Abstract

SYSTEM-LEVEL CODESIGN OF MIXED HARDWARE-SOFTWARE SYSTEMS

by

Asawaree Prabhakar Kalavade

Doctor of Philosophy in Electrical Engineering

University of California, Berkeley

Professor Edward A. Lee, Chair

This thesis provides a systematic approach to the system-level design of embedded signal processing applications. Such applications 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 the four key problems of partitioning, cosynthesis, cosimulation, and design methodology management. Applications are assumed to be specified using synchronous dataflow semantics.

A key contribution of the thesis is to formulate the *extended partitioning problem*. In system-level design, applications are represented as task graphs where tasks (called nodes) have moderate to large granularity and each node has several implementation options differing in area and execution time. The extended parti-

tioning problem involves the joint determination of the mapping (hardware or software), schedule, as well as the implementation option for each node, so that the overall area allocated to nodes in hardware in minimum. This problem is considerably harder (and richer) than the traditional binary partitioning that determines just the best mapping and schedule.

Both extended and binary partitioning problems are constrained optimization problems and are shown to be NP-hard. We first present an efficient heuristic, called GCLP, to solve the binary partitioning problem. The heuristic reduces the greediness associated with serial traversal-based algorithms by formulating a global criticality (*GC*) measure. The *GC* measure also permits an adaptive selection of the objective (optimizing either area or time). We then present an efficient heuristic for extended partitioning, called MIBS, that alternately uses GCLP and an implementation-bin selection procedure. The implementation-bin selection procedure chooses the bin that maximizes the area-reduction gradient. Solutions generated by both heuristics are shown to be reasonably close to optimal. Extended partitioning generates considerably smaller overall hardware area as compared to binary partitioning.

Our approach to cosynthesis is to generate synthesizeable descriptions of the hardware and software components and to use pre-existing synthesis tools such as Ptolemy and Hyper to generate the actual software and hardware implementations. Techniques for generating these synthesizeable descriptions, starting with a partitioned graph, are discussed. Cosimulation involves simulating the hardware, the processor, and the software that the processor executes, within a unified framework. The requirements of a cosimulation framework are discussed and the use of Ptolemy for hardware-software cosimulation is illustrated.

Due to the large system-level design space, the system-level design prob-

lem cannot, in general, be posed as a single well-defined optimization problem. Typically, the designer needs to explore the possible options, tools, and architectures. We believe that managing the design process plays an equally important role in system-level design, as do the tools used for different aspects of the design. To this end, we present a framework that supports design methodology management.

We embody all the above concepts in a tool called the Design Assistant. The Design Assistant contains tools for partitioning, synthesis, and simulation and operates in the design methodology management framework.

Edward A. Lee, Thesis Committee Chairman

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank my advisor Prof. Edward A. Lee for introducing me to this research area and for his continous support, guidance, and encouragement. I appreciate the freedom and collegial respect that he offered me while pursuing my research.

Discussions with Prof. Jan Rabaey have been instrumental in shaping a number of ideas presented in this thesis and have given me valuable insight into the overall area of highlevel synthesis and system-level design space exploration. I am grateful to him for his interest and guidance.

I would also like to thank Prof. Brayton and Prof. Adler for some of the stimulating discussions that were useful in improving the overall quality of this dissertation.

I gratefully acknowledge the financial support provided by the Semiconductor Research Consortium (contract number DC-324 and grant number DC-008) over the past five years.

I would like to thank the Ptolemy team for developing such a wonderful (and at times frustrating!) software environment. I would also like to thank the Hyper team for answering my questions on Hyper.

Most of all, I want to thank my family for being an infinite source of love and encouragement. I especially want to thank Pratyush Moghe for making this thesis a reality.

1

INTRODUCTION

System-level design usually involves designing an application specified at a large granularity. 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 basic components of a system-level specification are called *tasks* (or nodes). The task-level description of a particular application (modem) is shown in Figure 1.1. The specification has two characteristics. First, tasks are at a

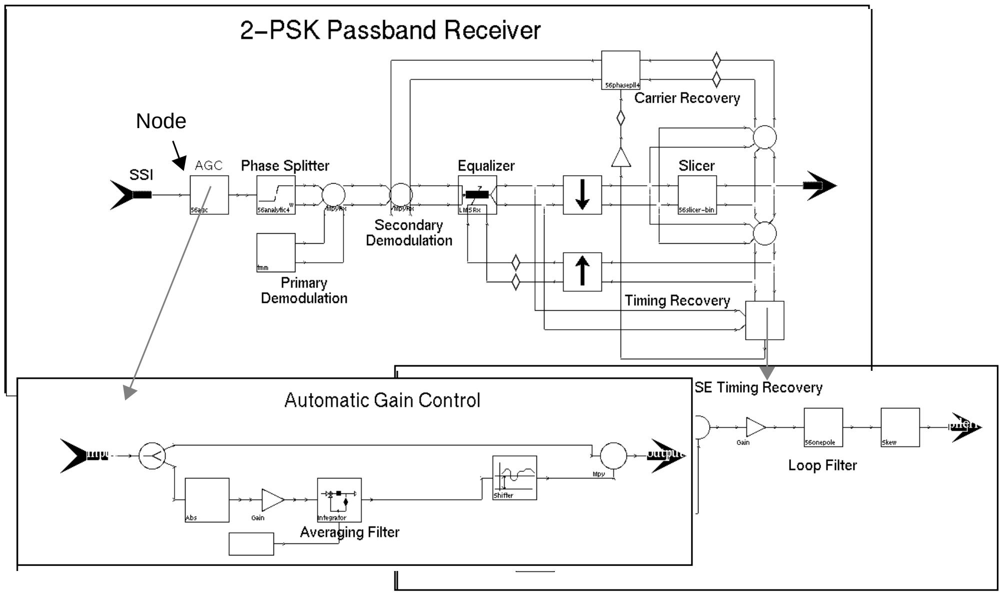


Figure 1.1. Task-level description of the receiver section of a modem.

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 especially important for the synthesis of complex applications whose cost and performance constraints often demand a mixed hardware-software implementation.

