INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

A hardware/software partitioning framework for the codesign of digital systems

Houria Oudghiri
B.sc, National institute of computer science, (Algeria) 1988
M.sc, National institute of Computer science, (Algeria) 1991

Department of electrical engineering
McGill University, Montréal
May1999

"A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfilment of the requirements for the degree of Philosophae Doctor"

ABSTRACT

The thesis provides a new approach to the codesign of digital systems. Complex systems tend to have mixed hardware-software components and are often subject to severe cost, performance, and design-time constraints. Our approach is to codesign these systems. The codesign approach allows the hardware and software designs to be tightly coupled throughout the design process. We focus on three key problems of system modeling, system analysis and system hardware-software partitioning.

A key contribution of the thesis is to use hierarchy as it is available in modeling tools to handle complex system models but also during the codesign process to provide different modeling alternatives for the same input system. This use of the hierarchy, during the codesign process, allows the extension of the design space explored to find the final implementation.

Another key contribution is the analysis of the hierarchical model for any target architecture using automatic estimators and scheduling algorithms. The performance estimation provides information on the performance of each block in the model when it is run on software resources or implemented on hardware. The scheduling algorithm is used to identify critical paths in the system execution and also to identify data dependency between the different model components.

The last main contribution is the hardware-software partitioning algorithm. The proposed algorithm is based on a weighted graph partitioning heuristic. The partitioning algo-

rithm performs into two nested loops. The outer loop is used to select the blocks which take a long time to execute in order to move them to hardware. This selection has the main objective of accelerating the system execution. The inner loop is used to select the neighbors of the block selected in the outer loop. The neighbors which have the most data dependency are first selected in order to minimize the hardware-software communication cost.

We embody all the above concepts in a framework containing tools for hierarchical modeling, system analysis and partitioning. This new framework provides designers with a list of hardware-software implementations corresponding to the different levels in the system hierarchy. These codesign alternatives are provided with their performance and cost measures to allow the designer selecting the final implementation.

SOMMAIRE

Cette thèse présente une nouvelle approche pour le codesign des systèmes digitaux. Les systèmes digitaux deviennent de plus en plus complexes et ont tendance à contenir des composants hybrides qui peuvent être du logiciel ou du matériel. La conception de tels systèmes est souvent sujette à des contraintes sévères sur le coût, la performance ainsi que sur le temps de conception. On propose, dans cette thèse une approche pour réaliser le codesign de tels systèmes. Cette approche permet de concevoir le logiciel aussi bien que le matériel dans le même processus de conception. Nous nous sommes intéressés, dans cette thèse, surtout à trois problèmes clés dans le domaine du codesign qui sont la modélisation au niveau système, l'analyse de performance ainsi que le problème de partitionnement logiciel-matériel.

L'une des contributions majeures de cette thèse, est l'utilisation de la hiérarchie non seulement pour permettre de modéliser des systèmes très complexes mais aussi lors du processus de codesign afin de fournir différentes alternatives de modélisation pour le même système en entrée. Cette utilisation de la hiérarchie, pendant le processus de codesign, permet d'étendre d'une façon considérable l'espace des solutions qu'on explore afin de trouver la solution finale. En effet, chaque niveau dans la hiérarchie donne un nouveau modèle sur lequel tout le processus de codesign est exécuté afin de trouver une solution logicielle-matérielle. Plusieurs niveaux de hiérarchie vont donc permettre de rechercher une solution pour chaque niveau. Ceci constitue une nette amélioration par rapport aux

travaux classiques où une solution est recherchée pour un seul modèle en entrée.

Une autre contribution concerne l'analyse du modèle hiérarchique pour n'importe quelle architecture cible. Ceci est réalisé en utilisant des outils d'estimation automatique ainsi que des algorithmes d'ordonnancement. L'estimation automatique des performances fournit, d'une façon rapide, des données sur les performances de chaque bloc du modèle lorsque celui-ci est exécuté sur une ressource logicielle ou bien implémenté sur le matériel. Les algorithmes d'ordonnancment servent à identifier les goulots d'étranglement dans l'exécution du système au complet et aussi à déterminer les dépendances en données entre les différents blocs du modèle.

La dernière contribution majeure est l'algorithme de partitionnement logiciel-matériel. L'algorithme proposé est basé sur une heuristique de partitionnement d'un graphe pondéré. Cet algorithme est réalisé en deux boucles imbriquées. La boucle externe sert à sélectionner les noeuds qui sont les plus lents et qui sont déterminants sur la performance finale du système. Cette première sélection sert donc à accélérer l'exécution du système. La boucle interne sert à sélectionner les voisins les plus dépendants du bloc sélectionné par la boucle externe afin de minimiser le coût de la communication entre les partitions logicielle et matérielle.

Toutes ces contributions ont été intégrées dans un environnement de codesign permettant de réaliser une modélisation hiérarchique, une analyse du système en entrée et enfin un partitionnement automatique du système en entrée en deux implémentations, l'une logicielle et l'autre matérielle.

Acknowledgements

I would like to acknowledge my supervisor, Januzs Rajski, for having accepted me as a Ph.D student, supported me and assisted me throughout my doctoral studies.

I would also like to acknowledge my co-supervisor, Bozena Kaminska, for her guidance and assistance throughout my doctoral thesis.

As a member of my supervisory comitte, I acknowledge Prof. Ted Szymanski for his suggestions during the earlier stages of my research work.

I am beholden to all my friends in the VLSI design laboratory at McGill and at the University of Montréal for their assistance. My special thanks to Claude Villeneuve for the pleasant work I had with him.

In addition, I would like to express my thanks to the Algerian Gouvernment for their support during my master and which allows me to continue towards Ph.D.

For financial support, I would like to acknowledge Prof. Yvon Savaria, the director of the PULSE project and all the project partners.

Finally, on a more personal level, I am grateful for my parents and to all my family members back home for their support throughout the course of my studies.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Claim of originality

The author claims originality for the following contributions of the dissertation:

- Chapters 1 through 5 contain reviews in their first sections, although these reviews are original in the sense of providing a new classification of related works according to determinant characteristics in the codesign field.

- Chapter 2 presents a novel codesign flow based on the hierarchy and performance estimation. This flow is reiterated for each level in the hierarchy.

- Chapter 3 develops a new use of hierarchy for models of digital systems described at high-level. The hierarchy is not presented as a new approach for design but is considered in a different way. The majority of modeling tools support hierarchy but only to allow handling complex systems. We exploit this same hierarchy used to simplify the modeling step to provide various input models for the same input system. This use of hierarchy provides an expansion of the modeling space because different input models are available for each level of the hierarchy, but also an expansion of the codesign space because a hardware-software solution may be found for each level of the hierarchy. Our Codesign process consider at each level of the hierarchy a set of blocks with different abstraction when moving through the hierarchy levels.

- Chapter 4 proposes a new analysis methodology for the codesign of digital systems. The analysis is performed in two stages: the performance estimation and the task scheduling. The performance estimation is performed using known automatic estimators but we provide these available estimators with technology files for each resource of our target architecture. The task scheduling is performed to find critical paths and bottlenecks in the system execution. The scheduling step identifies the concurrency between the system tasks in an independent way of the partitioning.

- Chapter 5 proposes a new hardware-software partitioning algorithm based on a dependency graph partitioning. The nodes and edges of the dependency graph are weighted in order to minimize the system execution time and the hardware-software communication overhead. The hardware-software partitioning is performed on the basis of two types of selection, global and neighborhood. The global selection selects the bottlenecks in the system to accelerate the system execution. The neighborhood selection operates on the neighbors of the bottlenecks in order to minimize the hardware-software communication overhead.

- Chapter 6 provides extensive experimental verification on two case studies. For each case study, two lists of hardware-software partitioning alternatives are provided: the timing constraint list and the hierarchy list. The timing constraint list gives the list of hardware-software partitioning alternatives obtained for different timing constraints. The hierarchy list provides a list of hardware-software partitioning alternatives for the same timing constraint but at different levels of the hierarchy.

- Chapter 6 also provides a comparison between the hardware-software alternatives obtained for the same timing constraint to show the necessity of a performance-area tradeoff.

1

INTRODUCTION

System-level design usually involves designing an application specified at a high-level abstraction. A typical design objective is to minimize cost (in terms of area or power) while the performance constraints are usually throughput or latency requirements. The specification at system level is built of basic components called tasks. This specification has two characteristics. First, tasks are at a higher level of abstraction than atomic operations or instructions. This allows for complex applications to be described easily and more naturally. Secondly, there is no commitment to how the system is implemented. Since the specification does not assume a particular architecture, it is possible to generate either a hardware, or a software, or a mixed implementation. This is specially important for the synthesis of complex applications whose cost and performance constraints often demand a mixed hardware-software implementation. For such applications, full-software implementations (program running on a programmable processor) often cannot meet the performance requirements, while custom-hardware solutions (custom ASIC) may increase design and

product costs. It is important therefore, not to commit each task in the application to a particular mapping (hardware or software) or implementation when specifying the application. The appropriate implementation for each task can be selected by a global optimization procedure after the specification stage. The task level of abstraction allows this flexibility. Manual development of a lower-level design specification (such as at the RTL level) is quite intense due to the complexity of the applications. As a result, it is desirable to specify the application at the task level and allow a design tool to generate lower levels of implementation from it.

Such a system-level design approach is now viable due to the maturity of lower level design tools and semiconductor technology. Computer-aided design tools that operate at lower levels of abstraction are quite robust. The next step is to use these CAD tools for system-level design. Also, advances in semiconductor manufacturing have made it possible to fabricate a "system on a chip".

Figure 1.1 summarizes the key issues in system-level design. These are system specification, modeling, partitioning, synthesis, simulation and design-space exploration. A system is first specified in a high-level language as a set of tasks. It is then transformed into a model to catch its functionality details at a high-level of abstraction. Several hardware and software implementation options are usually available for each task in the description. The partitioning process determines an appropriate mapping (hardware or software) for each task. A partitioned application has to be synthesized and simulated within a unified framework that involves the hardware and software components as well as the generated interfaces. The software synthesis of a task is to generate the code for the task on a given processor. Finally, the validation of the final implementation is often performed using co-

simulation tools. The system-level design space is quite large. Typically, the designer needs to explore the possible options, tools, and architectures, choosing either automated tools or manually selecting his/her choices (feedbacks in Figure 1.1). A design-space exploration framework attempts to ease this process.

The most important aspect of system-level design is the multiplicity of design and modeling options available for every task in the specification. Each task can be implemented in several ways in both hardware and software mappings, and can also be modeled as a big black box, a set of subblocks or a set of basic operations. The partitioning problem is to select an appropriate mapping of each task from a given model. In system-level design, there are a number of such tasks and overall design is to be optimized. Clearly, it is not enough to optimize each task independently. For example, if each task were fed to a high-level hardware synthesis tool that optimized for speed, then the overall area of the system might be too large. Hardware-software partitioning is the problem of determining an implementation of each task so that the overall design is optimized.

Once the appropriate implementation for each task has been determined, the hardware-software synthesis problem is that of synthesizing the implementation. Implementations for tasks mapped to hardware or software can be generated by feeding the task descriptions to synthesis tools. The hardware synthesis consists of high-level synthesis [McFarland 90] followed by logic synthesis [Brayton90] and layout synthesis [DeMicheli86]. The software synthesis comprises high-level software synthesis [Pino95], followed by compilation and assembly. System-level design is not a black-box process, but relies considerably on user creativity and interaction. For instance, the user might want to experiment with the design parameters. The design process could get quite anwieldy as the

user experiments with the design methodology. As a result, an infrastructure that supports design space exploration is also a key aspect of the system-level design process. Tools for design space exploration must include performance estimation tools. Estimation tools give quick predictions on the outcome of applying certain synthesis or transformation tools.

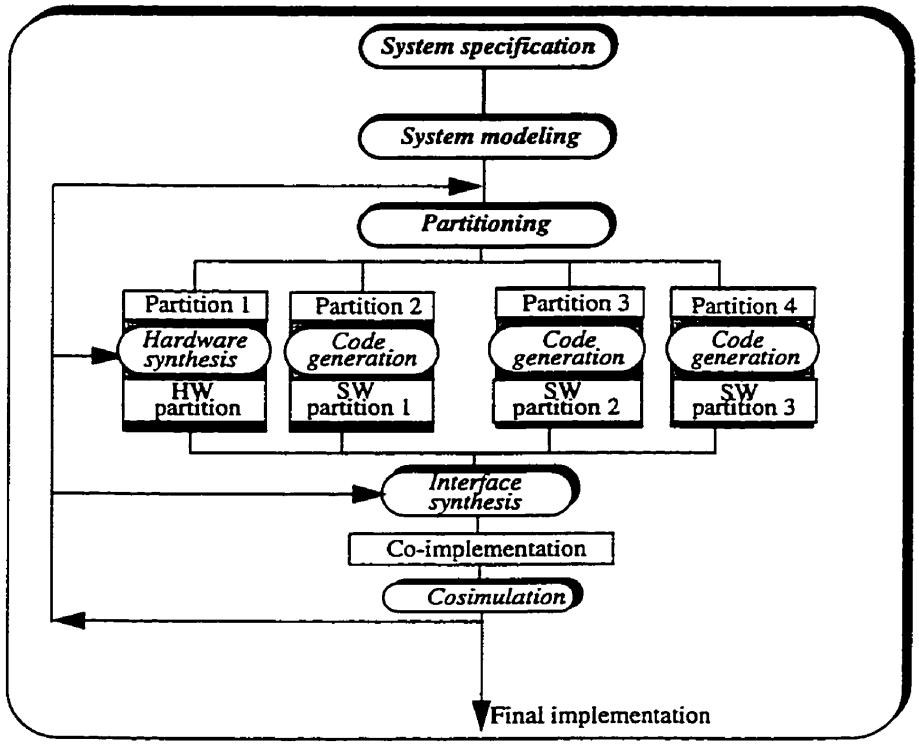


Figure 1.1. The hardware/software codesign process.

Designing systems containing both harware and software components is not a new problem. The traditional design approach has been somewhat hardware first in that the software components are designed after the hardware has been designed and prototyped. This leaves little flexibility in evaluating different design options and hardware-software mappings. With isolated hardware and software design paths, it also becomes difficult to optimize the design as a whole. Such a design approach is especially inadequate when designing systems requiring strict performance and a small design cycle time. The key tenet

in codesign is to avoid isolation between hardware and software designs to proceed in parallel, with feedback and interaction between the two as the design progresses. This is accomplished by developing tools and methodologies that support coupled design of hardware and software through a unified framework. The goal of codesigning the hardware and the software components of a system is to achieve high quality designs with a reduced design time.

Our main objective, in this thesis, is to provide efficient solutions to some relevant codesign problems. The main tasks involved in the codesign process have been presented in the above paragraphs. These are the system modeling, the system parformance estimation and the hardware/software partitioning. The techniques used to perform such tasks are determinant on the final hardware/software implementation. We propose new and original techniques to perform these tasks with the objective of providing more efficiency and better design space exploration. The proposed objectives with a statement of originality are presented below for each one of these tasks.

1. 1. Modeling

The specification at the system level is often modeled as a set of tasks. These tasks are described at the basic operation level in some modeling framework or at the process level (sequence of basic operations) in some others.

Digital systems are becoming more and more complex and this complexity involves the use of new modeling characteristics, as the hierarchy in order to make the modeling of such complex systems tractable. The hierarchy has been proven to be extremely useful when modeling highly complex systems. The majority of modeling tools include the hierarchy in their modeling approach. In this thesis, we consider the hier-

archical model available and we propose a new use of this available hierarchy. The hierarchy is not used only to ease the modeling process but all along the codesign process by considering the different modeling alternatives available in a hierarchical model, each one with a different complexity.

The main objective intended by this use of hierarchy is to have several modeling alternatives and to codesign each one of these alternatives. Many co-implementation alternatives are then provided since a possible hardware/software implementation with its own performance and cost is found for each one of the modeling alternatives. This use of hierarchy is a new and original way to expand the design space exploration which is a key tenet in the codesign process.

1. 2. Performance estimation

In literature, many tools are available for performance estimation on a specific target architecture. The available estimation tools are often tied to a specific architecture and new tools have to be developed if the target architecture is changed. We intend in this thesis to use generic estimation tools that can be tied dynamically to any target architecture. The target architecture resources are first described in a generic way and put into the data base of the estimation tools. Then, the estimators consider any input algorithmic specification and provide performance estimation measures when the input algorithm is run or implemented on the selected resources. The main objective, at this step, is the flexibility to consider many possible target architectures for the same input system specification rather than having one target architecture fixed at the very early stages of the design.

1. 3. Partitioning

Partitioning is the main task in the codesign process since the final hardware/software implementation is tightly dependent on the techniques used to select and map the tasks to hardware or to software. Many automatic partitioning algorithms have been proposed during the past years but without convincing codesigners to adopt the automatic solution. Codesigners have not enough confidence in an automatic solution for partitioning because they want to make the main decisions by themselves.

We propose a partitioning solution which is between the complete automated and manual partitioning. We propose an automatic partitioning technique which attempts to try several task mappings and then provides the desiger with the obtained alternatives and their performance estimations. The automatic partitioning algorithm is in charge of the complex and iterative search process while the decision of selecting the final implementation is left to the designer.

This thesis presents in detail each one of the original proposals presented above. Chapter 2 briefly presents an overview of related work in system design and computer-aided techniques developed for system synthesis, and the general scheme of the proposed codesign framework. The organization of the rest of thesis can be explained by relating it to the organization of our codesign CAD framework. The input to our system is an algorithmic description of the system functionality. The description is compiled into a hierarchical system graph model based on dataflow graphs whose features and properties are described in Chapter 3. Chapter 4 describes performance estimation and analysis techniques used to predict the performances of possible hardware/software implementations. In chapter 5, we define the problem of system partitioning and present an automatic approach to partitioning

digital systems for hardware/software cosynthesis. Chapter 6 describes case studies considered for hardware/software codesign and the results obtained after applying our codesign approach. Chapter 7 presents conclusions where the objectives defined above are recalled in the context of obtained results and finally the directions for future research are presented.

2

CODESIGN METHODOLOGY

While the CAD tools for the design of individual application specific ICs, or ASICs, are in a fairly mature state, in some application domains it is even possible to completely synthesize an ASIC from a high-level behavioral description in a matter of hours, CAD methodology for dedicated systems have not kept pace. Real-life systems are composed of a mix of software running on general purpose programmable hardware, ASICs and other dedicated hardware, electromechanical components, and mechanical interconnect and packaging, and a unified approach that encompasses the various software, hardware, and mechanical aspects of system design is desirable.

The increased functional and implementation complexity and heterogeneity of systems mean that one cannot just scale and apply chip design techniques to the design of a system. For example, it is difficult and unnatural to represent and simulate an entire system according to a single computation model as is usually done in the case of chips. This simple

architecture model of a single controller and a datapath as used for chips is inadequate for most board level systems. Clock synchronous hardware implementation is usually adequate for chip but not for an entire system. Software issues are absent in a chip design as ASICs mostly have hardwired controllers. In short, system level design is more than just a scaled version of chip design.

Since system-level design oversees high-level synthesis, logic synthesis, etc..., decisions made at the system level impact all the layers below it. In other words, if the objective in system-level design is to come up with the "best" system implementation, there are a large number of design options. The system-level designer is faced with the questions of selecting the best design options.

The general design methodology is shown in Figure 2.1. The inputs to the codesign tool include the design specification and the design constraints, and the output is an implementation for the system. This chapter describes the various components of the standard codesign tool. In section 2.2, the codesign problem is stated with all the involved steps. In section 2.3, a typical codesign flow is described and the various tools required in the codesign process are outlined. In section 2.4, we present a list of the most known frameworks related to the codesign of digital systems. In section 2.5, we outline the proposed methodology with the advantages it offers compared to the related works.

2. 1. Motivations for hardware/software codesign

Most digital functions can be implemented by software programs. The major reason for building dedicated application-specific hardware (ASICs) is the satisfaction of performance constraints. These performance constraints can be on the overall time (latency) to

perform a given task, or more specifically on the timing to perform a subtask and/or on the ability to sustain specified input/ouptut data rates over multiple executions of the system model. The hardware performance depends on the results of operation scheduling and the performance characteristics of individual hardware resources. The software performance depends on the number of instructions the processor must execute and the cycles-per-instruction metric of the processor. In general, application-specific hardware implementations are faster since the underlying hardware is optimized for the specific set of tasks. However, some parts of the description of an ASIC machine may be well suited to a commonly available reprogrammable processor while others may take too long to execute.

2. 2. The codesign problem statement

Hardware software codesign is a complex problem that involves the following subproblems:

1. **Modeling the system** functionality and performance constraints

   System modeling refers to the specification problem of capturing important aspects of system functionality and constraints to facilitate design implementation and evaluation. Among the important issues relevant to mixed system designs are:

   - *Explicit or implicit concurrency in the specification.*
   - *Model of communication- shared memory versus message passing-.*
   - *Control flow specification or scheduling information.*

   When the concurrency is implicit, the concurrency information is obtained by performing a dependency analysis for which the complexity depends on the sys-

tem model used.

2. **Choosing the granularity** of the hardware/software partitions. The system functionality can be handled either at the functional abstraction level, where a certain set of operations is partitioned, or at the process communication level where a system model composed of interacting process models is mapped onto either hardware or software. The former attempts fine-grain partitioning while the later attempts a high-level library binding through coarse-grain partitioning. Each choice has advantages and disadvantages. The first one allows efficient and refined analysis and transformations but at a higher computing complexity. The second one reduces the processing complexity with a loss in design efficiency since blocks or processes are considered instead of basic operations. This reduces the partitioning alternatives compared to those available at the basic operation level.

3. **Determining the feasible partitions** of application-specific and re-programmable components. The blocks in the system functionality are assigned to hardware or to software in such a way that the resulting implementation satisfies the functionality requirements and the constraints. This is a difficult problem because good system-level cost metrics, accurate techniques for estimating the cost, and techniques for reliable performance estimation of system-level hardware and software are required.

4. Specifying and **synthesizing the hardware, the software** and the hardware/software interface. Each one of the determined partitions is synthesized to obtain the final implementation using the automatic tools available on the mar-

ket. This synthesis is also done under performance and cost constraints.Hardware and software synthesis tools use different specification languages (for example, VHDL for hardware synthesis and C for software synthesis) and thus the resulting partitions must be translated to the specific description language in order to be automatically synthesized.

The next chapters focus on the various aspects of the codesign process with the proposition made for each one of these steps. In the next section, we follow the codesign problem presentation by showing a typical codesign flow in section 2.3. In section 2.4, a list of related works in codesign are classified according to their major characteristics. In section 2.5, our proposed codesign framework is presented.

2. 3. A typical codesign flow and tools for codesign

A typical codesign flow is shown in Figure 2.1. A task-level specification is transformed into the final implementation by a sequence of tools. The final implementation consists of custom and commodity programmable hardware components and the software running on the programmable components. The design constraints include the desired throughput and the architectural model (maximum allowable hardware area, memory size, communication model). The design flow describes the sequences of steps that operate on the design data to generate the final implementation. The components of the codesign flow include tools for modeling, analysis, partitioning, synthesis and simulation.

2.3.1 System modeling

Models are often needed in order to avoid creating detailed implementations. A mod-

el of a system helps to estimate relevant properties, like area and delay, of its implementations. The model is built from a set of interconnected basic elements. The model complexity depends on the basic element complexity. Indeed, if the basic element is an arithmetic or logical operation, the model may be very complex because the majority of systems are generally constituted of a large set of these simple operations. If the basic element is a task or a process (a set of interconnected basic operations), the model complexity may be reduced considerably. This means that the model complexity depends on the model granularity. The more the granularity is fine the more the model is complex and hard to handle. We propose a modeling technique with a variable granularity that allows the use of a simple model as long as the constraints are satisfied. We handle complex models only when the constraints cannot be satisfied with simple models. In chapter 3, we show the advantages of such a modeling technique.

2.3.2 System analysis

The system analysis involves two main tasks. These are the performance estimation and the scheduling of the model tasks. The performance estimation provides estimates of the implementation metrics (area and execution time requirements) for each of the tasks in the specification when different hardware and software realizations are considered. The scheduling determines the possible execution flows of the different tasks based on their data dependencies. The estimates and task scheduling guide the partitioning algorithm during the task mapping. Details of the estimation and scheduling tools are discussed in chapter 4.

2.3.3 Partitioning

Once the estimates of the area and execution time and the scheduling of the functional

blocks have been performed, the next step in the codesign flow is the partitioning. The goal of the partitioning is to determine, for each task the mapping to hardware or to software while optimizing the overall design. The partitioning is a non-trivial problem. Consider a task-level specification, typically in the order of 50 to 100 tasks or nodes. Each task can be mapped to either hardware or software. Furthermore, within a given mapping, a task can be implemented in one of several options. Suppose there are 5 design options. Thus, there are $(2\times5)^{100}$ design options in the worst case! Although a designer may have a preferred implementation for some p nodes, there are still a large number of design alternatives with respect to the remaining nodes $(2\times5)^{100-p}$. Determining the best design option for these remaining nodes is, in fact, a constrained optimization problem. In section 5.3, the partitioning algorithm that uses a graph partitioning heuristic is presented. The proposed heuristic is very efficient with the complexity $O(N^2)$, where N is the number of tasks in the design specification.

2.3.4 Cosynthesis

Once the application is partiotioned into hardware and software, the individual hardware, software, and interface components are synthesized. The particular synthesis tool used depends on the desired technology. The VHDL code can be generated and passed through synopsys tools to generate the hardware implementation. Similarly, different software synthesis strategies can be used; for instance, the C description may be generated. The interface generation depends on the desired architectural model. There are several published approaches to the problem of synthesizing hardware and software from high-level specification.

2.3.5 Cosimulation

Once the hardware, software, and the interface components are synthesized, a validation/verification step is required. This is often a cosimulation of the different components, hardware, programmmable hardware and software. Examples of such cosimulation tools are Ptolemy (Stanford University) and logic modeling [Synopsys95].

We propose a codesign tool as a framework for system-level design. The tool is a unified platform consisting of tools for modeling, analyzing and partitioning of mixed hardware-software systems. Simulation and synthesis tools may be used from those available on the market.

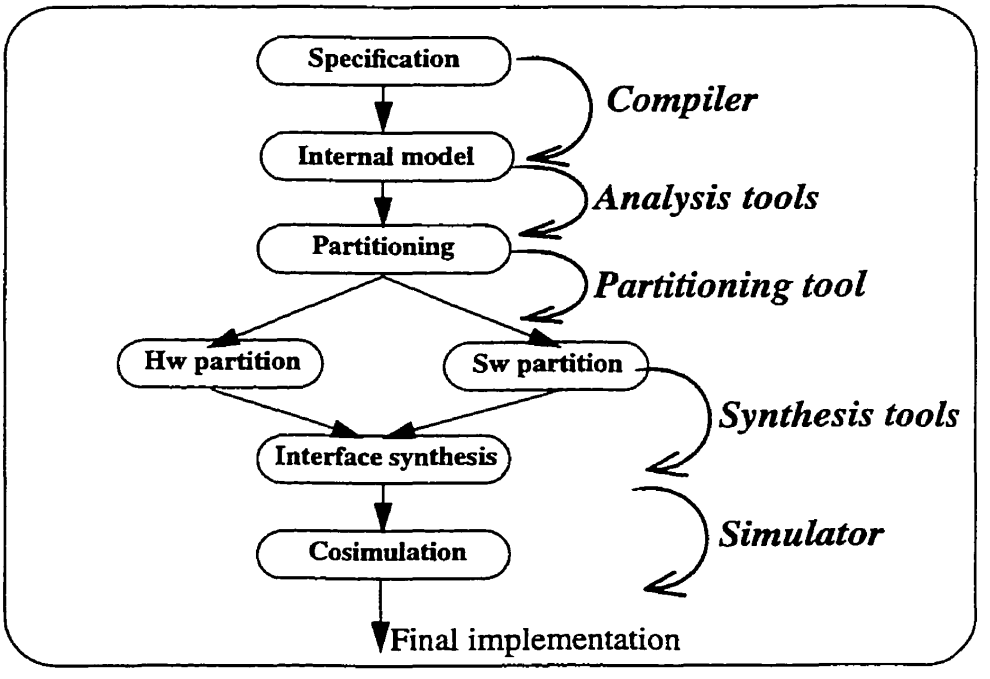


Figure 2.1. The typical codesign flow process.

2. 4. Related work

Work in the computer-aided approach to system design is relatively new. Recent in-

terest in system synthesis has been stimulated by the success and maturity of chip-level synthesis tools and the emergence of synthesis approaches at levels of abstraction higher than logic-level and RTL-level circuit description. In the follow sections, we briefly review related work that is directly relevant to the codesign problem.

Table 2.1 gives some pointers to fields in system design. A system design typically requires to address all these topics. But the relevance of each topic depends very much on the considered application domain. For example in avionics, the system security and a validated design are of the utmost importance, whereas in digital signal processing the high performance power and the chip complexity are the main design topics.

Achieving performance requires a careful tailoring of the system structure to the requirements imposed by the algorithms. For complex applications, this makes an analysis of the programs necessary to determining the requirements. Analysis tools are required to support this in a an efficient way. Table 2.1 gives the list of some codesign works with the main codesign problems each of them has addressed. In the next paragraphs, these problems are listed with the solutions proposed in literature.

2.4.1 Specification and verification

The input system is specified into a formal language that allows an easy and formal specification with functionality checking and verification facility. This specification is determinant for the next steps in the codesign flow. The more efficient the specification language is, the easier are the analysis and partitioning steps. Some codesign frameworks in literature are dedicated mainly for a strong and efficient formal language for specification and verification.

One of these works is CODES [Buchenrieder92] which stands for COncurrent DES-

sign. The system is specified as a set of communicating parallel random access machines (PRAMs). The design process is modeled using Petri Nets which are a well known strong formal modeling technique. The input specification is simulated using StateMate or SDL tools widely used for the communication protocol simulation and verification.

Another work concentrating on the formal specification language is COSMOS [BenIsmail94b]. The main objective of this codesign framework is to provide an intermediate format SOLAR [O'Brien95] which can model system-level constructs in a synthesis-oriented manner. The philosophy is to allow the designer to use customized languages to describe different aspects of his system and to provide translators to this intermediate form. SOLAR is able to model and synthesize a wide range of communication schemes between concurrent processes.

Another work related to specification and modeling is the SpecCharts formal language [Gajski93] developped by Gajski et al. at the university of California at Irvine. This formal language is an attempt to build a bridge between hardware and software specifications. The system's behavior is conceptualized as a hierarchy of sequential and concurrent behaviors. The hierarchical structure is described in the language SpecCharts while the basic behaviors are described in VHDL language. This has been a very elegant solution for modeling both hardware and software but raises the question of the need for a standard.

The last work in this category of frameworks concentrating on specification and modeling is the tool developped by Chiodo et al. [Chiodo94]. This tool uses a unified formal specification model called network of Codesign Finite State Machines (CFSMs) to describe control-dominant systems characterized by relatively low algorithmic complexity such as embedded controllers. The modeling tool is also used to describe techniques to realize a

CFSM as either a hardware or software FSM, to generate the interfaces between the resulting hardware and/or software FSMs.

2.4.2 Co-simulation

Simulation is a very uselful facility at the starting of the design to check the functionality and at the end of the design to verify the design implementation. The simulation is often used as an alternative to formal verification for complex systems.

The main framework known for system-level simulation is PTOLEMY [Buck94a]. PTOLEMY is a framework for the simulation, prototyping and software synthesis of digital signal processing systems. PTOLEMY's strength is its unified framework that primarily addresses the simulation of specifications as a set of heteregeneous computation models constituting a DSP system. Examples of supported models are synchronous data flow (SDF), dynamic data flow (DDF), discrete events (DE) and signal-level digital hardware (Thor).

