile.fail()|| outfile1.fail()) {
        cout << "Could not open output file" <<endl;
    }

// Read input file
    if(!infile.fail()) {
        for(int counter=0; counter<nn; counter++) {
            infile >> objt[counter].name
                >> objt[counter].hw_area
                >> objt[counter].sw_size
                >> objt[counter].hw_delay
                >> objt[counter].sw_delay
                >> objt[counter].pp;
            if ((objt[counter].pp=='H')||(objt[counter].pp=='S')) {
                objt[counter].locked=true;
            }
            else if (objt[counter].pp=='X') {
                objt[counter].locked=true;
            }
            else {objt[counter].locked=false;}
        }
        infile >> area >> size >> hdelay >> sdelay;
        for (counter=0; counter<np; counter++) { infile >> kk[counter]; }
    }else
        cout << "Could not open input file" <<endl;

//  Write input data to output file
    outfile << "The following data is read" << endl;
    outfile << "------------------------------------------------------------------------" << endl;
    outfile << "  Module              HW Area   SW Size   HW Delay  SW Delay
Locked" << endl;
    outfile << "                      (mm sq)    (Bytes)    (ns)  (Clk Cycles)   in" <<
endl;
    outfile << "------------------------------------------------------------------------" << endl;
    for (ii=0; ii<nn; ii++) {
        outfile << setiosflags(ios::left) << setw(20) << objt[ii].name <<
resetiosflags(ios::left);
        outfile << setw(10) << objt[ii].hw_area;
        outfile << setw(10) << objt[ii].sw_size;
        outfile << setw(10) << objt[ii].hw_delay;
```

```
        outfile << setw(10) << objt[ii].sw_delay;
        outfile << setw(8) << objt[ii].pp << endl;
    }
    outfile << "--------------------------------------------------------------------------" << endl;
    outfile << endl << "Constraints    " << setw(10) << area << setw(10) << size <<
setw(10)
            << setw(10) << hdelay << setw(10) << sdelay << endl;
    outfile << endl << "Weights ";
    for (ii=0; ii<np; ii++) { outfile << kk[ii] << ", "; }
    outfile << endl;

    cout << "Enter Processor Clock in MHz, Press '0' for default: ";
    cin >> clk;
    if(clk>0) {
        proc_clk=clk;
    }
//  Seed the random number
    srand(time(0));

// write input data to output file for graphs
    outfile1 << "Modules = " << nn
            << ", Constraints " << area << ", " << size
            << ", " << hdelay << ", " << sdelay << ", Weights ";
    for (ii=0; ii<np; ii++) { outfile1 << kk[ii] << ", "; }
    outfile1 << endl << "Initial Temp = " << init_temp << ", n equilibrium = "
            << n_eq << ", freez = " << freez << ", alpha = " << alpha << endl;

    // Generate Initial Partition
    for (ii=0; ii<nn; ii++) {
        if (objt[ii].pp=='H') {
            P_in[ii]='H';
        }
        else if (objt[ii].pp=='X') {
            P_in[ii]='X';
        }
        else P_in[ii]='S';
    }
    Gen_part(P_in);

    outfile << endl << "Partition generated : {";
    Part_file(P_in, outfile);

// Initial Partiton
    for (ii=0; ii<nn; ii++) {
        P[ii]=P_in[ii];
        best_P[ii]=P[ii];
```

```
}

temp=init_temp;  // Initialize Temp
cost=Cost_fct(P);
best_cost=cost;
y=0;
while (temp > freez)        // Loop while temperature is not frozen
{

    n=1;
    z=0;
    while(n < n_eq)    // loop while equilibrium is not reached
    {

        for (ii=0; ii<nn; ii++) {
            P_ten[ii]=P[ii];              // Tentative Partition
        }
        Random_move(P_ten);          // Randomly move an object in P_ten
        cost_ten=Cost_fct(P_ten);
        delta_cost=cost_ten-cost;
        random=float(rand () % 100000)/(100000);
        if (Accept(delta_cost, temp) > random) {
            for (ii=0; ii<nn; ii++) {
                P[ii]=P_ten[ii];
            }
            if (cost_ten > cost) { n++; }
            cost=cost_ten;
        }
        else {
            n++;
        }
        z++;
    }
    temp=alpha*temp;
    outfile1 << cost << endl;  // Output cost after reaching equilibrium
    if(cost < best_cost) {
        best_cost=cost;
        for (ii=0; ii<nn; ii++) {
                best_P[ii]=P[ii];
        }
    }
y++;
}
outfile << "\nTook " << y << " iterations." << endl
        << "Best cost= " << best_cost << " for Partition :{";
Part_file(best_P, outfile);
```

```
    Final_part(best_P, outfile);

}


float Cost_fct(char PX[])
{
    float sz=0, ha=0, sd=0, hd=0;
    float est_cost, x;
    for (int jj=0; jj<nn; jj++) {
        switch (PX[jj]) {
            case 'S': {
                sz += objt[jj].sw_size;
                sd += objt[jj].sw_delay;
                break;
            }
            case 'H': {
                ha += objt[jj].hw_area;
                hd += objt[jj].hw_delay;
                break;
            }
            case 'X': {
                sz += objt[jj].sw_size;
                sd += objt[jj].sw_delay;
                ha += objt[jj].hw_area;
                hd += objt[jj].hw_delay;
                break;
            }

        cout << "No valid case, Press any key to continue" << endl;
        cin >> x;
        break;
        }
    }
    sd=sd/proc_clk;
    est_cost = kk[0]*Func(sz, size) + kk[1]*Func(sd, sdelay) + kk[2]*Func(ha, area) +
kk[3]*Func(hd, hdelay);
    return est_cost;
}

float Func(float val, float constr)  // Estimate the normalized cost of a parameter for an
Object
{
    float t_val;
    t_val=(val-constr)/constr;
    if (t_val <= 0) {t_val=0;}
```

```
    return t_val;
}

void Pr_part(char Pt[])
{
    for (int pj=0; pj<nn; pj++) {
        cout << Pt[pj];
    }
    cout << "}" << endl;
}

void Part_file(char Pt[], fstream outf)
{
    for (int pj=0; pj<nn; pj++) {
        outf << Pt[pj];
    }
    outf << "}" << endl;
}

float Accept(float d_cost, float temp1)
{
    float exp1;
    exp1=exp(-d_cost/temp1);
    if (exp1 < 1) { return exp1; }
    else { return 1; }
}

void Gen_part(char Pr[])
{
    int k, m;
    for (m=0; m<nn; m ++) {
        if (!objt[m].locked) {
            k=rand() % 1000;
            if (k < 500) { Pr[m]='S';}
            else { Pr[m]='H';}
        }
    }
    cout << "Generated Partition : {";
    Pr_part(Pr);

}

void Random_move(char Pr[])
{
    int k, j=0;
```

```
    while (j<1) {
        k=rand() % nn;
        if(!objt[k].locked) {
            switch (Pr[k]) {
                case 'S': {
                    Pr[k]='H';
                    cout << "Object moved to HW" << endl;
                    break;
                }
                case 'H': {
                    Pr[k]='S';
                    cout << "Object moved to SW" << endl;
                    break;
                }
                case 'X': {
                    break;
                }
                cout << "Un-identified partiton code " << Pr[k] << endl;
                break;
            }
        j++;
        }
        else { j=0; }
    }
}

void Final_part(char Pt[], fstream outf2)
{
    float ha=0, hd=0, sz=0, sd=0;
    outf2 << endl << "Partitioned System, Processor clock= " << Proc_clk << " MHz" <<
endl;
    outf2 << "-------------------------------------------------------------------------------" <<
endl;
    outf2 << "  Module                  HW Area     SW Size   HW Delay  SW Delay
Implement" << endl;
    outf2 << "                        (micro m sq)  (Bytes)    (ns)  (micro sec)   in" <<
endl;
    outf2 << "-------------------------------------------------------------------------------" <<
endl;
    for (int ii=0; ii<nn; ii++) {
        outf2 << setiosflags(ios::left) << setw(20) << objt[ii].name <<
resetiosflags(ios::left);
        switch (Pt[ii]) {
        case 'H': {
            outf2 << setw(10) << objt[ii].hw_area << setw(10)
                  << "   - " << setw(10) << objt[ii].hw_delay
```

```
                << setw(10) << "    -  " << setw(10) << "HW" << endl;
            ha += objt[ii].hw_area;
            hd += objt[ii].hw_delay;
            break;
            }
        case 'S': {
            outf2 << setw(10) << "    -  " << setw(10) << objt[ii].sw_size << setw(10)
                << "    -  " << setw(10);
            outf2 << (objt[ii].sw_delay/Proc_clk) << setw(10) << "SW" << endl;
            sz += objt[ii].sw_size;
            sd += objt[ii].sw_delay;
            break;
        }
        case 'X': {
            outf2 << setw(10) << objt[ii].hw_area << setw(10)
                << objt[ii].sw_size << setw(10) << objt[ii].hw_delay
                << setw(10) << (objt[ii].sw_delay/Proc_clk) << setw(10) << "H-S" <<
endl;
            ha += objt[ii].hw_area;
            hd += objt[ii].hw_delay;
            sz += objt[ii].sw_size;
            sd += objt[ii].sw_delay;
            break;
        }

        cout << "Un-identified partiton code " << Pt[ii] << endl;
        break;
        }
    }
    outf2 << "-------------------------------------------------------------------------" <<
endl;
    outf2 << "Totals          " << setw(10) << ha << setw(10) << sz << setw(10) << hd;
    outf2 << setw(10) << (sd/Proc_clk) << endl;
    outf2 << "Constraints     " << setw(10) << area << setw(10) << size;
    outf2 << setw(10) << hdelay << setw(10) << sdelay << endl;
    outf2 << "-------------------------------------------------------------------------" <<
endl;
    outf2 << "Difference      " << setw(10) << ha-area << setw(10) << sz-size <<
setw(10)
        << hd-hdelay << setw(10);
    outf2 << (sd/Proc_clk)-sdelay << endl;