System-level design is very broad problem. In this thesis, we restrict our attention to the design of **embedded systems with real-time signal processing components**. Examples of such systems include modems for both tethered and wireless communication, cordless phones, disk drive controllers, printers, digital audio systems, data compression systems, etc. These systems are characterized by stringent performance and cost constraints.

Such applications often tend to have mixed hardware and software implementations. For instance, 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 node in the application to a particular mapping (hardware or software) or implementation (design style within hardware or software) when specifying the application. The appropriate mapping and implementation for each node can be selected by a global optimization procedure after the specification stage. The task level of abstraction allows this flexibility. Also, such embedded systems typically have a product cycle time of 18 months or so [Keutzer94] which imposes a severe time-to-market constraint. Manual development of a lower-level design specification (such as at the RTL level) is quite intensive due to the sheer 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 the 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 (as shown later in Figure 1.5) 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”. For instance, core-based ASICs that contain cores of programmable processors along with custom hardware on the same die are now a commercial reality.

In the next section, we discuss a few key issues in system-level design. Our approach to solving these problems is outlined in Section 1.2.

1.1 Issues in System-level Design

Figure 1.2 summarizes the key issues in system-level design. These are *partitioning*, *synthesis*, *simulation,* and *design-space exploration*. In this thesis, we address these issues in order. Several hardware and software implementation options are usually available for each node in the task-level description. The *partitioning* process determines an appropriate mapping (hardware or software) and an implementation for each node. 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 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. The *design-space exploration* framework attempts to ease this process.

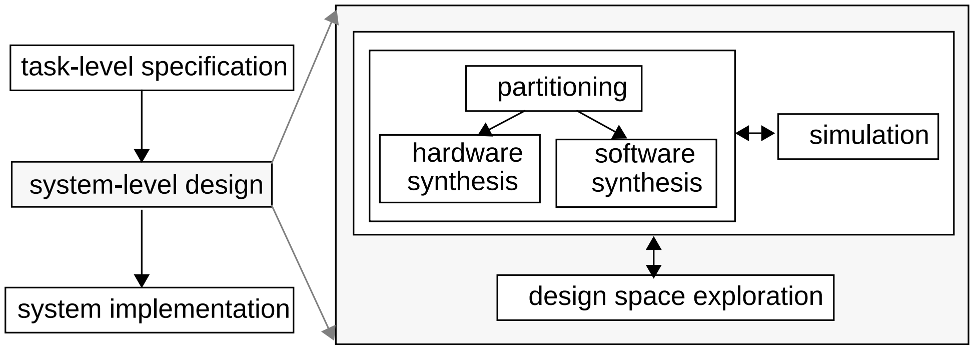


Figure 1.2. System-level design

We next discuss each of these issues in more detail.

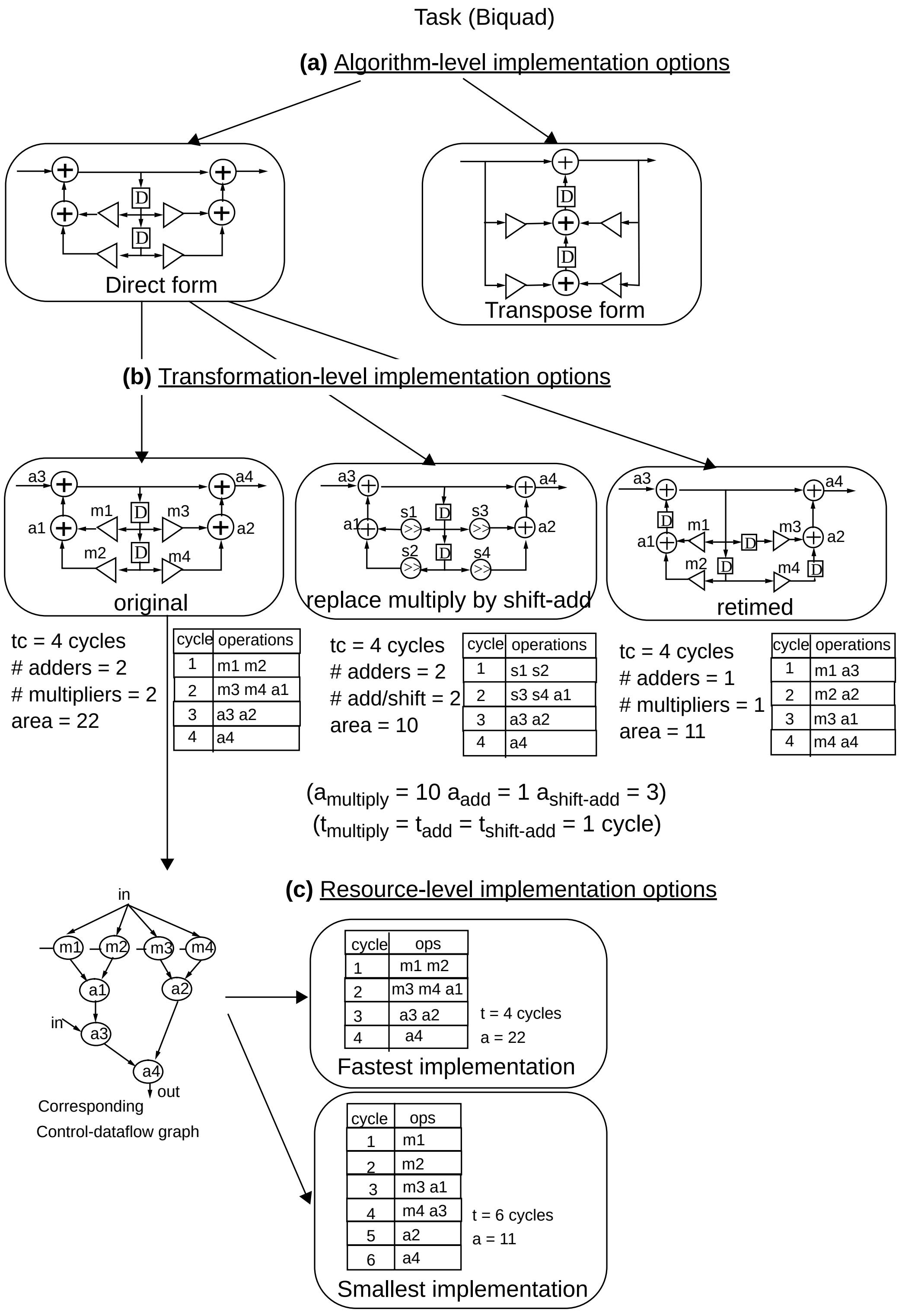
1.1.1 Partitioning

Perhaps the most important aspect of system-level design is the multiplicity of design options available for every node in the task-level specification. Each node can be implemented in several ways in both hardware and software mappings. The *partitioning* problem is to select an appropriate combination of mapping and implementation for each node.