In addition to a simulation framework, PTOLEMY also provides code generation abilities for its synchronous data flow (SDF) model using DSP processors, C, Silage or VHDL languages as targets. The strength in heterogeneity by use of diverse computation models in Ptolemy comes at the loss of an analytical handle on system properties when the input system is not from a DSP application..This is particularly true for system specifications that feature a significant amount of control flow. Nevertheless, Ptolemy represents an important step towards simulation of complex systems.

2.4.3 System analysis

The codesign process requires a profiling and a performance analysis of the input

specification in order to find a final implementation for the different system blocks such that the overall system performance is optimized. This analysis phase is determinant for an efficient hardware/software partitioning. Some codesign frameworks emphasize on this apsect as SpecSyn [Vahid92] and CASTLE [Theinbinger94].

SpecSyn is an automatic codesign framework which uses the SpecCharts language for the input specification. The hardware and software performance and area estimation may be determined for a wide range of resources described into technology files. The technology file library contains many standard processors and some custom ASIC types.

CASTLE is a complete cosynthesis environment in which data flow graph (DFG) representation is derived from an array of specification formats as Verilog, VHDL and C. This environment has very efficient profiling and analysis procedures which provide the designer with a wide range of information allowing him to make manually the appropriate mapping of the model's operations.

2.4.4 Partitioning

Several partitioning algorithms have been proposed in literature and are presented with more details in chapter 5. Here, we briefly review some of the codesign frameworks which are mainly intended for partitioning.

COSYMA [Henkel93], CO-SYnthesis for eMmbedded Architectures, performs partitioning of operations at the basic block level with the goal of providing speedup in program execution time using hardware co-processors. Input to COSYMA consists of an annoted C-program. This input is compiled into a set of basic blocks and corresponding Directed Acyclic Graph or DAG-based syntax graphs. The syntax graphs are helpful in performing dataflow analysis for definition and use of variables that helps in estimating

communication overheads accross hardware and software. The syntax graphs are partitioned using simulated annealing algorithm under a cost function. This process is repeated using exact performance parameters from synthesis results for a given partition.

The chief advantage of this approach is the ability to utilize advanced software structures that result in enlarging the complexity of system designs. However, selective hardware extraction on potential speedups makes this scheme relatively limited in exploiting potential use of hardware components. Further, the assumption that hardware and software components execute in an interleaved manner (and not concurrently) results in a system that under-utilizes its resources.

Another framework based on the language UNITY for the specification of concurrent systems [Barros94] uses clustering for partitioning. The partitioning scheme associated with it classifies UNITY assignments according to a set of five attributes which identify the degree of data dependency and parallelism between assignments. Associated with each of these attributes is a set of implementation alternatives. A two-stage clustering algorithm then selects assignments to be grouped according to similarity of implementation alternatives, data dependencies, resource sharing and performance. The clustered assignments are scheduled for a given target architecture. Finally, an interface graph is constructed based on clustering results. This process is then reiterated based on satisfaction of design constraints.

The last framework presented here is VULCAN II [Gupta93]. VULCAN II addresses the problem using I/O data rate constraints to partition a description in Hardware C such that a maximal set of operations is implemented as software running on a microprocessor while the remaining operations are mapped to ASICs in a common-memory shared-bus architecture. The partitioning algorithm is greedy and moves non-deterministic delay opera-

tions to software in order to meet hardware cost constraints while satisfying performance requirements.

2.4.5 Cosynthesis and prototyping

Once a system has been analyzed and partitioned, the different resulting partitions need to be implemented to obtain the final concurrent hardware/software implementation. This is performed by a synthesis phase of each one of the partitions using two types of techniques, standard synthesis tools or by protototyping each one of these partitions on an existing architecture. Academical examples for the two types are presented below.

From the first class, we present Chinook tool for the cosynthesis of real-time reactive embedded systems [Chou95]. The Chinook system consists of six tools. The first one is a front-end parser of system descriptions in annoted Verilog. The second one is the processor/ device library containing detailed generic specification of the processor device interfaces as well as timing schemas for software run-time estimation. The third tool is the device/ driver synthesizer that compiles the timing diagrams and Verilog devices into a customized code for the given processor. The fourth tool is the interface synthesizer which has the role of allocating I/O resources to connect a processor to the desired peripheral devices and customizing the access routines accordingly. The fifth tool is the communication synthesizer in charge of generating the hardware and software needed from inter-processor communication. The last tool is the scheduler which generates C code to meet real-time constraints in software with all resources allocated.

The chief advantage of this approach is its efficiency in building suitable input/output interfaces for controlling external devices.

The second class is based on prototyping. A typical example is the SIERA tool

[Srivastava95]. SIERA is a framework for rapid prototyping of systems that span across chip and multiprocessor boards in hardware as well as device drivers and operating system kernels in software. This work utilizes chip-level synthesis tools and DSP code synthesis tools to present a framework for performing both activities. A system is specified as a network of concurrent sequential processes in VHDL. The communication between processes is by means of queues. This specification is manually mapped into an architecture template. The main strength of this methodology lies in management of system complexity by using modularity and reusability afforded by existing libraries.

Other codesign aspects have been addressed in literature too. One of them is the code generation for the software synthesis. Typically, flexible processors are used for software implementation. Another codesign interest is related to case studies on specific examples. Applying design automation to real-life problems provides insight into the complexity and requirements demanded by these systems.

The related work review, presented above, is not exhaustive but merely representative of the contemporary work by examining important CAD frameworks for codesign.

Table 2.1: Fields of system design with related work

Topics	References
Specification & verification	**CODES** [Buchenrieder92], **COSMOS** [BenIsmail 94b], **SpecCharts** [Gajski93]
Simulation	**PTOLEMY** [Buck94b], Insulin [Sutarwala93]
Analysis	**ADAM** [Jain92], **SPECSYN** [Vahid92], **CASTLE** [Theibinger94]
Partitioning	**COSYMA** [Henkel94], **VULCAN II**[Gupta93], **UNITY**[Barros93]
Cosynthesis and prototyping	**Chinook** [Chou95], **SIERA**[Srivastava95]
Case studies and Code generation	**JPEG** [Gupta94c], Powertrain [Hu94], Video[Wilberga], **MPEG**[Wilberg96] **CAPSYS** [Auguin94], **CodeSyn** [Lien94]

2. 5. The proposed codesign framework

Our proposed methodology is presented according to the codesign problem statement already presented in section 2.1. Figure 2.2 shows the proposed codesign flow. The input specification is described in any HDL or programming language where a certain hierarchy is inherent. This description is then manually translated into a hierarchical model based on data flow graphs. The hierarchical model can also be taken from any modeling tool available on the market. Each element or task in the model is a node in a graph, this node may be a simple basic operation or may be decomposed into a subgraph where sub-nodes may be basic operations or other data flow graphs. Edges between nodes correspond to data dependencies between operations or tasks. The hierarchy of the input system allows the use of a variable granularity rather than a fixed one at the level of basic operations or at the level of tasks.

Figure 2.3(b) shows the different modeling views provided by the hierarchy for the FIR filter example described in Figure 2.3(a).

The FIR behavior may be built of one black box with all the inputs and outputs, or of three blocks where the outer loop (the first For loop) is decomposed, or five blocks where the two loops are decomposed or finally of seven blocks corresponding to the assignment, comparison, multiplication and addition basic operations.

This model is then estimated in order to determine for each node in the model the performance and area metrics when implemented into hardware or software using the Specsyn estimators. We provide the technology files of our specific resources to these estimators

[Gong95]. The technology files describe the target hardware and software.

The next step is the partitioning. A level of hierarchy or a given granularity is first selected. This results in a set of tasks corresponding to the system model at the chosen granularity. These tasks are then assigned to hardware or to software while minimizing a cost function and satifying a performance constraint. Figure 2.3(c) shows the number of possible hardware/software partitionings for each one of the possible models for the FIR filter. The more the number of blocks increases, the more the number of possible alternatives increases too and the more the partitioning algorithm becomes complex. We will show in the next chapters how the optimal modeling level may be found to satisfy the performance constraints while simplifying the partitioning algorithm.

Once a hardware or a software implementation has been determined for each element, the hardware and software synthesis are performed according to the processor on which the software will be run and to the hardware technology targeted for hardware. The interface between the two partitions is also synthesized. The final implementation is then validated using a cosimulation environement.

One of the main contributions in the proposed codesign framework, shown in Figure 2.2, is the possibility to choose different granularities. The tool starts at the simplest model of the input system by fixing a coarse-grain modeling. The model is built of a small number of components and the partitioning algorithm complexity is of course reduced. If the constraints are not satisfied using this modeling level, the tool may change the granularity towards fine-grain modeling. The number of components increases but the design exploration may be more efficient. We will show later in chapter 6 how the variation of the granularity allows a very efficient design space exploration.

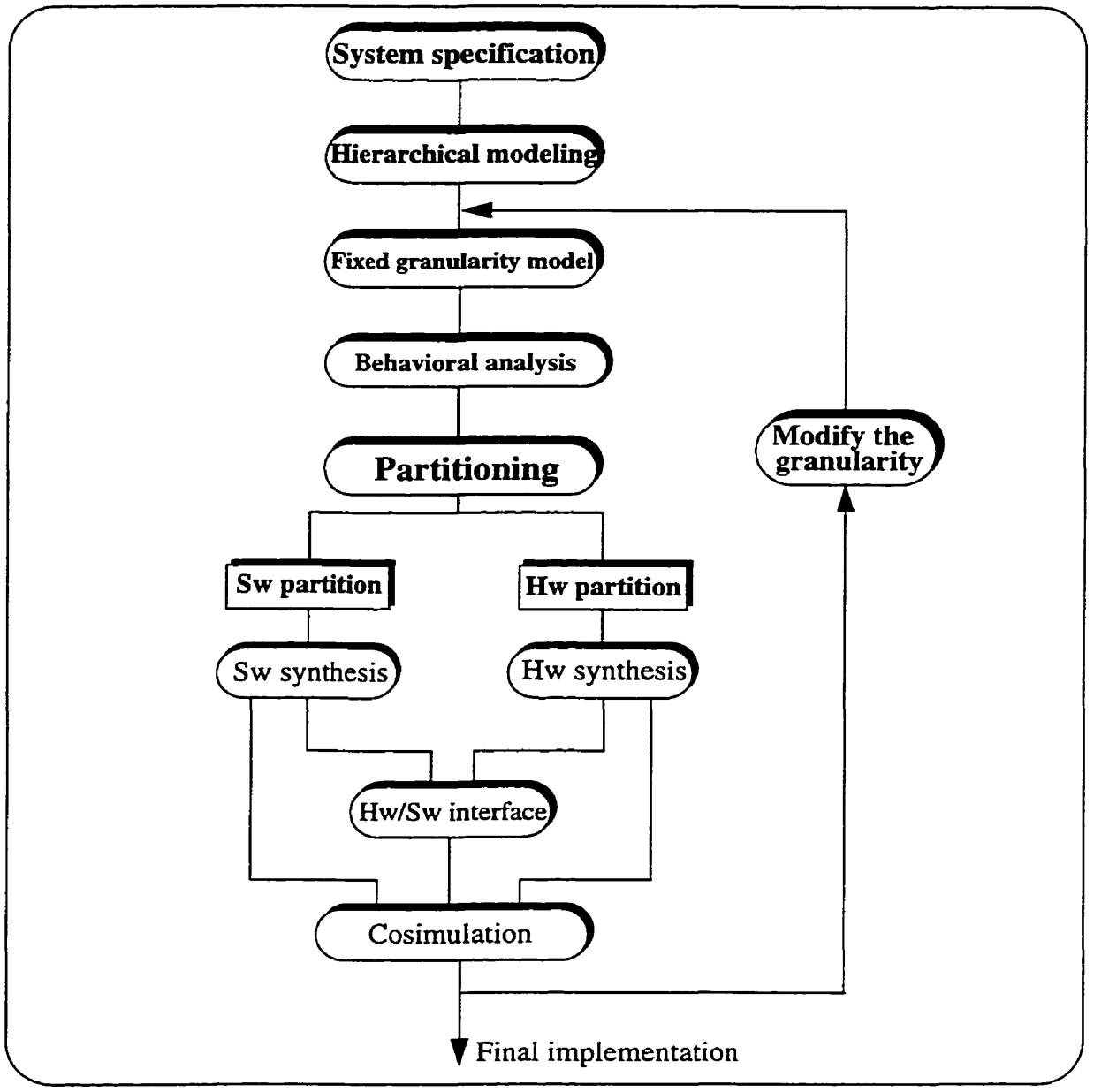
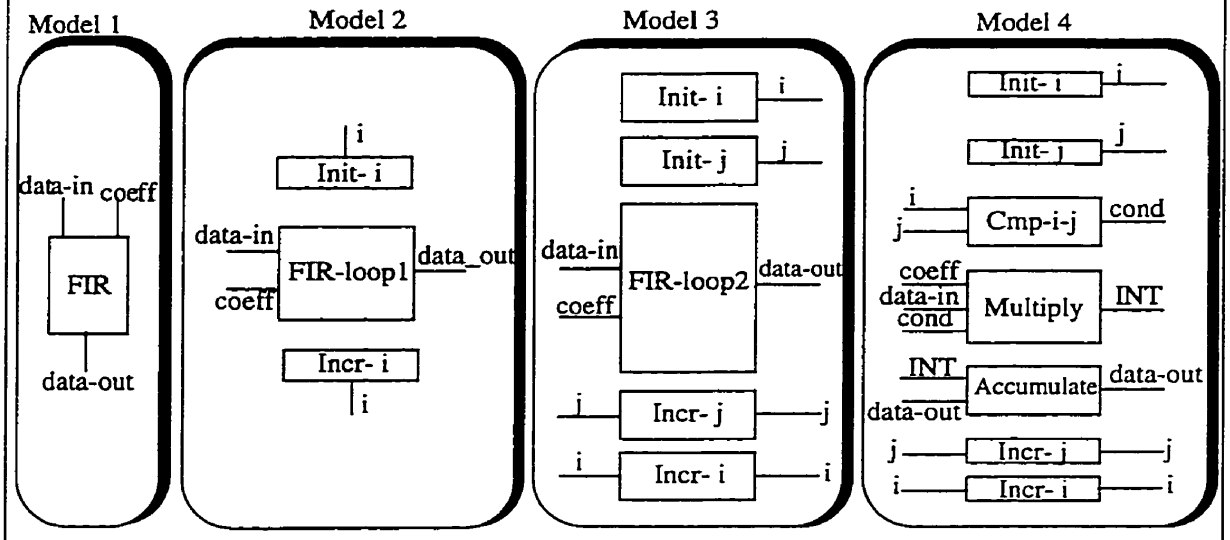


Figure 2.2. The proposed codesign flow.

```
FIR-filter()
begin
 For i = 1 to N
 do
   For j = 1 to M
   do
     if (i > j)
      then
        data-out[i] = data-in[i-j]*coeff[j] + data-out[i];
      end if;
    endo
 endo
end FIR-filter;
```

(a). High-level description of the FIR filter.



(b). The modeling level alternatives for the FIR filter example.

Model	*Model complexity*	*Possible hardware/ software alternatives*
Model 1	1 block	2
Model 2	3 blocks	8
Model 3	5 blocks	32
Model 4	7 blocks	128

(c). The codesign alternatives for the FIR filter example.

Figure 2.3. Modeling and codesign alternatives.

The proposed codesign approach provides new contributions for the following aspects:

1. *The proposed use of hierarchy overcomes many of the limitations of modeling techniques. Our approach supports various models from the simplest to the most complex for the same input system. These modeling alternatives are already available in any hierarchical modeling tool but we propose to take advantage of it during the codesign process. A main feature in codesign is to explore the design space efficiently in order to find the best final implementation. Our modeling technique allows an enlargement of the modeling space for the same input system and thus a large choice for the final implementation.*

2. *A model analysis step is performed to estimate the performance of each block and to determine the critical blocks with all the possible concurrency between them. This analysis step provides very important information that will later guide the partitioning algorithm in order to optimize the overall system performance.*

3. *The partitioning algorithms in the literature are various but a non-negligeable number of codesign frameworks perform partitioning manually. Those partitioning algorithms which are automatic consider a limited number of parameters in order to simplify the algorithm complexity. In our proposed algorithm, many parameters may be taken into account (performance requirements, hardware/software concurrency, and communication overhead).*

2. 6. Summary

This chapter showed a typical codesign flow, a review of the most popular codesign frameworks and then the proposed methodology. Our work is not an attempt to implement a complete codesign framework, but we concentrate on specific tasks to improve the standard codesign flow. All our proposed tools and techniques are intended to be integrated into

available cosynthesis tools. Developing the compiler which translates the input specification into an internal model and the hardware and software synthesis tools is out of the scope of our work. Compilers and synthesis tools are already available as industrial tools. Our objective is to propose efficient techniques for specific tasks in the codesign flow that can be easily integrated into available codesign frameworks to improve their performance and design exploration techniques. The major improvements provided by our proposition are related to the granularity flexibility that expands considerably the codesign space exploration and the pseudo-automatic partitioning approach. The follow chapters present the different codesign steps in detail with the solutions and approaches adopted in our codesign methodology. The next chapter presents the modeling technique in detail. We also concentrate in chapters 4 and 5 on the partitioning algorithm using a graph partitioning heuristic which takes into account the performance estimation and concurrency between model's tasks.

3

SYSTEM SPECIFICATION

This chapter examines issues in the specification and modeling of system functionality for systems that are the target of hardware/software consynthesis. The essential idea is to capture properties of a system without regard to its implementation.

Specification and design approaches provide many advantages through a designer's lifecycle. First, by creating a test-bench early in the design process and simulating the behavior, functional errors and omissions are detected early and easily corrected. Similar corrections can be extremely difficult to make late in the design cycle. Second, by defining module behavior completely, fewer integration problems are likely to occur after concurrent design of each of the modules. Third, by using a machine readable language, automated estimators and synthesis tools can be applied to reduce the design time or to rapidly evaluate alternative implementations. Finally, by writing a behavioral specification independent of any implementation information, redesign is greatly simplified. A variety of languages have been proposed for behavioral specification, such as VHDL, Verilog, HardwareC, CSP, and

StateCharts. A good language should support a conceptual model useful for the particular system to be specified. Existing languages support conceptual models such as finite-state machines or data flow graphs.

In section 3.1, a list of the modeling techniques generally used in related work is given. In section 3.2, the proposed model is presented on examples to show the efficiency of the modeling technique.

3. 1. Modeling techniques

A model refers to an abstraction over its object, capturing important relationships between components of the object. Models are often needed in order to avoid creating detailed implementation.

A formal model of a design should consist of the following components:

1. A functional specification (implicit or explicit relations involving inputs and outputs)
2. A set of properties that the design must satisfy.
3. A set of performance indexes that evaluate the quality of the design.
4. A set of constraints on performance indexes.

Common models used to capture the functionality of digital systems are listed in the next sections. Each one of these models is appropriate for a specific application field. A combination of one or two of these models may be required when using hybrid systems.

3.1.1 Finite State Machines

Traditional FSMs are good for modeling sequential behavior, but are impractical for modeling concurrency or memory because of the so-called state explosion. Several speci-

fication languages are based on finite-state machines, like StateCharts and SDL [BenIsmail94b]. In order to avoid their limitations, the FSMs are often improved by three characteristics to reduce the state space size: hierarchy, concurrency and non-determinism. Figure 3.1(a) shows an example of a simple FSM state digram. This state diagram shows the system control flow from one state to another or what is called state transitions. The state transitions are the edges on the state digaram. These transitions are initiated by input conditions or environmental events (values on the edges). Several common systems may be modeled as FSMs but these are more appropriate for control-dominated systems because no data processing is specified in the FSM behavior. To overcome such a limitation, extended FSMs have been proposed where the data processing is added to each state definition.

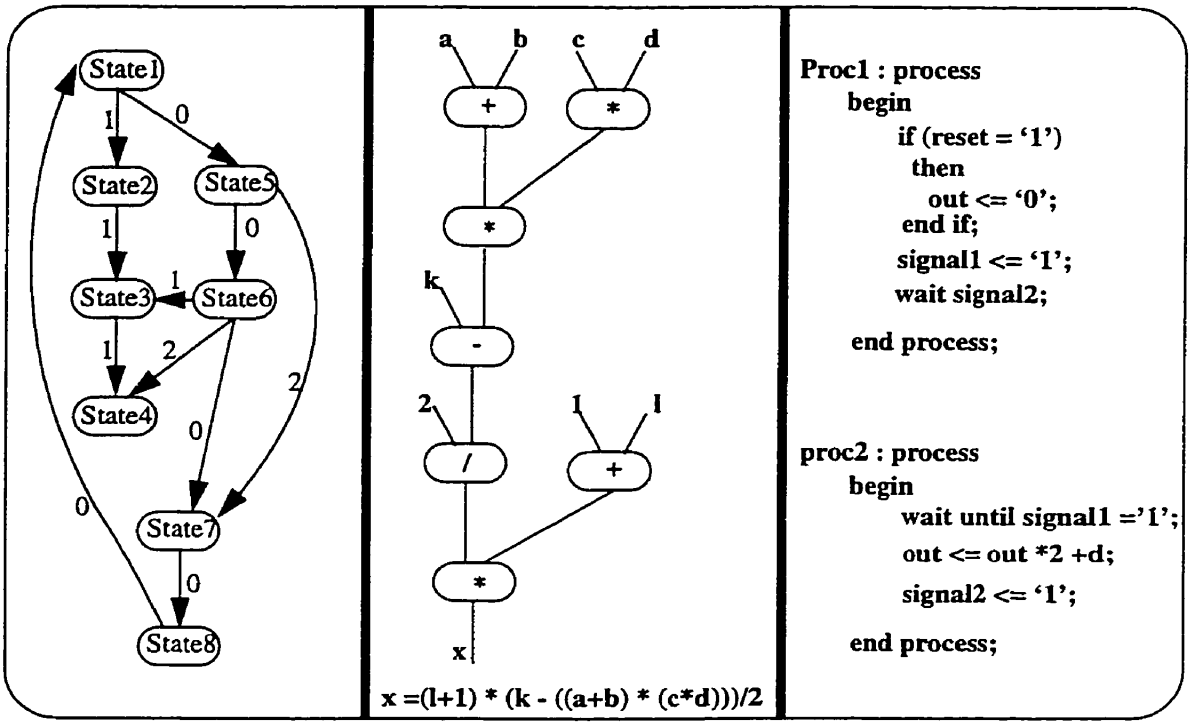
3.1.2 Data flow graphs

A program is specified by a directed graph where the nodes represent computations and the arcs represent totally ordered sequences of events. Examples of languages based on data flow graphs are Esterel [Dembinski], HardwareC [Gupta94], and Verilog [Hu94]. Figure 3.1(b) shows a small data flow graph where the nodes are basic arithmetic operations and the arcs shows the sequencing and data dependencies between the operations. The bottom of Figure 3.1(b) shows the arithmetic expression modeled by the data flow graph. This kind of modeling has been shown to be appropriate for data-driven applications especially digital signal processing applications. Data flow graphs have been extended to Control-Data flow graphs in order to support control-driven applications.

3.1.3 Communicating processes

The system behavior is described as a set of concurrent processes communicating via

message passing or shared data. This model is appropriate for control-dominated systems with concurrency which is a characteristic supported neither by finite-state machines nor by data flow graphs. An Example from this category is CSP [McFarland92] and VHDL [Eles94]. Figure 3.1(c) shows a VHDL code sample where two processes execute concurrently. The two processors are synchronized using wait statements on common signals (signal1 and signal2). Internally, each process may be modeled as an FSM or a data flow graph.



(a). A finite state machine (b). A data flow graph (c). Concurrent processes

Figure 3.1. Examples of the different modeling techniques.

3.1.4 Object-oriented modeling

The system behavior is described as a set of objects with their associated procedures and functions. This modeling concept is largely used in software development but its list of users and application fields is continuously increasing. This makes it a future candidate for

modeling high-level systems which can be implemented in hardware or in software. An attempt to use the C++ language for high-level systems has been purposed [Forrest92].

3. 2. The proposed model

In our codesign framework, we consider the input system as a hierarchical data flow graph. This means that the system behavior is represented as a set of processes or functions, each one described as another set of processes and functions until basic nodes are reached. Basic nodes are simply basic operations like arithmetic or logical operations. Blocks or functions may be the processes from VHDL description or the procedures in a C specification. Then, each block or function is decomposed into subblocks or basic statements. When basic statements are reached, the hierarchy is stopped while non-basic statements are decomposed until their basic statements are reached. Examples of non-basic statements are loops, conditional branches or simply a block of sequential statements. Each level in the hierarchy provides a different model, in terms of complexity, for the same input system.

Figure 3.2(a) shows an example from the communications field, the HDLC entity with its environment [Berry91]. An HDLC entity is the set of functions needed to perform the communication between a user and the network, in such a way that all low-level functions related to the network are transparent at the user side. The HDLC entity communicates with the user via the variables, *user-Input,* and *user-output*, and with the network via the variables, *frame, endframe, iline, sline,* and *nrline.* Internally the HDLC functions are structured into three main blocks, the *window manager*, the *emission manager* and the *frame receiver*, as shown in Figure 3.2(b).

Figure 3.3 shows the HDLC entity as a hierarchy of interacting and dependent ele-

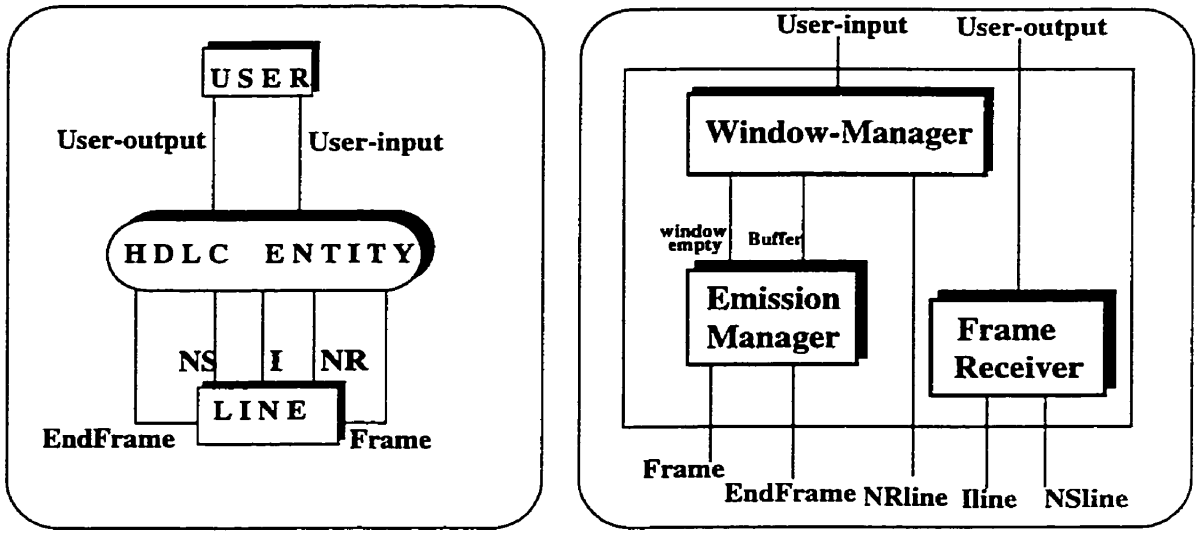
ments. We propose a new use of the already available hierarchy to consider different views of different complexity for the same system functionality. Indeed, level 1 of hierarchy shows the main block, then at level 2 the main block is decomposed into 3 blocks, the *window manager* block, the *emission manager* block, and the *frame receiver* block. At level 3 of the hierarchy, each one of these blocks is decomposed into 3, 2, and 2 subblocks respectively. Note that the *frame receiver* block is constituted at this level of one basic operation (assignment statement) for which no further decomposition is possible, and a loop structure which can be decomposed in the next level as the set of basic operations of the loop body.

This example shows how the behavior is structured from few complex boxes at low levels of hierarchy to the most detailed description which is simply a lot of basic and simple operations at high levels of hierarchy. These different modeling views are available in all modeling languages supporting the hierarchy. Our main contribution is to use each one of these modeling alternatives during the codesign process.

Each level of the hierarchy in the model is described as a data flow graph. Figure 3.4 shows examples of data flow graphs obtained for two different levels in the HDLC hierarchy.

A node in the data flow graph is characterized by its inputs (input edges to boxes), its outputs (ouput edges from the boxes) and the internal functionality of the node. The node may be a simple operation or a pointer to another data flow graph. Edges between nodes in the data flow graph correspond to data dependencies. Nodes that may be executed concurrently do not have any data dependencies as the *user-input handler* and *acknowledge handler* blocks. Dependent nodes have to be executed sequentially as is the case of the *frame emitter* and *line manager* blocks.

This data flow graph modeling allows the analysis to identify concurrent and dependent blocks in the model. In Figure 3.4(b), the blocks *acknowledge handler*, *buffer manager* and *frame emitter* have to be executed sequentially because they have data dependencies. The *buffer manager* block waits for the variable *na* which is an output of the block *acknowledge handler* and the *frame emitter* block waits for the variable *nu which is* output by the *buffer manager* block. The blocks *line manager* and *buffer manager* can be executed in parallel since there is no common variable shared by the two blocks.



(a). The HDLC inputs, outputs, and environment

(b). The HDLC block diagram

Figure 3.2. The HDLC entity environment and block digram.