}
```

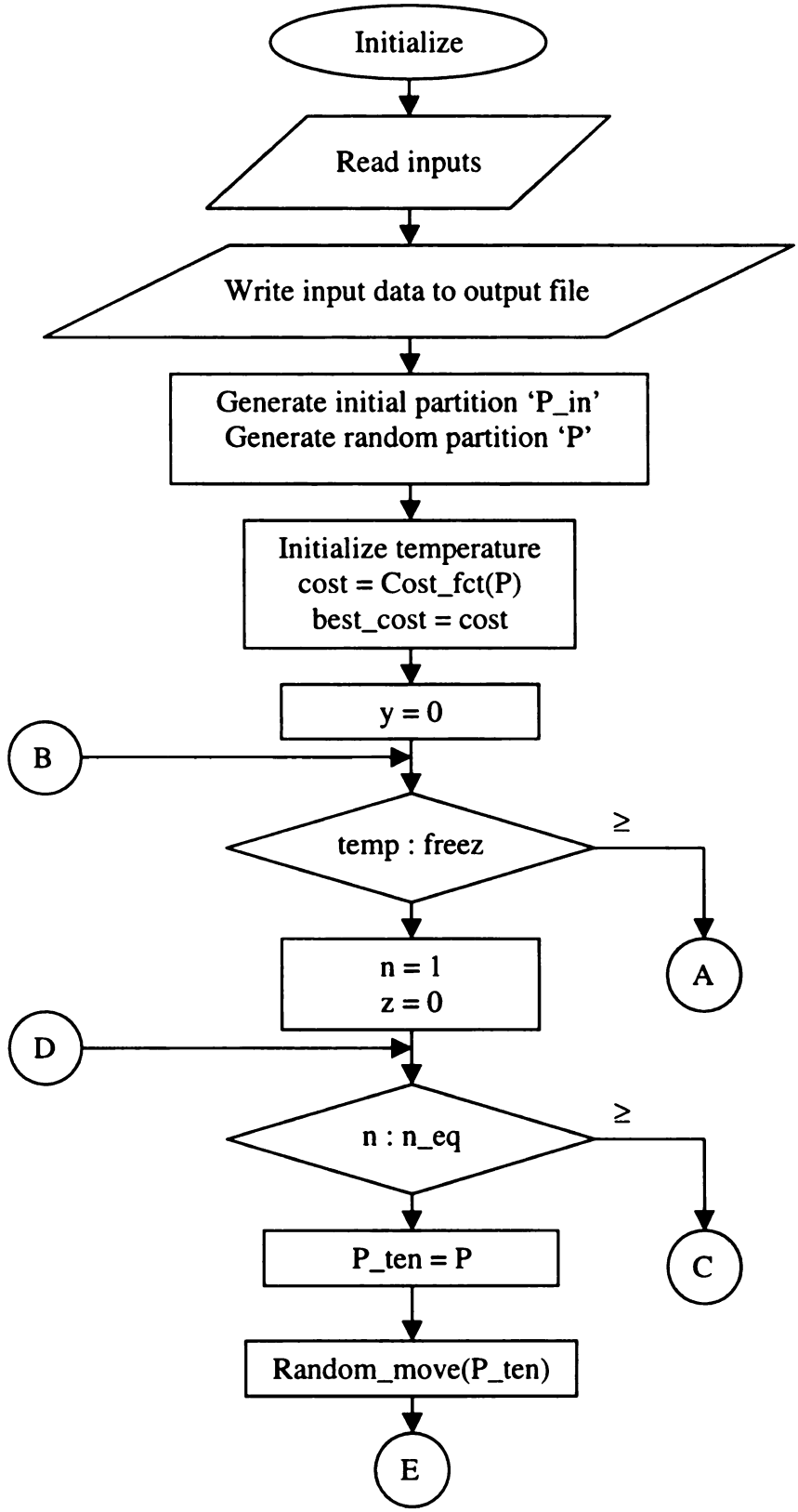

Figure 5.2: Flow chart of SA algorithm program.

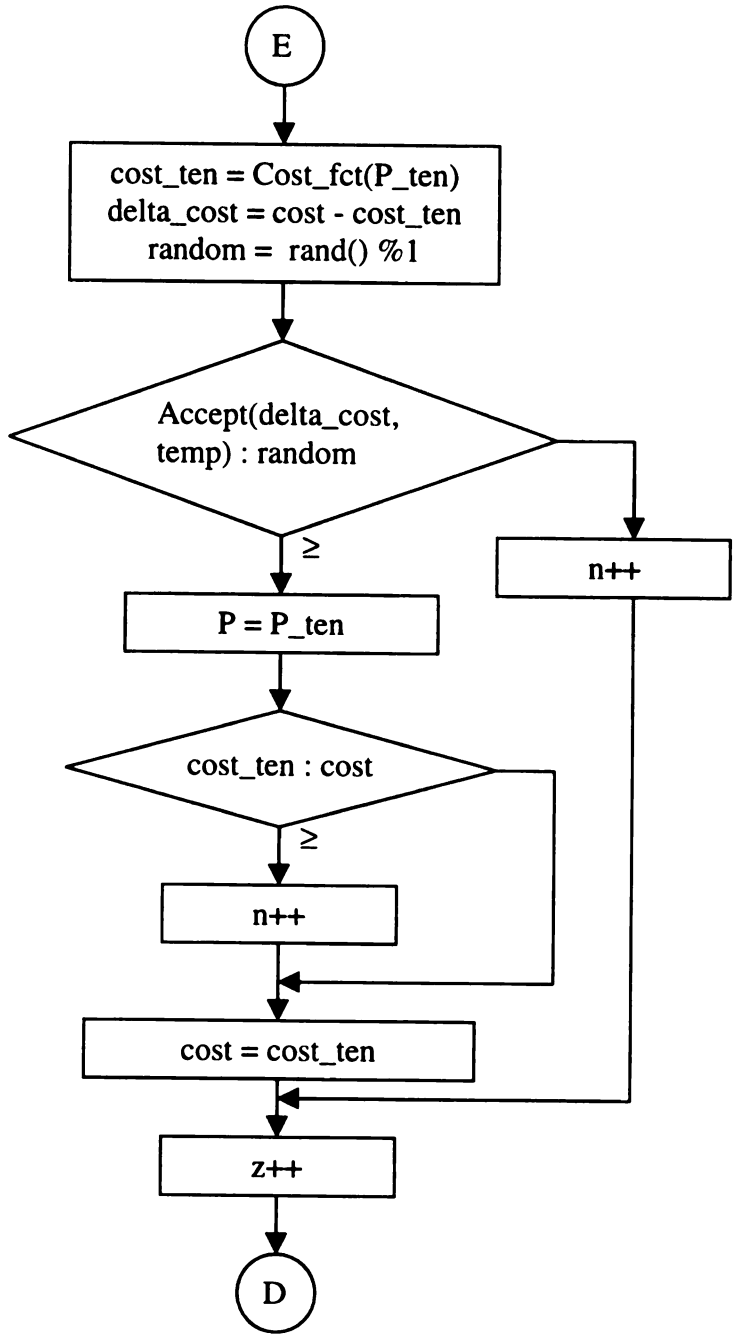

Figure 5.2: (Continued).

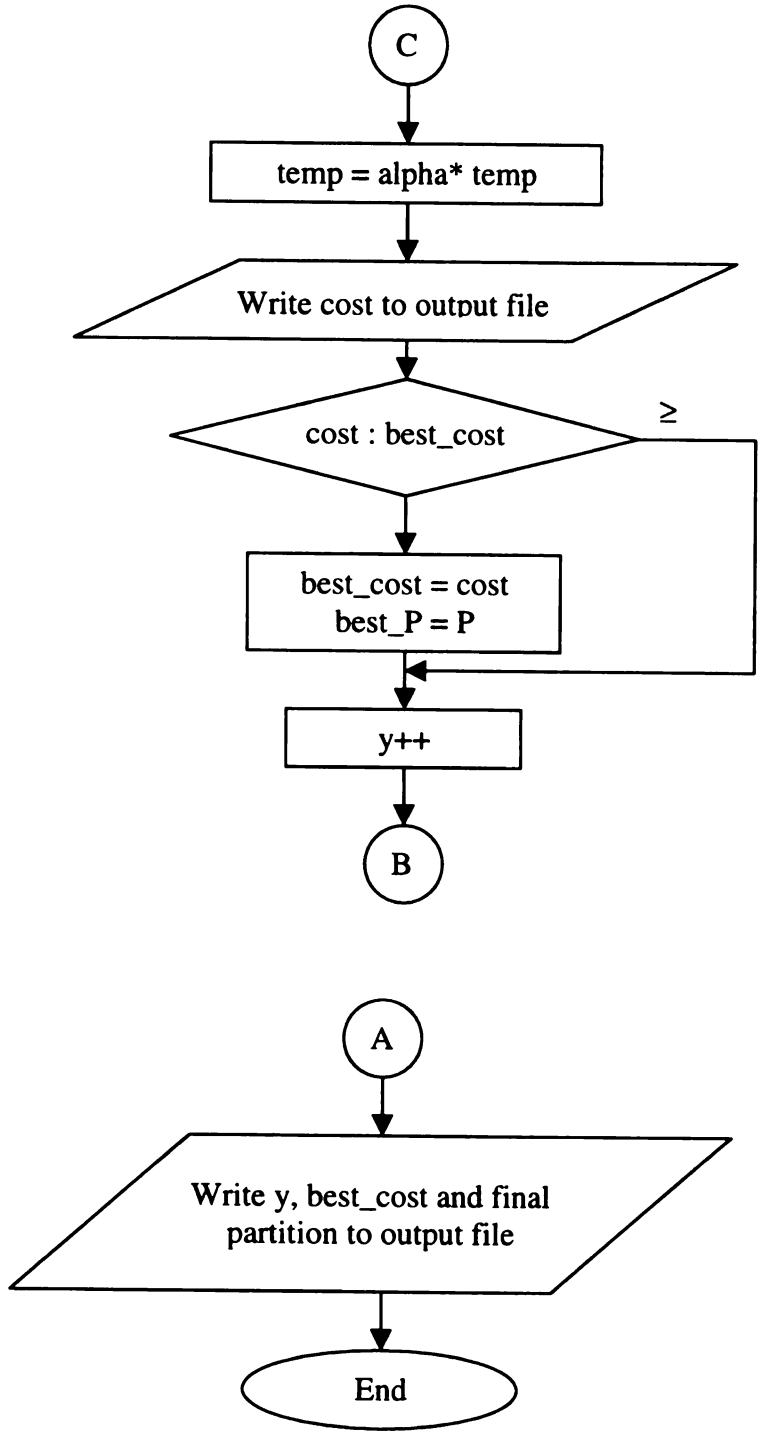

Figure 5.2: (Continued).

APPENDIX C

GROUP MIGRATION ALGORITHM RESULTS

Results of the application of Group Migration Algorithm to two hypothetical systems consisting of five and 50 modules respectively are discussed in this appendix. Table C.1 shows the input file for the system of five modules. Table C.2 shows the partitioned system based on the given inputs. Figures C.1 to C3 show the hill-climbing nature of the algorithm.

Figure C.1 is a plot of best cost found after each iteration of the algorithm for the small example. Table C.3 lists the input data for the large example consisting of 50 modules. Tables C.4 and C.5 list the partitions found for different values of weights. Figures C.2 and C.3 show the plot of cost vs. iterations in each case.

Module	Hw_Area	Sw_Size	Hw_Delay	Sw_Delay
A	234	482	28	100
B	374	500	48	240
C	198	273	40	190
D	328	430	38	245
E	283	381	59	120
Constraints	500	1000	100	500
Weights	0.25, 0.25, 0.25, 0.25			

Table C.1: Input data for the small example.

ALGORITHM STEPS

P_in= {A,B,C,D,E} All software partition. Generate a random partition.
P={**A**,**B**,C,**D**,**E**}; A,B, D and E (bold) are in hardware and C is in software.
Do Loop {
prev_cost = 0.542, bestpart_cost = infi
Reset moved flag of all objects as false.
j=0, Move objects to opposite side
bestmove_cost = infi
k=0, P={A,**B**,C,**D**,**E**} Cost = 0.355 // A is moved to Sw
k=1, P={**A**,B,C,**D**,**E**} Cost = 0.235 // B is moved to Sw
k=2, P={**A**,**B**,C,**D**,**E**} Cost = 0.741 // C is moved to Hw
k=3, P={**A**,**B**,C,D,**E**} Cost = 0.283 // D is moved to Sw
k=4, P={**A**,**B**,C,**D**,E} Cost = 0.253 // E is moved to Sw
bestmove_cost = 0.235, bestpart_cost = 0.235, B is moved and locked in software
bestcost_P={**A**,B,C,**D**,**E**}, P={**A**,~~**B**~~,C,**D**,**E**}
j=1, Move objects to opposite side
k=0, P={A,~~**B**~~,C,**D**,**E**} Cost = 0.134
k=2, P={**A**,~~**B**~~,C,**D**,**E**} Cost = 0.434
k=3, P={**A**,~~**B**~~,C,D,**E**} Cost = 0.1467
k=4, P={**A**,~~**B**~~,C,**D**,E} Cost= 0.0945
bestmove_cost = 0.0945, bestpart_cost = 0.0945, E is moved and locked in software
bestcost_P ={**A**,B,C,**D**,E}, P={**A**,~~**B**~~,C,**D**,~~**E**~~}
j=2, Move objects to opposite side
k=1, P={A,~~**B**~~,C,**D**,~~**E**~~} Cost = 0.234
k=3, P={**A**,~~**B**~~,C,D,~~**E**~~} Cost = 0.145
k=4, P={**A**,~~**B**~~,C,D,~~**E**~~} Cost = 0.2935
bestmove_cost = 0.145, bestpart_cost = 0.0945, C is moved and locked in hardware
bestcost_P={**A**,B,C,**D**,E}, P={**A**,~~**B**~~,~~**C**~~,**D**,~~**E**~~}
j=3, Move objects to opposite side
k=1, P={A,~~**B**~~,~~**C**~~,**D**,~~**E**~~} Cost = 0.10375
k=4, P={**A**,~~**B**~~,~~**C**~~,D,~~**E**~~} Cost = 0.13025
bestmove_cost = 0.10375, bestpart_cost = 0.0945, A is moved and locked in software
bestcost_P={**A**,B,C,**D**,E}, P={~~**A**~~,~~**B**~~,~~**C**~~,**D**,~~**E**~~}
j=4, Move objects to opposite side
k=4, P={~~**A**~~,~~**B**~~,~~**C**~~,D,~~**E**~~} Cost = 0.30075
bestmove_cost = 0.30075, bestpart_cost = 0.0945, D is moved and locked in software
bestcost_P={**A**,B,C,**D**,E}, End of 1st Iteration
bestpart_cost is less then prev_cost therefore
P= bestcost_P
while (bestpart_cost is less then prev_cost) }
if cost of P is less then best_cost then
best_cost = Cost_Fct(P), best_P = P
Repeat the algorithm nn times with new randomly generated partitions.

The best cost found after 52 iterations is 0.0945

Partitioned System (Processor Clock = 1 MHz)

Module	Hw_Area	Sw_Size	Hw_Delay	Sw_Delay	Implement
A	234	-	28	-	HW
B	-	500	-	240	SW
C	-	273	-	190	SW
D	328	-	38	-	HW
E	-	381	-	120	SW
Totals	562	1154	66	550	
Constraints	500	1000	100	500	
Difference	62	154	-34	50	

Table C.2: Partitioned system of the small example.

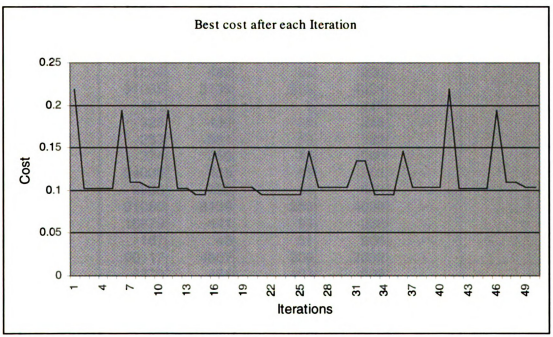

Figure C.1: Plot of cost vs. iterations corresponding to partition in Table C.2.

Module	HW Area	SW Size	HW Delay	SW Delay	Locked
Mod1	12499	712	87	944	-
Mod2	8838	720	56	501	-
Mod3	90117	4807	306	7392	-
Mod4	91065	3139	255	4034	-
Mod5	9501	211	37	176	-
Mod6	24036	405	85	301	-
Mod7	16879	441	93	238	-
Mod8	23339	267	85	218	-
Mod9	741	228	11	216	-
Mod10	461	91	8	115	-
Mod11	1234	482	88	200	-
Mod12	374	350	48	240	-
Mod13	198	273	40	190	-
Mod14	328	430	38	245	-
Mod15	283	381	59	120	-
Mod16	1339	167	85	118	-
Mod17	6879	461	103	338	-
Mod18	167	43	51	200	-
Mod19	9017	207	36	392	-
Mod20	29840	870	209	873	-
Mod21	4561	189	48	285	-
Mod22	40985	4509	307	2874	-
Mod23	3289	732	98	569	-
Mod24	12499	712	87	944	-
Mod25	90117	4807	306	7392	-
Mod26	9501	211	37	176	-
Mod27	24036	405	85	301	-
Mod28	1234	482	88	200	-
Mod29	91065	3139	255	4034	-
Mod30	461	91	8	115	-
Mod31	328	430	38	245	-
Mod32	283	381	59	120	-
Mod33	374	350	48	240	-