For instance, a given task can be implemented in hardware using design options at several levels.

1. *Algorithm level*: Several algorithms can be used to describe the same task. For instance, a finite impulse response filter can be implemented either as an inner product or using the FFT in a shift-and-add algorithm. As a trivial example, a biquad can be described using the direct form or the transpose form (Figure 1.3).

2. *Transformation level*: For a particular algorithm, several transformations [Potkonjak94] can be applied on the original task description. Figure 1.3-b shows two such transformations. In the first transformation, multiplication by a constant (filter coefficients) is replaced by equivalent shift and add



tc = 4 cycles
# adders = 2
# multipliers = 2
area = 22

cycle	operations
1	m1 m2
2	m3 m4 a1
3	a3 a2
4	a4

tc = 4 cycles
# adders = 2
# add/shift = 2
area = 10

cycle	operations
1	s1 s2
2	s3 s4 a1
3	a3 a2
4	a4

tc = 4 cycles
# adders = 1
# multipliers = 1
area = 11

cycle	operations
1	m1 a3
2	m2 a2
3	m3 a1
4	m4 a4

$(a_{multiply} = 10\ a_{add} = 1\ a_{shift\text{-}add} = 3)$
$(t_{multiply} = t_{add} = t_{shift\text{-}add} = 1 \text{ cycle})$

cycle	ops
1	m1 m2
2	m3 m4 a1
3	a3 a2
4	a4

t = 4 cycles
a = 22

cycle	ops
1	m1
2	m2
3	m3 a1
4	m4 a3
5	a2
6	a4

t = 6 cycles
a = 11

Figure 1.3. Hardware design options for a “node” at the algorithmic, transformational, and resource levels.

operations. Since the area of a shift-add operator is smaller than that of a multiplier, the transformed design has a smaller area for the same execution time. The second transformation shows the retimed design. By changing relative positions of delay elements, the operations can be distributed over the various clock cycles to get a better resource utilization and consequently a smaller area for the same execution time.

3. *Resource level*: A task, for a specified algorithm and transformation set, can be implemented using varying numbers of resource units. Typical resource units are adders and multipliers. Varying the resource units changes the values of the implementation metrics (area and execution time). Figure 1.3-c shows two resource-level implementation options for the direct form biquad. It shows the schedule and area for the fastest (critical path of 4 cycles) and smallest (only one resource of each kind) designs.

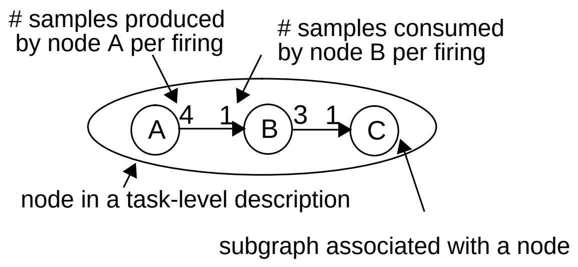
Similarly, different software synthesis strategies can be used to implement a given node in software. For instance, inlined code is faster than code using subroutine calls, but has a larger code size. Thus, there is a trade-off between code size and execution time. Figure 1.4-a shows a 3 node subgraph associated with a node. The schedule describes the sequence in which nodes have to be executed such that at the end of one pass through the graph, the number of samples on all the arcs is restored to the number at start (i.e., there is no accumulation of data). A possible schedule, along with the corresponding generated code, is shown in Figure 1.4-b. Figure 1.4-c shows three possible software implementations differing in size and execution time.

Thus each node in the task-level description can be implemented in several ways in either hardware or software. Current design tools can generate multiple implementations for every task. In system-level design, there are a number of such tasks and the *overall* design is to be optimized. Clearly, it is not enough to optimize each task independently. For example, if each task in the task-level specification were fed to a highlevel hardware synthesis tool that is optimized for speed (i.e., generates the fastest implementation), then the overall area of the system might be too large. *Hardware-software partitioning* is the problem of determining an implementation for each node so that the overall design is optimized. There are two parts to this problem: (1) *Binary partitioning* is the problem of determining, for each node, a hardware or a software mapping and a schedule. (2) *Extended partitioning* is the problem of selecting an appropriate implementation, over and above binary partitioning.

1.1.2 System Synthesis

Once the appropriate implementation for each task has been determined,

(a)



# samples produced by node A per firing
# samples consumed by node B per firing
A
4
1
B
3
1
C
node in a task-level description
subgraph associated with a node

(b)

a possible schedule	corresponding code
A(4BCCC)	see below

```
code for node A;
for(j=1;j<=4; j++)
{
        code for node B;
        code for node C;
        code for node C;
        code for node C;
}
```

(c)

possible schedules	code size	data size	execution time
ABBBBCCCCCCCCCCCC	17c	16b	17t
A(4BCCC)	5c	7b	17t + 4y
A(4B(3C))	3c	7b	17t + 16y

<u>Assumptions</u>
code size of one node = c words
code size for loop constructs = 0
execution time of one node = t cycles
execution overhead for loop constructs = y cycles
buffer size per sample communicated = b words

Figure 1.4. Software design options for a node

the hardware-software *synthesis* problem is that of synthesizing the implementations. Implementations for nodes mapped to hardware or software can be generated by feeding the nodal descriptions to synthesis tools. Figure 1.5 shows both the hardware and the software design paths followed by existing synthesis tools. The hardware design path consists of highlevel hardware synthesis [McFarland90a], followed by logic synthesis [Brayton90], and layout synthesis [DeMicheli86]. The software design path comprises highlevel software synthesis [Pino95a], followed by compilation and assembly.