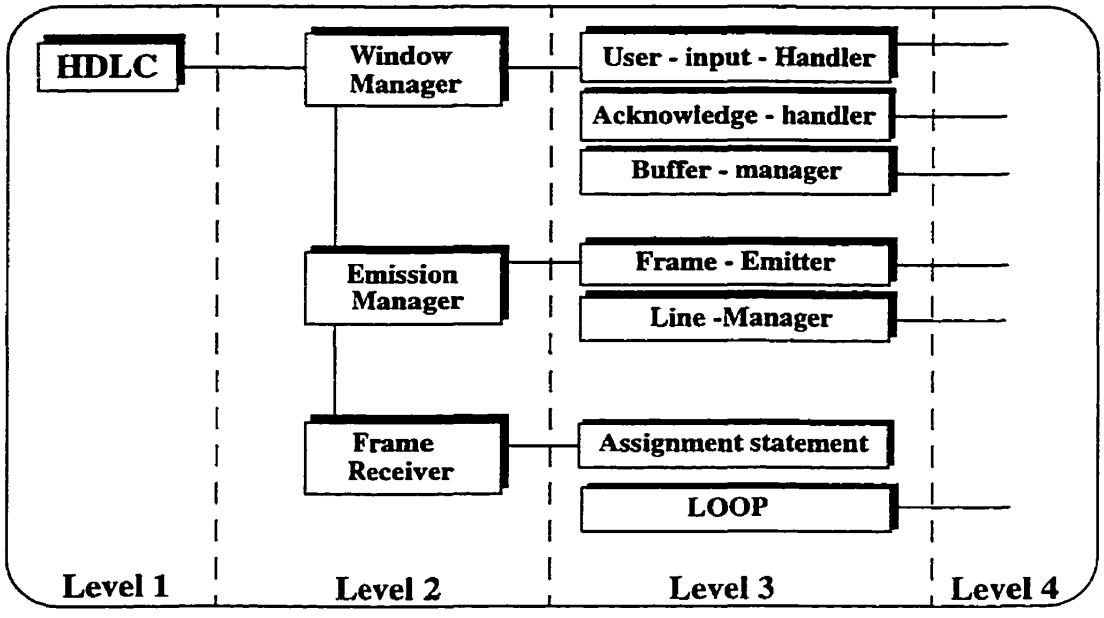
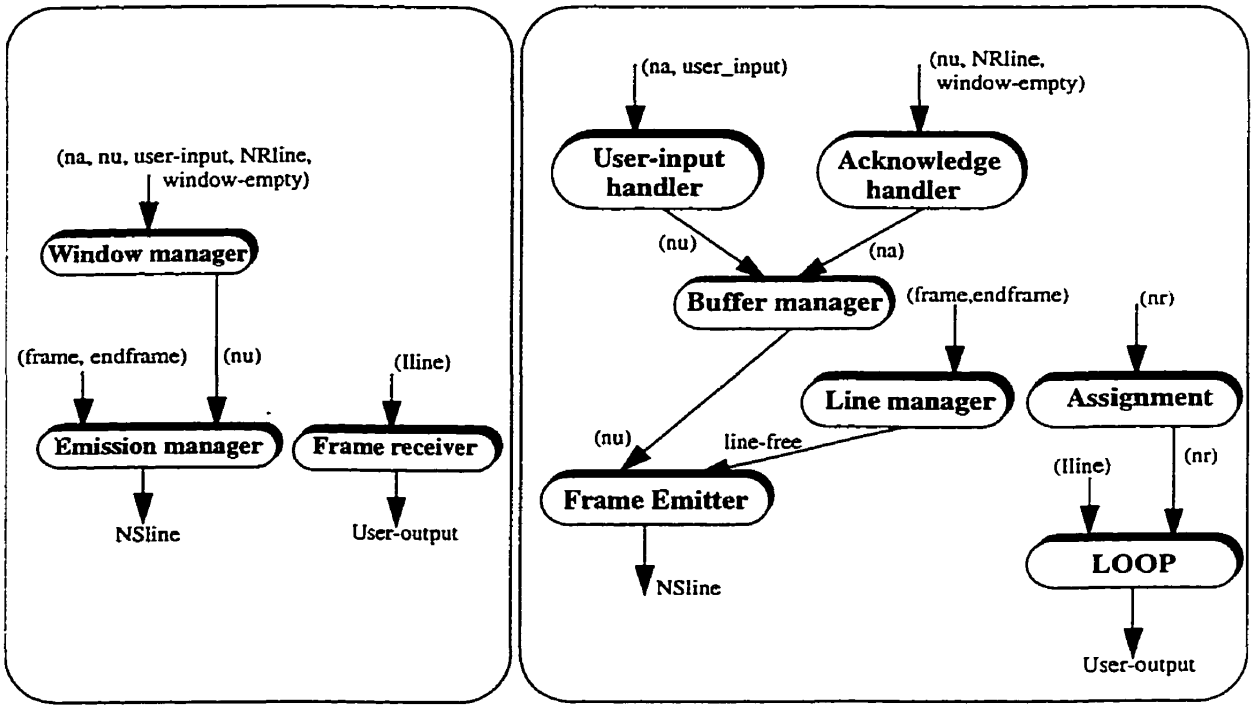


Figure 3.3. The hierarchical model of the HDLC entity.



(a). The data flow graph at level 1.

(b). The data flow graph at level 2.

Figure 3.4. The data flow graph models corresponding to different levels of hierarchy.

When codesigning the HDLC function, the designer may start to find the best partitioning at level 1. In this case, the partitioning algorithm handles only three functional and interdependent blocks which are *the window manager*, the *emission manager* and the *frame receiver* (Figure 3.4(a)). Each block is assigned to hardware or to software based on its performance estimation determined during the performance estimation step.

The data flow graph at this level shows that the block *frame receiver* is independent of the other two blocks, the *window manager* and the *emission manager*. If the two later blocks were assigned to software and the block *frame receiver* to hardware, there will be no communication overhead because no communication is needed between the two partitions, Figure 3.5(a). Executing the *frame receiver* block concurrently with the blocks *window manager* and *emission manager* reduces the overall execution time. The total execution time is the maximum of two execution times: the *frame receiver* execution time in hardware and the sum of the *window manager* and *emission manager* software execution times.

A complete software solution where all the blocks are assigned to software would execute sequentially all the blocks even if they can be executed in parallel. The total execution time is then the sum of the execution times of the three blocks, *window manager*, *emission manager* and *frame receiver*.

If no partitioning that satisfies the performance requirements is found when considering the HDLC as three interconnected blocks, the number of blocks may be increased by moving to level 2 of the hierarchy.

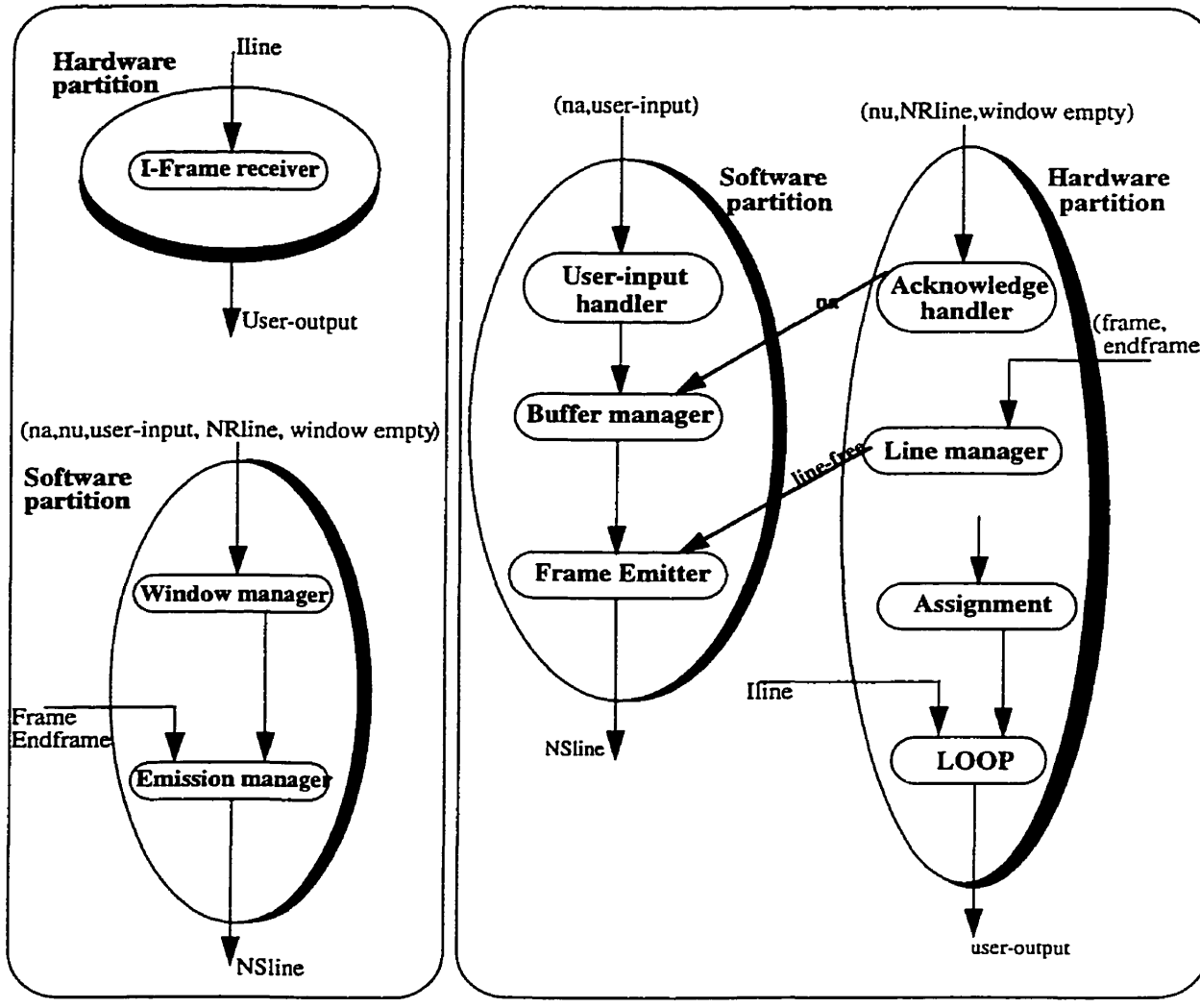
At this level, seven blocks build the system model, the *user-input handler*, the *acknowledge handler*, the *buffer manager*, the *frame emitter*, the *line manager*, the *assignment statement* and the *loop* (Figure 3.4 (b)). This detailed model allows more partitioning

alternatives. Indeed, more partitioning combinations are performed with seven blocks than with 3 blocks. All the possible alternatives may be evaluated before proposing the final implementation.

The data flow in Figure 3.4(b) shows that the blocks *line manager*, *buffer manager* and *loop* may be executed concurrently while the blocks *user-input handler*, *acknoweldge handler*, *buffer manager* and *frame emitter* have data dependencies and should be executed sequentially.

Figure 3.5(b) shows a possible hardware/software partitioning. Two variables, *na* and *line-free*, need to be transferred between the hardware and the software partitions. The overall execution time is equivalent to the sum of the execution times of the following three blocks, the *user-input handler*, the *buffer manager* and the *I-frame emitter*. The execution time is of course less than the execution time of the complete software solution which would be the sum of the execution times of the seven blocks in the input model.

The level of abstraction may be reduced more and more as long as the required performance is not satisfied or until the most detailed model of the input system, i.e the basic operation description level, is reached.



(a). Hardware/software partition at level 1. **(b). Hardware/software partition at level 2.**

Figure 3.5. Possible hardware/software partitioning.

3. 3. The model data structure

In this section, we describe the data structure used to implement the proposed model in a C++ framework. We present the principal classes and the list of associated functions and procedures. Figure 3.6 shows the two major classes needed to implement our hierarchical data flow graph model. These classes correspond to two objects: the task and the data flow node.

The first class describes each task in the hierarchical model, Figure 3.6. The task description block contains information on the task identifier, the task type, the list of variables used by the task, the list of performance and area estimates of the task, the pointer to its subtasks if the task is not a basic operation.

The lower part of the description block shows the list of procedures associated with the task object. These are the class constructor to build the description block for each task in the model, the procedures transcoding, mapping and type-computing to extract the performance estimates read from a technology file for each task in the model. The transcoding procedure is used to determine the block read from the data files. The mapping procedure matches the read block with the corresponding block in the model. The type-computing procedure determines the block type (basic or non-basic). Finally, the last procedure is span-hierarchy to span the hierarchy of the task in order to determine the basic operations used by the task. This procedure is used to compare two tasks in order to determine the common basic operations between them.

The second class shown in Figure 3.7 is the basic element or the node in the data flow graph structure. The description block of the node contains information on the node identifier which is the same as the one used in the hierarchical model, the list of the nodes connected to the current node. This list is split into two lists, one for the successors and the other for the predecessors. The list of successors correspond to the list of tasks in the model that use an input variable which is ouput by the current task while the predecessors are those tasks that provide the input variables of the current task.

The node description block contains also the scheduling cycle of the node when scheduled As Soon As Possible, ASAPly and As Late As Possible, ALAPly. These two val-

ues determine the critical nodes in the data flow graph. We will see in chapter 4 how these values are determined. Finally, the last field is the next node in the data flow graph.

The list of procedures associated with the object "data flow graph node" are: the constructor procedure to build the description block for each node in the data flow graph and two procedures to determine the list of successors and the list of predecessors for each node in the data flow graph. These lists are determined by extracting data dependencies of the current node with all the other nodes in the graph.

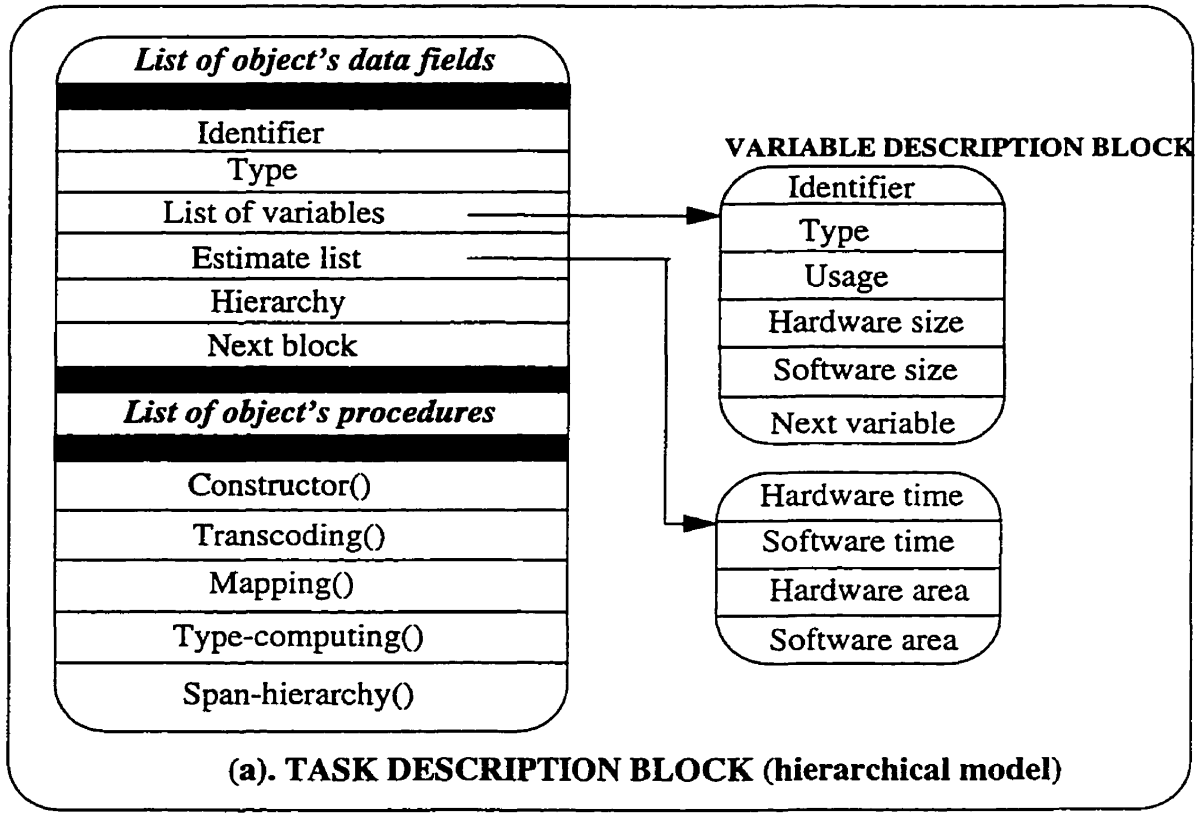


Figure 3.6. The data structures used to implement the hierarchical model.

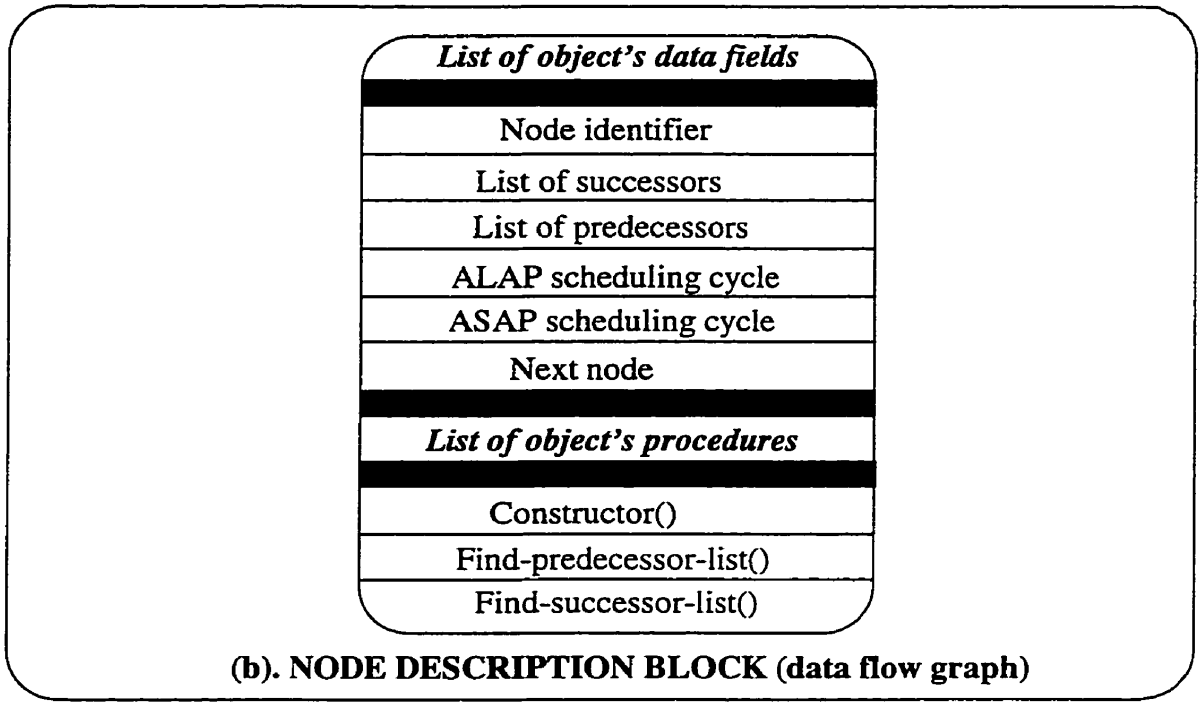


Figure 3.7. The data structures used to implement the data flow graph.

3. 4. Summary

As shown above, the proposed model has two main characteristics, the hierarchy and a variable granularity. The use of these features allow an expansion of the design space. For each level of the hierarchy, a different granularity is provided for the same input system.

Such a way of using the hierarchy enhances the codesign process with a new feature which has four main advantages:

1. *Many input models are possible for the same input system. The difference between all these models is their complexity while they describe the same behavior.*

2. *For each input model, the codesign process explores the design space to find a solution.*

3. *No need to handle a complex input model when the performance constraints are*

satisfied using simple input models.

4. *The above advantage may reduce considerably the CPU runtime of the codesign process.*

We will show later that the generation of different implementation alternatives for the same system is achieved only by varying the level of hierarchy.

Once the system model is constructed, the analysis of the behavioral blocks is performed to estimate the performance of each block in the model on the target architecture and to determine the concurrency and the dependency between the model blocks. Such analysis data is used during the partitioning to find the best assignment for each block. The next chapter presents the system analysis approach with the proposed performance estimation and scheduling techniques.

4

SYSTEM ANALYSIS

In this chapter, we describe two major analysis tasks. The first one is the performance estimation which provides the performance estimates used for block mapping during the partitioning. The second one is the scheduling to determine the execution flow of the different blocks in the design. This scheduling determines the concurrency between the different blocks. This information is also used during the partitioning.

Section 4.1 describes the performance estimation problem and in section 4.2 , the proposed technique for the performance estimation is presented. Sections 4.3 and 4.4 describe the scheduling problem and the proposed scheduling technique respectively.

4. 1. Performance estimation

System design is a set of tasks which convert the system-level specification into a set of completely specified interconnected modules implementing the specification. Each module could be implemented in hardware or as software executing on a processor. A hardware implementation has better time performance whereas a software implementation has lower

cost, shorter development time and allows changes late in the design cycle. Thus, the most efficient implementation has a minimal amount of costly application-specific hardware while still meeting the required timing performance constraints.

Due to the large search space associated with system-level design, it becomes a necessity to have the capability of obtaining estimates of design parameters such as area and performance that will characterize any implementation of the design. In the absence of estimates, the designer cannot make synthesis decisions and perform tradeoffs without actually synthesizing each partition and then evaluating the implementation.

Estimates of design parameters assist the designer providing him with the capability of exploring large design search in a relatively short time. The savings in time are evident if the designer were to synthesize the design completely before realizing that an undesirable design decision had been made early is the design cycle.

Furthermore, hardware/software partitioning requires performance estimations that will predict the execution time in order to identify which portion in the specification can be migrated from hardware to software while not violating the constraints or which portion needs to be implemented in hardware to satisfy the timing constraints.

4.1.1 Hardware performance estimation

The hardware estimation techniques are related to area and performance of a design intended for a hardware implementation. Area metrics are concerned with the area of entire processes or behaviors and consequently with the chip area. Thus, the designer will be able to determine as to how much area will a particular behavior require or wether a set of given behaviors be assigned to the same chip without violating the area constraints specified for that chip. The performance metrics are concerned with the execution time for processes and

inter-process communication times.

4.1.2 Software performance estimation

In order to rapidly explore large design space encountered on hardware/software systems, automatic software estimation is indispensable in hardware/software partitioning in which designers or partitioning tools must trade off a hardware with a software implementation for the whole or a part of the system under design. Software estimation provides three software metrics, execution time, program memory size and data memory size for a given target processor.

4. 2. The proposed estimation technique

Once the model is obtained from the specification, we perform an analysis on each element in the model to characterize its performance and the effect it has on the whole system performance. This analysis is performed in two steps, the performance estimation and the scheduling steps (Figure 4.1). The first step is the performance estimation which is performed given a data base library of the available resources. Four parameters are determined during this step: the execution time and the area of each partition (hardware and software). We used the Specsyn estimators developed at the university of California at Irvine to perform this step. These estimators are presented in the next section [Narayan92b] [Gong93] [Huang95]. The second analysis task is the scheduling of the different blocks to determine the overall execution time and the critical execution path in the model. The proposed scheduling technique will be presented in section 4.4. The information provided by the behavioral analysis step is then given to the partitioning algorithm in order to take the right decisions

when mapping blocks to hardware or to software.

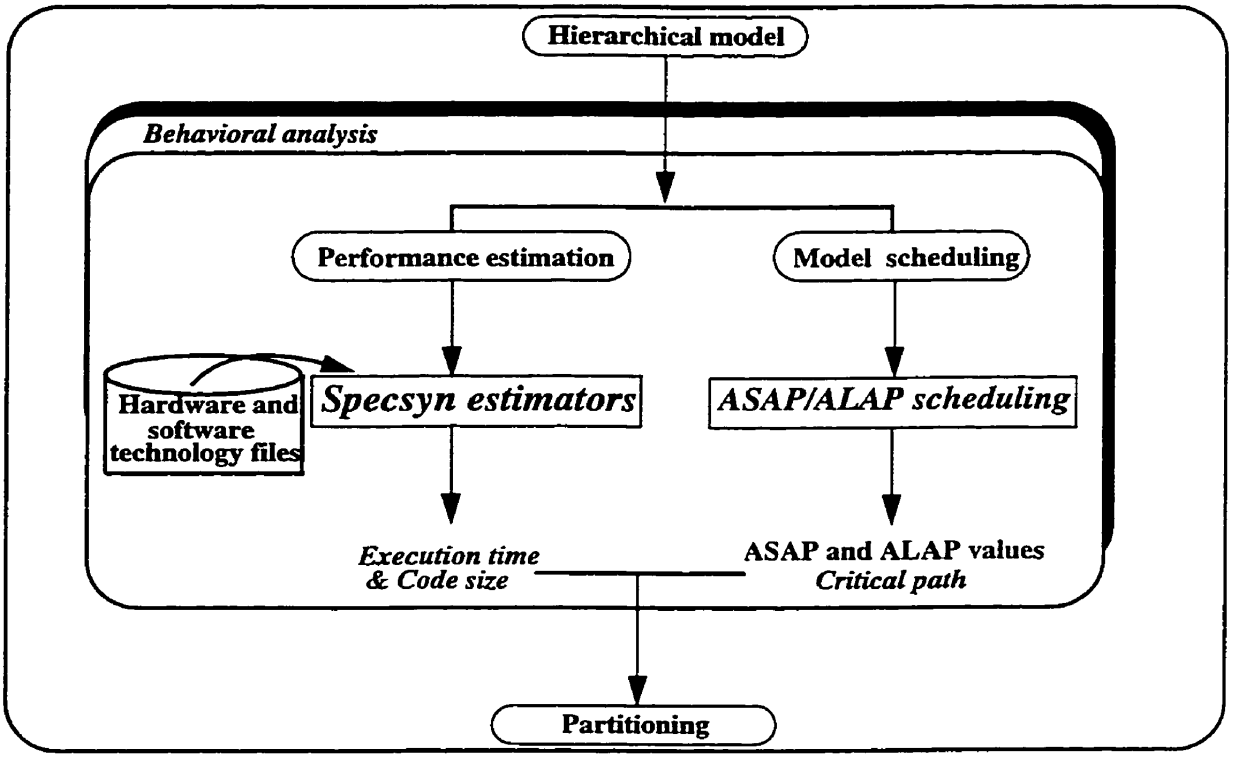


Figure 4.1. The behavioral analysis step.

4.2.1 The Specsyn estimators

Our estimation technique is based on two estimators for software and hardware developed at the university of California at Irvine. The software estimator is based on a generic model and does not require different estimators for different target processors. The hardware estimator is based on data path mapping of the processes with a given clock cycle.

1. Hardware estimation

The inputs to the area and performance estimators are:

1. *A SpecChart description* representing the design for which the area has to be estimated. A SpecChart consists of hierarchical concurrent state diagrams built on top of the VHDL language. An example of a SpecChart description is given

in Figure 4.3. The system is first defined as an entity with its input and output ports. This entity description is similar to the entity definition in VHDL. The architecture is then built of three sequential blocks, *read_data*, *processing* and *write_data*. Each one of these blocks is then described as a VHDL process, i.e a set of sequential instructions [Narayan92a].

2. *An allocation list* consisting of the number of available operators of each type to implement the design and their delays. If no allocation list is specified, the estimator will allocate one operator of each class (adder, multiplier, etc.) needed in the design. Figure 4.2 shows the content of an allocation list file. Each line in this file describes a resource block. The block is defined by its class, type, delay (number of cycles needed to execute the block), area/bit (the average number of transistors required to implement one bit slice of the block) and the number of copies available.

3. *The clock cycle* which will be used to determine the number of microstates in the design. The first line in Figure 4.2 specifies the clock cycle which is equal to 50 ns in this example.

The design model for estimation is a datapath/control logic model on which the scheduled behavior is mapped. The scheduling technique used is very simple, the ASAP scheduling.

Using the scheduled behavior with the provided clock cycle, a time estimation is evaluated following the execution flow determined during scheduling.

The strategy for area estimation is based on dividing the total design area into the following components:

1. *Datapath blocks*, which consists of registers, function units such as adders and interconnect units such as multiplexers. The datapath blocks are assumed to consist of a stack of bit-sliced components.

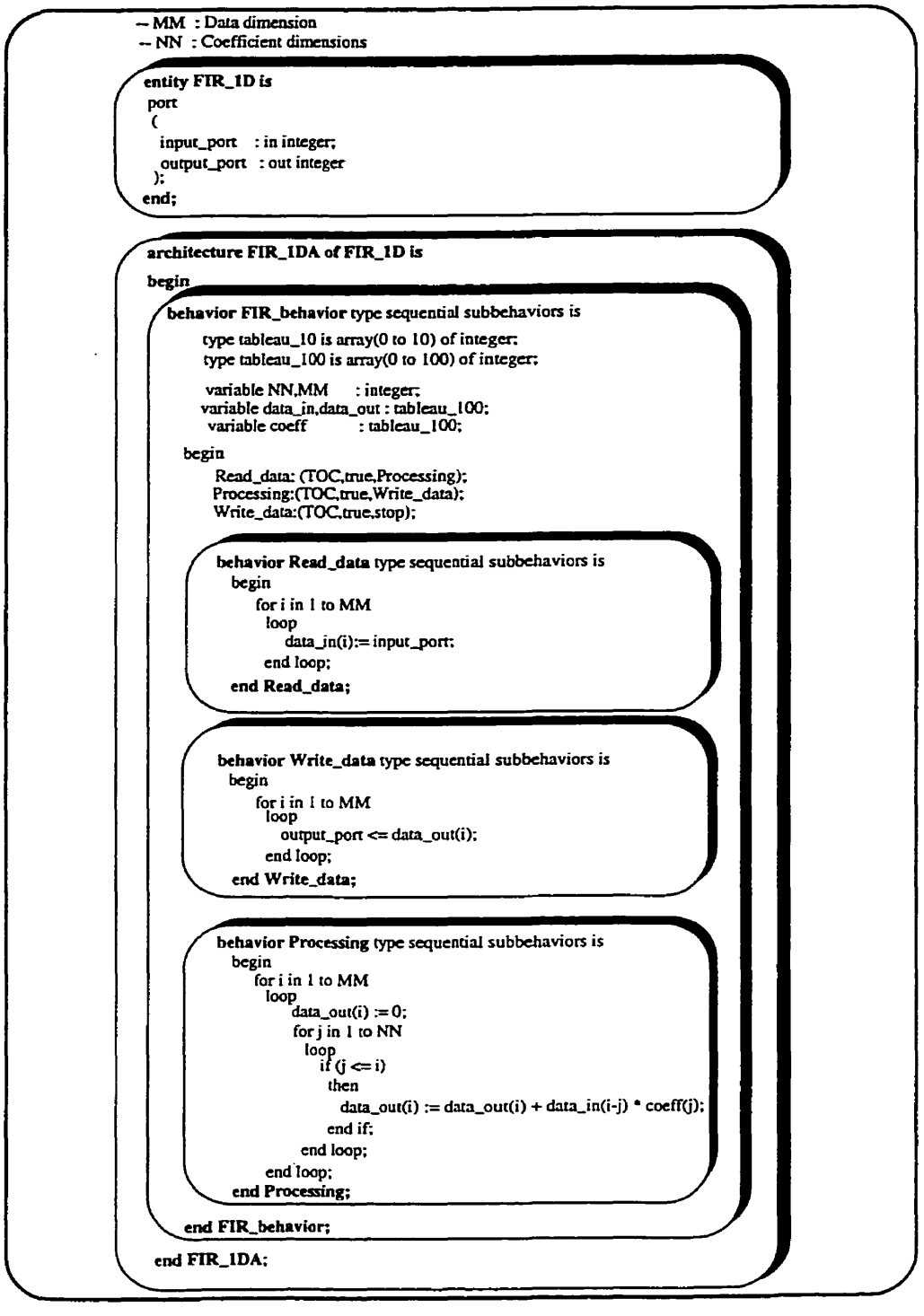
2. *Control units*, which controls the data transfers within the datapath components. The control unit could be a random logic implementation consisting of two-level AND-OR gates or of a ROM.

3. *Memories* which are used to implement the arrays in the design specification.