Mod34	24036	405	85	301	-
Mod35	8838	720	56	501	-
Mod36	91065	3139	255	4034	-
Mod37	16879	441	93	238	-
Mod38	167	43	51	200	-
Mod39	90117	4807	306	7392	-
Mod40	6879	461	103	338	-
Mod41	16879	441	93	238	-
Mod42	374	350	48	240	-
Mod43	16879	441	93	238	-
Mod44	6879	461	103	338	-
Mod45	23339	267	85	218	-
Mod46	167	43	51	200	-
Mod47	198	273	40	190	-

Table C.3: Input data for the large example.

Mod48	283	381	59	120	-
Mod49	90117	4807	306	7392	-
Mod50	23339	267	85	218	-
Constraints	1000000	16000	1000	10000	
Weights 025, 0.25, 0.25, 0.25					

Table C.3: Input data for the large example (Continued).

The best cost found after 1002 iterations is 0.37585

Module	HW Area	SW Size	HW Delay	SW Delay	Implement
Mod1	12499	-	87	-	HW
Mod2	-	720	-	501	SW
Mod3	90117	-	306	-	HW
Mod4	91065	-	255	-	HW
Mod5	-	211	-	176	SW
Mod6	-	405	-	301	SW
Mod7	-	441	-	238	SW
Mod8	-	267	-	218	SW
Mod9	741	-	11	-	HW
Mod10	461	-	8	-	HW
Mod11	-	482	-	200	SW
Mod12	-	350	-	240	SW
Mod13	-	273	-	190	SW
Mod14	-	430	-	245	SW
Mod15	-	381	-	120	SW
Mod16	-	167	-	118	SW
Mod17	-	461	-	338	SW
Mod18	-	43	-	200	SW
Mod19	-	207	-	392	SW
Mod20	-	870	-	873	SW
Mod21	-	189	-	285	SW
Mod22	40985	-	307	-	HW
Mod23	-	732	-	569	SW
Mod24	12499	-	87	-	HW
Mod25	90117	-	306	-	HW
Mod26	-	211	-	176	SW
Mod27	-	405	-	301	SW
Mod28	-	482	-	200	SW
Mod29	91065	-	255	-	HW
Mod30	461	-	8	-	HW
Mod31	-	430	-	245	SW
Mod32	-	381	-	120	SW
Mod33	-	350	-	240	SW
Mod34	-	405	-	301	SW
Mod35	-	720	-	501	SW

Table C.4: Partitioned system of the large example with equal weights.

Mod36	91065	-	255	-	HW
Mod37	-	441	-	238	SW
Mod38	-	43	-	200	SW
Mod39	90117	-	306	-	HW
Mod40	-	461	-	338	SW
Mod41	-	441	-	238	SW
Mod42	-	350	-	240	SW
Mod43	-	441	-	238	SW
Mod44	-	461	-	338	SW
Mod45	-	267	-	218	SW
Mod46	-	43	-	200	SW
Mod47	-	273	-	190	SW
Mod48	-	381	-	120	SW
Mod49	90117	-	306	-	HW
Mod50	-	267	-	218	SW
Totals	701309	13882	2497	10064	
Constraints	1000000	16000	1000	10000	
Difference	-298691	-2118	1497	64	

Table C.4: Partitioned system of the large example with equal weights (Continued).

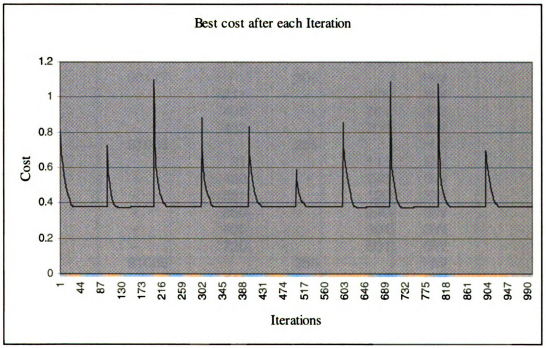

Figure C.2: Plot of cost vs. iterations corresponding to partition in Table C.4.

Weights 0.35, 0.15, 0.25, 0.25
The best cost found after 2402 iterations is 0.35948
Partitioned System Processor clock = 1 MHz

Module	HW Area	SW Size	HW Delay	SW Delay	Implement
Mod1	-	712	-	944	SW
Mod2	-	720	-	501	SW
Mod3	90117	-	306	-	HW
Mod4	91065	-	255	-	HW
Mod5	-	211	-	176	SW
Mod6	-	405	-	301	SW
Mod7	-	441	-	238	SW
Mod8	-	267	-	218	SW
Mod9	741	-	11	-	HW
Mod10	-	91	-	115	SW
Mod11	-	482	-	200	SW
Mod12	-	350	-	240	SW
Mod13	-	273	-	190	SW
Mod14	-	430	-	245	SW
Mod15	-	381	-	120	SW
Mod16	-	167	-	118	SW
Mod17	-	461	-	338	SW
Mod18	-	43	-	200	SW
Mod19	-	207	-	392	SW
Mod20	-	870	-	873	SW
Mod21	-	189	-	285	SW
Mod22	40985	-	307	-	HW
Mod23	-	732	-	569	SW
Mod24	-	712	-	944	SW
Mod25	90117	-	306	-	HW
Mod26	-	211	-	176	SW
Mod27	-	405	-	301	SW
Mod28	-	482	-	200	SW
Mod29	91065	-	255	-	HW
Mod30	-	91	-	115	SW
Mod31	-	430	-	245	SW
Mod32	-	381	-	120	SW
Mod33	-	350	-	240	SW
Mod34	-	405	-	301	SW
Mod35	-	720	-	501	SW
Mod36	91065	-	255	-	HW
Mod37	-	441	-	238	SW
Mod38	-	43	-	200	SW
Mod39	90117	-	306	-	HW
Mod40	-	461	-	338	SW
Mod41	-	441	-	238	SW
Mod42	-	350	-	240	SW

Table C.5: Partitioned system when Hw_Area is more important.

Mod43	-	441	-	238	SW
Mod44	-	461	-	338	SW
Mod45	-	267	-	218	SW
Mod46	-	43	-	200	SW
Mod47	-	273	-	190	SW
Mod48	-	381	-	120	SW
Mod49	90117	-	306	-	HW
Mod50	-	267	-	218	SW
Totals	675389	15488	2307	12182	
Constraints	1000000	16000	1000	10000	
Difference	-324611	-512	1307	2182	

Table C.5: Partitioned system when Hw_Area is more important (Continued).

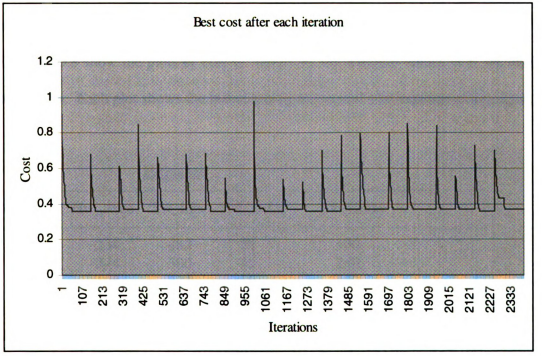

Figure C.3: Plot of cost vs. iterations corresponding to partition in Table C.5.

APPENDIX D

SIMULATED ANNEALING ALGORITHM RESULTS

Results of the application of simulated annealing algorithm to two hypothetical systems consisting of five and 50 modules respectively are discussed in this appendix. Table D.1 shows the input file for the system of five modules. Table D.2 shows the partitioned system based on the given inputs. Figure D.1 is a plot of costs recorded after reaching equilibrium at each temperature. Figures D.2 to D.6 show the plots of cost vs. iterations with various values of **n_eq, init** and **freez** temperatures and **alpha**. Table D.3 shows the input file for the system of 50 modules. Table D.4 shows the partitioned system. Figures D.7 and D.8 show the plots of cost vs. iterations for different values of **n_eq, init, freez** and **alpha**. Each plot shows the hill-climbing nature and convergence of the algorithm.

	Estimates			
Module	Hw_Area	Sw_Size	Hw_Delay	Sw_Delay
A	234	482	28	100
B	374	500	48	240
C	198	273	40	190
D	328	430	38	245
E	283	381	59	220

Table D.1: Input data for the small example.

Took 459 iterations. Best cost = 0.0945

Partitioned System — Processor clock = 1 MHz