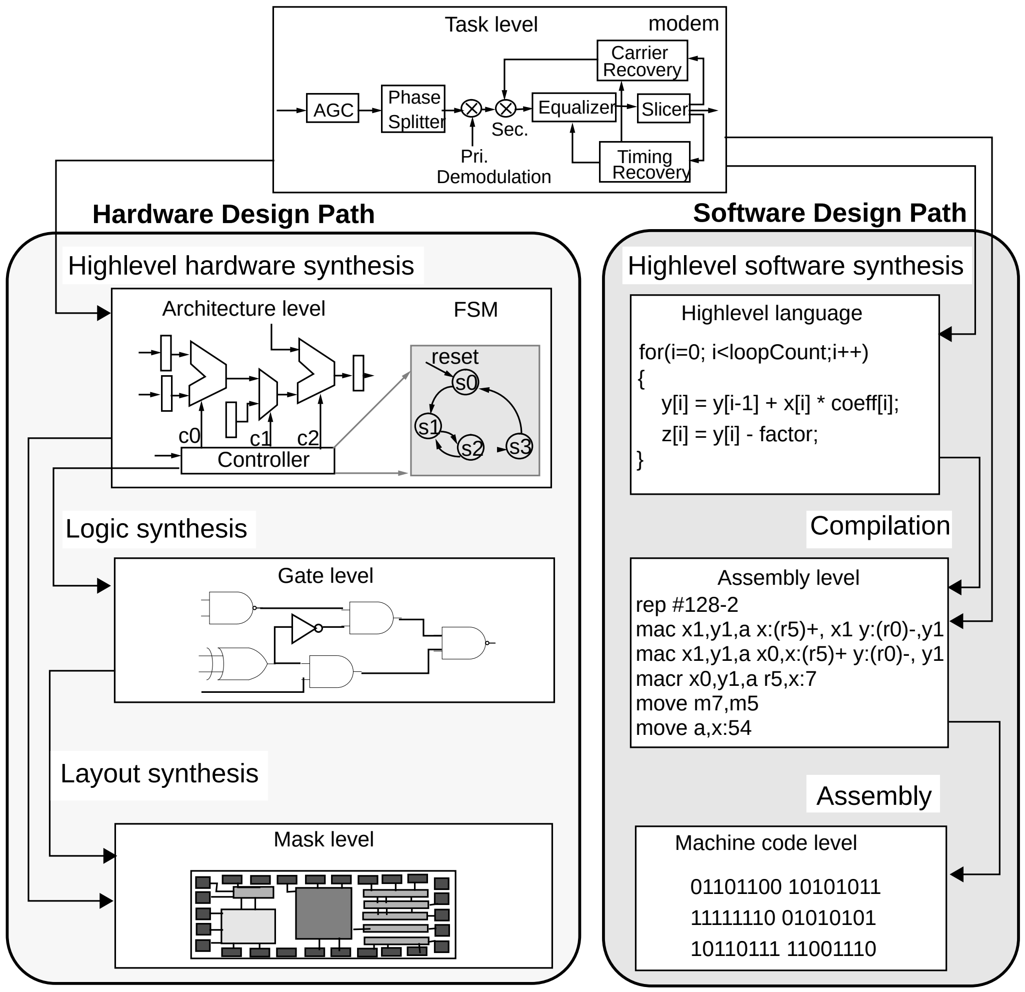


Figure 1.5. Design specification hierarchy and translation across levels of abstraction.

1.1.3 System Simulation

Simulation plays an important role in the system-level design process. At the specification level, it is possible that the design may be specified using a combination of one or more semantic models. Tools that allow the simulation of such heterogeneous specifications are required. At the implementation level, simulation tools that support mixed hardware-software systems are needed.

Throughout the design process, it should be possible to simulate systems where the components are described at different levels of abstraction. As parts of the design evolve from specification to their final implementation, functional models can be replaced by more lower-level structures. A simulation environment should be capable of supporting these aspects.

1.1.4 Design-Space Exploration

System-level design is not a black-box process, but relies considerably on user creativity. For instance, the user might want to experiment with the design parameters, the tools used, or the sequence in which these tools are applied. As such, there is no hardwired design methodology. The design process could get quite unwieldy 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 fall under two categories: estimation and management. Estimation tools are primarily used for *what-if* analysis, i.e., they give quick predictions on the outcome of applying certain synthesis or transformation tools. Management tools orchestrate the design process, i.e., for systematic control of the design data, tools, and flow.

Design methodology management includes tools for visualizing the design process and managing the invocation of tools. Since tools involved in the system-level design process are often computationally intensive, it is important to avoid unnecessary invocation of tools. This requires that the design flow be specified in a modular way so that only desired tools may be invoked. Further, the designer should not have to manually keep track of the tools that have been run. Thus, in order to support efficient exploration of the design space, a mechanism to manage the design flow, tools, and data is required.

1.2 The Codesign Philosophy for System-level Design

Designing systems containing both hardware 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. Our approach to designing such systems is to adopt the *codesign*[1] philosophy.

The key tenet in codesign is to avoid isolation between hardware and software designs. The strategy allows the 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 the

1. One of the early definitions of codesign was given by Franke and Purvis [Franke91] as "The system design process that combines the hardware and software perspectives from the earliest stages to exploit design flexibility and efficient allocation of function". Wolf [Wolf94] emphasizes that "the hardware and software must be designed together to make sure that the implementation not only functions properly but also meets performance, cost, and reliability goals".

tightly coupled design of the hardware and software through a unified framework. The goal of codesigning the hardware and software components of a system is to achieve high-quality designs with a reduced design time.

This thesis presents a systematic approach to the system-level design of embedded signal processing systems, based on the codesign philosophy.

2

THE DESIGN ASSISTANT

Since system-level design oversees highlevel 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. Each task can be represented by a multiplicity of algorithms, each algorithm selection can in turn be represented by several transformation-level and resource-level options. The system-level designer is faced with the question of selecting the best design option. In the context of mixed hardware-software systems, system-level design offers even greater challenges. How do we take advantage of the inherently distinct properties of hardware and software and still couple tightly their design processes? How can the mixed hardware-software system be simulated? In the light of the numerous tools involved in this process, can we relieve the designer of the burden of design management?