CLK 50

CLASS	multiplexer	TYPE	multiplexer	DELAY	7	AREA/BIT	2	NUM	1
CLASS	memory	TYPE	memory	DELAY	19	AREA/BIT	3	NUM	1
CLASS	memory	TYPE	register	DELAY	15	AREA/BIT	9	NUM	1
CLASS	operator	TYPE	&	DELAY	0	AREA/BIT	0	NUM	1
CLASS	operator	TYPE	abs	DELAY	56	AREA/BIT	9	NUM	1
CLASS	operator	TYPE	mod	DELAY	163	AREA/BIT	38	NUM	1
CLASS	operator	TYPE	rem	DELAY	163	AREA/BIT	38	NUM	1
CLASS	operator	TYPE	**	DELAY	163	AREA/BIT	38	NUM	1
CLASS	operator	TYPE	*	DELAY	163	AREA/BIT	38	NUM	1
CLASS	operator	TYPE	/	DELAY	163	AREA/BIT	38	NUM	1
CLASS	operator	TYPE	+	DELAY	49	AREA/BIT	9	NUM	1
CLASS	operator	TYPE	-	DELAY	56	AREA/BIT	18	NUM	1
CLASS	operator	TYPE	/=	DELAY	23	AREA/BIT	7	NUM	1
CLASS	operator	TYPE	>=	DELAY	23	AREA/BIT	7	NUM	1
CLASS	operator	TYPE	<=	DELAY	23	AREA/BIT	7	NUM	1
CLASS	operator	TYPE	>	DELAY	23	AREA/BIT	7	NUM	1
CLASS	operator	TYPE	<	DELAY	23	AREA/BIT	7	NUM	1
CLASS	operator	TYPE	=	DELAY	23	AREA/BIT	7	NUM	1
CLASS	gate	TYPE	xor	DELAY	6	AREA/BIT	4	NUM	1
CLASS	gate	TYPE	not	DELAY	2	AREA/BIT	2	NUM	1
CLASS	gate	TYPE	nor	DELAY	5	AREA/BIT	2	NUM	1
CLASS	gate	TYPE	nand	DELAY	3	AREA/BIT	2	NUM	1
CLASS	gate	TYPE	or	DELAY	6	AREA/BIT	3	NUM	1
CLASS	gate	TYPE	and	DELAY	5	AREA/BIT	3	NUM	1

Figure 4.2. The allocation list for hardware estimation.



```
-- MM  : Data dimension
-- NN  : Coefficient dimensions
entity FIR_1D is
 port
 (
  input_port   : in integer;
  output_port  : out integer
 );
end;

architecture FIR_1DA of FIR_1D is
begin
  behavior FIR_behavior type sequential subbehaviors is
      type tableau_10 is array(0 to 10) of integer;
      type tableau_100 is array(0 to 100) of integer;

      variable NN,MM      : integer;
      variable data_in,data_out : tableau_100;
      variable coeff      : tableau_100;

    begin
      Read_data: (TOC,true,Processing);
      Processing:(TOC,true,Write_data);
      Write_data:(TOC,true,stop);

      behavior Read_data type sequential subbehaviors is
        begin
          for i in 1 to MM
           loop
             data_in(i):= input_port;
           end loop;
        end Read_data;

      behavior Write_data type sequential subbehaviors is
        begin
          for i in 1 to MM
           loop
             output_port <= data_out(i);
           end loop;
        end Write_data;

      behavior Processing type sequential subbehaviors is
        begin
          for i in 1 to MM
           loop
              data_out(i) := 0;
              for j in 1 to NN
                loop
                  if (j <= i)
                    then
                      data_out(i) := data_out(i) + data_in(i-j) * coeff(j);
                  end if;
                end loop;
           end loop;
        end Processing;
    end FIR_behavior;
end FIR_1DA;
```

Figure 4.3. The SpecChart description of the FIR filter.

2. Software estimation

The software estimator is based on a generic model and does not require different estimators for different target processors. A single estimator and a set of technology files for different target processors are used. This makes the estimator fast and easy to extend for different target processors.

In order to obtain the estimates for processes, these process code must be compiled into machine instructions of the target processor. For example, if the process will be implemented on an Intel 8086 processor, it needs to be compiled into the 8086 instruction set. Using the timing and size information associated with each type of instruction such as how many clock cycles the 8086 instruction executes and how many bytes it takes, we can obtain the performance and program size of the process.

Instead of using different compilers and estimators for different processors, a generic estimation model is used. The processes described in SpecChart are converted into a set of generic instructions shown in Figure 4.4. The list of generic instruction is given with the possible addressing modes for each one of them. The estimator computes the software metrics based on the generic instructions and the technology files for the target processors. For example, if the process is going to be implemented on a Motorola 68000 processor, then the technology file for the 68000 processor is used during the estimation. The technology file for the target processor supplies information about how many clock cycles each type of generic instruction needs and how many bytes it takes if the generic instruction is executed on that target processor. The technology file for a target processor is derived from

the timing and size information of the processor's instruction set. Examples of technology files are given in Appendix A.

The advantages encountered by this estimation techniques are:

1. With a generic model, we do not need to use different compilers and different estimators for different target processors. Instead, only a single compiler, estimator and a set of technology files are required for the estimation.
2. The generic model makes it much easier to apply the estimator to other target processors. The estimation can be carried out as long as the technology file for the target processor is supplied.
3. It is much easier and faster to compile the specification into a generic instruction model than those associated with specific processors because the translation from the high-level specification to the generic code is automatic.

Instruction	Destination	Source 1	Source 2
ALU	Dest_addressing (1)	Src1_addressing (2)	Src2_addressing (2)
MUL	Dest_addressing	Src1_addressing	Src2_addressing
DIV	Dest_addressing	Src1_addressing	Src2_addressing
COMPARE	Dest_addressing	EMPTY	Src2_addressing
MOVE	EMPTY	EMPTY	EMPTY
CJUMP	EMPTY	EMPTY	EMPTY
JUMP	EMPTY	EMPTY	EMPTY
RETURN	EMPTY	EMPTY	EMPTY
CALL	EMPTY	EMPTY	EMPTY
NOP	EMPTY	EMPTY	EMPTY

(1) {Register, Direct Memory}
(2) {Constant, Register, Direct Memory, Indirect Memory}

Figure 4.4. The list of generic instructions.

The software execution time of a process is determined using flow analysis of the execution time of its constituent basic blocks. The execution time of each basic block is computed by summing the time execution of its constituent generic in-

structions. The execution time of each generic instruction is taken from the technology files supplied to the estimator.

The software area estimation is to determine how much program memory (bytes used to store the compiled program representing the process) and how much data memory (bytes used to store the data manipulated by the process) are needed.

Based on the size of each generic instruction, the program memory size of each basic block is the sum of that of all generic instructions in that basic block. The data memory size is determined based on the data declaration parts in the specification. The data memory size of each declared type is specified in a configuration file. The information used in the configuration file of Specsyn estimators is shown in Table 4.1

Table 4.1: Memory size of the base types

Base type	Data memory size (bytes)
Bit	1
Bitvector	n/8, n is the number of bits in the vector
Boolean	1
Character	1
Integer	4
Natural	4
Real	8
String	4
Time	4

4. 3. Scheduling

The performance estimates obtained for each block in the model are not sufficient to determine the execution time for the complete model. This is possible only after performing a scheduling of the blocks. The scheduling problem is to find an efficient sequencing of the tasks to optimize or tend to optimize the required resources and the total execution time. With the estimated performance for each

task, performing scheduling provides a measure of the overall execution time.

The scheduling problem is a well known problem, also known to be NP-Complete. Many heuristics have been proposed to solve this problem in a polynomial time. The simplest techniques are the ASAP (As Soon As Possible) and ALAP (As late As Possible) scheduling. In the ASAP approach, each task is scheduled as soon as a resource is available to execute it while the ALAP approach tends to schedule a task as late as possible. Figure 4.6 shows the algorithmic description of these two scheduling techniques. The two approaches allow, when applied to the same data flow graph, to determine the critical path. Figure 4.5 shows a scheduling example using ASAP (a) and ALAP (b) approaches on the same data flow graph. The critical path is determined by the list of tasks for which the execution cycle is the same independently of the approach applied (the sequence T1, T3, T4 and T6 constitutes the critical path).

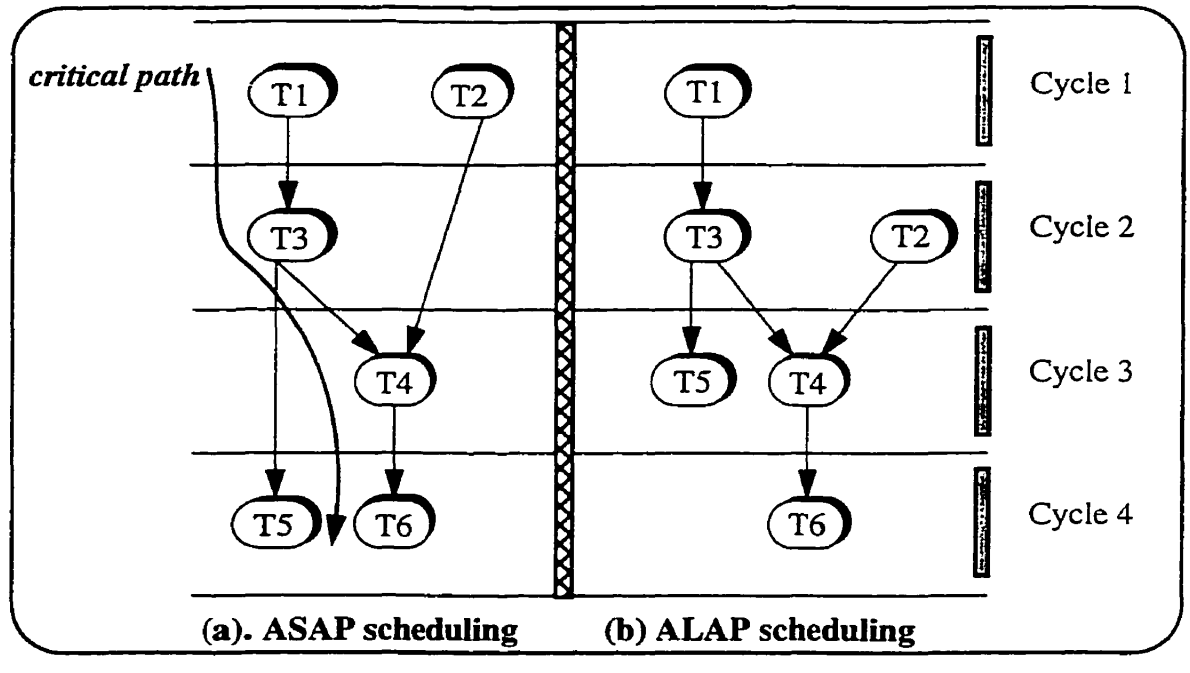


Figure 4.5. An example of ASAP and ALAP scheduling.

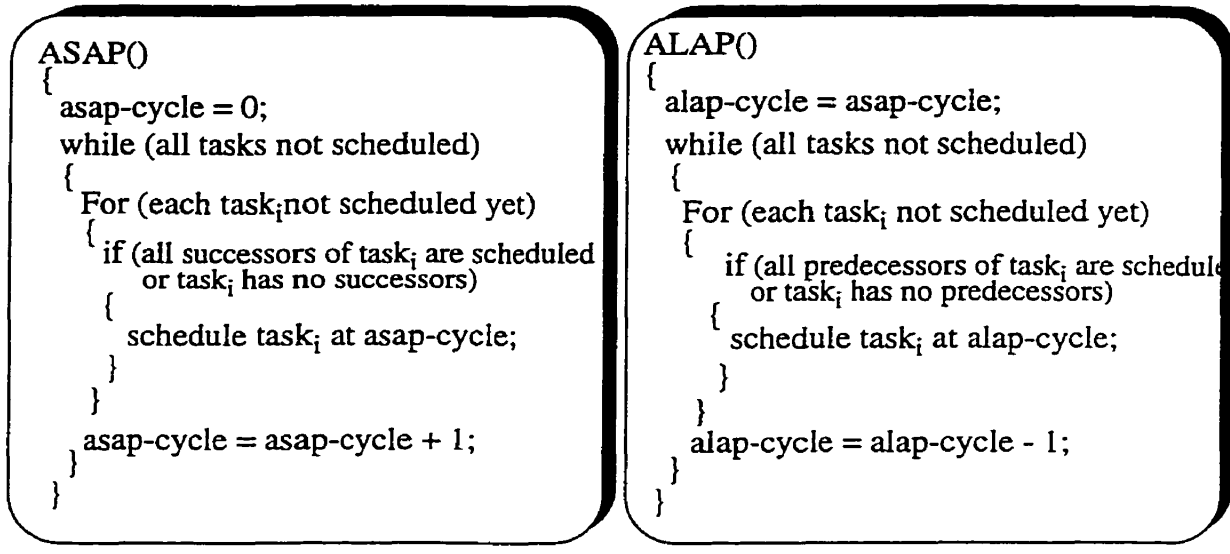
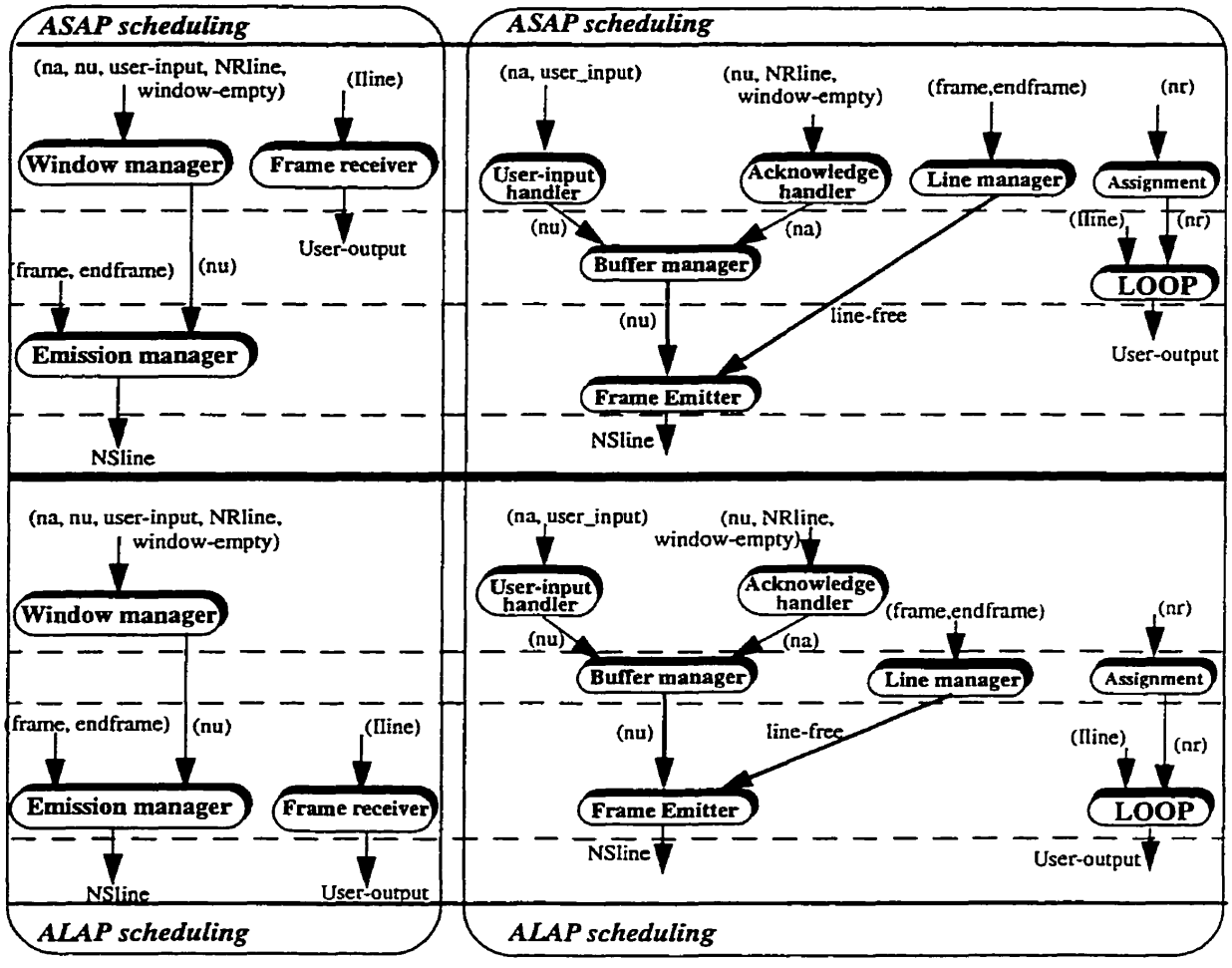


Figure 4.6. The ASAP and ALAP scheduling procedures.

The performance estimation step allows the characterization of each block independently of other blocks while the scheduling step determines the position of blocks in the complete system execution flow. At each level of hierarchy, the selected blocks are ASAP (As Soon As Possible) and ALAP (As Late As Possible) scheduled to determine the critical path and thus to identify critical blocks (Figure 4.1).

Figure 4.7 shows the obtained scheduling at two different levels of hierarchy for the HDLC example. At level 1, three blocks are considered and two possible execution paths are found. The critical path is the one starting at the block *window manager* and finishing at the block *emission manager*. At level 2, seven blocks are selected and four longest paths are found. Two critical paths are found in this case, the first one starts at the block *user-input handler* and ends at the block *frame emitter* while the second starts at the block *acknowledge handler* and ends at the block *frame emitter*.



(a). The ASAP and ALAP scheduling at level 1 of the hierarchy.

(b). The ASAP and ALAP scheduling at level 2 of the hierarchy.

Figure 4.7. The task scheduling for different levels in the hierarchy.

The scheduling step has two main objectives:

a). Identify the critical paths in the model execution.

b). Determine all the possibilities of concurrency between the hardware and software partitions by determining clearly all data dependencies.

4. 4. Summary

We have seen in this chapter that the performance estimation and scheduling tasks allow to determine an estimate of the execution time for the whole system. The next chapter shows the way these estimates are used during partitioning to guide the partitioning decisions.

5

SYSTEM PARTITIONING

In this chapter we address the problem of system functionality partitioning with the objective of achieving an implementation into separate components. The partitioning problem is of two types: homogeneous and heterogeneous. The objective of homogeneous partitioning is to partition a system functionality into a minimal number of parts such that all parts are implemented completely in hardware or in software. Homogeneous partitioning for hardware is typically done under size constraints on each of the parts, whereas for software implementations, the objective of partitioning is typically to increase resource utilization in order to achieve speedup in overall execution time.

We focus here on the heterogeneous partitioning problem, where the objective is to partition the system model for implementation into hardware and software components. The difference in the rates of computations causes variations in the rates of communication between hardware and software components and thus entails a higher communication over-

head than purely hardware or software partitions, due to necessary handshake and buffering mechanisms. Clearly, the problem of partitioning into hardware and software is much more complex than partitioning for implementations into purely hardware or software. The partitioning procedure presented in this chapter attempts to perform a division of functionality at diferent levels of the model, from the basic operation level to complex block level. The partitioning procedure attempts to supplement the conceptual design process by providing the system designer a means to handle the compexity associated with a detailed design description like at the language-level operations.

The partitioning problem for flow graphs refers to the assignment of operations in the graph to hardware or software. This assignment to hardware or software determines the delay of the operation. Further, the assignment of operations to a processor and to one or more application-specific hardware circuits involves additional delays due to communication overheads. All partitioning schemes must attempt to minimize this communication.

In this chapter, we focus on the hardware-software partitioning problem. As discussed in chapter 2, the task-level description of an application is specified as a DFG (Data Flow Graph) representing precedences and this DFG is the input to the partitioning tools.

The partitioning problem is to map each node of the DFG to hardware or software, and to determine the schedule for each node. The hardware-software partitioning problem is not just limited to making a binary choice between a hardware or software mapping.

The partitioning problem is a difficult one because good system-level cost metrics, accurate techniques for estimating the cost, and the techniques for reliable performance estimation of system-level hardware and software are not always available.

Partitioning is, in general, a hard problem. The design parameters can often be used

to formulate it as an integer optimization problem. Exact solutions are intractable for even moderately small problems. We propose and evaluate a heuristic solution.

The chapter is organized as follows. In section 5.1, the partitioning problem is defined. In section 5.2, we discuss the related work in the area of hardware-software partitioning. In section 5.3, we present the HAP (Hierarchy, Analysis and Partitioning) algorithm to solve the partitioning problem. Its performance is analyzed in section 5.4.

5.1. Problem definition

First, we state the major assumptions underlying the partitioning problem.

1. The precedences between the tasks are specified as a DFG. A performance constraint on the DFG is given as a deadline D, i.e, the execution time of the DFG should not exceed D clock cycles.
2. The target architecture consists of many processors (which execute the software component) and many custom datapaths (the hardware components). The software and hardware components have capacity constraints. The communication costs of the hardware-software interface are represented by three parameters, the hardware (software) area required to communicate one sample of data accross the hardware-software interface and the number of cycles required to transfer the data. This cost represents the area of the interface glue logic and the size of the code that sends or receives the data.
3. The area and time estimates for the hardware and software implementation of every node are assumed to be known. The specific techniques used to compute these estimates have been desribed in chapter 4.

The hardware-software partitioning problem (PP):

Given a DFG, area and time estimates for hardware and software mappings of all nodes, and communication costs, subject to resource capacity constraints and a deadline D, determine for each node i, the hardware or software mapping (Mi) and the start time for the execution of the node (schedule i), such that the total area occupied by the nodes mapped to hardware is minimum.

PP is combinatorial in the number of nodes ($\Theta(2^N)$ by enumeration). The problem is known to be NP-hard [Kalavade93].

Many techniques have been proposed to solve this problem. The next section presents a list of the most known works in the field followed by our approach presented in section 5.3.

5.2. Related work

Partitioning methods can be classified according to four characteristics, the specification model supported, the granularity at which the partitioning is performed, the cost function to be minimized and the partitioning algorithm used.

According to [Edwards97], the main partitioning related works are presented and classified below according to four characteristics: the input model, the granularity, the cost function and the partitioning procedure.

5.2.1 The input model

In chapter 2, different specification languages have been presented for codesign. The input specification is always translated into an intermediate representation which is used

during the other codesign steps like analysis, partitioning and synthesis. Column 2 in Table 5.1 shows a list of input models used by the principal partitioning algorithms described in the literature. In this list, we notice that the Control Data Flow Graph (CDFG) model is the most used [Henkel93] [Kalavade94] [Steinhausen93] [Gupta93]. This model is widely used because it may be extracted from different input descriptions, like an HDL specification as well as a programing language specification. The CDFG model is a unified modeling technique for both hardware and software but suffers some limitations as size explosion when it is used to modeling complex systems or some modeling failures when it is used for control-dominant systems.

To overcome such limitations, other modeling techniques are sometimes used as shown in Table 5.1. These may be HDL languages like VHDL [Thomas93] [Eles96] [Luk94], timing diagram [Chou94], set-based [Kumar92] or communicating processes [BenIsmail94b].

5.2.2 The granularity

The intermediate representation may have different complexity levels according to the model used and to its granularity too. Column 3 in Table 5.1 shows the different possible levels of granularity used in the literature. These are mainly two, the operation level or the task level. The operation level is more used than the task level because of its inheritance from the high-level synthesis field [Henkel93] [Kalavade94] [Gupta93] [Steinhausen93]. The basic modeling element in CDFGs is often the basic operation and the use of this level of granularity in the codesign field becomes intractable even for systems with medium complexity.

To overcome such a limitation, some codesign frameworks moved to the task level in order to reduce the modeling complexity [BenIsmail94b] [Olokutum94] [Hu94] [Eles96]. The system is built of complex blocks called tasks and these tasks are mapped to hardware or to software. The limitation of such level of granularity is the loss of system details available at the operation level. More mapping alternatives are possible at the operation level than at the task level.

COSYMA [Henkel93] has now a new version where the variable granularity has been introduced [Henkel97]. The system tries to reduce the complexity of the basic operation level by grouping some basic operations in what they called macro-instructions. These macro-instructions are generated using an optimization procedure

Our proposed variable granularity allows to overcome both problems by selecting any granularity in the system model from the most complex tasks to basic operations. There is no computation overhead added in the partitioning process because the blocks are available directly from the hierarchy. The Blocks can be macro-instructions at the first levels of the hierarchy or operations at the last levels of hierarchy. Unlike the COSYMA system, the blocks in our modeling have not to be generated but are already embedded in the hierarchical model.

5.2.3 The cost function

The hardware/software partitioning is peformed with the objective to optimize the system final performance. The performance is often measured by two parameters, the system execution time and the hardware area. Different techniques are used to estimate or to evaluate such performance as shown in column 4 of Table 5.1. Three main techniques are

used to measure the system performance: Profiling, synthesis and simulation.

The profiling consists in identifying the performance critical regions and bottlenecks in the input specification. The profiling is generally performed using compilers or analysis tools on a software specification [Henkel93] [Steinhausen93] [Kumar92]. The profiling has the main objective of identifying bottlenecks in the software implementation in order to move critical regions to a hardware implementation.

The synthesis technique consists in generating a final implementation using synthesis tools when a hardware/software solution is proposed. The execution time and the hardware area are evaluated for the synthesized implementation and this information is then refeeded into the partitioning process to find other implementation alternatives [Olukutum94] [Thomas93]. This process is reiterated until the design constraints are satisfied. The use of such a technique to evaluate the system performance is sometimes impratical because the synthesis phase may be time consuming and a complex task to perform at each iteration. To overcome such a limitation, some tools adopt the estimation approach instead of the synthesis one.

The third main technique used to evaluate the implementation cost is the simulation. The simulation is used for each proposed hardware/software implementation to determine the performance of each partition and also to determine the cost of the communication between the two partitions [Henkel93]. The simulation technique is very effective because it provides a real evaluation of the system but it is a very time consuming phase.

Other techniques have been proposed to evaluate the cost of the final implementation quickly with a certain loss of efficiency. These techniques are based on estimation tools or some analysis criteria like the similarity, the concurrence/sequence between modules

[Barros92], the closeness between operations [Henkel93], the schedulability of the tasks [Kalavade94] or the rate matching of different tasks [Luk94].

5.2.4 The partitioning algorithm

The last characteristic to classify partitioning techniques is the partitioning algorithm. Two main classes are first identified: manual or automatic partitioning. In the manual partitioning, a complete analysis and modeling environment is given to the designer who will entirely decide where operations or tasks are mapped [Steinhausen93] [BenIsmail94b] [Thomas93] [Luk94]. The other class of algorithms propose an automatic solution for the partitioning problem.

Many algorithms have been proposed in the second class. Some of these algorithms are very known and have already been used in many circuit design fields like high-level synthesis, logic synthesis, place and routing etc... .

D'Ambrosio et al. [Hu94] describes a branch and bound based approach for partitioning applications where each node has a deadline constraint (instead of an overall throughput deadline). Each node has three attributes: the deadline, the number of software instructions needed to execute it, and the type of hardware units it can be implemented on. The target architecture consists of a single software processor and a set of different hardware modules. The input specification is transformed into a set of constraints. The set of constraints is solved by an optimizing tool called GOPS, which uses a branch and bound approach, to determine the mapping. The approach suffers from limitations similar to those in a ILP formulation, that is, solving even moderated-sized problems can become computationally infeasible.

Kalavade et al. use an acyclic depending graph derived from a DFG (data flow graph where nodes are basic operations) to simultaneously map each node to software or hardware and schedule the execution of the tasks. The approach is heuristic and gives approximate solution to very large problem instantiations [Kalavade94].

Vahid et al. perform the partitioning of a variable-grained SpecCharts specification. SpecCharts is a hierarchical model in which the leaves are "states" of hierarchical State-Charts-like FSMs. Classical clustering and simulated annealing partitioning algorithms are applied. A refinement step may be performed after partitioning where each partition is synthesized to get better area, pin, chip count, and performance constraint satisfaction measure [Vahid].

Chou et al. describe a specialized scheduling-based algorithm for interface partitioning. The cost function is time for software and area for hardware. The algorithm is based on a min-cut procedure. This tool attempts to implement the interfaces as hardware or software partitions [Chou94].

Gupta et al. discuss a scheme where all nodes (except the data dependent delay tasks) are initially mapped to hardware. Nodes are at an instruction level of granularity (basic operations). Nodes are progressively moved from hardware to software subject to timing constraints. A hardware mapped node is selected (this node is an immediate successor of the node previously moved to software). This node is moved to software if the resultant solution is feasible (meets specified throughput) and the cost of the new partition is smaller than the earlier cost. The cost is a function of the hardware and software sizes. The algorithm is greedy and is not designed to find a global minimum [Gupta93].

Clustering is another heuristic used to perform partitioning. Units are clustered ac-

cording to some criteria like similarity, concurrency, sequencing, and mutual exclusion [Barros93] [Barros94].

A Kernighan-Lin swapping procedure is also used to perform partitioning. The procedure is applied on an initial solution where operations are classified according to their synthesizability [Olukutum94].

Ernest et al. use a graph-based model with nodes corresponding to basic operations in C. The cost function is derived using profiling to discover bottlenecks, estimation of operations closeness and estimation of the communication overhead incurred when blocks are moved across partitions. The partitioning is performed in two loops. The inner loop uses a simulated annealing with a quick estimation of the gain derived by moving an operation between hardware and software to improve an initial partition. The outer loop is manually performed by the designer and uses synthesis to refine the estimates used in the inner loop [Henkel93]. The authors of this system have updated it recently to consider a variable granularity as we have seen in the section 5.2.2.

The last tool is HMS (Hardware/Multi-Software partitioning) is a heuristic partitioning tool with scheduling [Sheliga94]. The scheduling process is not used as an analysis step before partitioning but the partitioning algorithm itself is based on the schedulability. Operations are selected for a hardware or a software implementation according to their needability. The needability measures the constraint of an operation to be scheduled in the current execution flow with the current hardware/software partitioning.

The variant proposed solutions have been shown to be effective for various applications. The field is new and any contribution is welcome. The list of tools presented above is not exhaustive but the main algorithms have been listed. The proposed algorithms in lit-

erature have been shown to be effective even if some of them suffer from some limitations. Due to the inherent nature of simulated annealing, this scheme requires long run times and the quality of the solution depends on the cooling schedule.The min-cut and clustering examples have local solution problems. The mathematical programming technique has the design space explosion problem even for moderately-sized systems.

It is very difficult to propose an exhaustive algorithm for partitioning since the problem is known to be NP-Complete and that is the main reason for the use of heuristics.

Our proposed partitioning technique has a major advantage not available in any of these works. Indeed, all the works listed above consider a fixed granularity as shown in column 3 of Table 5.1. This granularity may be at the operation level or at the task or process level. Our proposed methodology allows a use of a variable granularity according to the performance constraints. The most abstract model is used first and while the performance constraints are not satisfied, more details are taken into account until the operation level if needed.

We also propose a pseudo-automatic partitioning heuristic based on graph partitioning techniques. In table 5.1, most of the related work use a manual partitioning or an automatic partitioning. An automatic approach for partitioning reduces the codesign time and increases the number of alternatives considered by the partitioning tool before proposing the best implementation. The automatic search of the best implementation is clearly faster and more efficient. But the automatic solutions did not gain a lot of success in the codesign field because designers want to have the facility to make the design decisions. For this reason, we propose a pseudo-automatic partitioning tool which looks for different design alternatives, proposes these alternatives with their performance measures and lets the

designer select the appropriate final implementation.

Table 5.1: Comparison of the common partitioning methods.

Partitioning tool	Model	Granularity	Cost function	Algorithm
COSYMA [Henkel93] [Henkel 97]		operation variable granularity	profiling (Sw), synchronization & simulation (Hw)	Manual (outer) *Simulated annealing* (inner)
SpecSyn [Vahid]		operation	Profiling (SW) Processor cost (HW) Communication cost	*Clustering* *Simulated Annealing* Manual
UNITY [Barros92]		operation hierarchy	Similarity Concurrence/sequence	*Clustering*
PTOLEMY [Kalavade94]		operation	Schedulability	Heuristic with look-ahead
CASTLE [Steinhausen93]		operation	Profiling	Manual
COSMOS [BenIsmail 94a]		task	N/A	Manual
VULCAN [Gupta93]		operation	Execution time	Heuristic, *greedy*
[Chou94]		operation	Time (SW), area (HW)	*Min-Cut*
[Olukotum94]		task	Profiling (SW) Synthesis (HW)	*Kernighan and Lin*
[Kumar92]		task	Profiling	*Mathematical programming*
[Hu 94]		task	Profiling scheduling analysis	*Branch and Bound*
[Thomas93]		task	Profiling (SW) Synthesis (HW)	Manual
[Eles96]		task	Profiling	*Simulated annealing*
[Luk94]		operation hierarchy	Rate matching	Manual
[Sheliga94]		opration	Scheduling cycles	Schedulability based heuristic

5.3. The proposed partitioning technique

In this section, we first introduce the graph partitioning problem and then its use for hardware/software partitioning problem formulation. The proposed graph partitioning algorithm is based on the techniques presented in [Kernighan70] and [Oudghiri92]]. These techniques consider a graph G = (E, V), where V is the set of nodes and E the set of edges. Each edge in E is also weighted by a cost value as in [Oudghiri92]. The graph G is then partitioned into the minimal number of cliques. Nodes are assigned to cliques while keeping the weights on edges from different cliques at a minimal value. This means that the edges with large weights must be assigned to the same clique. The graph partitioning techniques have been used to formulate and solve a wide range of problems for the following reasons:

1. *The graph partitioning has the simplicity of constructive-iterative algorithms but uses a global formulation of the problem.*
2. *At each step of the iterative algorithm, the graph formulation may include all the required data to make the best selection. This formulation provides a flexible way to include such data (weights on the edges).*

This technique has been used to perform high-level synthesis [Oudghiri92] and is now extended to perform hardware/software partitioning of a digital system as we show in the next paragraphs [Oudghiri97].