Module	Hw_Area	Sw_Size	Hw_Delay	Sw_Delay	Implement
A	234	-	28	-	HW
B	-	500	-	240	SW
C	-	273	-	190	SW
D	328	-	38	-	HW
E	-	381	-	120	SW
Totals	562	1154	66	550	
Constraints	500	1000	100	500	
Difference	62	154	-34	50	

Table D.2: Partitioned system of the small example.

Case 1: With init = 1.0, n_eq = 3, alpha = 0.99, freez = 0.01:

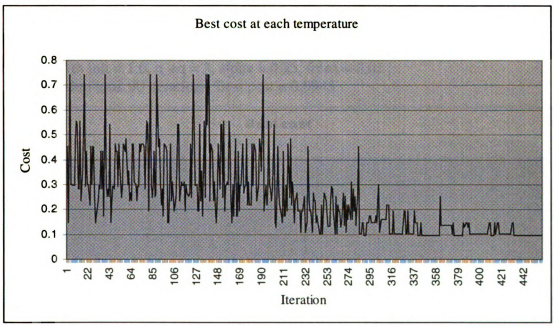

Figure D.1: Plot of costs vs. iterations for the small example (Case1).

Case 2: With init = 1.0, n_eq = 5, alpha = 0.99, freez = 0.01
The algorithm took 459 iterations. Best cost = 0.0945 for Partition: {HSSHS}

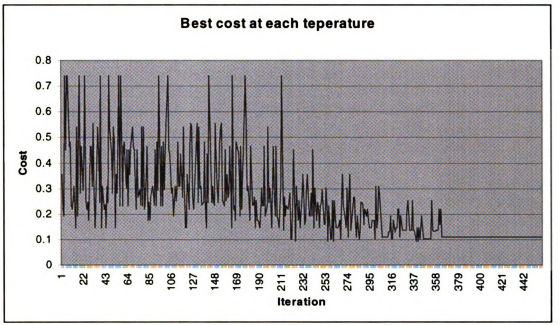

Figure D.2: Plot of costs vs. iterations for the small example (Case 2).

Case 3: With init = 1.0, n_eq = 3, alpha = 0.95, freez = 0.01
The algorithm took 90 iterations. Best cost = 0.0945

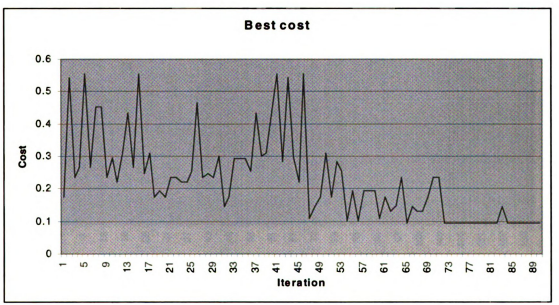

Figure D.3: Plot of costs vs. iterations for the small example (Case 3).

Case 4: With init = 1.0, n_eq = 5, alpha = 0.95, freez = 0.01
Took 90 iterations. Best cost = 0.0945

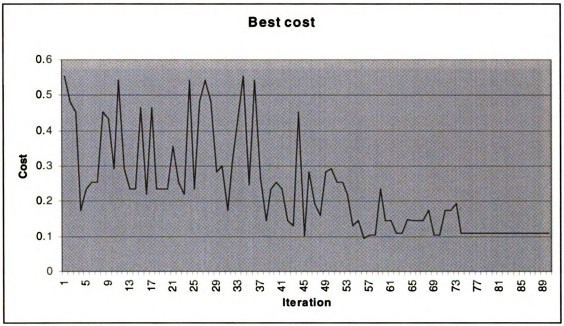

Figure D.4: Plot of costs vs. iterations for each temperature value (Case 4).

Case 5: init = 10.0, n_eq = 3, alpha = 0.95, freez = 0.01
Took 135 iterations. Best cost = 0.0945

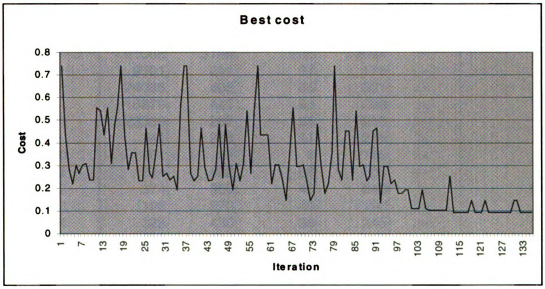

Figure D.5: Plot of costs vs. iterations for the small example (Case 5).

Case 6: init = 10.0, n_eq = 3, alpha = 0.99, freez = 0.01
Took 688 iterations. Best cost = 0.0945 for Partition: {HSSHS}

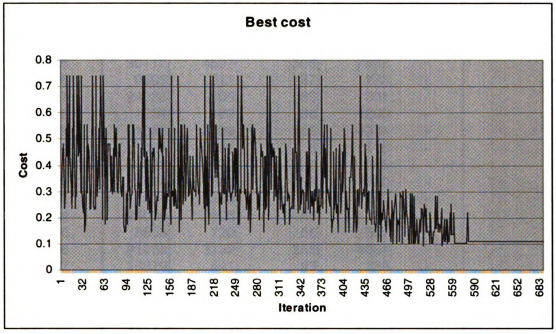

Figure D.6: Plot of costs vs. iterations for the small example (Case 6).

Module	HW Area	SW Size	HW Delay	SW Delay	Locked
Mod1	12499	712	87	944	-
Mod2	8838	720	56	501	-
Mod3	90117	4807	306	7392	-
Mod4	91065	3139	255	4034	-
Mod5	9501	211	37	176	-
Mod6	24036	405	85	301	-
Mod7	16879	441	93	238	-
Mod8	23339	267	85	218	-
Mod9	741	228	11	216	-
Mod10	461	91	8	115	-
Mod11	1234	482	88	200	-
Mod12	374	350	48	240	-
Mod13	198	273	40	190	-
Mod14	328	430	38	245	-
Mod15	283	381	59	120	-
Mod16	1339	167	85	118	-
Mod17	6879	461	103	338	-
Mod18	167	43	51	200	-

Table D.3: Input data for the large example.

Mod19	9017	207	36	392	-
Mod20	29840	870	209	873	-
Mod21	4561	189	48	285	-
Mod22	40985	4509	307	2874	-
Mod23	3289	732	98	569	-
Mod24	12499	712	87	944	-
Mod25	90117	4807	306	7392	-
Mod26	9501	211	37	176	-
Mod27	24036	405	85	301	-
Mod28	1234	482	88	200	-
Mod29	91065	3139	255	4034	-
Mod30	461	91	8	115	-
Mod31	328	430	38	245	-
Mod32	283	381	59	120	-
Mod33	374	350	48	240	-
Mod34	24036	405	85	301	-
Mod35	8838	720	56	501	-
Mod36	91065	3139	255	4034	-
Mod37	16879	441	93	238	-
Mod38	167	43	51	200	-
Mod39	90117	4807	306	7392	-
Mod40	6879	461	103	338	-
Mod41	16879	441	93	238	-
Mod42	374	350	48	240	-
Mod43	16879	441	93	238	-
Mod44	6879	461	103	338	-
Mod45	23339	267	85	218	-
Mod46	167	43	51	200	-
Mod47	198	273	40	190	-
Mod48	283	381	59	120	-
Mod49	90117	4807	306	7392	-
Mod50	23339	267	85	218	-
Constraints	1000000	16000	1000	10000	
Weights 025, 0.25, 0.25, 0.25					

Table D.3: Input data for the large example (Continued).

Took 6905 iterations. Best cost= 0.37585
Partitioned System, Processor clock= 1 MHz

Module	HW Area	SW Size	HW Delay	SW Delay	Implement
Mod1	12499	-	87	-	HW
Mod2	-	720	-	501	SW
Mod3	90117	-	306	-	HW
Mod4	91065	-	255	-	HW
Mod5	-	211	-	176	SW
Mod6	-	405	-	301	SW

Table D.4: Partitioned system of the large example.

Mod7	-	441	-	238	SW
Mod8	-	267	-	218	SW
Mod9	741	-	11	-	HW
Mod10	461	-	8	-	HW
Mod11	-	482	-	200	SW
Mod12	-	350	-	240	SW
Mod13	-	273	-	190	SW
Mod14	-	430	-	245	SW
Mod15	-	381	-	120	SW
Mod16	-	167	-	118	SW
Mod17	-	461	-	338	SW
Mod18	-	43	-	200	SW
Mod19	-	207	-	392	SW
Mod20	-	870	-	873	SW
Mod21	-	189	-	285	SW
Mod22	40985	-	307	-	HW
Mod23	-	732	-	569	SW
Mod24	12499	-	87	-	HW
Mod25	90117	-	306	-	HW
Mod26	-	211	-	176	SW
Mod27	-	405	-	301	SW
Mod28	-	482	-	200	SW
Mod29	91065	-	255	-	HW
Mod30	461	-	8	-	HW
Mod31	-	430	-	245	SW
Mod32	-	381	-	120	SW
Mod33	-	350	-	240	SW
Mod34	-	405	-	301	SW
Mod35	-	720	-	501	SW
Mod36	91065	-	255	-	HW
Mod37	-	441	-	238	SW
Mod38	-	43	-	200	SW
Mod39	90117	-	306	-	HW
Mod40	-	461	-	338	SW
Mod41	-	441	-	238	SW
Mod42	-	350	-	240	SW
Mod43	-	441	-	238	SW
Mod44	-	461	-	338	SW
Mod45	-	267	-	218	SW
Mod46	-	43	-	200	SW
Mod47	-	273	-	190	SW
Mod48	-	381	-	120	SW
Mod49	90117	-	306	-	HW
Mod50	-	267	-	218	SW
Totals	701309	13882	2497	10064	
Constraints	1000000	16000	1000	10000	
Difference	-298691	-2118	1497	64	

Table D.4: Partitioned system of the large example (Continued).

Case 1: init = 1.0, n_eq = 3, alpha = 0.999, freez = 0.001

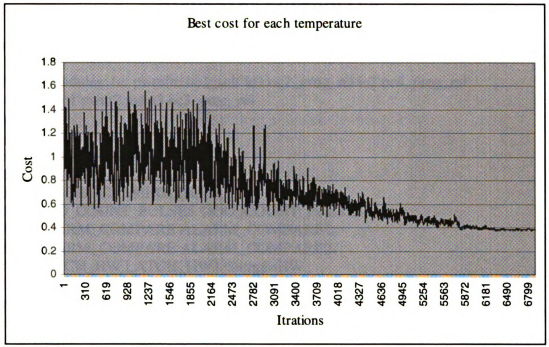

Figure D.7: Plot of costs vs. iterations for the large example (Case 1).

Case 2: init = 0.5, n_eq = 3, alpha = 0.999, freez = 0.001

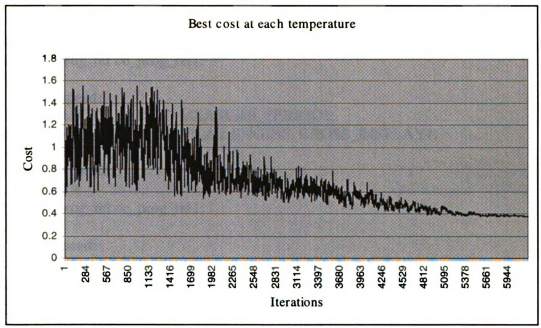

Figure D.8: Plot of costs vs. iterations for the large example (Case 2).

APPENDIX E

DASHBOARD CONTROLLER

List of Modules