In order to address these issues, we present a framework called the *Design Assistant*. The Design Assistant provides a platform that spans the entire gamut of system-level design. It consists of: (1) specific tools for partitioning, synthesis, and simulation that are configured for a particular codesign flow by the user, and (2) an underlying design methodology management infrastructure for design-space exploration. The architecture of the Design Assistant is shown in Figure 2.1. The inputs to the Design Assistant include the design specification and the design constraints, and the output is an implementation for the system. The user configures the tools and constructs a design flow by determining the connectivity and the parameters of the tools. The design methodology manager is a transparent infrastructure that manages the design data, tools, and flow.

This chapter describes the various components of the Design Assistant. In Section 2.1, a typical codesign flow is discussed and the various tools required in the codesign process are outlined. In Section 2.2, the requirements of a design methodology management infrastructure are discussed. In Section 2.3, we outline the assumed specification semantics. The assumed target architecture is described in Section 2.4. Key restrictive assumptions made in the thesis are listed in Section 2.5.

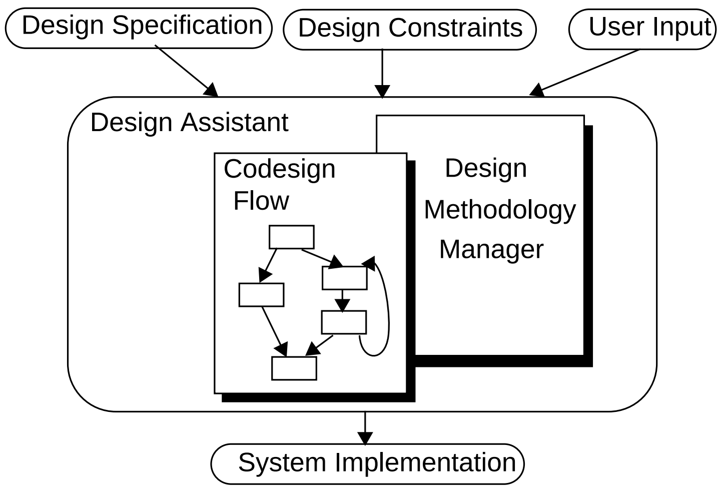


Figure 2.1. The Design Assistant

2.1 A Typical Codesign Flow and Tools for Codesign

A typical codesign flow is shown in Figure 2.2. A task-level specification (for example, a modem specified by a dataflow graph as in Figure 1.1) 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 objective is assumed to be to minimize the total hardware area. The design constraints include

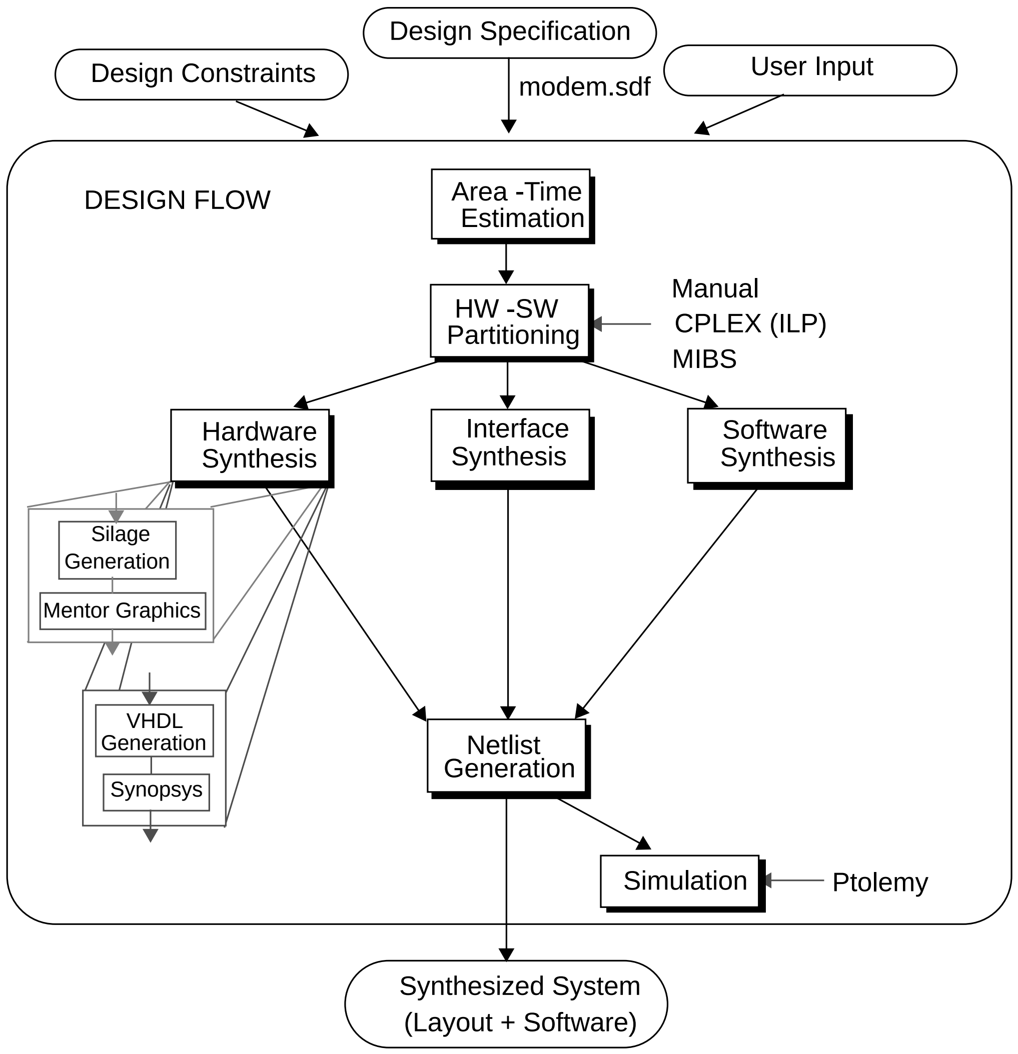


Figure 2.2. A Codesign Flow.

the desired throughput and the architectural model (maximum allowable hardware area, memory size, communication model, etc.). The *design flow* describes the sequence of tools that operate on the design data to generate the final implementation.

The components of the codesign flow include tools for estimation, partitioning, synthesis, and simulation.