First, the graph G is constructed in such a way that nodes are the blocks in the behavioral

model considered at a given level of hierarchy. G is a complete graph because all the blocks are connected by an edge. Edges of the graph are weighted by a closeness function, which corresponds to the number of common variables between the two nodes. Two operations with a large number of common variables have to be assigned to the same partition in order to minimize the communication needed between the final partitions. The weights on the graph edges provides an easy way to clustering strong dependent tasks.

This graph is then partitioned into a fixed number of cliques using the heursitic in [Kernighan70]. These cliques may be different software implementations and different hardware implementations. In our case, two cliques are considered and correspond to the implementations on software and hardware. The steps of the partitioning algorithm are shown in Figure 5.1.

The algorithm puts all blocks into the software partition at the first step. If the constraints are not satisfied by the software solution, blocks are moved to hardware.

First, **Algorithm 1** (Figure 5.1) selects the most time consuming node in the behavior (step 3.i), based on the performance estimates already determined in chapter 4, and assigns it to the hardware (step 3.ii). If the constraints still not be satisfied, the next node is selected among the neighbors of the node selected in step 3.i (step 3.iii). The neighbors are considered according to the increasing order of their edge weights. The neighbor with the maximum number of common variables with the current node is selected.

At each assignment, the dependency graph is updated. The update consists in deleting or merging no more needed edges. The edges connecting the assigned block to blocks from the same partition are deleted. The edges connecting two blocks from the same partition to a node from a different partition are merged into one edge. The weight on edges correspond

to the number of common variables between two blocks. When two edges are merged, the weight of the new edge is the cardinality of the union of the two common variable sets. In Figure 5.2(c), the edges (buffer-manager, frame-emitter) and (user-input-handler, frame-emitter) are merged into one edge when the block frame-emitter is assigned to hardware as shown in Figure 5.2(d).

This selection and graph update are repeated until the performance constraints are satisfied or all the model blocks have been moved from software to hardware.

At each partitioning step, the system execution time is compared to the performance constraints. The system execution time is computed using three parameters: the performance estimation of each block, the concurrency and dependency between blocks (result of the scheduling step) and the communication time. The execution time is computed using the set of equations shown below.

$$Execution-time = \sum_{i} time-h(i)\ [time-s(i)\ ] + comm(i) \qquad \textit{Eq.1}$$

$$comm(i) = \sum_{j} transf(j) \qquad \textit{Eq.2}$$

$$transf(j) = k\ cycles.$$

Time-h(i) is the hardware time execution of the block *i* in the input model.

Time-s(i) is the software time execution of the block *i* in the input model.

Comm(i) in the communication time required by the block *i* to transfer data between the hardware and software partitions and *j* is one of the transferred variables by the block *i*.

Time-h is selected if the block *i* has been assigned to hardware and *time-s* is selected if the block *i* has been assigned to software.

The first equation computes the complete execution time considering blocks on the critical path. For each critical block, the estimated hardware or sofware execution time (time_h

and time_s) is used if the block is assigned to hardware or to software respectively. The communication time is computed by the second equation which is the sum of the transfer time of all variables used by the block and which are updated by the blocks assigned to a different partition.

The transfer time from hardware to software or vice-versa for one variable is k cycles and is determined according to the transfer speed of the available resources (processors and ASICs).

Eq.1 is used in a recursive procedure that searchs the longest path in the design flow as shown in **Algorithm 2** (Figure 5.3). The procedure *longest-path()* starts from a root block and spans all the possible paths starting from the root block successors. Each one of the successors is considered as a new root and the procedure is recalled for that block. At the end, the procedure provides the slowest path. The system execution time is then computed for this path by the procedure *longest-path()*.

Figure 5.2 shows the partitioning flow steps for the HDLC example. The HDLC is considered at level 2 of its hierarchy, i.e the model is built of seven blocks which have the dependencies shown in Figure 5.2(a).

We considered an execution time constraint equal to 18 ms. The software implementation of the system runs during 27 ms and does not satisfy the time constraint equal to 18 ms. When considering these performance values, **Algorithm 1** peforms as shown in Figures 5.2(b) to 5.2(d).

First, the node *LOOP* is selected and assigned to hardware. This assignment reduces the total execution time to 25 ms. The input constraint (18 ms) is not yet satisfied. At the next step, the *LOOP* block neighbors are considered (*assignment*) and the maximum weighted

edge is selected (only one in this case). The *assignment* block is then assigned to hardware and the total execution time becomes 22.65 ms. The dependency graph is updated as shown in Figure 5.2(c). The edge (*LOOP*, *Assignment*) is moved to hardware and is now considered as one node in the hardware partition.

At this step, the constraints have not been satisfied yet and another block is selected, the *frame emitter* block because this block has the maximum execution time in the list of blocks assigned to software. The assignment of the block *frame emitter* to hardware reduces the total execution time to 19.02 ms but this time is still beyond the time constraint. One of the *frame emitter* neighbors is selected (the one with the maximum weighted edge), the *line manager* block. This last assignment provides a total execution time equal to 17.58 ms with one variable to be transferred between hardware and software partitions, as shown in Figure 5.2(d).

At this step, the time constraint is satisfied, the algorithm is stopped and the final partitioning is output. Four blocks, *loop*, *assignment*, *Frame emitter* and *line manager*, have been assigned to hardware, and three blocks, user-input handler, acknowledge handler and buffer manager have been kept in software.

We have shown in this section the partitioning procedure and the way the dependency graph is built to formalize the partitioning problem. The selection strategy of nodes has also been shown for each step of the algorithm. The proposed heuristic has the following main objective: minimizing the hardware partition while satisfying the performance constraints. The proposed heuristic is greedy and the final assignment depends on the selection order. In the next section, we present the different modules defined to implement the partitioning algorithm.

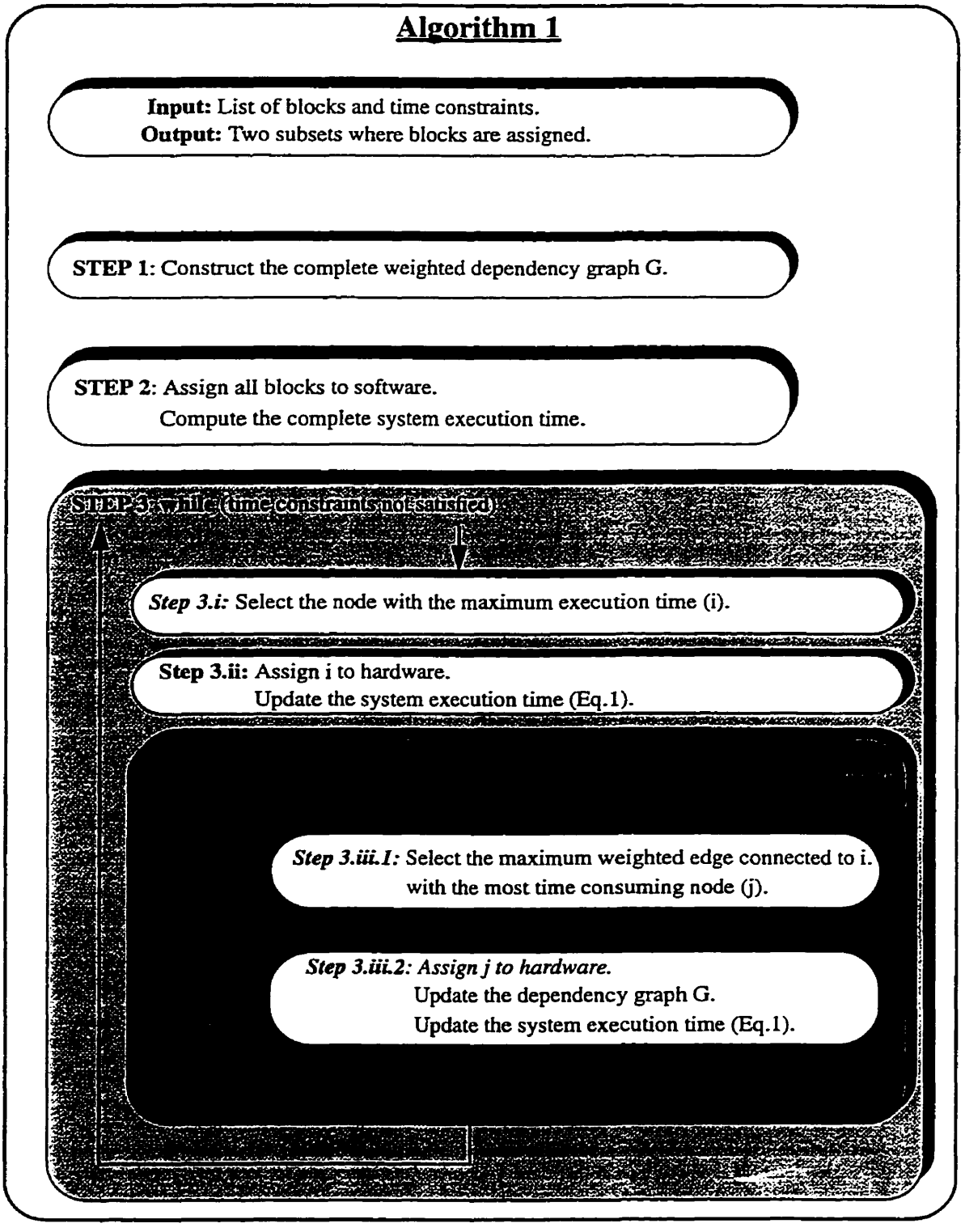
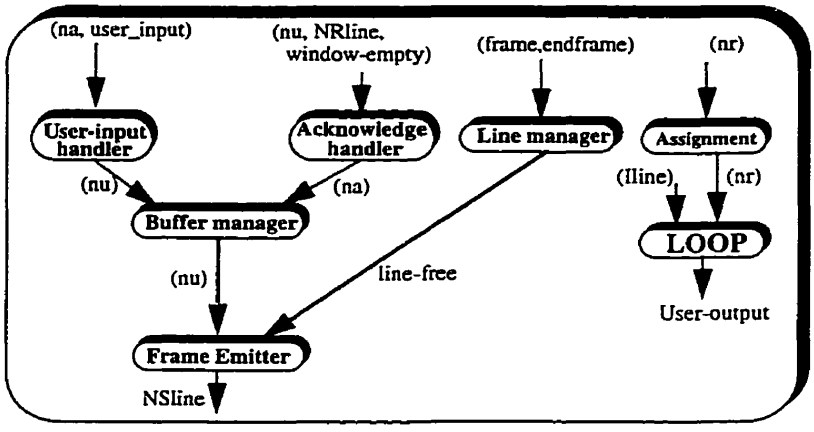
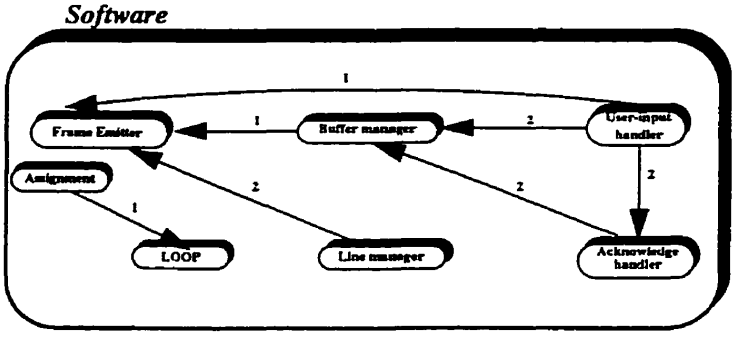


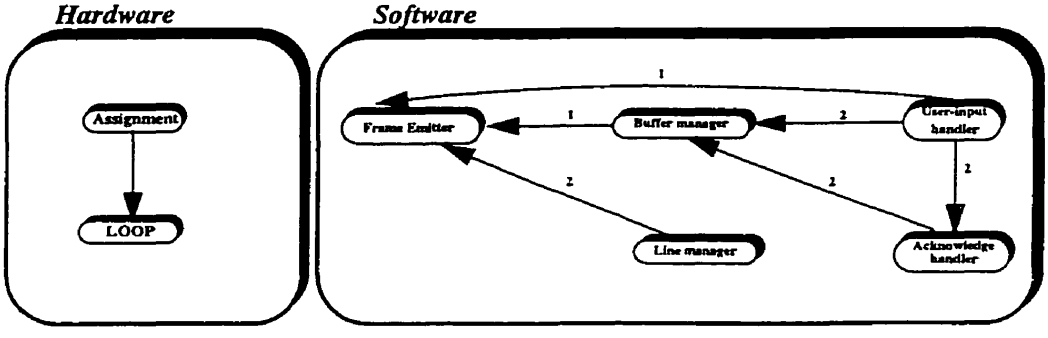
Figure 5.1. The proposed hardware-software partitioning procedure.



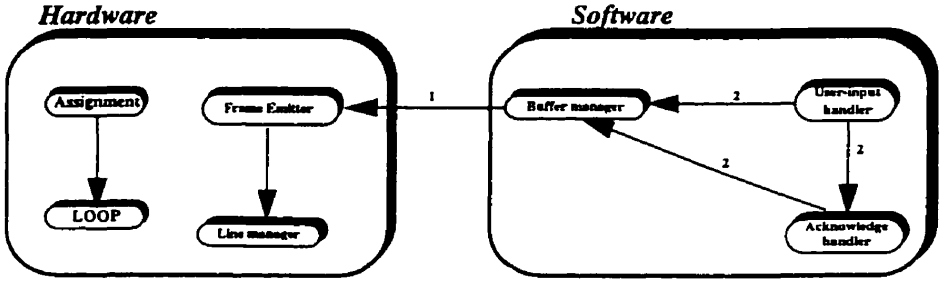
a. The input data flow graph.



b. The initial dependency graph and the initial partition.



c. Hardware/software partitioning when the node LOOP is selected. (iteration 1)



d. Hardware/software partitioning when the frame-emitter node is selected (iteration 2).

Figure 5.2. The partitioning procedure flow for the HDLC example.

5.4. The partitioning algorithm implementation

The proposed hardware/software partitioning algorithm has been implemented as a set of modular blocks in C++. In this section, we will give a modular description of the partitioning algorithm. Only the main procedures in the partitioning algorithm will be presented.

Figure 5.4 shows the principal procedures and functions involved in the partitioning algorithm. These are five procedures and two functions. The procedures are: the weighted graph construction, the partition initialization, search the maximum weighted edge in the graph, search the maximum weighted node in the graph, and reduce the dependency graph. The two functions are : compute time and compute area of the complete system at each partitioning step.

The general functionality of each one of these modules is described below.

1. *Construct-weighted-graph()* : this procedure is provided with the number of building blocks in the model at a given level of hierarchy and provides, at the output, a graph where the blocks are connected by edges. Each edge in the graph in weighted by an integer value corresponding to the number of common variables between the two blocks connected by the edge.
2. *Initialize-partition()* : this procedure assigns all the blocks in the dependency graph to the software partition. This provides the initial assignment of all the blocks and also the total software execution time computed by the function compute-time().This time will be compared to the time constraints to make the decision of moving blocks to hardware.

3. *Search-max-node()* : this procedure determines the block which has the largest execution time. This is performed by visiting all the graph nodes and checking their performance estimate.

4. *Search-max-weighted-edge()* : this procedure determines the maximum weighted edge connected to the current block. This search corresponds to finding among the neighbors of the current block the one which has the largest number of common variables with the current block.

5. *Reduce-graph()* : each time a node is moved from software to hardware, the dependency graph is updated. This update may consist in deleting some edges or updating the edge weights.

6. *Compute-time()* : this function determines the total execution time of the current implementation of the system considering the blocks in hardware, the blocks in software and the required communication time. This procedure takes into account the concurrency to provide a realistic evaluation of the system overall execution time.

7. *Compute-area()* : this function determines the total area of the system or the code size if the system is completely implemented in software.

In the next section, we consider the complexity of the proposed heuristic built of the presented procedures.

Algorithm 2

```
void critical_path()
{
  while (there is an invisited block i )
      longest_path(blocki);
}

void longest_path(blocki)
{
  if (blocki is assigned to HW)
    then
      cumul += time-h(blocki) + comm(blocki);
    else
      cumul += time-s(blocki) + comm(blocki);
  end if;
  for each successor of blocki
      longest_path( successor);
  if (cumul > glob-cumul) glob-cumul = cumul;
}
```

Figure 5.3. The procedure to find the critical path in the data flow graph.

5.5. The algorithm complexity

In order to evaluate the complexity of our proposed partitioning technique, we first compute the complexity of each procedure used in the heuristic. These procedures have been presented in the previous section and their complexity is shown in Figure 5.5.

The main loop is iterated until all blocks have been assigned or as soon as the performance constraints are met. This loop has a computing complexity of $O(N)$ in the worst case, where N is the number of blocks that build up the input model.

The procedure Search-max-weighted-node() identifies the most weighted block, i.e, the block with the maximum execution time. In the worst case, this procedure requires N iterations.

The procedure Search-max-weighted-edge() finds the most weighted edge connected to the current node. This procedure iterates, in the worst case, *(N-1)* times.

The procedure Compute-system-performance() determines the current performance of the system according to the current assignments of blocks. This procedure has a *O(N)* complexity.

The last procedure, Reduce-graph, has the role of reducing the dependency graph each time a node has been assigned to one of the two partitions. This procedure visits all the common neighbors of the assigned block and all the blocks in the same partition. In the worst case, this procedure has a *O(N)* complexity.

The four procedures all together have a *O(N)* complexity. These procedure are called in the main loop body. Thus, the partitioning algorithm has a $O(N^2)$ complexity in the worst case.

The heuristic complexity of $O(N^2)$ is considerably reduced compared to a $O(2^N)$ complexity of the exhaustive solution. This heuristic provides a polynomial time solution for a hard problem known to be NP-Hard.

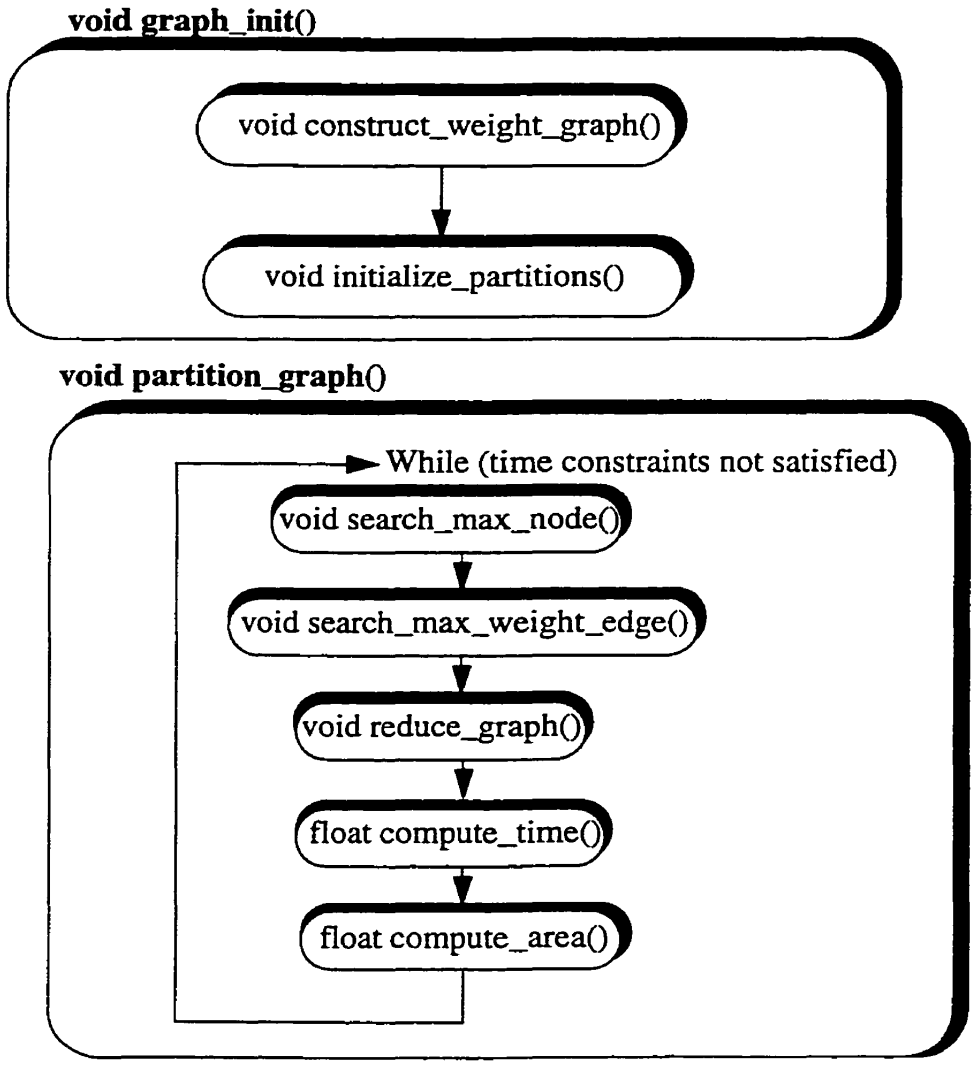


Figure 5.4. The principal procedures used in HAP.

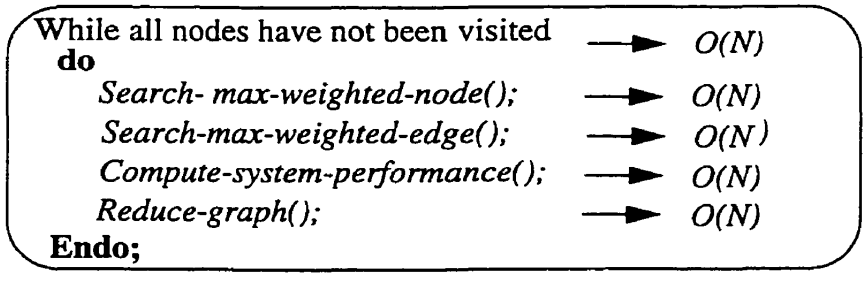


Figure 5.5. The complexity of the principal procedures in the partitioning algorithm.

5.6. Summary

The proposed partitioning algorithm has the following main advantages:

1. *The global dependency graph formulation is a complete description of the dependency between the input model blocks. The graph nodes are the blocks in the application model and the edges are the interactions between the blocks.*
2. *The weighted nodes identify the time consuming blocks while the weighted edges in the dependency graph quantify the dependency between blocks. Thus, the classification and the comparison between blocks are easy to perform.*
3. *The area and time estimation values associated with each node in the graph allows an estimation of the system overall area and time at each partitioning step. Thus, partitioning decisions are taken based on this available data.*
4. *The node scheduling performed during the analysis step allows taking into account the possible concurrency between hardware and software partitions. Indeed, dependent and independent blocks are clearly identified during the scheduling step.*
5. *The nested loop structure used in* **Algorithm 1** *is used to perform a two-level selection. The first selection (outer loop) is performed on nodes which require a long time to execute in order to accelerate the complete system execution time. The second selection (inner loop) is performed on neighbors of the block already selected in the outer loop in order to minimize the communication cost.*
6. *The heuristic complexity is $O(N^2)$. A feasible solution is possible in polynomial time.*

This completes the presentation of our approach. The availability of such an approach for codesigners provides new characteristics for efficient design space exploration. Indeed, different input models for the same input system are considered and each block in the model has time and area estimates, while the concurrency and dependency between blocks are also taken into account. Hence, the proposed tool performs a wide (different input models corresponding to levels in the hierarchy) and deep (weighted graph partitioning at each level)

search before proposing a solution to the designer as shown in Figure 5.6.

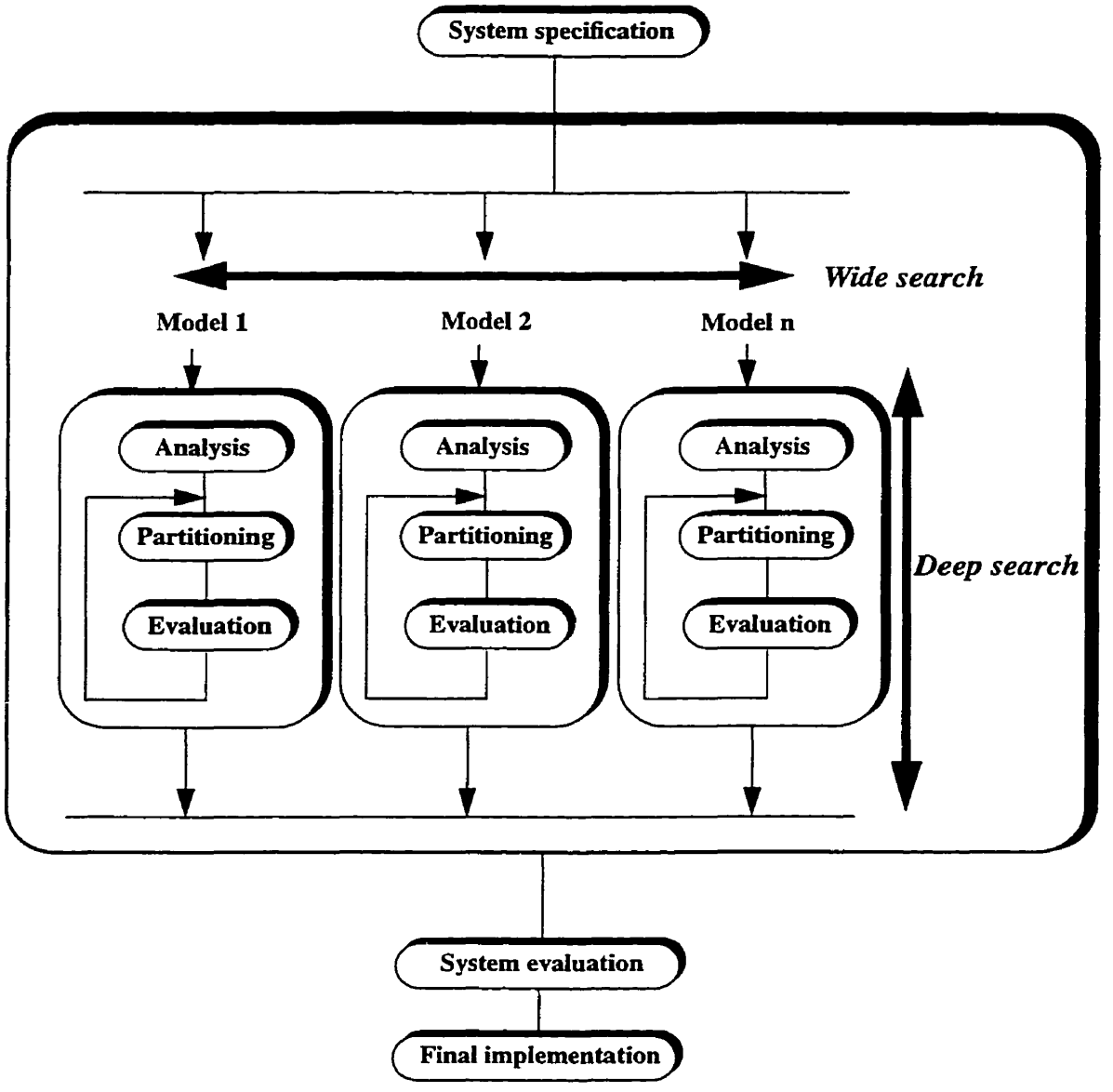


Figure 5.6. The search directions in the codesign space exploration.

6

CASE STUDY AND RESULTS

In this chapter, we will show a DSP example and a network simulation algorithm codesigned on a specific architecture to show the codesign results obtained by our approach for two examples with different complexity. The FFT transform is a DSP example while the power network simulation algorithm is a complex algorithm used to simulate real power networks. Section 6.1 presents the resources involved in the target architecture. The considered examples, the FFT algorithm and the power network simulation algorithm have been codesigned on the target architecture.Two sections are used to describe the results obtained for each one of the two case studies, sections 6.1 and 6.2. Each section contains the following subsections: the high-level description of the example, its hierarchical and variable-grain modeling, its performance estimates and finally the different codesign alternatives found for the example using our partitioning procedure. These case studies show the efficiency of our technique and the wide range of alternatives it is capable to provide for each

input design.

6. 1. The target architecture

In our case, systems to be codesigned are intended to be implemented on the architecture in Figure 6.1.

The target architecture includes a DSP standard processor with a specific SIMD processor. The codesign process has to partition the application algorithm into two execution codes, one for the DSP processor (software) and the other for the custom SIMD processor (hardware). This kind of codesign may be considered as pseudo-hardware/software partitioning and our technique is enough general to perform it.

This architecture is based on two types of processors as shown in Figure 6.1. The Texas Instruments DSP processor TMS320C40 [Texas92] is used as the master processor and the custom SIMD processor PULSE [Marriot98] as the slave processor. PULSE is dedicated to massive data processing which are very common in high-speed DSP applications. Internally, PULSE is a SIMD processor with four parallel processing units. This processor is a custom circuit developed at Ecole Polytechnique of Montréal and is intended to run concurrently with the C40 processor in order to accelerate time consuming DSP applications. PULSE has very strong parallel instructions and communications features within one processing unit as well as between different processing units. The two processors communicate data via a program memory and a local data memory or via asynchronous communication ports, as shown in Figure 6.1.

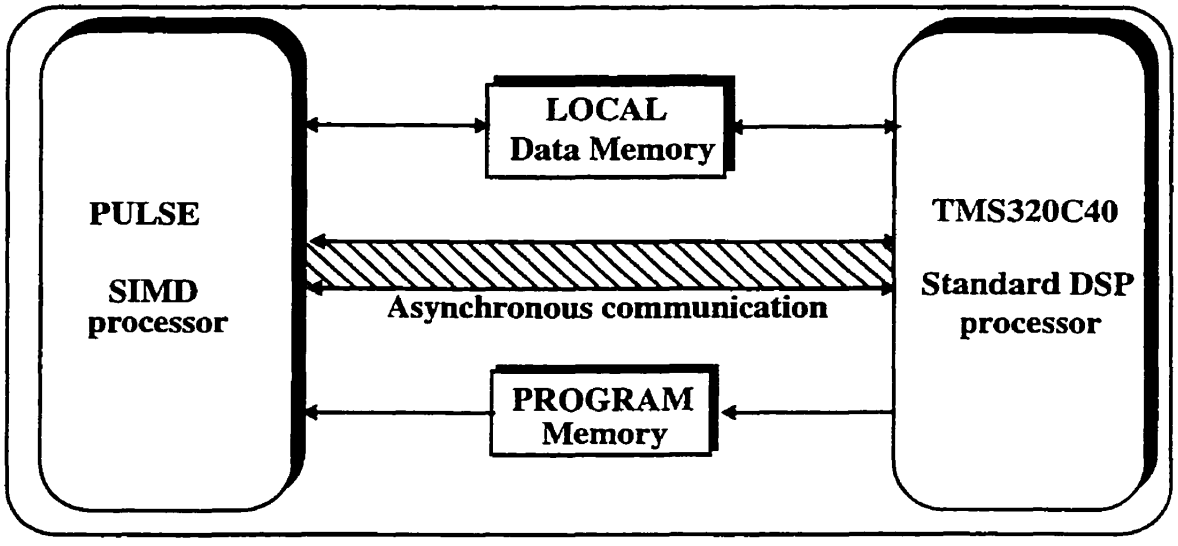


Figure 6.1. The codesign target architecture.