```
# user_lib modules: frc pwmfrc oc1_self ic1 oc2_prog_rel ic2 oc4_prog_rel
# pwm0 pwm1 pwm2 pwm3 oc3_prog_rel

net dac_demo
module DEBOUNCE DEBOUNCE1;
module DEBOUNCE DEBOUNCE2;
module DEBOUNCE DEBOUNCE3;
module ODO_COUNT_PULSES ODO_COUNT_PULSES1;
module ALARM_COMPARE ALARM_COMPARE1;
module ALARM_COMPARE ALARM_COMPARE2;
module LATCH_HW LATCH_HW1 %impl=HW ;
module LATCH_HW LATCH_HW2 %impl=HW ;
module LATCH_HW LATCH_HW3 %impl=HW ;
module belt_control belt_control1 ;
module engine_speed engine_speed1;
module wheel_speed wheel_speed1;
module frc frc1;
module oc_self oc_self1;
module ic ic1;
module pwmfrc pwmfrc1;

net belt_control:
module BELT BELT1;
module oc_prog_rel oc_prog_rel1;

net engine_speed:
module MEASURE_PERIOD MEASURE_PERIOD1;
module ENGINE_CROSS_DISPLAY ENGINE_CROSS_DISPLAY1;
module pwm pwm1;
module pwm pwm2;
module ic ic1 ;
module oc_prog_rel oc_prog_rel1;