Estimation

The *Estimation* tool generates estimates of the implementation metrics (area and execution time requirements) for each of the nodes in the graph in different implementations in hardware and software realizations. These estimates are used by the partitioning tool. Details of the estimation tool are discussed in Section 3.4.1 and Appendix A8.

Partitioning

After obtaining the estimates of the area and execution times, the next step in the codesign flow is *partitioning*. The goal of partitioning is to determine, for each task, three parameters: mapping (whether it is in hardware or software), schedule (*when* it executes, relative to other tasks), and implementation (which *type* of implementation option to use with respect to the algorithm, transformation, and area-time value). The Design Assistant framework allows the user to experiment with different partitioning tools. The figure shows three possible partitioning tools: manual, an ILP solver (CPLEX), or an efficient heuristic (MIBS).

Partitioning is a non-trivial problem. Consider a task-level specification, typically in the order of 50 to 100 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 $(2*5)^{100}$ design options in the worst case! Although a designer may have a preferred implementation for some (say p) nodes, there are still a large number of design alternatives with respect to the remaining nodes ($(2*5)^{100-p}$). Determining the best design option for these remaining nodes is, in fact, a constrained optimization problem. In this thesis, we will focus on analytical tools to solve this problem, although the Design Assistant framework itself does not presuppose a particular tool.

In Chapter 3, we formulate the *extended partitioning* problem. Given the estimates for area and time values of each design option for all the tasks, we develop heuristics to determine the design options that result in a minimum overall area approximately, and still satisfy the throughput constraints. Heuristics are required because exact methods are computationally intensive. A heuristic, called MIBS (Mapping and Implementation Bin Selection), that solves the extended partitioning problem is presented in Chapter 3. MIBS is developed in two stages. First, in Section 3.3, an algorithm (Global Criticality/Local Phase, or GCLP) that solves just the hardware/software mapping and scheduling problem[1] is presented. GCLP is a very efficient heuristic; the complexity is $O(|N|^2)$, where $|N|$ is the number of tasks in the design specification. It uses a combination of global and local measures in order to determine the best mapping. The MIBS algorithm, described in Section 3.5, determines the mapping and implementation bin by a joint optimization process. It uses GCLP as a core and has cubic complexity in the order of number of tasks.

Cosynthesis

Once the application is partitioned into hardware and software, the individual hardware, software, and interface components are synthesized. The particular synthesis tool used depends on the desired technology. The figure shows just two

1. We call this the “binary partitioning problem”.

possibilities for hardware synthesis: (1) Silage [Hilfinger85] code can be generated for the nodes mapped to hardware, and the generated code can be passed through the Mentor Graphics [MentorGraphics1] tools to generate a standard-cell implementation (2) VHDL code can be generated and passed through Synopsys [Synopsys] tools to generate a gate-array implementation. Similarly, different software synthesis strategies can be used; for instance, the software synthesis tool could generate C or assembly code. The interface generation depends on the desired architectural model. For instance, in a memory-mapped communication model, the address decoders and latches as well as the code that writes to and reads from the memory locations need to be synthesized.

There are several published approaches to the problem of synthesizing hardware and software from highlevel specifications. Typically, the complexity of the input to these synthesis systems corresponds to that of a single task in a system-level specification. Instead of reinventing the task-level synthesis tools, we have developed a system-level synthesis approach that builds upon them. The key aspects of this synthesis approach are discussed in Chapter 4.

Cosimulation

Once the hardware, software, and interface components are synthesized, the *Netlist Generator* puts them together and the system is simulated. Ptolemy [Buck94a] is a simulation environment that allows different models of computation to interact seamlessly. Due to its support for multiparadigm simulation, Ptolemy is suitable for simulating heterogeneous systems through the entire design process, from the specification to the implementation levels. The use of Ptolemy for hardware-software cosimulation is discussed in Chapter 4.

2.2 Requirements for Design-Space Exploration

As discussed in Chapter 1, the design process is not a black-box push-button process; it involves considerable user interaction. The user experiments with different design choices; design-space exploration is the key to system-level design. Managing the complexity of this design process is non-trivial. The features needed for efficient design-space exploration include:

1. Modular and configurable flow specification mechanisms

   In Figure 2.2, the user might be interested in first determining whether a feasible partition exists. At this point only the *Estimation* and *Partition* tools need to be invoked; subsequent tools need not be run. Unnecessary tool invocations can be avoided if the flow specification is modular.

   A number of design options are available at each step in the design process. For instance, the *Partition* tool can be either a manual partitioning tool, or an exact (but time consuming) tool such as CPLEX using integer linear programming techniques, or an efficient heuristic such as MIBS. Depending on the available design time and desired accuracy, one of these is selected. This selection can be done either by the user, or by embedding this design choice within the flow. A design flow with a configurable flow specification mechanism is thus compact and efficient.

2. Mechanisms to systematically track tool dependencies and automatically determine the sequence of tool invocations

   The user should not have to keep track of tools that have already run and those that need to be run. Also, if a specific tool is changed on the fly, the entire system need not be re-run; only those tools that are affected should be run. A mechanism that automatically determines the sequence of tool invocations is needed. For instance, if the user changes the hardware synthesis mechanism (perhaps a standard-cell based design instead of one based on field programmable gate arrays), the system should only invoke the dependencies of the hardware synthesis tool (in this case: *Hardware Synthesis, Netlist Generation, Simulation*).

3. Managing consistency of design data, tools, and flows

   Different types of tools, with varying input and output formats, are used in the system-level design process. At the very least, a mechanism to automatically detect incompatibilities between tools is required. Data translators could also be invoked automatically.

   At the system level, it is no longer sufficient to keep track of versions of data — versions of tools and design flows also need to be maintained.

A *design methodology management* infrastructure that supports these requirements is described in Chapter 5.