6. 2. The FFT example

The FFT transform is a function which is often used in the DSP field and is one of the basic functions in signal and image processing. It allows the transformation of a temporal function to a frequency one in order to make the function analysis much easier than using temporal functions. In the next sections, we show the high-level C description of the FFT, its hierarchical model, performance estimates and finally the partitioning results obtained when this DSP transform is codesigned on the architecture described in Figure 6.1.

6.2.1 The high-level description of the FFT transform

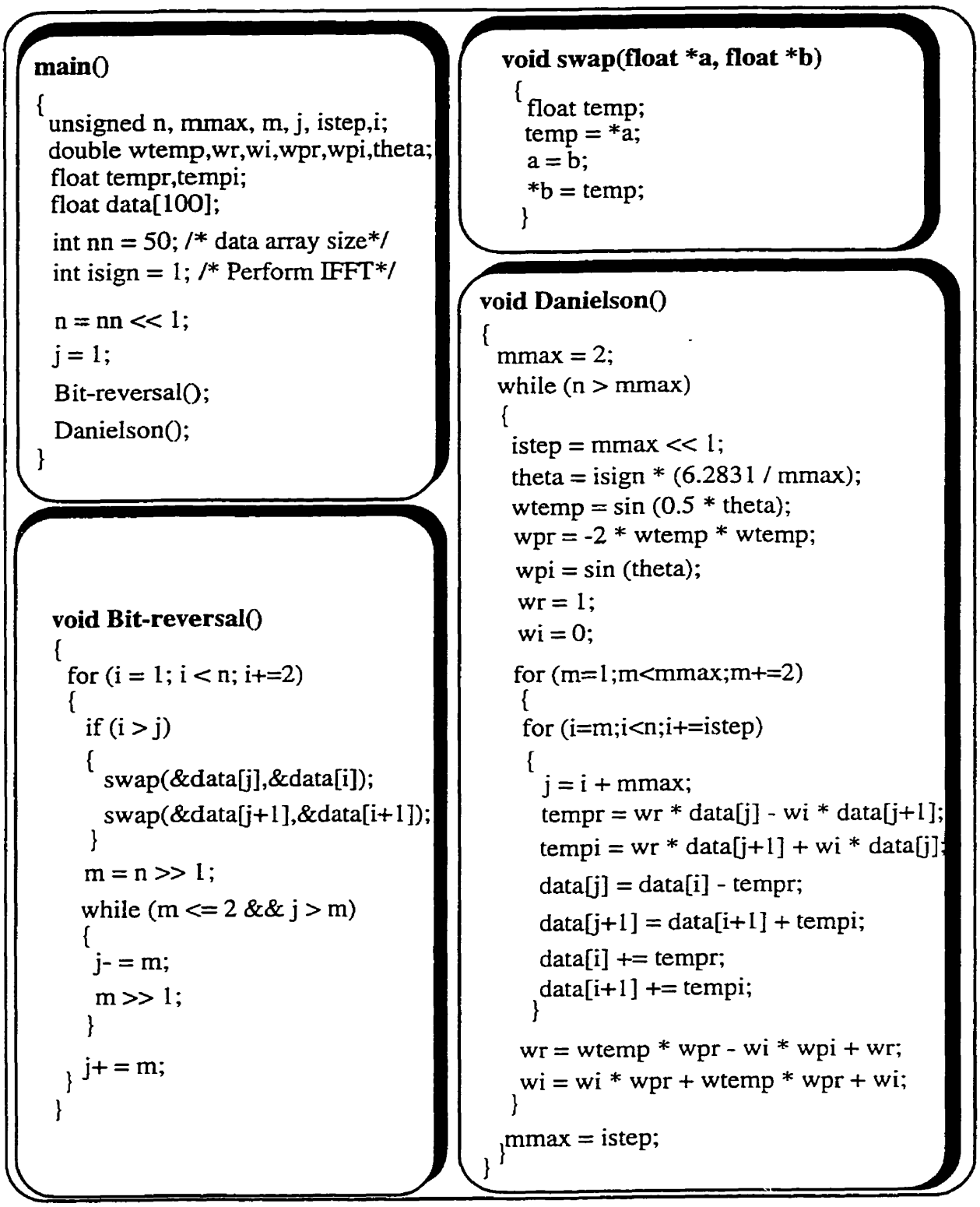
The C program of the FFT transform is shown in Figure 6.2. The inputs are the data array (data), the data array dimension (nn) and the kind of transform FFT or Inverse FFT (isign). The output is the output data array (data). The function involves many computations and requires rapid implementations [Christopher92].

The main program includes the declarations of the variables and calls to the functions

bit-reversal and danielson. The function bit-reversal reverses the positions of the values in the data array according to the desired transform. The data are reversed if the inverse FFT is performed. This function calls a simple function swap(a, b) which swaps two input values a and b.

The main function in the algorithm is danielson() and it corresponds to a fast technique to perform the FFT transform. It is constituted of a main loop and two nested loops. The main loop contains the two nested loops and a set of data preparation and update statements. These preparation steps provide the values of the FFT coefficients at each iteration of the loop. The FFT coefficients are not stored in a memory but are computed iteratively and when needed. The nested loops compute the output values of the FFT for the input data and provides the result in the same array.

The presented C program shows some hierarchical characterisitics. Indeed, all the nested structures have implicit hierarchical structure and some dependent instructions may be grouped into a task structure. In the next section, we show the hierarchical model of the FFT transform.



```
main()
{
 unsigned n, mmax, m, j, istep,i;
 double wtemp,wr,wi,wpr,wpi,theta;
 float tempr,tempi;
 float data[100];
 int nn = 50; /* data array size*/
 int isign = 1; /* Perform IFFT*/
 n = nn << 1;
 j = 1;
 Bit-reversal();
 Danielson();
}
```

```
void swap(float *a, float *b)
 {
  float temp;
  temp = *a;
  a = b;
  *b = temp;
 }
```

```
void Bit-reversal()
{
 for (i = 1; i < n; i+=2)
 {
  if (i > j)
  {
   swap(&data[j],&data[i]);
   swap(&data[j+1],&data[i+1]);
  }
  m = n >> 1;
  while (m <= 2 && j > m)
  {
   j- = m;
   m >> 1;
  }
  j+ = m;
 }
}
```

```
void Danielson()
{
 mmax = 2;
 while (n > mmax)
 {
  istep = mmax << 1;
  theta = isign * (6.2831 / mmax);
  wtemp = sin (0.5 * theta);
  wpr = -2 * wtemp * wtemp;
  wpi = sin (theta);
  wr = 1;
  wi = 0;
  for (m=1;m<mmax;m+=2)
  {
   for (i=m;i<n;i+=istep)
   {
    j = i + mmax;
    tempr = wr * data[j] - wi * data[j+1];
    tempi = wr * data[j+1] + wi * data[j];
    data[j] = data[i] - tempr;
    data[j+1] = data[i+1] + tempi;
    data[i] += tempr;
    data[i+1] += tempi;
   }
   wr = wtemp * wpr - wi * wpi + wr;
   wi = wi * wpr + wtemp * wpr + wi;
  }
  mmax = istep;
 }
}
```

Figure 6.1. The FFT transform C program.

6.2.2 The FFT hierarchical modeling

Figure 6.3 shows the FFT transform as a hierarchy of interacting and dependent blocks using the modeling technique shown in chapter 3. At level 1, the main FFT block is decomposed into 3 blocks, the Initialization module, the Bit reversal module and the Danielson control module which is the main processing task in the FFT transform.

At the next level of the hierarchy, each of the previous modules is decomposed into 2, 3 and 2 subblocks respectively. Note that the initialization module is made up, at this level, of two blocks, initialize variables and initialize data. The variable initialization block is not decomposed, at the next level, because it contains only basic operations for which no further decomposition is possible, while the data initialization loop body is decomposed into 3 blocks, Initialize the index, Read the indexed data and increment the index.

The blocks can be decomposed in this way until the basic operations are reached. The initialize block has a three levels of hierarchy and the bit-reversal block has four levels of hierarchy. The most complex block, Danielson, has eight levels of hierarchy. Thus, the FFT hierarchical modeling provides eight levels of hierarchy and 39 blocks.

During the codesign of such a function, the designer may start to find the best partitioning only at level 1. In this case, the partitioning algorithm handles only three functional and interdependent blocks. Each block is assigned to hardware or to software based only on its performance estimation already computed. Dealing with all the block details and basic operations is not required.

If no partitioning that satisfies the performance requirements is found, the number of blocks may be increased by moving to level 2. At this level, seven (7) blocks build the system model. At the last level, level 8, twenty four (24) blocks build up the FFT model.

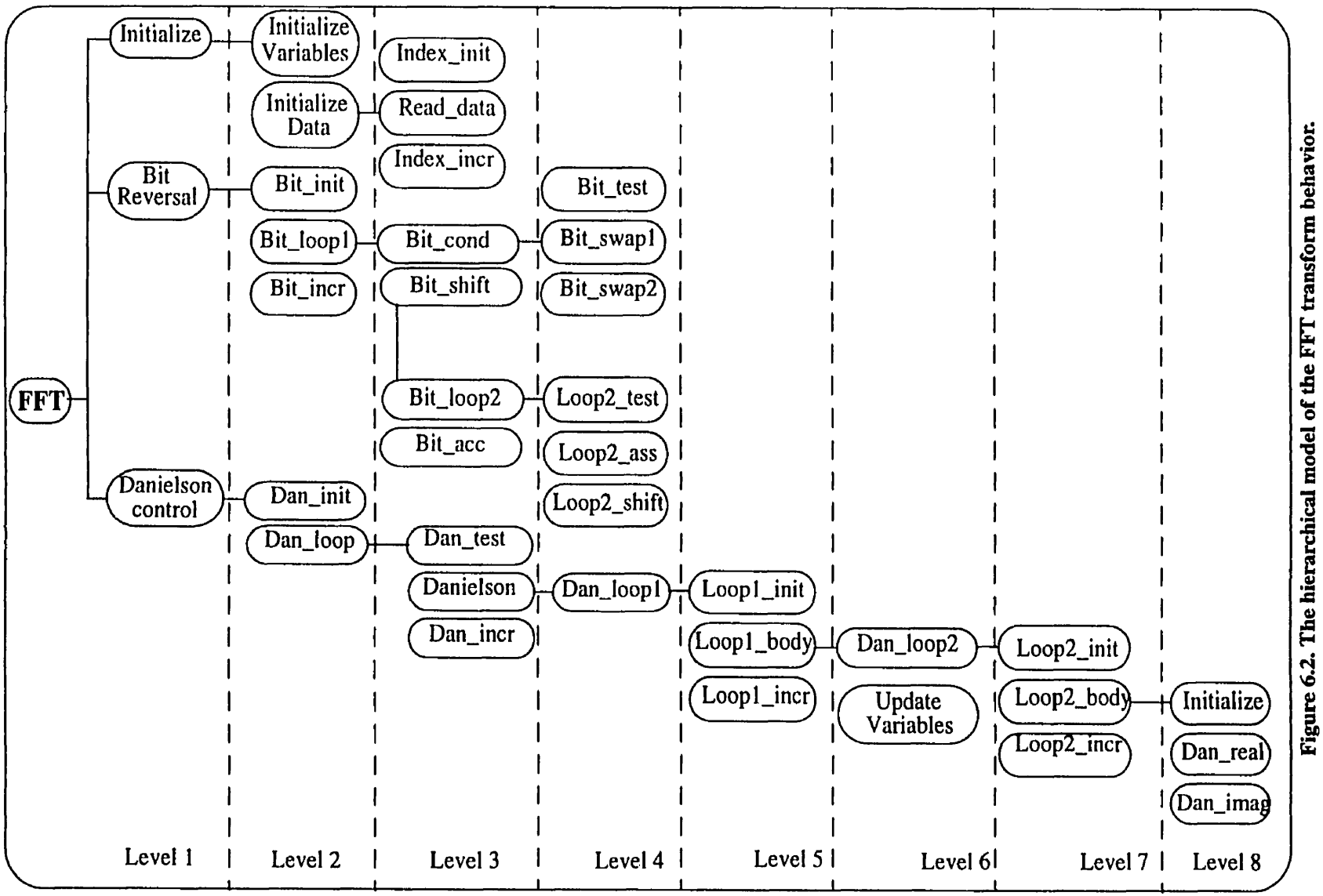


Figure 6.2. The hierarchical model of the FFT transform behavior.

6.2.3 The performance estimation and scheduling

Table 6.1 shows the execution time and the code size of the different blocks in the FFT model when each of them is run on the processors C40 and PULSE.

The first four rows in Table 6.1 show the execution time and the code size of the three blocks that build the FFT at level 1. The remaining rows show the performance estimation of some blocks from other levels in the hierarchy. Note that the Danielson block is the slowest block in the FFT behavior because it is the most complex and it contains two nested data processing loops, Dan-loop1 and Dan-loop2. This performance estimation step allows the identification of the bottlenecks in the FFT behavior.

Table 6.1: Performance estimation of the FFT blocks.

Module	Execution time (ms)		Code size (Bytes)	
	PULSE	C40	PULSE	C40
Initialize	1.48	2.28	184	92
Bit-reversal	3.6	7.04	432	192
Danielson	14.16	26.36	992	440
Initialize-data	0.96	1.92	80	40
Bit-loop1	1.4	2.72	368	160
Dan-loop1	12.52	23.12	912	408
Dan_loop2	10.4	20.84	736	312

The next step determines the data dependencies between the FFT blocks. This consists first in determining the list of successors and predecessors for each block in the model and then scheduling the blocks ASAPly and ALAPly to identify the possible critical paths in the FFT execution flow.

Figure 6.3 shows the FFT transform data flow graph at two different levels of hierar-

chy.

The first data flow graph, Figure 6.3(a), is built of three nodes. Only one critical path is possible because the nodes are data dependent and must execute sequentially. The second data flow graph, Figure 6.3(b), is built of seven blocks. The ASAP and ALAP scheduling provides the critical path whose execution flow is constituted by the following sequence of blocks, Initialize-data, Bit-loop1 and Dan-loop.

These graphs are used to determine the total execution time of the FFT when the blocks are assigned to PULSE or to C40.

For example, in the initial partition, the complete FFT behavior is assigned to the processor C40 and the total execution time is the sum of all the block execution times. The concurrency cannot be used for the initial partition because the C40 processor executes the instructions sequentially even if they are independent. We may take advantage of the concurrency when the concurrent blocks are assigned to two different partitions or processors.

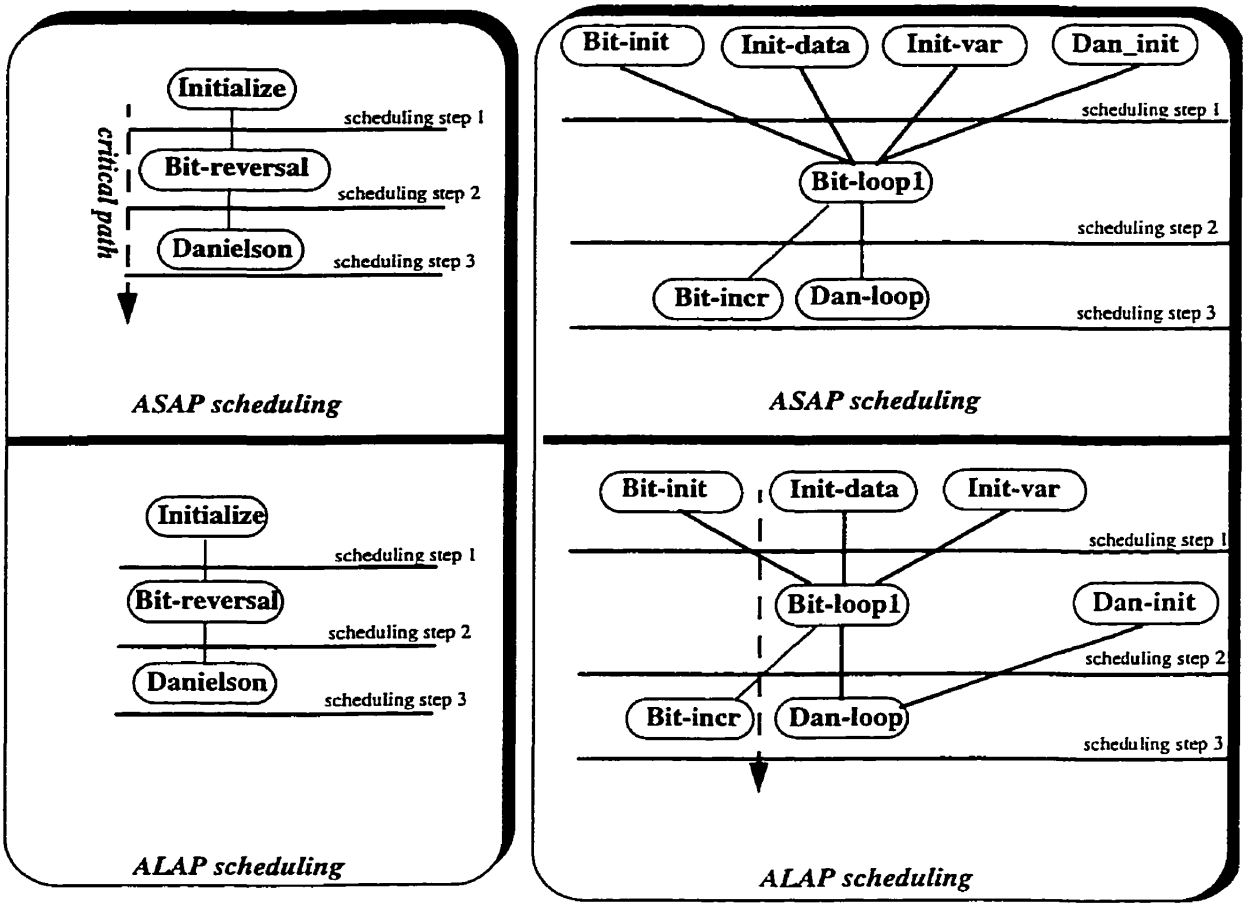
In the next section, we present the different partitioning alternatives for the FFT transform.

6.2.4 Partitioning alternatives

Here, a typical signal processing function, the FFT transform, is used to show the proposed codesign framework results. The FFT transform program has been codesigned on the architecture shown in Figure 6.1. Three sets of results have been generated to validate our methodology.

The first set shows the codesign results under performance constraints. The second set of results shows codesign determined for the same input system but at different levels

of the hierarchy. Finally, the third set of results is a comparison between the obtained code-sign implementations and the complete software or hardware implementations.



(a). Scheduling at level 1 **(a). Scheduling at level 2**

Figure 6.3. ASAP/ALAP scheduling at two different levels of the FFT hierarchy.

6.2.4.1 Partitioning under time constraints

The analysis values for the FFT subblocks have been already shown in Table 6.1. Based on these parameters and using our heuristic partitioning procedure, the first set of the FFT transform partitioning results is shown in Table 6.2. Table 6.2 shows co-implementation alternatives generated for the FFT transform when different timing constraints are specified.

The FFT transform is considered at the level 1 of the hierarchy. If the constraint is completely relaxed (column 1 in Table 6.2), which means that the constraint is equal or greater than the complete software solution, the complete C40 solution is adopted without any partitioning. The more the constraint is decreased the more blocks are moved to PULSE to accelerate the execution.

For example, to reach the 25 ms constraint (column 4 in Table 6.2), two blocks are executed on PULSE (Danielson and Bit_reversal) and one block on C40 (Initialize). When the timing constraint is equal to 20 ms (column 5 in Table 6.2), the three blocks are assigned to PULSE. This set of results shows the first characteristic of our framework, codesign with input time constraints.

Table 6.2: The FFT transform partitioning under timing constraints

Modules	C1= 40 ms	C2 = 35 ms	C3 = 25 ms	C4 = 20 ms
Initialize	C40	C40	C40	PULSE
Bit-reversal	C40	C40	PULSE	PULSE
Danielson	C40	PULSE	PULSE	PULSE
C40-time	38.2 ms	11.84 ms	5.80 ms	0 ms
PULSE-time	0 ms	14.16 ms	18.15 ms	19.24 ms
Total time	38.2 ms	26 ms	23.95ms	19.24 ms

6.2.4.2 Partitioning alternatives at different levels of the hierarchy

In the next group of results, we considered the behavior at different levels of hierarchy, from 1 to 7 with the same timing constraint. These partitioning results are shown in Table 6.3.

At level 1, all initialization and input operations (Initialize blocks) are implemented on the TMS320C40 processor, while data preprocessing and processing operations (Bit_reversal and Danielson blocks) are assigned to the PULSE processor. At level 2, only the data processing parts in both Bit_reversal and Danielson blocks are assigned to PULSE. At level 3, only the Dan-loop algorithm in the Danielson block with bit swapping blocks in the Bit_reversal module are assigned to PULSE. At level 7, only the data processing blocks related to bit swapping functions and the multiplication of complex numbers are assigned to PULSE.

The three last columns in Table 6.3 show the execution time for each proposed solution from level 1 to 7. These values show how the computation load is distributed between the two available processors as a function of the level of hierarchy. At levels 1 and 2, the obtained performance satisfy largely the input constraint but PULSE is taking all the computation load. Note that this will lead in more memory size since the instruction size in PULSE is twice the C40 one (66 and 32 respectively). The same thing is observed for levels 6, 7. At the middle levels, 4 and 5, the partitioning is more balanced than at other levels. The resulting implementation performance is very close to the constraint but the memory size is well balanced.

6.2.4.3 Partitioning tradeoff

The last group of results provide a comparison between the complete hardware and the complete software solutions. In Table 6.4, the partitions obtained at level 1 and level 7 are compared to complete C40 implementation and complete PULSE implementation. Note that the system performance is 20.44 ms if implemented on PULSE and 38.64 ms if imple-

mented on C40. An intermediate implementation where all initialization and read operations are assigned to C40 and all data preprocessing and processing operations are assigned to PULSE has an execution time equal to 23.95 ms (solution#1 in Table 6.4). The last implementation proposed, where only processing operations are assigned to PULSE, leads to an execution time equal to 23.84 ms (solution#7 in Table 6.4).

The solution #1 reduces the execution time by 40% of the complete C40 implementation execution time while the memory size is reduced by 23% of the memory size required by the complete PULSE implementation. In solution #7, the execution time is reduced by 38% while the memory size is reduced by 65%. The two solutions satisfy the input constraint equal to 25 ms but solution #1 needs three times the memory size needed by solution #7. This shows that a trade-off has to be found when the memory size is taken into account. In this section, we presented the FFT transform behavior to show the ability of our tool to find co-implementations using the proposed algorithm. This algorithm allows the generation of a list of possible implementations when different levels of hierarchy are considered.

Table 6.3: Block assignment at different hierarchical levels of the FFT model.

Level	Nb. of Blocks	C40	PULSE	Time(ms) / time constraint = 25 ms]		
				PULSE	C40	Total
1	4	2	2	18.14	4.8	22.94
2	8	6	2	18.8	2.96	21.76
3	19	11	8	15.56	9	24.56
4	29	18	10	14.68	10.24	24.92
5	36	17	13	14.56	10.4	24.94
6	47	40	7	6.82	17.72	24.54
7	52	40	12	7	17.92	24.52

The objective of exploring the major points in the codesign space has been reached by varying the level of hierarchy. This means that our tool is able to find as many implementation possibilities as hierarchy depth allows, which is a new and advantageous feature compared to previous works.

The generated alternatives are compared to the lower-bound performance (the hardware solution) and the upper-bound performance (the software solution) implementations in order to find the best trade-off. The final decision may be taken by the designer or by an automatic tool to be developed.

Table 6.4: Alternative comparison for the FFT transform.

	Execution time (ms)			Code size (Bytes)		
Partition	PULSE	C40	Reduction	PULSE	C40	Reduction
1 (C40)	0	38.64	----	0	1260	----
2 (PULSE)	20.44	0	----	2196	0	----
solution #1	18.15	4.80	40%	1424	264	23%
solution #7	5.2	18.64	38%	352	412	65%

6. 3. The power network simulation algorithm

The power network simulation algorithm is a very time consuming algorithm used by power network companies to simulate their electrical networks. Various and long processing involving a lot of vector and matrix computations are required in such simulations. In the next sections, we show the high-level matlab description of the algorithm, its hierarchical model, performance estimates and finally the partitioning results obtained when this algorithm is codesigned on the architecture described in Figure 6.1.

6.3.1 The high-level description of the power network simulation

Figure 6.4 shows the matlab description of the power network simulation algorithm.

The algorithm is composed of three main pieces. The first one is the data and constant preparation phase. The second one is a list of matrix computations and finally the computation of the vectors Ug, Ig and Vb which are the main charcateristics of the network that we want to determine. Theses vectors are computed and updated at each simulation iteration. The simulation algorithm is then executed into two nested loops. The first loop reiterates 600 times which corresponds to 5 seconds in the network real lifetime. The network characteristics, Ug, Ig and Vb, have to be computed each 5 seconds. The second loop is the convergence loop for the characteristics values of the network. These characteristics have to be determined at a stable state and this stable state corresponds to the given precision. When the characteristics values do not change from one iteration to another by more than the precision value, the precision loop is stopped and the current values of the network characteristics are stored. In the next section, we show the hierarchical modeling of such a behavior.

```
fid = fopen('data_in.bin','r');
Ug = fread(fid, 21, int32);
Vb = fread(fid, 18, int32);
Ig = fread(fid, 6, int32);
invAg = fread(fid, [21,21], int32);
invAgRgu = fread(fid, [21,21], int32);
invAgRgk = fread(fid, 21, int32);
invYeifg = fread(fid, [18, 6], int32);
invYepfg = fread(fid, [18, 6], int32);
fclose(fid);

Vbm = [Vb(1:2) Vb(3:4) Vb(5:6) Vb(7:8) Vb(9:10)
       Vb(11:12) Vb(13:14) Vb(15:16) Vb(17:18) ]';
Ugm = [Ug(1:7) Ug(8:14) Ug(15:21)]';
res(1,:) = [Vbm(5,:) Ugm(2, 2) Ugm(2, 1) Ugm(1, 2)];
invYeg = invYeifg;
For k =1 : 600
 If (k ==11) invYeg = invYepfg; end
 invAgRg = invAgRgu * Ug + invAgRgk;
 oPe = zeros(3,1);
 dPe = ones(3,1);
 While (max(abs(dPe)) > 1E-2)
    Dg = Ugm(:, 2);
    Dg = Dg - fix(Dg/(2*pi)) * (2*pi);
    Dg = Dg - fix(Dg/pi) * (2*pi);
    sgnsncs = [ sign(Dg) sign((pi/2) - abs(Dg))];
    Dg = abs(fix(Dg/(pi/2)) * pi - Dg);
    chgsncs = fix(Dg/(pi/4));
    Dg = abs(chgsncs * (pi/2) - Dg);
    Dgp2 = Dg .* Dg;
    Dgp3= Dgp2 .* Dg;
    Dgp4 = Dgp3 .* Dg;
    Dgp5 = Dgp4 .* Dg;
    sncsDg = [ones(3,1) Dg Dgp2 Dgp3 Dgp4 Dgp5] *
             [0 1 0 -1/6 1/120; 1 0 -1/2 - 1/24 0]';
    If chgsncs(1)
       sncsDg(1, :) = sncsDg(1, 2:-1:1); end
    If chgsncs(2)
       sncsDg(1, :) = sncsDg(2, 2:-1:1); end
    If chgsncs(3)
       sncsDg(1, :) = sncsDg(3, 2:-1:1); end

    sncsDg = sgnsncs .* sncsDg;
    Bg1 = [0 0 0 0 0 1 0;
           [0 0 0 0 0 0 1]' * sncsDg(1,1) +
           [0 0 0 0 0 0 1;
           [0 0 0 0 0 -1 0]' * sncsDg(1,2);
    Bg2 = [0 0 0 0 0 1 0;
           [0 0 0 0 0 0 1]' * sncsDg(2,1) +
           [0 0 0 0 0 0 1;
           [0 0 0 0 0 -1 0]' * sncsDg(2,2);
    Bg1 = [0 0 0 0 0 1 0;
           [0 0 0 0 0 0 1]' * sncsDg(3,1) +
           [0 0 0 0 0 0 1;
           [0 0 0 0 0 -1 0]' * sncsDg(3,2);

    Bg = [ Bg1 zeros(7,2) zeros(7,2)
           zeros(7,2) Bg2 zeros(7,2)
           zeros(7,2) zeros(7,2) Bg3];

    Cg = Bg';
    Ig = Cg * invAgRg;

    Vbm = [Vb(1:2) Vb(3:4) Vb(5:6) Vb(7:8)
           Vb(9:10) Vb(11:12) Vb(13:14)
           Vb(15:16) Vb(17:18)]';

    Ug = invAg * Bg * Vb(1:6) + invAgRg;

    dPe =Ugm(:, 3) - oPe;
    oPe =Ugm(:, 3);
 end %precision loop
 res(k+1,:) = [Vbm(5,:) Ugm(2, 2)
               Ugm(2, 1) Ugm(1, 2)];

end %iterative loop

fid = fopen('data_out.bin','w');
fwrite(fid,res, 'int32');
fclose(fid);;
```

Figure 6.4. The matlab program of the network simulation algorithm.

6.3.2 The hierarchical modeling of the power network simulator

Figure 6.5 shows the power network simulation algorithm as a hierarchy of interacting and dependent blocks using the modeling technique shown in chapter 3. At level 1, the simulation algorithm is decomposed into 2 blocks, the data preparation module and the network simulation module which is the main processing task in the simulation algorithm.

At the next level of the hierarchy, each of the previous modules is decomposed into 2 subblocks. Note that the data preparation module is made up, at this level, of two blocks, vector initialization and coefficient initialization. The vector initialization block is decomposed, at the next level, into 3 subblocks corresponding to the initialization of the three vectors Ug, Ig and Vb. The coefficient initialization is decomposed into 2 blocks, one block for the initialization of the network generator coefficient and the other block for the initialization of the network admittance coefficient.

The blocks can be decomposed until the basic operations are reached. The vector initialization block has a two level hierarchy and the coefficient initialization block has a three level hierarchy. The most complex block, Net simulation, has an eight level hierarchy. This hierarchy level is the result of the successive decompositions on the nested loops until the basic operations are reached. Thus, the network simulation algorithm has a hierarchical model with eight levels and a complexity equal to 94 blocks

During the codesign of such a function, the designer may start to find the best partitioning at level 1. In this case, the partitioning algorithm handles only two functional and interdependent blocks. Each block is assigned to hardware or to software based only on its performance estimation already computed. Dealing with all the block details and basic operations is not required unless the design performance constraints are not satisfied.