net wheel_speed:
module SPEED_COUNT_PULSES SPEED_COUNT_PULSES1;
module SPEED_CROSS_DISPLAY SPEED_CROSS_DISPLAY1;
module oc_prog_rel oc_prog_rel1;
module pwm pwm1;
module pwm pwm2;
```

Auxiliary File containing information on interconnection of modules:

 # dac_demo.aux
#
user_lib modules: frc pwmfrc oc1_self ic1 oc2_prog_rel ic2 oc4_prog_rel
pwm0 pwm1 pwm2 pwm3 oc3_prog_rel
user lib module constants (500 KHz timer clock, 500 KHz pwm clock)
MAXCOUNT: 65536
PRESCALER: 1 in simulation (prescaling in Absclock) 4 in reality
OC1PERIOD: 500 (10 msec)
PWMMAXCOUNT 1024
PWMPRESCALER: 2 in simulation (prescaling in Absclock) 8 in reality
mv BELT1_e_OC2_START 32768
mv MEASURE_PERIOD1_e_OC4_START 32768
mv SPEED_COUNT_PULSES1_e_OC3_START 32768
mv CONST_MAX_FRC_NUMBER 131072
mv CONST_TIME_UP_ENGINE_OFF 131072
mv CONST_ENGINE_CROSS_DISPLAY 262144
mv K_ACC 262144
mv CONST_TIME_UP_WHEEL_PULSE 131072
mv MAXCOUNT 131072
mv OC1PERIOD 131072
mv X 1024
mv Y 1024
mv ENGINE_COIL_0 1
mv ENGINE_COIL_1 1
mv SPEED_COIL_2 1
mv SPEED_COIL_3 1
mv PARTIAL_RESET 1
mv KEY 1
mv BELT 1

net dac_demo:
input KEY , BELT , RESET , ECLK , ENGINE_PULSE , WHEEL_PULSE ,
PARTIAL_RESET , FUEL_LEVEL , WATER_LEVEL ;
output BELT_ALARM_HW , ENGINE_COIL_0 , ENGINE_COIL_1 , SPEED_COIL_2
, SPEED_COIL_3 , PARTIAL_TENTH_KM_RUN , TOTAL_TENTH_KM_RUN ,
FUEL_ALARM_HW , WATER_ALARM_HW ;
module DEBOUNCE DEBOUNCE1 [RESET/RESET, SENS_IN/KEY,
OC1_END/oc_self1_e_OC_END, EVENT_ON/DEBOUNCE1_e_EVENT_ON,
EVENT_OFF/DEBOUNCE1_e_EVENT_OFF, CONST_N_SAMPLES/10] ;
module DEBOUNCE DEBOUNCE2 [RESET/RESET, SENS_IN/BELT,
OC1_END/oc_self1_e_OC_END, EVENT_ON/DEBOUNCE2_e_EVENT_ON,
EVENT_OFF/DEBOUNCE2_e_EVENT_OFF, CONST_N_SAMPLES/10] ;