2.3 Specification Semantics

A complete system-level design is likely to involve various subsystems, each of which is specified in a different model of computation. Figure 2.3 shows

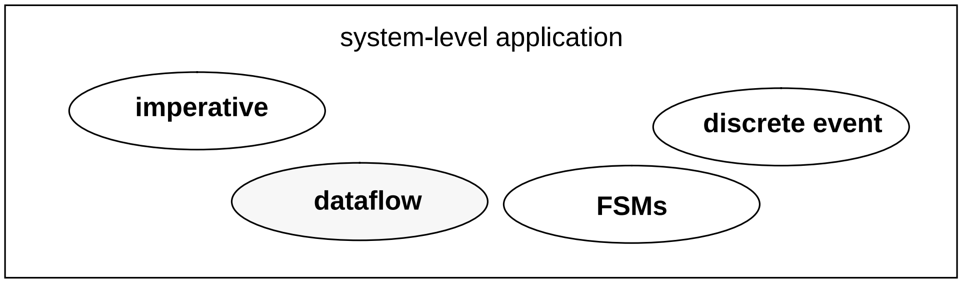


Figure 2.3. Models of computation used in system-level modeling.

four classes of semantics that can be used for abstract system-level modeling. A particular model may be better-suited for a certain part of an application, based on its properties. For instance, control-dominated components of an application are best described by hierarchical FSMs, while computation-dominated components are better-suited to dataflow semantics. FSM and dataflow semantics have been shown to be amenable to both hardware and software synthesis. Discrete-event models are commonly used for hardware modeling and are usually implemented in hardware. Another possible class of semantics is the imperative model, such as that found in C or C++. Such a model is usually better-suited for a software implementation. Thus, an application can be specified using a combination of these semantics, with each component subsystem being represented in a model of computation best suited for it.

We focus on periodic real-time signal processing applications with fixed throughput constraints. Many such applications can be specified in the synchronous dataflow (SDF) model of computation [Lee87] quite naturally. In this model, an application is represented as a graph, where nodes represent computations and arcs indicate the flow of data. A node fires when it receives data on all of its inputs, and on firing, it generates data on all of its outputs. SDF is a special case of dataflow, where the number of samples consumed and produced by a node firing is known statically (at compile time). This makes it possible to develop efficient compile-time scheduling techniques. The SDF specification supports manifest iterations, hierarchy, delays, feedback, and multirate operations. Synthesis of both hardware [Rabaey91] and software [Pino95a] from the SDF model of computation has been demonstrated and the model has proven useful for a reasonable set of signal processing applications.

In this thesis we focus on the design of the components of an application that are represented in the SDF model of computation; the entire application itself will likely use a variety of models, however.

Figure 1.1 shows a typical task-level SDF specification in Ptolemy. The SDF graph can be translated into a directed acyclic graph (DAG), representing precedences between tasks, using techniques discussed in [Lee87]. This DAG is the input to the hardware-software partitioning algorithms. We assume that a hierarchically specified node in the SDF graph (such as the AGC in Figure 1.1) is not flattened when the SDF graph is converted to a DAG, i.e., it retains its hierarchical structure. The user can guide the partitioning process by choosing the hierarchy. The hierarchical representation of a node is called its *subgraph*.

2.4 Target Architecture

Although the Design Assistant does not assume a target architecture, the particular partitioning and synthesis tools used depend on the underlying target architecture. Figure 2.4 shows two typical architectures of mixed hardware-software systems. Figure 2.4-a illustrates a core-based ASIC. This is an emerging design style, where a programmable processor core is combined with a custom datapath within a single die [Bier95b]. Such core-based designs offer numerous advantages: performance improvement (due to critical components being implemented in custom datapaths, and faster internal communication between the hardware and software), field and mask programmability (due to the programmable core), and area and power reduction (due to integration of hardware and software within a single core). This architecture is especially attractive for portable applications, such as those typically found in digital cellular telephony. Figure 2.4-b illustrates a board-level architecture comprising several programmable and custom

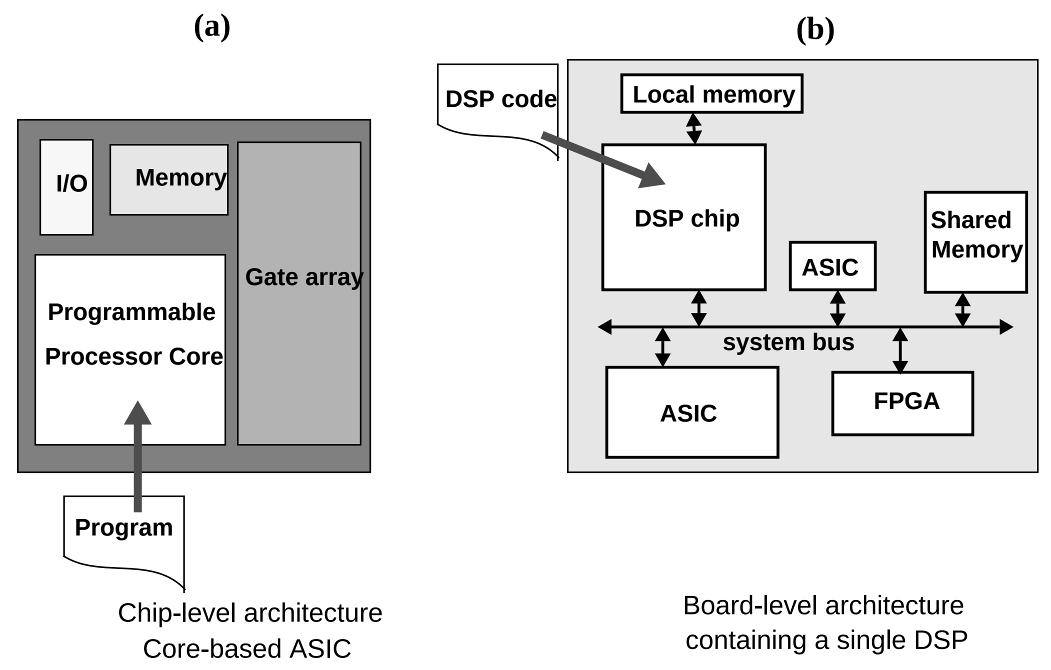


Figure 2.4. Typical target architectures for mixed hardware-software systems.

hardware and software components. These include FPGAs, ASICs, general purpose programmable processors, special purpose processors (DSP for instance), and domain-specific processors (processors optimized for a class of applications).