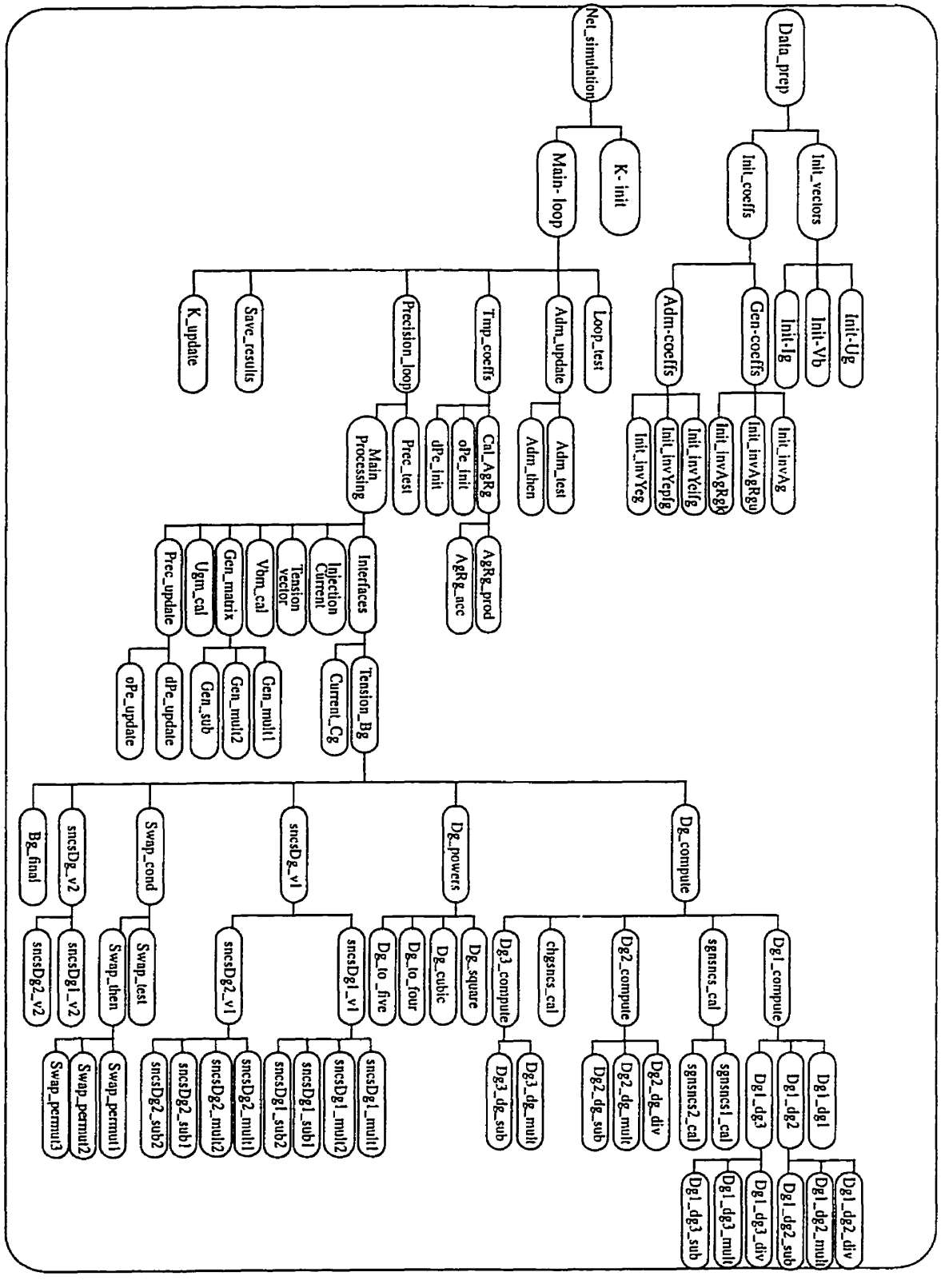


Figure 6.5. The hierarchical model of the network simulation behavior.

6.3.3 Performance estimation

Table 6.5 shows the execution time and the code size of the different blocks in the network simulation algorithm when each of them is run on the processors C40 and PULSE.

The execution time of some blocks of the simulation algorithm are presented in columns 2 to 4 in Table 6.5. The initialization blocks are very small and do not take a long time to execute as shown in Table 6.5, columns 3 and 4. Blocks like the main loop and the precision loop are the more complex and take a lot of time to execute. For example, the main loop takes 978,000 cycles to execute its 600 iterations. One iteration of the precision loop lasts 889,000 cycles while the loop may reiterate several times before reaching the desired precision. Only one iteration of the precision loop is considered because the number of iterations is unknown. This performance estimation step allows the identification of the critical regions in the simulation algorithm. These are clearly, the main loop at level 2 of the hierarchy, the precision loop at level 3, the main processing block at level 4, and interfaces block at level 5.

Columns 5 to 7 in Table 6.5 show the number of micro-instructions needed by each block and the corresponding memory size required for each processor. The main loop is built of 145 instructions which correspond to 580 bytes on the C40 processor and to 1305 bytes on the PULSE processor. The gen-matrix block is built of 7 instructions and the corresponding memory size is only 28 bytes on C40 and 163 on PULSE. Note that the Init-coeffs block needs much more instructions even if it takes less execution time than the gen-matrix block. This is due to the large number of sequential instructions in the Init-coeffs block while the gen-matrix block is built of only 7 instructions but in a 21 iteration loop. This values show the time-memory tradeoff we have to take into account.

Table 6. 5: The performance estimation for the power network simulator blocks.

Module	Nb.cycles	Execution time (ms)		Nb. instructions	Code size (Bytes)	
		PULSE	C40		PULSE	C40
Init_vectors	45	0.008	0.018	10	90	40
Init_coeffs	140	0.025	0.56	21	189	84
Main_loop	978,000	17.6	39.12	145	1305	580
Tmp_coeffs	54,000	2.16	0.97	10	90	40
Precision_loop	889,000	16.2	36	111	999	444
Interfaces	340,200	7.74	17.2	87	783	348
Gen_matrix	369,000	6.64	14.76	7	163	28

6.3.4 Partitioning alternatives

Table 6.6 shows co-implementation alternatives generated for the network simulation at different levels of hierarchy, from 1 to 10 with the same timing constraint.

At level 1, all initializations and data preparing steps (Data_prep) are implemented on the TMS320C40 processor, while data preprocessing and processing operations (Net_simulation) are assigned to the PULSE processor. At level 2, only the data processing blocks in the net_simulation block are assigned to PULSE. At level 3, the main_processing block is assigned to PULSE. Finally, at level 9, only data processing blocks are assigned to PULSE. The three last columns in Table 6.6 show the execution time for each proposed solution from level 1 to 10. These values show how the processing charge is distributed between the two available processors as a function of the level of hierarchy. At levels 1 to 4, the obtained performance satisfy largely the input constraint but PULSE is taking all the computation load. Note that PULSE solutions require more memory size since the instruction size in PULSE is more than twice the C40 one (66 and 32 bits respectively). In levels, 5 to 10, the partitioning is balanced by moving less blocks towards the processor PULSE

than at the first levels. The resulting implementation performance is still satisfying the same constraint with very close values and the memory size is reduced. The performance values of the final partitions, for each level in the hierarchy, are provided to the designer in order to select the appropriate implementation.

Table 6. 6: Block assignment at different hierarchical levels of the network simulation algorithm

Level	Nb. of Blocks	C40	PULSE	Time(ms) / time constraint = 30 ms]		
				PULSE	C40	Total
1	2	1	1	17.64	0.08	17.64
2	4	3	1	17.6	0.074	17.6
3	12	10	2	16.2	3.127	18.53
4	20	15	5	15.74	3.05	17.83
5	27	17	10	5.06	27.76	29.93
6	31	22	9	4.64	27.41	29.29
7	36	27	9	4.06	28.68	27.04
8	46	31	15	4.06	28.35	27.04
9	60	46	14	4.06	28.01	27.04
10	64	55	9	4.06	27.96	27.04

Table 6.7 shows, with more details, some of the partitioning obtained at different levels of the hierarchy. For each one of these alternatives, we provide the performance estimation which is the execution time of the complete system and the code size corresponding to the required data memory space.

The system performance is 17.80 ms if the system is completely implemented on PULSE and 39.12 ms if implemented on C40. An intermediate implementation where all initialization and output operations are assigned to C40 and all data preprocessing and processing operations are assigned to PULSE has an execution time equal to 28.68 ms (solution

#1). The last implementation proposed, where only processing operations are assigned to PULSE, leads to an execution time equal to 27.41 ms (solution #2). These two solutions are found for the same input constraint equal to 30 ms. Both solutions satisfy this timing constraint but the difference is the code size needed to implement each one of them.

The solution #1 is approaches the input constraint by 9% while the required memory size is reduced by 45% of the memory size required by the complete PULSE implementation. The solution #2 approaches the timing constraint by 4% and the required memory size is reduced by 75%. The two solutions satisfy the input constraint equal to 30 ms but the solution #1 needs 1.67 times the memory size needed by the solution #2. Of course, the solution #1 is a better solution when the comparison is based on the timing performance but for the same input constraint, solution #2 is better than solution #1 because the desired performance is obtained with less area cost.

Table 6. 7: Codesign alternatives for the network simulation algorithm

	Execution time (ms)			Code size (Bytes)		
Partition	PULSE	C40	Solution Precision	PULSE	C40	%Area
1 (C40)	0	39.12	----	0	772	----
2 (PULSE)	17.64	0	----	3088	0	----
solution #1	4.64	27.41	9%	1424	264	45%
solution #2	4.06	28.68	4%	352	412	75%

6. 4. Summary

These case studies provide the following main conclusions on the impact the hierarchy has on the generated partitioning alternatives.

1. *An optimal level in the hierarchy may be identified to obtain an optimal and balanced partitioning of blocks between hardware and software partitions.*

2. *The use of the most detailed model, the operation level, does not always mean obtaining the best solution.*

3. *Considering the first levels in the hierarchy, during partitioning, improve considerably the time performance but this is not the case for the code size. This is due to the fact that big blocks are moved from software to hardware when using first levels of the hierarchy.*

4. *The use of the medium and the last levels may decrease considerably the code size or the area with a very little degradation in performance while the use of first levels reduces the execution time but requires more memory storage space.*

7

CONCLUSIONS

The thesis studies a systematic approach to partition system-level designs into hardware and software. The key contributions are summarized in section 7.1. In section 7.2. we conclude with a discussion of some of the future directions to this research.

7. 1. Contributions

In this thesis we developed techniques for the codesign of digital systems. Pure hardware or software implementations often cannot meet constraints like cost, performance or time-to-market. Due to the algorithmic complexity of systems and also to avoid early commitment to a particular hardware or software implementation, the systems are specified at a high-level of abstraction.

Our approach to codesign is to design the hardware and the software in parallel with feedback and interaction between the two as the design progresses. The codesign approach

enables the exploration of a wide variety of implementation alternatives simply by using the available hierarchical model of the input system. Thus, the system can be optimized in its entirety.

Five key problems are identified in the context of system-level codesign: modeling, partitioning, analysis, synthesis and simulation. We provide new and original solutions to the three former problems and these solutions are summarized in the next sections.

At the system-level, designs are typically represented as task graphs, where tasks have moderate to large granularity already fixed at the specification level even if the input specification is hierarchical. We present a modeling technique for system level tasks with variable granularity. The key contribution is summarized in section 7.1.1.

Each task in the system specification can be implemented in hardware or in software. The resulting implementation typically differ in area and execution time. The objective in system level design is to select the "best" implementation for the system as a whole. We presented an automatic approach to solve this problem by formulating the partitioning problem as a dependency graph partitioning problem. The key contributions are summarized in section 7.1.2.

In order to select the best implementation, one needs efficient estimation tools to compare the possible implementations. The estimation avoids the complex and time consuming synthesis steps by providing fast and efficient evaluations. The key aspects of the estimation steps are summarized in section 7.1.3.

7.1.1 Hierarchical modeling

The use of hierarchy as presented in our codesign approach is an original contribution

of this thesis. We are not aware of any other work that attempts to do this at the system-level even if the majority of systems are described using hierarchical modeling. All the proposed systems use a flattened model of the input system at the task level or at the operation level. We proposed a solution which takes advantage of the hierarchical model at the input.

The way we use the hierarchy provides the flexibility to select a given complexity. Each level in the hierarchy provides a different input model. The models have large complexity when fine grain models are reached (hierarchy flatten) but these models are not used unless no satisfying solution is found with the simple models. Our contribution has three main advantages:

1. The number of models with different complexity that the hierarchy provides allows the expansion of the design space exploration without any computation overhead because the hierarchy is a charactersitic already available in all modeling tools.
2. The model complexity used during partitioning may be selected among the range of possible models provided by the use of hierarchy. The simple models are used first and when no satisfying solution is found, more complex models may be considered.
3. The possibility to use only simple models as long as the found solutions satisfy the constraints allows the reducing of the overall design CPU time.

7.1.2 System partitioning

Hardware/software partitioning is the problem of determining, for each node in the application, a hardware or a software mapping and schedule for execution. The end-objective is to minimize the total hardware area subject to timing constraints. Since the problem

is known to be NP-Hard, we developed an efficient heuristic.

We presented the proposed heuristic to solve the hardware/software partitioning problem. This heuristic has several unique features:

1. The mapping selection of a task is performed such that the finish time of the task is minimized. The task is selected using a global-time criterion.

2. In addition to global consideration, neighborhood preference is taken into account. The next tasks to be selected are those which have strong data dependency with the global selected task.

3. The global selection has the main objective of minimizing the execution time in order to satisfy the timing constraint. The local preference has the main objective of minimizing the communication cost between the hardware and the software partitions.

4. The weighted dependency graph used to formulate the partitioning problem uses a global formulation where the graph nodes are the application tasks. The graph nodes are weighted by their execution time while the graph edges are weighted by the number of shared data between the two tasks in the graph edge. All the weights are determined from the estimated performance and cost.

5. The partitioning heuristic takes into account all the possible concurrency and communication overhead between the hardware and the software partitions before proposing the "best" implementation.

6. Our proposed heuristic is computationally efficient ($O(N^2)$) when compared to the $O(2^N)$ theoritical complexity. N is the number of tasks in the system model.

7.1.3 System analysis

Behavioral analysis is the problem of determining fast performance and area estimates for the whole system to ease the decision making step during the automatic hardware/ software partitioning. We used available estimation tools but our key contribution is the extent of such estimators for our specific target architecture.

1. The specsyn estimators provide the time and area estimates for each block in the model independently of its hierarchy or of the subblocks that build it. Our key contribution is to determine the estimates hierarchically for each block. At a given level of hierarchy, the block time and area estimates include the time and area estimates of all its subblocks. During partitioning, the estimate is ready and we do not need to span the block hierarchy to find its performance estimates at each iteration.
2. The technology files for our specific architecture components have also been determined and added to the estimator data base. These technology files correspond to the two processors used in our target architecture, the C40 and PULSE processors.

7. 2. Future directions

7.2.1 Framework integration

Our contribution focuses on partitioning, analysis and modeling. These steps are a key tenet in codesign process but need to be integrated in existing or future codesign frameworks.

In order to perform such an integration, several questions need to be answered. 1. what is the specification language and the modeling concept supported? 2. If the hierarchy

is implicit, how is it extracted? 3. how the partitioning algorithm would be integrated in the available codesign tools that do not support partitioning?

All common specification languages used in industrial tools support the hierarchy because of the growing complexity of specified systems. We think that integrating our use of hierarchy in the available frameworks would be an easy task while providing a powerful feature for future codesign tools by simply taking advantage of the hierarchy already embedded in these frameworks.

The available cosynthesis tools do not include automatic partitioning facility but focuses on hardware high-level synthesis and software code generation for different processors. The integration of our automatic partitioning heuristic would be very easy because it is a completely independent module. The heuristic needs a set of dependent blocks with their performance estimates as inputs and provides, at the output, a mapping of these blocks to hardware or to software.

7.2.2 Hardware and software synthesis

The partitions found by our proposed partitioning algorithm have to be synthesized in order to provide the final implementation. High-level synthesis tools may be used to perform hardware synthesis. A good candidate for this synthesis would be the Synopsys behavioral compiler which is available at universities [Synopsys97]. As for the software synthesis, standard compilers may be used. The key issue for the future is to translate the obtained partitions to the appropriate description language, like VHDL for Synopsys and C for compilers.

7.2.3 Interface synthesis

In the proposed partitioning heuristic, we used a simple formula to determine the communication time overhead. This equation needs to be more elaborated in order to reflect the real time and also the area overhead which is not considered in the equations presented in chapter 5. An interface synthesis or an efficient interface estimation tool is required. The interface evaluation has a big impact on the final implementation and using an interface estimation or synthesis tool is a must.

Appendix A. Technolgy files for the system estimation

In this appendix, complementary information is provided for the performance estimation phase. The performance estimation has been presented in chapter 4 and is performed using two automatic estimators developed at the university of California at Irvine.

In order to use such estimators, some information has to be provided. This information is the SpecCharts description of all the behaviors we want to estimate and the technology files for our target architecture resources. In the next section, we present the SpecCharts description of one of the case studies, the FFT example. The other sections of this appendix present the technology files for both the Texas Instruments C40 processor and the SIMD processor PULSE.

A. 1. SpecCharts description

The FFT C description has been presented in chapter 6. Here, we show a SpecCharts description which is used to estimate the performance of the FFT algorithm when it is run on the C40 or PULSE processors.

The description below shows the SpecCharts description of the FFT example. This description has the same structure as VHDL and is composed of two parts. The first part is the entity where the input and output ports of the system are defined. The second part is the architecture where the functionality and the structure of the system are described. The architecture heading is composed of all the variables and constants used in the FFT behavior. The FFT behavior is composed of three main blocks, Initialize, Bit-reversal and danielson. The control flow between these three blocks is described at the beginning of the FFT architecture. For each block, a control flow is defined by the following statement:

Block name: (TOC, condition, next block)

The different fields in this statement are:

1. ***Block name***: It is the current block where the control flow is.
2. ***TOC (Terminate On Completion)***, this key word means that the control is given to another block only after the completion of the current block.
3. ***Condition*** : This key word means that the control is given to another block only if the Condition is true even if the block processing is completed. If this field is True, this means that the flow is given to the next block at the completion unconditionaly.
4. ***Next block*** is the name of the next block to which the control is given. If the current block is the last block in the control flow, the key word Stop is used to describe the end of the control flow graph.

After the definition of the control flow, each block in the flow graph is then described internally. The block may be hierarchical and thus composed of other blocks. These blocks are first described in a control flow and then their internal description is given, or may be a simple behavior described as a set of VHDL statements. In the description below, the block Initialize is of type sequential behaviors and is composed of a hierarchy of two sub-blocks, Data-initilaize and Var-initialize. An example of simple behavior is Var-initialize which is of type code and is described by a set of VHDL statements to initialize the variables n, i and j.

```
--use work.data.all;
entity FFT_E is
port
(
 input_port   : in integer;
 output_port   : out integer
);
end;
```

```
architecture FFT_A of FFT_E is
begin
 behavior FFT_behavior type sequential subbehaviors is
  type tableau is array (0 to 100) of integer;
  variable n, mmax,m,j,istep,i,nn,isign                    : integer;
  variable wtemp,wr,wpr,wpi,wi,theta,tempr,tempi,tmp1,tmp2 :integer;
  constant six  : integer:= 6;
  constant half : integer:= 5;
  constant deux : integer:=-2;
  constant un   : integer:= 1;
  constant zero : integer:= 0;
  variable data: tableau;
  begin
   Initialize:(TOC,true,Bit_reversal);
   Bit_reversal:(TOC,true,Danielson_ctl);
   Danielson_ctl:(TOC,true,Stop);
   behavior Initialize type sequential subbehaviors is
    begin
     Data_initialize:(TOC,true,Var_initialize);
     Var_initialize:(TOC,true,stop);
     behavior Data_initialize type sequential subbehaviors is
      begin
       Init_index:(TOC,true,Read_data);
       Read_data:(TOC,true,Incr_index);
       Incr_index:(TOC,(i<nn),Read_data),
              (TOC,(i>=nn),stop);
       behavior Init_index type code is
        begin
         i := 0;
         nn := 50;
         isign := 1;
        end Init_index;
       behavior Read_data type code is
        begin
```

```
    data(i):=input_port;
   end Read_data;

  behavior Incr_index type code is
   begin
    i:= i+1;
   end Incr_index;

 end Data_initialize;

 behavior Var_initialize type code is
  begin
   n := nn;
   j := 1;
   i := 1;
  end Var_initialize;

end Initialize;

behavior Bit_reversal type sequential subbehaviors is

begin

 Bit_init  : (TOC,true,Bit_loop1);

 Bit_loop1 : (TOC,true,Bit_incr);

 Bit_Incr   : (TOC,(i<n),Bit_loop1),
              (TOC,(i>=n),stop);

  behavior Bit_init type code is
   begin
    i := 1;
   end Bit_init;


  behavior Bit_Incr type code is
   begin
    i:=i+2;
   end Bit_Incr;

  behavior Bit_loop1 type sequential subbehaviors is

   begin

    Bit_condition:(TOC,true,Bit_shift);

    Bit_shift :(TOC,true,Bit_loop2);

    Bit_loop2 :(TOC,true,Bit_acc);

    Bit_acc : (TOC,true,stop);
```

```
behavior Bit_shift type code is
 begin
  m:=nn; --Shift
 end Bit_shift;

behavior Bit_acc type code is
 begin
  j:=j+m;
 end Bit_acc;

behavior Bit_condition type sequential subbehaviors is

begin

 Bit_test : (TOC,(i<j),Bit_swap1),

         (TOC,(i>=j),stop);

 Bit_swap1: (TOC,true,Bit_swap2);

 Bit_swap2: (TOC,true,stop);

 behavior Bit_test type code is
  begin
   null;
  end Bit_test;


 behavior Bit_swap1 type code is
  begin
   tempr := data(i);
   data(i) := data(j);
   data(j) := tempr;
  end Bit_swap1;


 behavior Bit_swap2 type code is
  begin
   tempr := data(i+1);
   data(i+1) := data(j+1);
   data(j+1) := tempr;
  end Bit_swap2;

end Bit_condition;

behavior Bit_loop2 type sequential subbehaviors is

begin

 Loop2_test:  (TOC,(m<2 and j<m),Loop2_assign),

              (TOC,(m>=2 or j>=m),stop);
```

```
    Loop2_assign:(TOC,true,Loop2_shift);
    Loop2_shift:(TOC,true,Loop2_test);

    behavior Loop2_test type code is
     begin
      null;
     end Loop2_test;

    behavior Loop2_assign type code is
     begin
      j:= j - m;
     end Loop2_assign;


    behavior Loop2_shift type code is
     begin
      m:=m; --Shift
     end Loop2_shift;
   end Bit_loop2;
  end Bit_loop1;
end Bit_Reversal;

behavior Danielson_ctl type sequential subbehaviors is
 begin
  Dan_init : (TOC,true,Dan_loop);
  Dan_loop : (TOC,true,stop);


  behavior Dan_init type code is
   begin
    mmax := 2;
   end Dan_init;

  behavior Dan_loop type sequential subbehaviors is
   begin
    Dan_test : (TOC,(n > mmax),Danielson),
               (TOC,(n <= mmax),stop);
    Danielson : (TOC,true,Dan_incr);
    Dan_incr : (TOC,true,Dan_test);
```

```
behavior Dan_test type code is
 begin
  null;
 end Dan_test;

behavior Dan_incr type code is
 begin
  mmax := istep;
 end Dan_incr;

behavior Danielson type sequential subbehaviors is

 begin

  Dan_initializations:(TOC,true,Dan_loop1);

  Dan_loop1 :(TOC,true,stop);


 behavior Dan_initializations type code is
  begin
   istep := mmax;
   theta := isign * (6 /mmax);
   wtemp := theta / 2; --sinus
   wpr := 2 * wtemp * wtemp;
   wpi := theta; --sinus
   wr := 1;
   wi := 0;
  end Dan_initializations;

 behavior Dan_loop1 type sequential subbehaviors is

  begin

   Loop1_init    : (TOC,true,Loop1_body);

   Loop1_body: (TOC,true,Loop1_incr);

   Loop1_incr: (TOC,(m<mmax),Loop1_body),
                (TOC,(m>=mmax),stop);


   behavior Loop1_init type code is
    begin
     m := 1;
    end Loop1_init;

   behavior Loop1_incr type code is
    begin
     m := m + 2;
```

```
end Loop1_incr;

behavior Loop1_body type sequential subbehaviors is
begin
 Dan_loop2    : (TOC,true,Dan_var_update);
 Update_var: (TOC,true,stop);

 behavior Update_var type code is
  begin
   wr := wtemp * wpr - wi * wpi + wr;
   wi := wi * wpr + wtemp * wpi + wi;
  end Dan_var_update;

 behavior Dan_loop2 type sequential subbehaviors is
  begin
   Loop2_init:(TOC,true,Loop2_body);
   Loop2_body:(TOC,true,Loop2_incr);
   Loop2_incr:(TOC,(i<=n),Loop2_body),
              (TOC,(i>n),stop);

   behavior Loop2_init type code is
    begin
     i := m;
    end Loop2_init;

   behavior Loop2_incr type code is
    begin
     i := i + istep;
    end Loop2_incr;

   behavior Loop2_body type sequential subbehaviors is
    begin
     Initialize-j : (TOC,true,Dan_real);
     Dan_real : (TOC,true,Dan_imag);
     Dan_imag : (TOC,true,stop);
     behavior Initialize-j type code is
      begin
       j := i + mmax;
      end Initialize-j;
```

```
            behavior Dan_real type code is
             begin
              tempr := wr * data(j) - wi * data(j+1);
              data(j) := data(i) - tempr;
              data(i) := data(i) + tempr;
             end Dan_real;

            behavior Dan_imag type code is
             begin
              tempi := wr * data(j+1) + wi * data(j);
              data(j+1) := data(i+1) + tempi;
              data(i+1) := data(i+1) + tempr;
             end Dan_imag;

           end Loop2_body;

          end Dan_loop2;

         end Loop1_body;

        end Dan_loop1;

       end Danielson;

      end Dan_loop;

     end Danielson_ctl;

end FFT_behavior;

end FFT_A;
```

A. 2. Technology files

In order to use the specsyn estimators, one has to provide the SpecCharts input like the one shown in section A.1. This specification is then run on any resource from the data base to estimate the performance. The performance estimates provided by the specsyn tools are the execution time (number of clock cycles) and the code size (bytes). The resource data base provided with the specsyn estimators contains many standard processors like the processor 68000 of Motorola or the 6800 of Intel. Our target architecture already shown in chapter 6 is built of two processors, the Texas instruments C40 and the custom SIMD processor PULSE developped at Ecole Polytechnique of Montréal. In order to estimate our case studies on the target architecture, we need to have the technology files for both the C40 and PULSE processors. We have constructed such models, shown in Figures A.1 and A.2. In these files, each line describes the number of clock cycles and bytes required for each generic instruction. If the processor has the parallelism capability, this feature is described into a complementary file to specify which generic instructions may be run in parallel. The files to specify the parallelism have also been created but are not presented in this appendix because of their complexity.

```
## Anything after # are comments.
## This is the technology file for the PULSE processor.
## DirectMem means direct memory addressing.
## IndirectMem means indirect memory addressing.
```

# OP	DESTINATION	SOURCE1	SOURCE2	time (clock cycles)	size(bytes)
ALU	Register	Constant	Constant	1	8
ALU	Register	Constant	Register	1	8
ALU	Register	Register	Constant	1	8
ALU	Register	Register	Register	1	8
ALU	Register	DirectMem	Constant	1	8
ALU	Register	Constant	DirectMem	1	8
ALU	Register	DirectMem	Register	1	8
ALU	Register	Register	DirectMem	1	8
ALU	Register	DirectMem	DirectMem	1	8
ALU	Register	IndirectMem	Constant	2	16
ALU	Register	Constant	IndirectMem	2	16
ALU	Register	IndirectMem	Register	2	16
ALU	Register	Register	IndirectMem	2	16
ALU	Register	IndirectMem	DirectMem	2	16
ALU	Register	DirectMem	IndirectMem	2	16
ALU	Register	IndirectMem	IndirectMem	3	24
ALU	DirectMem	Constant	Constant	1	8
ALU	DirectMem	Constant	Register	1	8
ALU	DirectMem	Register	Constant	1	8
ALU	DirectMem	Register	Register	1	8
ALU	DirectMem	DirectMem	Constant	1	8
ALU	DirectMem	Constant	DirectMem	1	8
ALU	DirectMem	DirectMem	Register	1	8
ALU	DirectMem	Register	DirectMem	1	8
ALU	DirectMem	DirectMem	DirectMem	1	8
ALU	DirectMem	IndirectMem	Constant	2	16
ALU	DirectMem	Constant	IndirectMem	2	16
ALU	DirectMem	IndirectMem	Register	2	16
ALU	DirectMem	Register	IndirectMem	2	16
ALU	DirectMem	IndirectMem	DirectMem	2	16
ALU	DirectMem	DirectMem	IndirectMem	2	16
ALU	DirectMem	IndirectMem	IndirectMem	3	24
ALU	Register	Empty	Constant	1	8
ALU	Register	Empty	Register	1	8
ALU	Register	Empty	DirectMem	1	8
ALU	Register	Empty	IndirectMem	2	16
ALU	DirectMem	Empty	Constant	1	8
ALU	DirectMem	Empty	Register	1	8
ALU	DirectMem	Empty	DirectMem	1	8
ALU	DirectMem	Empty	IndirectMem	2	16
DIV	Register	Constant	Constant	3	12
DIV	Register	Constant	Register	3	12
DIV	Register	Register	Constant	3	12
DIV	Register	Register	Register	2	8
DIV	Register	DirectMem	Constant	3	12
DIV	Register	Constant	DirectMem	2	8
DIV	Register	DirectMem	Register	2	8
DIV	Register	Register	DirectMem	2	8
DIV	Register	DirectMem	DirectMem	2	8
DIV	Register	IndirectMem	Constant	3	12
DIV	Register	Constant	IndirectMem	2	8
DIV	Register	IndirectMem	Register	2	8
DIV	Register	Register	IndirectMem	2	8
DIV	Register	IndirectMem	DirectMem	2	8
DIV	Register	DirectMem	IndirectMem	2	8
DIV	Register	IndirectMem	IndirectMem	2	8
DIV	DirectMem	Constant	Constant	4	16
DIV	DirectMem	Constant	Register	3	12
DIV	DirectMem	Register	Constant	4	16
DIV	DirectMem	Register	Register	3	12
DIV	DirectMem	DirectMem	Constant	4	16
DIV	DirectMem	Constant	DirectMem	3	12
DIV	DirectMem	DirectMem	Register	3	12
DIV	DirectMem	Register	DirectMem	3	12
DIV	DirectMem	DirectMem	DirectMem	3	12
DIV	DirectMem	IndirectMem	Constant	4	16
DIV	DirectMem	Constant	IndirectMem	3	12
DIV	DirectMem	IndirectMem	Register	3	12
DIV	DirectMem	Register	IndirectMem	3	12
DIV	DirectMem	IndirectMem	DirectMem	3	12
DIV	DirectMem	DirectMem	IndirectMem	3	12
DIV	DirectMem	IndirectMem	IndirectMem	3	12
NOP	Empty	Empty	Empty	1	8
CJUMP	Empty	Empty	Empty	1	8
JUMP	Empty	Empty	Empty	1	8
RET	Empty	Empty	Empty	1	8
CALL	Empty	Empty	Empty	1	8
DEFAULT	Empty	Empty	Empty	1	8