```
module DEBOUNCE DEBOUNCE3 [ RESET/RESET, SENS_IN/PARTIAL_RESET,
OC1_END/oc_self1_e_OC_END, EVENT_ON/DEBOUNCE3_e_EVENT_ON,
EVENT_OFF/DEBOUNCE3_e_EVENT_OFF, CONST_N_SAMPLES/10 ] ;
module ODO_COUNT_PULSES ODO_COUNT_PULSES1 [
PARTIAL_RESET/DEBOUNCE3_e_EVENT_ON, WHEEL_PULSE/ic1_e_IC_END,
PARTIAL_TENTH_KM_RUN/PARTIAL_TENTH_KM_RUN,
TOTAL_TENTH_KM_RUN/TOTAL_TENTH_KM_RUN,
CONST_ODOMETER_MAX_VALUE/1000, CONST_ODOMETER_TENTH_KM/200
] ;
module ALARM_COMPARE ALARM_COMPARE1 [ RESET/RESET,
LEVEL/FUEL_LEVEL, ALARM/ALARM_COMPARE1_e_ALARM,
CONST_THRESHOLD_LEVEL/20, CONST_INIT_ALARM/1 ] ;
module ALARM_COMPARE ALARM_COMPARE2 [ RESET/RESET,
LEVEL/WATER_LEVEL, ALARM/ALARM_COMPARE2_e_ALARM,
CONST_INIT_ALARM/0, CONST_THRESHOLD_LEVEL/120 ] ;
module LATCH_HW LATCH_HW1 [ IN/LATCH_HW1_e_IN,
OUT/BELT_ALARM_HW ] %impl=HW ;
module LATCH_HW LATCH_HW2 [ IN/ALARM_COMPARE1_e_ALARM,
OUT/FUEL_ALARM_HW ] %impl=HW ;
module LATCH_HW LATCH_HW3 [ IN/ALARM_COMPARE2_e_ALARM,
OUT/WATER_ALARM_HW ] %impl=HW ;
module belt_control belt_control1 [ KEY_ON/DEBOUNCE1_e_EVENT_ON,
RESET/RESET, KEY_OFF/DEBOUNCE1_e_EVENT_OFF,
BELT_ON/DEBOUNCE2_e_EVENT_ON, TCLK/frc1_e_TCLK,
ALARM/LATCH_HW1_e_IN, MAXCOUNT/65536, SIMSCALE/0 ] ;
module engine_speed engine_speed1 [ PWMCLK/pwmfrc1_e_PWMCLK,
RESET/RESET, IC2_START/ENGINE_PULSE, TCLK/frc1_e_TCLK,
ENGINE_COIL_0/ENGINE_COIL_0, ENGINE_COIL_1/ENGINE_COIL_1,
MAXCOUNT/65536, PWMMAXCOUNT/1024, TIME_UP_ENGINE_OFF/60000,
SIMSCALE/0 ] ;
module wheel_speed wheel_speed1 [ PWMCLK/pwmfrc1_e_PWMCLK,
RESET/RESET, WHEEL_PULSE/ic1_e_IC_END, TCLK/frc1_e_TCLK,
SPEED_COIL_2/SPEED_COIL_2, SPEED_COIL_3/SPEED_COIL_3,
MAXCOUNT/65536, PWMMAXCOUNT/1024, SIMSCALE/0 ] ;
module frc frc1 [ ECLK/ECLK, TCLK/frc1_e_TCLK, SIMSCALE/0,
MAXCOUNT/65536, PRESCALER/1 ] ;
module oc_self oc_self1 [ RESET/RESET, TCLK/frc1_e_TCLK,
OC_END/oc_self1_e_OC_END, SIMSCALE/0, MAXCOUNT/65536, OCPERIOD/256,
N/1 ] ;
module ic ic1 [ IC_START/WHEEL_PULSE, TCLK/frc1_e_TCLK,
IC_END/ic1_e_IC_END, MAXCOUNT/65536, N/1 ] ;
module pwmfrc pwmfrc1 [ ECLK/ECLK, PWMCLK/pwmfrc1_e_PWMCLK,
PWMPRESCALER/1, SIMSCALE/0, PWMMAXCOUNT/1024 ] ;
.
```

net belt_control:
input KEY_ON , RESET , KEY_OFF , BELT_ON , TCLK , MAXCOUNT,
SIMSCALE;
output ALARM ;
module BELT BELT1 [RESET/RESET, KEY_ON/KEY_ON, KEY_OFF/KEY_OFF,
BELT_ON/BELT_ON, OC2_END/BELT1_e_OC2_END,
OC2_START/BELT1_e_OC2_START, ALARM/ALARM, CONST_ALARM_OFF/0,
CONST_MAX_FRC_NUMBER/16384, CONST_PRE_SCAL_5/76,
CONST_PRE_SCAL_10/152] ;
module oc_prog_rel oc_prog_rel1 [RESET/RESET,
OC_START/BELT1_e_OC2_START, TCLK/TCLK, OC_END/BELT1_e_OC2_END,
SIMSCALE/SIMSCALE, MAXCOUNT/MAXCOUNT, N/2] ;
.

net engine_speed:
input PWMCLK , RESET , IC2_START , TCLK , TIME_UP_ENGINE_OFF,
MAXCOUNT, PWMMAXCOUNT, SIMSCALE;
output ENGINE_COIL_0 , ENGINE_COIL_1 ;
module MEASURE_PERIOD MEASURE_PERIOD1 [RESET/RESET,
ENGINE_PULSE/MEASURE_PERIOD1_e_ENGINE_PULSE,
OC4_END/MEASURE_PERIOD1_e_OC4_END,
PERIOD/MEASURE_PERIOD1_e_PERIOD,
OC4_START/MEASURE_PERIOD1_e_OC4_START,
CONST_TIME_UP_ENGINE_OFF/TIME_UP_ENGINE_OFF,
CONST_MAX_FRC_NUMBER/MAXCOUNT] ;
module ENGINE_CROSS_DISPLAY ENGINE_CROSS_DISPLAY1 [RESET/RESET,
PERIOD/MEASURE_PERIOD1_e_PERIOD, X/ENGINE_CROSS_DISPLAY1_e_X,
Y/ENGINE_CROSS_DISPLAY1_e_Y, CONST_ENGINE_CROSS_DISPLAY/125000,
PWMMAXCOUNT/PWMMAXCOUNT,
CONST_TIME_UP_ENGINE_OFF/TIME_UP_ENGINE_OFF] ;
module pwm pwm1 [PWMCLK/PWMCLK,
PWM_DC/ENGINE_CROSS_DISPLAY1_e_X, PWM_OUT/ENGINE_COIL_0,
PWM_POLARITY/1, SIMSCALE/SIMSCALE,
PWMMAXCOUNT/PWMMAXCOUNT, N/0] ;
module pwm pwm2 [PWMCLK/PWMCLK,
PWM_DC/ENGINE_CROSS_DISPLAY1_e_Y, PWM_OUT/ENGINE_COIL_1,
PWM_POLARITY/1, SIMSCALE/SIMSCALE,
PWMMAXCOUNT/PWMMAXCOUNT, N/1] ;
module ic ic1 [IC_START/IC2_START, TCLK/TCLK,
IC_END/MEASURE_PERIOD1_e_ENGINE_PULSE, MAXCOUNT/MAXCOUNT,
N/2] ;
module oc_prog_rel oc_prog_rel1 [RESET/RESET,
OC_START/MEASURE_PERIOD1_e_OC4_START, TCLK/TCLK,
OC_END/MEASURE_PERIOD1_e_OC4_END, SIMSCALE/SIMSCALE,
MAXCOUNT/MAXCOUNT, N/4] ;
.

net wheel_speed:
input PWMCLK , RESET , WHEEL_PULSE , TCLK , MAXCOUNT,
PWMMAXCOUNT, SIMSCALE;
output SPEED_COIL_2 , SPEED_COIL_3 ;
module SPEED_COUNT_PULSES SPEED_COUNT_PULSES1 [RESET/RESET,
WHEEL_PULSE/WHEEL_PULSE,
OC3_END/SPEED_COUNT_PULSES1_e_OC3_END,
OC3_START/SPEED_COUNT_PULSES1_e_OC3_START,
WHEEL_PULSES/SPEED_COUNT_PULSES1_e_WHEEL_PULSES,
CONST_TIME_UP_WHEEL_PULSE/62500] ;
module SPEED_CROSS_DISPLAY SPEED_CROSS_DISPLAY1 [RESET/RESET,
WHEEL_PULSES/SPEED_COUNT_PULSES1_e_WHEEL_PULSES,
X/SPEED_CROSS_DISPLAY1_e_X, Y/SPEED_CROSS_DISPLAY1_e_Y,
CONST_NUM_FRAME/4, CONST_ALPHA_MAX/270,
CONST_NUM_PULSE_MAX/1500, PWMMAXCOUNT/PWMMAXCOUNT] ;
module oc_prog_rel oc_prog_rel1 [RESET/RESET,
OC_START/SPEED_COUNT_PULSES1_e_OC3_START, TCLK/TCLK,
OC_END/SPEED_COUNT_PULSES1_e_OC3_END, SIMSCALE/SIMSCALE,
MAXCOUNT/MAXCOUNT, N/3] ;
module pwm pwm1 [PWMCLK/PWMCLK,
PWM_DC/SPEED_CROSS_DISPLAY1_e_X, PWM_OUT/SPEED_COIL_2,
PWM_POLARITY/1, SIMSCALE/SIMSCALE,
PWMMAXCOUNT/PWMMAXCOUNT, N/3] ;
module pwm pwm2 [PWMCLK/PWMCLK,
PWM_DC/SPEED_CROSS_DISPLAY1_e_Y, PWM_OUT/SPEED_COIL_3,
PWM_POLARITY/1, SIMSCALE/SIMSCALE,
PWMMAXCOUNT/PWMMAXCOUNT, N/2] ;
.

Module **belt_control**:

Module	HW Area	SW Size	HW Delay	SW Delay	Locked
BELT	-	720	-	501	S
oc_prog_rel	218544	532	23	691	-
Constraints	200000	1000	1000	10000	

Multipliers 0.25, 0.25, 0.25, 0.25

Took 459 iterations, Best cost= 0.02318
Partitioned System, Processor clock= 1 MHz

Module	HW Area	SW Size	HW Delay	SW Delay	Implement
BELT	-	720	-	501	SW
oc_prog_rel	218544	-	23	-	HW
Totals	218544	720	23	501	
Constraints	200000	1000	1000	10000	
Difference	18544	-280	-977	-9499	

Table E.1: Input and partition data for belt_control module.

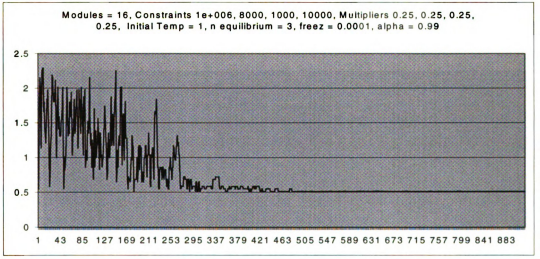

Figure E.1: Plot of cost vs. iterations for belt_control module.

Module **engine_speed**

Module	HW Area	SW Size	HW Delay	SW Delay	Locked
MEASURE_PERIOD	839376	441	60	238	-
ENGINE_CROSS_DISPLAY	-	3139	-	4034	S
pwm	461216	458	47	853	-
pwm	461216	458	47	853	-
ic	152656	91	15	161	-
oc_prog_rel	218544	532	23	691	-

Took 459 iterations. Best cost= 0.244119

Partitioned System				Processor clock = 1 MHz	
MEASURE_PERIOD	-	441	-	238	SW
ENGINE_CROSS_DISPLAY	-	3139	-	4034	SW
pwm	461216	-	47	-	HW
pwm	461216	-	47	-	HW
ic	-	91	-	161	SW
oc_prog_rel	218544	-	23	-	HW
Totals	1140976	3671	117	4433	
Constraints	1000000	2000	1000	10000	
Difference	140976	1671	-883	-5567	

Table E.2: Input and partition data for engine_speed module.

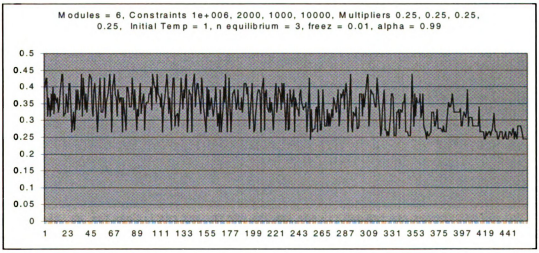

Figure E.2: Plot of cost vs. iterations for engine_speed module.

Module **wheel_speed**

Module	HW Area	SW Size	HW Delay	SW Delay	Locked
SPEED_COUNT_PULSES	450080	211	37	176	-
SPEED_CROSS_DISPLAY	0	4807	0	7392	S
oc_prog_rel	218544	532	23	691	-
pwm	461216	458	47	853	-
pwm	461216	458	47	853	-

Took 917 iterations. Best cost= 0.412494
Partitioned System Processor clock= 1 MHz

SPEED_COUNT_PULSES	-	211	-	176	SW
SPEED_CROSS_DISPLAY	-	4807	-	7392	SW
oc_prog_rel	218544	-	23	-	HW
pwm	461216	-	47	-	HW
pwm	461216	-	47	-	HW
Totals	1140976	5018	117	7568	
Constraints	1000000	2000	1000	10000	
Difference	140976	3018	-883	-2432	

Table E.3: Input and partition data for wheel_speed module.

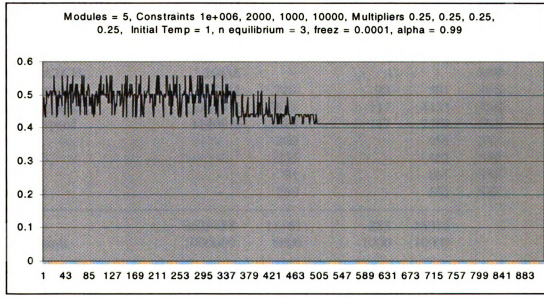

Figure E.3: Plot of cost vs. iterations for wheel_speed module.

net **dac_demo**

Module	HW Area	SW Size	HW Delay	SW Delay	Locked
DEBOUNCE	1286208	267	101	218	-
DEBOUNCE	1286208	267	101	218	-
DEBOUNCE	1286208	267	101	218	-
ODO_COUNT_PULSES	2148320	405	152	301	-
ALARM_COMPARE	33872	228	7	216	-
ALARM_COMPARE	33872	228	7	216	-
HW_LATCH	20416	-	4	-	H
HW_LATCH	20416	-	4	-	H
HW_LATCH	20416	-	4	-	H
belt_control	218544	720	23	501	X
engine_speed	1140976	3671	117	4433	X
wheel_speed	1140976	5018	117	7568	X
frc	586960	260	62	486	-
oc_self	170288	235	17	482	-
ic	152656	91	15	161	-
pwmfrc	575360	260	51	486	-

Took 917 iterations. Best cost= 0.64233

Partitioned System — Processor clock = 1 MHz

DEBOUNCE	-	267	-	218	SW
DEBOUNCE	-	267	-	218	SW
DEBOUNCE	-	267	-	218	SW
ODO_COUNT_PULSES	-	405	-	301	SW
ALARM_COMPARE	33872	-	7	-	HW
ALARM_COMPARE	33872	-	7	-	HW
HW_LATCH	20416	-	4	-	HW
HW_LATCH	20416	-	4	-	HW
HW_LATCH	20416	-	4	-	HW
belt_control	218544	720	23	501	H-S
engine_speed	1140976	3671	117	4433	H-S
wheel_speed	1140976	5018	117	7568	H-S
frc	-	260	-	486	SW
oc_self	-	235	-	482	SW
ic	-	91	-	161	SW
pwmfrc	-	260	-	486	SW
Totals	2629488	11461	283	15072	
Constraints	1000000	8000	1000	10000	
Difference	1629488	3461	-717	5072	

Table E.4: Input and partition data for net dac_demo (1 MHz clock).

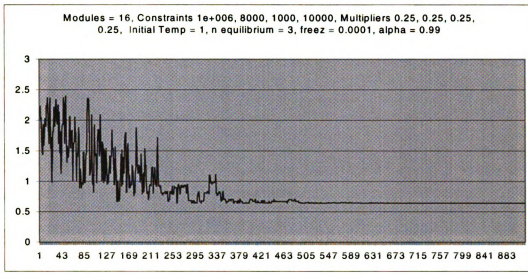

Figure E.4: Plot of cost vs. iterations for net dac_demo (1 MHz clock).

Took 917 iterations. Best cost= 0.512844
Partitioned System Processor clock = 4 MHz

Module	HW Area	SW Size	HW Delay	SW Delay	Implement
DEBOUNCE	-	267	-	54.5	SW
DEBOUNCE	-	267	-	54.5	SW
DEBOUNCE	-	267	-	54.5	SW
ODO_COUNT_PULSES	-	405	-	75.25	SW
ALARM_COMPARE	-	228	-	54	SW
ALARM_COMPARE	-	228	-	54	SW
HW_LATCH	20416	-	4	-	HW
HW_LATCH	20416	-	4	-	HW
HW_LATCH	20416	-	4	-	HW
belt_control	218544	720	23	125.25	H-S
engine_speed	1140976	3671	117	1108.25	H-S
wheel_speed	1140976	5018	117	1892	H-S
frc	-	260	-	121.5	SW
oc_self	-	235	-	120.5	SW
ic	-	91	-	40.25	SW
pwmfrc	-	260	-	121.5	SW
Totals	2561744	11917	269	3876	
Constraints	1000000	8000	1000	10000	
Difference	1561744	3917	-731	-6124	

Table E.5: Partition data for net dac_demo (4 MHz clock).

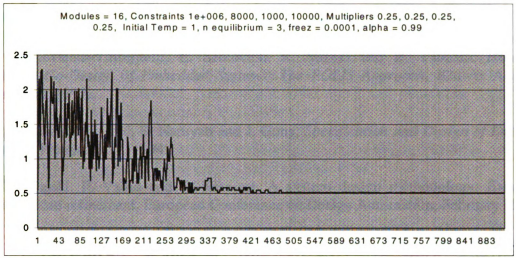

Figure E.5: Plot of cost vs. iterations for net dac_demo (4 MHz clock)

Took 459 iterations. Best cost= 0.203781
Partitioned System Processor clock = 1 MHz

Module	HW Area	SW Size	HW Delay	SW Delay	Implement
DEBOUNCE	-	267	-	218	SW
DEBOUNCE	-	267	-	218	SW
DEBOUNCE	-	267	-	218	SW
ODO_COUNT_PULSES	-	405	-	301	SW
ALARM_COMPARE	33872	-	7	-	HW
ALARM_COMPARE	33872	-	7	-	HW
HW_LATCH	20416	-	4	-	HW
HW_LATCH	20416	-	4	-	HW
HW_LATCH	20416	-	4	-	HW
belt_control	218544	720	23	501	H-S
engine_speed	1140976	3671	117	4433	H-S
wheel_speed	679760	5476	70	8421	H-S
frc	586960	-	62	-	HW
oc_self	170288	-	17	-	HW
ic	152656	-	15	-	HW
pwmfrc	575360	-	51	-	HW
Totals	3653536	11073	381	14310	
Constraints	4000000	8000	10000	10000	
Difference	-346460	3073	-9619	4310	

Table E.6: Partition data for net dac_demo (Hw_Area constraint = 4000000).

BIBLIOGRAPHY

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

2. D. Gajski, F. Vahid, S. Narayan and J. Gong, *Specification and Design of Embedded Systems*, Prentice-Hall, 1994.

3 D. Gajski, F. Vahid and S. Narayan, A system design methodology: Executable specification refinement, European Conference on Design Automation, February 1994.

4. J. Rozenblit, K. Buchenrieder, *Codesign: Computer Aided Software/Hardware Engineering*, IEEE Press @ 1995.

5. C.U. Smith, F.A. Geoffrey and J.L. Cuadrado, "An Architecture design and Assessment system for Software/Hardware Co-design," Proceedings of the 22nd Design Automation Conference, pp. 417-424 June 1985.

6. G. Berry, A quick Guide to Esterél Version 5.10, release 1.1, [Online] Available berry@cma.inria.fr 14 February1998

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

8. R.K. Gupta and G. De Micheli, Hardware-Software Co-synthesis for Digital Systems, IEEE Design & Test of Computers, pp. 29-41, September 1993.

9. R.K. Gupta, C. Coelho and G. De Micheli, Synthesis and simulation of digital systems containing interacting hardware and software components, Proceedings of the 29th Design Automation Conference, pp. 225-230, June 1992.

10. R. Ernst, J. Henkel and T. Benner, Hardware/Software Co-synthesis of microcontrollers, Design and Test of Computers, pp. 64-75, December, 1992.

11. D. Herrmann, J. Henkel and R. Ernst, W. YE, N. Serafimov and G. Glawe, COSYMA: A Software-Oriented Approach to Hardware-Software Co-design, The Journal of Computer and Software Engineering, Vol. 2, No.3 pp. 293-314, 1994.

12. J. Madsen, J. Grode, P.V. Knudsen, M.E. Petersen and A. Haxthausen, "LYCOS: The Lyngby Co-Synthesis System," Design Automation of Embedded Systems, vol. 2, no.2, March 1997 [Online] Available http://www.it.dtu.dk/~lycos/publications.html 24 February 1998.

13. P.V. Knudsen and J. Madsen, Aspects of System Modeling in HW/SW Partitioning, Proceeding: 7th IEEE International Workshop on Rapid Prototyping RSP'96 [Online] Available http://www.it.dtu.dk/~lycos/publications.html 3 March 1998.

14. P.V. Knudsen and J. Madsen, PACE: A Dynamic Programming Algorithm for Hardware/Software Partitioning, Proceedings of 4th International Workshop on Hardware/Software Codesign, Codes/CASHE'96 [Online] Available http://www.it.dtu.dk/~lycos/publications.html 3 March 1998.

15. P.V. Knudsen, Fine Grain Partitioning in Co-design, [Online] Available http://www.it.dtu.dk/~lycos/publications.html 3 March 1998.

16. R. Miller H. Carter K. Davis, Hardware/Software CoSynthesis: Multiple Constraint Satisfaction and Component Retrieval, [Online] Available email: ahmadhab@pilot.msu.edu from rmiller@quest.ece.uc.edu, April 27, 1998.

17. J. Buck, S. Ha E.A. Lee and D.G. Messerschmitt, Ptolemy: A framework for simulating and prototyping heterogeneous systems, Int. Journal of Computer Simulation, special issue on Simulation Software Development," vol. 4, pp. 155-182, April 1994. [Online] Available ftp://ptolemy.eecs.berkeley.edu/pub/ptolemy/www/Ptolemy.html 4 May 1998

18. A. Kalavade and E. Lee, A global critically/local phase driven algorithms for the constrained hardware/software partitioning problem, The 3rd Int'l Workshop on Hardware/Software Codesign, pp. 42-48, September 1994.

19. A. Kalavade and E. Lee, A Hardware-Software Co-design Methodology for DSP Applications, IEEE Design & Test of Computers, pp. 16-28, September, 1993.

20. S. Antoniazzi, A. Balboni, W. Fornaciari and D. Sciuto, The Role of VHDL with in the TOSCA hardware/software codesign framework, Proceedings of the European Design Automation Conference, pp. 612-617, 1994.

21. D. Herel, Statecharts: A visual Formalism for Complex Systems, Science of Computer Programming. 8. North-Holland, 1987.

22. F. Vahid, A specification-level intermediate format for system design, Technical Report CS-94-06, Dept of Computer Science, University of California, Riverside, September 1994.

23. F. Vahid and D. Gajski, Clustering for Improved system-level functional partitioning, 8th International Symposium on System Synthesis, September 1995.

24. A. Jantsch, P. Ellervee, J. Oberg, A. Hermani and H. Tenhunen, A case study on hardware/software partitioning, Proceedings of the IEEE Workshop on FPGAs for custom computing machines, April, 1994.

25. A. Jantsch, P. Ellervee, J. Oberg, A. Hermani and H. Tenhunen, Hardware/Software partitioning and minimizing memory interface traffic, Proceedings of the European Design Automation Conference, pp. 226-231, 1994.

26. T. H. Corma, C.E. Leiserson and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1992

27. E. Barros and W. Rosentiel, A clustering approach to support hardware/software partitioning, in J. Rozenblit, K. Buchenrieder, editors, *Codesign* pp. 230-262, IEEE Press @ 1995.

28. T. Ismail., K, O'brien, and A. Jerraya, Interactive system-level partitioning with PARTIF, in European Conference on Design Automation, February, 1994

29. R. Camposano and J. Willberg, Embedded system design, Design Automation for Embedded Systems, 1(1-2): 5-50, January 1996.

30. P. Chou, R. Ortega, and G. Borriello, The Chinook Hardware-Software Co-Synthesis System, Technical Report 95-03-04 Dept. of CS&E University of Washington Seattle, WA, March 1994.

31. The VIS Group, VIS : A system for Verification and Synthesis, Proceedings of the 8th International Conference on Computer Aided Verification, pp 428-432, Springer Lecture Notes in Computer Science, #1102, Edited by R. Alur and T. Henzinger, New Brunswick, NJ, July 1996

32. E.M. Sentovicn K.J. Singh, L. Lavagno, SIS: A system for Sequential Circuit Synthesis," Memorandum No. UCB/ERL M92/4 May 1992, [Online] Available http://www-cad.eecs.berkeley.edu/Respep/Research/hsc/abstract.html 4 May 1998.

33. F. Balarin, M. Chiodo, *POLIS version 0.3 User's Manual*, [Online] Available http://www-cad.eecs.berkeley.edu:80/Software/software.html December, 1997.

34. B.W. Kernighan and S.Lin, An efficient heuristic procedure for partitioning graphs, Bell System Journal, Vol. 49, pp. 291-307, February 1970.

35. C.M. Fiduccia and R.M. Mattheyses, A linear-time heuristic for improving network partitions," Proceedings of the Design Automation Conference, 1982.

36. B. Krishnamurthy, An improved min-cut algorithm for partitioning VLSI networks, IEEE Transactions on Computers May 1984.

37. S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi, Optimization by simulated annealing, Science, 220 pp. 671-680, 1983.

38. P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli, Performance Guided System Level HW SW Partitioning with Iterative Improvement Heuristics, Technical Report, Linköping University, Dept. of Computer and Information Science, No. 26, 1995 [Online] Available http://www.ida.liu.se 3 April, 1998.

39. P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli, System Level HW/SW Partitioning Based on Simulated Annealing and Tabu Search, Kluwer Journal on Design Automation for Embedded Systems, vol. 2, no. 1, pp. 5-32, January, 1997,