OP	DEST.	Source1	Source2	time	size
MUL	Register	Constant	Constant	1	8
MUL	Register	Constant	Register	1	8
MUL	Register	Register	Constant	1	8
MUL	Register	Register	Register	1	8
MUL	Register	DirectMem	Constant	1	8
MUL	Register	Constant	DirectMem	1	8
MUL	Register	DirectMem	Register	1	8
MUL	Register	Register	DirectMem	1	8
MUL	Register	DirectMem	DirectMem	1	8
MUL	Register	IndirectMem	Constant	2	16
MUL	Register	Constant	IndirectMem	2	16
MUL	Register	IndirectMem	Register	2	16
MUL	Register	Register	IndirectMem	2	16
MUL	Register	IndirectMem	DirectMem	2	16
MUL	Register	DirectMem	IndirectMem	2	16
MUL	Register	IndirectMem	IndirectMem	3	24
MUL	DirectMem	Constant	Constant	1	8
MUL	DirectMem	Constant	Register	1	8
MUL	DirectMem	Register	Constant	1	8
MUL	DirectMem	Register	Register	1	8
MUL	DirectMem	DirectMem	Constant	1	8
MUL	DirectMem	Constant	DirectMem	1	8
MUL	DirectMem	DirectMem	Register	1	8
MUL	DirectMem	Register	DirectMem	1	8
MUL	DirectMem	DirectMem	DirectMem	1	8
MUL	DirectMem	IndirectMem	Constant	2	16
MUL	DirectMem	Constant	IndirectMem	2	16
MUL	DirectMem	IndirectMem	Register	2	16
MUL	DirectMem	Register	IndirectMem	2	16
MUL	DirectMem	IndirectMem	DirectMem	2	16
MUL	DirectMem	DirectMem	IndirectMem	2	16
MUL	DirectMem	IndirectMem	IndirectMem	3	24
CMP	Register	Constant	Constant	1	8
CMP	Register	Constant	Register	1	8
CMP	Register	Register	Constant	1	8
CMP	Register	Register	Register	1	8
CMP	Register	DirectMem	Constant	1	8
CMP	Register	Constant	DirectMem	1	8
CMP	Register	DirectMem	Register	1	8
CMP	Register	Register	DirectMem	1	8
CMP	Register	DirectMem	DirectMem	1	8
CMP	Register	IndirectMem	Constant	2	16
CMP	Register	Constant	IndirectMem	2	16
CMP	Register	IndirectMem	Register	2	16
CMP	Register	Register	IndirectMem	2	16
CMP	Register	IndirectMem	DirectMem	2	16
CMP	Register	DirectMem	IndirectMem	2	16
CMP	Register	IndirectMem	IndirectMem	3	24
CMP	DirectMem	Constant	Constant	1	8
CMP	DirectMem	Constant	Register	1	8
CMP	DirectMem	Register	Constant	1	8
CMP	DirectMem	Register	Register	1	8
CMP	DirectMem	DirectMem	Constant	1	8
CMP	DirectMem	Constant	DirectMem	1	8
CMP	DirectMem	DirectMem	Register	1	8
CMP	DirectMem	Register	DirectMem	1	8
CMP	DirectMem	DirectMem	DirectMem	1	8
CMP	DirectMem	IndirectMem	Constant	2	16
CMP	DirectMem	Constant	IndirectMem	2	16
CMP	DirectMem	IndirectMem	Register	2	16
CMP	DirectMem	Register	IndirectMem	2	16
CMP	DirectMem	IndirectMem	DirectMem	2	16
CMP	DirectMem	DirectMem	IndirectMem	2	16
CMP	DirectMem	IndirectMem	IndirectMem	3	24
MOV	Register	Empty	Constant	1	8
MOV	Register	Empty	Register	1	8
MOV	Register	Empty	DirectMem	1	8
MOV	Register	Empty	IndirectMem	2	16
MOV	DirectMem	Empty	Constant	1	8
MOV	DirectMem	Empty	Register	1	8
MOV	DirectMem	Empty	DirectMem	1	8
MOV	DirectMem	Empty	IndirectMem	2	16
MOV	IndirectMem	Empty	Constant	2	16
MOV	IndirectMem	Empty	Register	2	16

Figure A.1. The technology file of the PULSE processor.

```
## Anything after `#' are comments.
## This is the technology file for the C40 DSP processor.
## DirectMem means direct memory addressing.
## IndirectMem means indirect memory addressing.
```

# OP	DESTINATION	SOURCE1	SOURCE2	time (clock cycles)	size(bytes)
ALU	Register	Constant	Constant	2	8
ALU	Register	Constant	Register	2	8
ALU	Register	Register	Constant	2	8
ALU	Register	Register	Register	1	4
ALU	Register	DirectMem	Constant	2	8
ALU	Register	Constant	DirectMem	2	8
ALU	Register	DirectMem	Register	2	8
ALU	Register	Register	DirectMem	2	8
ALU	Register	DirectMem	DirectMem	2	8
ALU	Register	IndirectMem	Constant	2	8
ALU	Register	Constant	IndirectMem	2	8
ALU	Register	IndirectMem	Register	1	4
ALU	Register	Register	IndirectMem	1	4
ALU	Register	IndirectMem	DirectMem	2	8
ALU	Register	DirectMem	IndirectMem	2	8
ALU	Register	IndirectMem	IndirectMem	1	4
ALU	DirectMem	Constant	Constant	3	12
ALU	DirectMem	Constant	Register	3	12
ALU	DirectMem	Register	Constant	3	12
ALU	DirectMem	Register	Register	2	8
ALU	DirectMem	DirectMem	Constant	3	12
ALU	DirectMem	Constant	DirectMem	3	12
ALU	DirectMem	DirectMem	Register	3	12
ALU	DirectMem	Register	DirectMem	3	12
ALU	DirectMem	DirectMem	DirectMem	3	12
ALU	DirectMem	IndirectMem	Constant	3	12
ALU	DirectMem	Constant	IndirectMem	3	12
ALU	DirectMem	IndirectMem	R egister	2	8
ALU	DirectMem	Register	IndirectMem	2	8
ALU	DirectMem	Indir ectMem	DirectMem	3	12
ALU	DirectMem	DirectMem	IndirectMem	3	12
ALU	DirectMem	IndirectMem	IndirectMem	2	8
ALU	Register	Empty	Constant	1	4
ALU	Register	Empty	Register	1	4
ALU	Register	Empty	DirectMem	1	4
ALU	Register	Empty	IndirectMem	1	4
ALU	DirectMem	Empty	Constant	2	8
ALU	DirectMem	Empty	Register	2	8
ALU	DirectMem	Empty	DirectMem	2	8
ALU	DirectMem	Empty	IndirectMem	2	8
DIV	Register	Constant	Constant	3	12
DIV	Register	Constant	Register	3	12
DIV	Register	Register	Constant	2	12
DIV	Register	Register	Register	3	8
DIV	Register	DirectMem	Constant	2	12
DIV	Register	Constant	DirectMem	2	8
DIV	Register	DirectMem	Register	2	8
DIV	Register	Register	DirectMem	2	8
DIV	Register	DirectMem	DirectMem	3	8
DIV	Register	IndirectMem	Constant	2	12
DIV	Register	Constant	IndirectMem	2	8
DIV	Register	IndirectMem	Register	2	8
DIV	Register	Register	IndirectMem	2	8
DIV	Register	IndirectMem	DirectMem	2	8
DIV	Register	DirectMem	IndirectMem	2	8
DIV	Register	IndirectMem	IndirectMem	2	8
DIV	DirectMem	Constant	Constant	4	16
DIV	DirectMem	Constant	Register	3	12
DIV	DirectMem	Register	Constant	4	16
DIV	DirectMem	Register	Register	3	12
DIV	DirectMem	DirectMem	Constant	4	16
DIV	DirectMem	Constant	DirectMem	3	12
DIV	DirectMem	DirectMem	Register	3	12
DIV	DirectMem	Register	DirectMem	3	12
DIV	DirectMem	DirectMem	DirectMem	3	12
DIV	DirectMem	IndirectMem	Constant	4	16
DIV	DirectMem	Constant	IndirectMem	3	12
DIV	DirectMem	IndirectMem	Register	3	12
DIV	DirectMem	Register	IndirectMem	3	12
DIV	DirectMem	IndirectMem	DirectMem	3	12
DIV	DirectMem	DirectMem	IndirectMem	3	12
DIV	DirectMem	IndirectMem	IndirectMem	3	12
NOP	Empty	Empty	Empty	1	4
CJUMP	Empty	Empty	Empty	1	4
JUMP	Empty	Empty	Empty	1	4
RET	Empty	Empty	Empty	1	4
CALL	Empty	Empty	Empty	1	4
DEFAULT	Empty	Empty	Empty	1	4

OP	DEST.	Source1	Source2	time	size
MUL	Register	Constant	Constant	2	8
MUL	Register	Constant	Register	2	8
MUL	Register	Register	Constant	2	8
MUL	Register	Register	Register	1	4
MUL	Register	DirectMem	Constant	2	8
MUL	Register	Constant	DirectMem	2	8
MUL	Register	DirectMem	Register	2	8
MUL	Register	Register	DirectMem	2	8
MUL	Register	DirectMem	DirectMem	2	8
MUL	Register	IndirectMem	Constant	2	8
MUL	Register	Constant	IndirectMem	2	8
MUL	Register	IndirectMem	Register	1	4
MUL	Register	Register	IndirectMem	1	4
MUL	Register	IndirectMem	DirectMem	2	8
MUL	Register	DirectMem	IndirectMem	2	8
MUL	Register	IndirectMem	IndirectMem	1	4
MUL	DirectMem	Constant	Constant	3	12
MUL	DirectMem	Constant	Register	3	12
MUL	DirectMem	Register	Constant	3	12
MUL	DirectMem	Register	Register	2	8
MUL	DirectMem	DirectMem	Constant	3	12
MUL	DirectMem	Constant	DirectMem	3	12
MUL	DirectMem	DirectMem	Register	3	12
MUL	DirectMem	Register	DirectMem	3	12
MUL	DirectMem	DirectMem	DirectMem	3	12
MUL	DirectMem	IndirectMem	Constant	3	12
MUL	DirectMem	Constant	IndirectMem	3	12
MUL	DirectMem	IndirectMem	Register	2	8
MUL	DirectMem	Register	IndirectMem	2	8
MUL	DirectMem	IndirectMem	DirectMem	3	12
MUL	DirectMem	DirectMem	IndirectMem	3	12
MUL	DirectMem	IndirectMem	IndirectMem	2	8
CMP	Register	Constant	Constant	2	8
CMP	Register	Constant	Register	2	8
CMP	Register	Register	Constant	2	8
CMP	Register	Register	Register	2	8
CMP	Register	DirectMem	Constant	2	8
CMP	Register	Constant	DirectMem	2	8
CMP	Register	DirectMem	Register	2	8
CMP	Register	Register	DirectMem	2	8
CMP	Register	DirectMem	DirectMem	2	8
CMP	Register	IndirectMem	Constant	2	8
CMP	Register	Constant	IndirectMem	2	8
CMP	Register	IndirectMem	Register	2	8
CMP	Register	Register	IndirectMem	2	8
CMP	Register	IndirectMem	DirectMem	2	8
CMP	Register	DirectMem	IndirectMem	2	8
CMP	Register	IndirectMem	IndirectMem	2	8
CMP	DirectMem	Constant	Constant	3	12
CMP	DirectMem	Constant	Register	3	12
CMP	DirectMem	Register	Constant	3	12
CMP	DirectMem	Register	Register	3	12
CMP	DirectMem	DirectMem	Constant	3	12
CMP	DirectMem	Constant	DirectMem	3	12
CMP	DirectMem	DirectMem	Register	3	12
CMP	DirectMem	Register	DirectMem	1	12
CMP	DirectMem	DirectMem	DirectMem	3	12
CMP	DirectMem	IndirectMem	Constant	3	12
CMP	DirectMem	Constant	IndirectMem	3	12
CMP	DirectMem	IndirectMem	Register	3	12
CMP	DirectMem	Register	IndirectMem	3	12
CMP	DirectMem	IndirectMem	DirectMem	3	12
CMP	DirectMem	DirectMem	IndirectMem	3	12
CMP	DirectMem	IndirectMem	IndirectMem	3	12
MOV	Register	Empty	Constant	1	4
MOV	Register	Empty	Register	1	4
MOV	Register	Empty	DirectMem	1	4
MOV	Register	Empty	IndirectMem	1	4
MOV	DirectMem	Empty	Constant	2	8
MOV	DirectMem	Empty	Register	1	4
MOV	DirectMem	Empty	DirectMem	2	8
MOV	DirectMem	Empty	IndirectMem	2	8
MOV	IndirectMem	Empty	Constant	2	8
MOV	IndirectMem	Empty	Register	1	4

Figure A.2. The technology file for the C40 processor.

REFERENCES

[Alhayek96]. G. Al Hayek, Y. Le Traon and C. Robach, "*Test economics criterion for hardware/software partitioning*", International Test Conference 1996.

[Auguin94]. A. Auguin, F. Boeri and C. Carriere, "*Automatic exploration of VLIW processor architectures from a designer's experience based specification*", CODES'94, pp 108-115.

[Bakhshi94]. S. Bakhshi and D.D. Gajski, "*A component selection algorithm for high performance pipelines*", Technical report #94-01, Univ. of California, Irvine, June 1994.

[Barros93]. E. Barros, W. Rosenstiel and X. Xiong, "*Hardware/software partitioning with UNITY*", Proceedings of Int. workshop on hardware/software codesign, October 1993.

[Barros94]. E. Barros and A. Sampaio, "*Toward provably correct hardware/software partitioning using OCCAM*", Proceedings of CODES'94, October 1994.

[BenIsmail94a]. T. Ben Ismail, K. O'Brien, and A. Jerraya, "*Interactive system-level partitioning with PARTIF*", ICCAD'94, pp 464-468.

[BenIsmail94b]. T. Ben Ismail, M. Abid and A. Jerraya, "*COSMOS: A codesign approach for communicating systems*", CODES'94, September 1994, pp 17-24.

[Berry91]. G. Berry, and G. Gonthier, "*Incremental development of an HDLC entity in Esterel*", Computer Networks and ISDN systems, Vol.22, No. 1, 1991, pp35-49.

[Binh96]. N. Binh, M. Imai, A. Shiomi, and N. Hikichi, "*A hardware/software partitioning algorithm for designing pipelined ASIPs with least gate count*", 33rd DAC 1996, (*http://kona.ee.pitt.edu/33dac/papers/1996/dac96/htmfiles/*).

[Boriello92]. G. Boriello ans A. Sangiovanni-Vincentelli, "*Models for the hardware/software codesign of embedded controllers*", CODES'92.

[Buchenrieder92]. K. Buchenrieder and C. Veith, "*CODES: a practical concurrent design environment*", CODES'92.

[Buchenrieder93]. K. Buchenrieder et al., "*Hardware/software codesign with PRAMs using CODES*", in Computer hardware desription languages, IFIP transcations, vol.A-32, 1993, Edited by D. Agneur et al.

[Buck94a]. J.T. Buck, S. HA, E.A. Lee and D.G. Messerschmitt, "*PTOLEMY: a framework for simulating and prototyping heterogeneous systems*", International journal of computer simulation, special issue on "simulation software development", vol.4, April 1994, pp 155-182.

[Buck94b]. J.T. Buck, "*A dynamic dataflow model suitable for efficient mixed hardware/software implementations of DSP applications*", 3rd CODES'94, Grenoble.

[Camurati94]. P. Camurati, F. Corno, P. Prinetto, C. Bayol, and B. Soulas, "*System-level modeling and verification: a comprehensive design methodology*", ICCAD'94. pp636-640.

[Cheng94]. W. Cheng, and Y. Lin, "*Code generation for a DSP processor*", ICCAD'94, pp 82-87.

[Chiodo92]. M. Chiodo, A. Sangiovanni-Vincentelli, "*Design methods for reactive real-time system codesign*", Int. CODES'92.

[Chiodo93a]. M. Chiodo, P. Giusto, H. Hsieh, A. Jureka, L.Lavagno, and A. Sangiovanni-Vincentelli, "*A formal specification model for hardware/software codesign*", Technical Report, June 1993.

[Chiodo93b]. Chiodo, P. Giusto, H. Hsieh, A Jurescka, l. Lavagno, and A. Sangiovanni-Vincentelli, " *Synthesis of mixed hardware/software implementation from CFSM specifications*", June 1993.

[Chiodo94]. M. Chiodo, P. Giusto, A. Jurescska, M. Marelli, H.C. Hsieh, A. Sangiovanni-Vincentelli, and L. Lavagno, "*Hardware/software codesign of embedded systems*", IEEE Micro, August 1994, pp 26-36.

[Chou92]. P. Chou, R. Ortega, and G. Boriello, " *Synthesis of the hardware/software interface in microcontroller-based systems*", ICCAD'92, pp 488-495.

[Chou94]. P. Chou, E.A. Walkup and G. Boriello, "*Scheduling for reactive real-time systems*", IEEE Micro, August 1994, pp 37-47.

[Chou95]. P. Chou, R. Ortega and G. Boriello, "*The Chinook hardware/software Co-synthesis system*", International symposium on system synthesis, Cannes, France, September 13-15, 1995. pp 22-27.

[Christopher92]. R. Christopher, "*Signal processing in C*", Wiley 1992. pp 496-535.

[Dembinski]. P. Dembinski, and S. Budkouski, "*Specification language Estelle*".

[Eduards94]. M. Eduards and J. Forrest, "*A development environment for the cosynthesis of embedded hardware/software systems*", EDAC'94, pp 469-473.

[Edwards97]. S. Edwads, L. Lavagno, E. Lee and A. Sangiovanni-Vincentelli, "*Design of embedded systems: formal models, validation and synthesis*", Proceedings of the IEEE, vol. 85, no. 23, march 1997, pp 366-390.

[Eles94]. P. Eles, Z. Peng and A. Doboli, "*VHDL system-level specification and partition-*

ing in a hardware/software codesign environment", CODES'94.

[Eles96]. P. Eles, Z. Peng and A. Doboli, "*Hardware/software partitioning of VHDL system specification*", EURO-DAC'96.

[Ernst92]. R. Ernst and J. Henkel, "*Hardware/software codesign of embedded controllers based on hardware extraction*", CODES'92.

[Forrest92]. J. Forrest, "*Multiple abstraction-level descriptions using C++*", International workshop on Codesign, Colorado, September 1992.

[Gajski93]. D. Gajski, F. Vahid, S. Narayan, "*SpecCharts: a VHDL front-end for embedded systems*", TR93-31, University of California at Irvine, June 1993.

[Gajski93]. D. Gajski, J. Gong, F. Vahid, and S. Narayan, "*The specsyn design process and human interface*", TR93-3, University of California, Irvine, 1993.

[Gajski94a]. G. Gajski, F.Vahid, and S. Narayan, "*A system design methodology: Executable specification refinement*", ICCAD'94, pp 458-463.

[Gajski94b]. D. D. Gajski, F. Vahid and S. Narayan, "*System-level methodology and technology*", Univ. of California, Irvine, 1994.

[Gong93]. J. Gong, D.D. Gajski and S. Narayan, "*Software estimation from executable specifications*", technical report ICS-93-5, March 1993. University of California, Irvine.

[Gong95]. J. Gong, F. Vahid and S. Narayan, "*The SpecCharts/Specsyn User's manual version 3.2*", University of California, Irvine, September 1995.

[Gupta92]. R.K. Gupta and G. De Micheli, "*System level design*", Internal report CS-92, Stanford university, 1992.

[Gupta93]. R.K. Gupta, and G. De Micheli, "*Hardware/software cosynthesis of digital systems*", IEEE Micro, September 1993, pp 29-41.

[Gupta94a]. R. Gupta, and G. De Micheli, "*Constrained software generation from hardware/software systems*", International workshop on codesign, 1994, pp 56-63.

[Gupta94b]. R.K Gupta, C.N. Coelho,Jr and G. De Micheli, "*Program implementation schemes for hardware/software systems*", IEEE Computer, January 1994, pp 48-55.

[Gupta94c]. R. Gupta et al., "*Experience with image compression chip design using a unified system construction tools*", DAC'94, pp 250-256.

[Gupta96]. R.K. Gupta, "*Analysis of operation delay and execution rate constraints for embedded systems*", 33rd DAC 1996, (*http://kona.ee.pitt.edu/33dac/papers/1996/dac96/htmfiles/*).

[Henkel93]. J. Henkel, T. Benner and R. Ernst, "*Hardware generation and partitioning effects in the COSYMA system*", CODES'93.

[Henkel94]. J. Henkel, R. Ernst, U. Holtmann and T. Benner, "*Adaptation of partitioning and high-level synthesis in hardware/software cosynthesis*", Proceedings of the Int. Conf. on CAD, November 1994.

[Henkel96]. J. Henkel and R. Ernst, "*The interplay of run-time estimation and granularity in hardware/software partitioning*", CODES'96, Pittsburgh 1996.

[Henkel97]. J. Henkel and R. Ernst, "*A hardware/software partitioner using a dynamically determined granularity*", 34th Design Automation Conference, 1997, pp691-696.

[Herman94]. D. Hermann, J. Henkel and R. Ernst, "*An approach to the adaptation of esti-*

mated cost parameters in the COSYMA system", CODES'94, pp 100-107.

[Hu94]. X. Hu, J.G. D'Ambrosio, B.T. Musray and D. Tang, "*Codesign of architecture for automotive powertrain modules*", IEEE Micro, August 1994, pp 17-25.

[Huang93]. I. Huang, and A. Despain, "*Hardware/software resolution of pipeline hazards in pipeline synthesis of instruction set processors*", ICCAD'93, pp 594-599.

[Huang95]. C. Huang, and D. Gajski, "*Software performance estimation for pipeline and superscalar processors*" TR95-20, University of California at Irvine, June 95.

[Jain92]. R. Jain et al., "*Predicting system-level area and delay for pipelined and non-pipelined designs*", IEEE transactions on CAD, vol. 11, no. 8, August 1992, pp 955-965.

[Kalavade92]. A. Kalavade and E.A. Lee, "*Hardware/software codesign using PTOLEMY- A case study*", CODES'92.

[Kalavade93]. A. Kalavade, and E.A. Lee, "*A hardware/software codesign methodology for DSP applications*", IEEE Design and Test of computers, September 1993, pp 16-28.

[Kalavade94]. A. Kalavade and E.A. Lee, " *A global criticality/local pahse driven algorithm for the constrained hardware/software partitioning problem*", Proc. CODES'94.

[Kernighan70]. B.W. Kernighan and S. Lin, "*An efficient heuristic procedure for partitioning graphs*", The Bell system technical journal, February 1970, pp 291-307.

[Korf94]. F. Korf, R. Schlor, "*Interface controller synthesis from requirement specification*", ICCAD'94, pp 385-394.

[Krishnakumar90]. A.S. Krishnakumar and K. Sabini, "*VLSI implementation of communication protocols: a survey*", IEEE journal on selected areas in communication, vol 7

no.7, September 1989. pp 1082-1090.

[Kramer92]. H. Kramer and J. Miller, "*Assignment of global memory elements for multi-process VHDL specifications*", ICCAD'92, pp 496-501.

[Kumar92]. S. Kumar, J.Ayler, B. Johnsin and W. Wulf, "*A framework for hardware/software codesign", Int. workshop on hardware/software codesign*", Colorado, September 1992.

[Leupers94]. R. Leupers, W. Schenk, and P. Marwedel, "*Retargetable assembly code generation by bootstrapping*", ICCAD'94, pp 88-93.

[Lien94]. C. Lien, T. May, P. Paulin, "*Register allocation through resource classification for ASIP microcode generation*", ICCAD'94.

[Lin96]. B. Lin, "*A system design methodology for hardware/software co-development of telecommunication network applications*", 33rd DAC, 1996, (http://kona.ee.pitt.edu/33dac/papers/1996/dac96/htmfiles).

[Luk94]. W.Luk and T. Wu, "*Toward a declarative framework for hardware/software codesign*", CODES'94.

[Lundberg92]. L. Lundberg, "*Generating VHDL for simulation and synthesis from a high-level DSP design tool*", VHDL simulation, synthesis and formal proofs of hardware, 1992 Kluwer Academic Publishers, pp 149-161.

[Mancini94]. G. Mancini, "*Hardware/software coverification in ATM*", ICCAD'94, pp 1-7.

[Marriot98]. P. Marriott, J.C. Kraljic and Y. Savaria, "*Parallel Ultra Large Scale Engine SIMD architectures for real-time Digital Signal Processing Applications", Proc. of*

ICCAD98.

[Marwedel93]. P. Marwedel, "*Tree-based mapping of algorithms to predefined structures*", ICCAD'93, pp 586-593.

[McFarland92]. M.C. McFarland, T.J. Kowalski and M.J. Pemain, "*Language and formal semantics of the specification system CPA*", CODES'92.

[Menez92]. G. Menez, M. Auguin, F. Boeri and N. Carriere, "*A partitioning algorithm for system-level synthesis*", ICCAD'92, pp 482-487.

[DeMicheli94]. G. De Micheli, "*Computer-aided hardware/software codesign*". IEEE Micro, August 1994. pp 10-16.

[Mitra93]. R. Mitra, B. Guha, and A. Basu, "*Rapid prototyping of microprocessor-based systems*", ICCAD'93, pp 600-603.

[Narayan92a]. S. Narayan, F. Vahid and D. Gajski, "*Modeling with SpecCharts*", TR90-20, University of California at Irvine, October 1992.

[Narayan92b]. S. Narayan and D.D. Gajski, "*Area and performance estimation from system level specifications*", Technical report ICS-92-16, December 1992, University of California at Irvine.

[O'Brien95]. K. O'Brien, T. Ben Ismail, and A.A. Jerraya, "*A flexible communication modeling paradigm for system-level synthesis*", GMD Institut Set, 1995.

[Olukotun94]. K.A. Olukotun, R. Belaihel, J. Levitt and R. Ramirez, "*A hardware/software cosynthesis approach to digital system simulation*", IEEE Micro, August 1994, pp 48-58.

[Oudghiri92]. H. Oudghiri and B. Kaminska, "*A global and weighted algorithm for scheduling and allocation in high-level synthesis*", European design automation conference, March 1992, Belgium, pp 491-495.

[Oudghiri97]. H. Oudghiri, B. Kaminska and J. Rajski, "*A hardware/software partitioning technique with hierarchical design space exploration*", Custom integrated circuit conference, May 1997, Santa Clara, pp 95-98.

[Pino95]. J.L. Pino, S. Ha, E.A. Lee and J.T. Buck, "*Software synthesis for DSP using PTOLEMY*", Journal of VLSI signal processing, no.9, 1995, pp7-21.

[Potkonjak94]. M. Potkonjak, J. Rabaey, "*Algorithm selection: a quantitative computation- intensive optimization approach*", ICCAD'94, pp 90-94.

[Puri94]. R. Puri, and J. Gu, "*A divide-and-conquer approach for asynchronous interface synthesis*", ICCAD'94, pp 118-125.

[Rao93]. D.S. Rao and F.J. Kurdali, "*Hierarchical design space exploration for a class of digital systems*", IEEE transcations on VLSI, vol. 1, no.3, September 1993, pp 282-295.

[Schnaider96]. B. Schnaider, and E. Yogev, "*Software development in a hardware simulation environment*", 33rd DAC 1996, (*http://kona.ee.pitt.edu/33dac/papers/1996/dac96/htmfiles/*).

[Sheliga94]. M. Sheliga and E. H-M. Sha, "System partitioning and scheduling for hardware/software codesign", Technical Report, University of Notre-Dame, Department of CSE, 1994.

[SKS92]. SKS Group, "*On an experiment in system codesign: a mass flowmeter*", CODES'92.

[Srivastava95]. M.B. Srivastava and R.W. Brodersen, "*SIERRA : a unified framework for rapid prototyping of system level hardware and software*", IEEE transactions on CAD, vol. 14, no.6. June 1995, pp 676-693.

[Steinhausen93]. U. Steinhausen et al., "*System synthesis using hardware/software codesign*", CODES'93.

[Subrahmanyam92]. P.A. Subrahmanyam, "*Hardware/software codesign: what is needed for success?*", CODES'92.

[Sun92]. J.S. Sun and R.W. Brodersen, "*Design of system interface modules*", ICCAD'92. pp478-481.

[Susuki96]. K. Suzuki and A. Sangiovanni-Vincentelli, "*Efficient software performance estimation methods for hardware/software codesign*", 33rd DAC, 1996, (*http://kona.ee.pitt.edu/33dac/papers/1996/dac96/htmfiles/*).

[Sutarwala93]. S. Sutarwala, P.G. Paulin and Y. Kumar, "*Insulin: an instruction set simulation environment*", Proc. CHDL, Ottawa, April 1993, pp 355-362.

[Synopsys95]. "SmartModel library reference manual", Synopsys Logic Modeling Group, 1995.

[Synopsys96]. "The Design Compiler Reference", Synopsys, Inc. 1997.

[Texas92]. "*The TMS320C40 parallel processing seminar workbook*" Texas Instruments, Inc., 1992.

[Theibinger94]. M. Theibinger, P. Stravers, and H. Veit, "*CASTLE: an intercative environment for hardware/software codesign*", 3rd workshop on hardware/software codesign, Grenoble september 22-23, 1994, pp 203-209.

[Theoen97]. F.Theoen, J. Dersteen. G.Jong, G. Goossens, and H. De Man, "*Multi-Thread-Graph: a system model for real-time embedded software synthesis*" International European Design Automation Conference, 1997, pp 476-481.

[Thomas93]. D.E. Thomas, J.K. Adams and H. Schmit, "*A model and methodology for hardware/software codesign*", IEEE Design and Test of computers, September 1993, pp 6-15.

[Thomas96]. J.K. Thomas, and D.E. Thomas, "*The design of mixed hardware/software systems*", 33rd DAC, 1996, (*http://kona.ee.pitt.edu/33dac/papers/1996/dac96/htmfiles/*).

[Vahid91]. F. Vahid, "*A survey of behavioral-level partitioning systems*", TR91-71, University of California, Irvine, October 1991.

[Vahid92]. F. Vahid, and D. Gajski, "*Specification partitioning for system design*", 29th DAC, 1992, pp 219-224.

[Vahid]. F. Vahid, J. Gong and D. Gajski, "*A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning*", Dept.of Information and Computer Science, UC at Irvine

[Vercauteren96a]. S. Vercauteren, B. Lin, and H. De Man, "*Constructing application-specific heterogenoeus embedded architectures from custom hardware/software applications*", 33rd DAC, Las Vegas, June 1996, (*http://kona.ee.pitt.edu/33dac/papers/1996/dac96/htmfiles/*).

[Vercauteren96b]. S. Vercauteren, B. Lin, and H. De Man, "*A strategy for real-time kernel support in application-specific hardware/software embedded architectures*" 33rd DAC 1996, (*http://kona.ee.pitt.edu/33dac/papers/1996/dac96/htmfiles/*).

[Wenban92]. A. Wenban, J. O'Leary, and G. Brown, " *Codesign of communication protocols*", CODES'92, Colorado, September 1992.

[Weiskamp]. K. Weiskamp and B. Flamig, " *The complete C++ primer*", Academic press, Inc. 1989.

[Wilberg95]. J. Wilberg, A. Kuth, R. Composano, W. Rosentiel, and H.T. Vierhaus, "*Design exploration in CASTLE*", Workshop on high-level synthesis algorithms tools and design (HILES), GMD studies, vol.276, Stanford University, November 4th, 1995.

[Wilberg96]. J. Wilberg, P. Plager, R. Composano, M. Langevin, and H.T. Vierhaus, "*Codesign of hardware, software and algorithms - a case study-*", International symposium on circuits and systems, Atlanta, Georgia,May 12-15, 1996.

[Wilberg a]. J. Wilberg, and R. Composano, "*VLIW processor for video processing*"

[Wilberg b]. J. Wilberg, R. Composano and W. Rosenstiel, "*Design flow for hardware.software cosynthesis of a video compression system*".

[Wilson94]. T. Wilson, G. Grewal, B. Halley, and D. Banerji, "*An integrated approach to retargetable code generation*", ICCAD'94, pp 70-75.

[Wolf93]. W. Wolf, and R. Manno, "*High-level modeling and synthesis of communicating processes using VHDL*" IEEE transactions on information and systems, vol E76.D, no. 9, September 1993, pp 1039-1046.

[Woo94]. N. Woo, and A. Dunlop, "*Codesign from cospecification*", IEEE computer, January 1994, pp 42-47.

[Zirojnovic96]. V. Zirojnovic, and H. Meyr, "*Compiled hardware/software co-simulation*", 33rd DAC, 1996, (*http://kona.ee.pitt.edu/33dac/papers/1996/dac96/htmfiles/*).