In this thesis we assume that the target architecture consists of a single programmable processor and multiple hardware modules. Figure 2.5 shows the chosen architecture. The hardware consists of the custom or semi-custom datapath

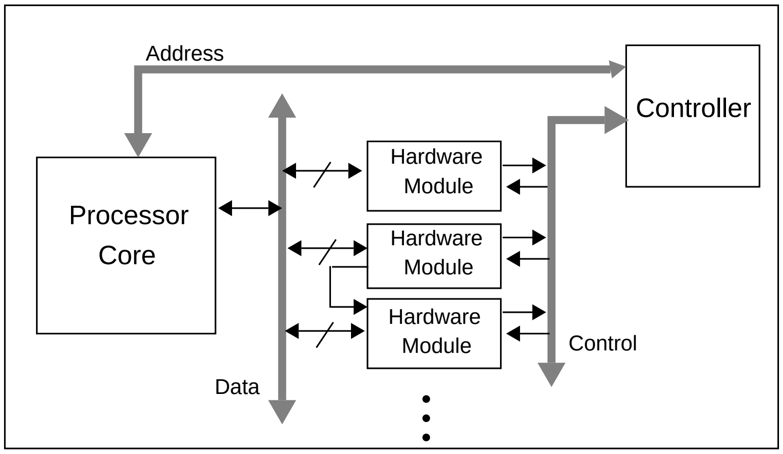


Figure 2.5. The target architecture.

components, and the software is the program running on the programmable component. The hardware-software interface, consisting of the glue logic and controllers, depends on the communication mechanism selected. The specifics of the interface are discussed in Chapter 4. This target architectural model subsumes both the design styles shown in Figure 2.4. Further details of this architecture are given in Section 4.1.1.

2.5 Restrictions in the Design Assistant

In summary, within the large design space, we restrict the applications of interest to be of the type shown in Figure 2.6. To reiterate the key limitations of our approach:

1. We restrict our techniques to the design of applications specified as SDF graphs. In particular, we assume that the DAG generated from such a SDF graph is the input to our partitioning and synthesis tools.
2. The target architecture is assumed to be of the type shown in Figure 2.5, it consists of a single programmable processor and multiple hardware modules.

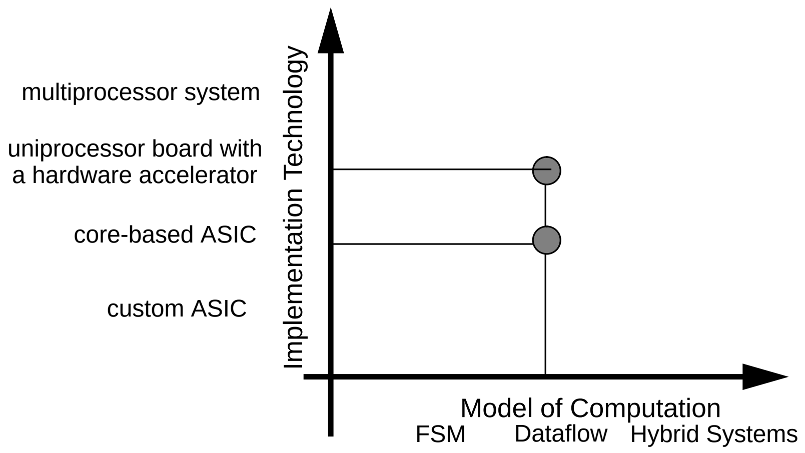


Figure 2.6. Specification semantics and target architecture.

3. In the partitioning and synthesis techniques, we assume that hardware is not reused between modules. The implications of this assumption are discussed in Chapter 4.

2.6 Summary

We propose the *Design Assistant* as a framework for system-level design. The Design Assistant is a unified platform consisting of tools for partitioning, simulating, and synthesizing mixed hardware-software systems. System-level design is not a black-box process. The designer may wish to experiment with different design parameters, tools, and flows. The Design Assistant supports efficient design space exploration; embedded in it is a design methodology manager that manages the design flow, tools, and data.

The next three chapters focus on the various aspects of the Design Assistant. In Chapter 3, an analytical framework for hardware-software partitioning is proposed. In Chapter 4, we discuss the approach used to cosynthesize and cosimulate a partitioned application. The design methodology management infrastructure is presented in Chapter 5.

3

EXTENDED PARTITIONING

In this chapter, we focus on the hardware-software partitioning problem for embedded signal processing applications. As discussed in Chapter 2, the task-level description of an application is specified as an SDF graph. The SDF graph is translated into a DAG representing precedences and this DAG is the input to the partitioning tools.

The *binary partitioning problem* is to map each node of the DAG 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. As discussed in Chapter 1, within a given mapping, a node can be implemented using various algorithms and synthesis mechanisms. These implementations typically differ in area and delay characteristics; we call them "implementation bins". The *extended partitioning problem* is the joint problem of mapping nodes in a DAG to hardware or software, and within each mapping, selecting an appropriate implementation bin (Figure 3.1).

Partitioning is, in general, a hard problem. The design parameters can often be used to formulate it as an integer optimization problem. Exact solutions to such formulations (typically using integer linear programming (ILP)) are intractable for even moderately small problems. We propose and evaluate heuristic solutions. The heuristics will be shown to be comparable to the ILP solution in quality, with a much reduced solution time.

The chapter is organized as follows. In Section 3.1, the *binary partitioning* problem is defined. This is followed by a motivation and a definition for the *extended partitioning* problem. In Section 3.2, we discuss the related work in the area of hardware-software partitioning. In Section 3.3, we present the GCLP algorithm to solve the binary partitioning problem. Its performance is analyzed in Section 3.4. In Section 3.5, we present the MIBS heuristic to solve the extended partitioning problem. The MIBS algorithm essentially solves two problems for each node in the precedence graph: hardware/software mapping and scheduling, followed by implementation-bin selection for this mapping. The first problem, that of mapping and scheduling, is solved by the GCLP algorithm. For a given mapping, an appropriate implementation bin is selected using a bin selection procedure. The bin selection procedure is described in Section 3.6. The details of the MIBS algorithm are described in Section 3.7, and its performance is analyzed in Section 3.8.

3.1 Problem Definition

In Section 3.1.1, we state the major assumptions underlying the partitioning problem. The binary partitioning problem is defined in Section 3.1.2. The extended partitioning problem is motivated and defined in Section 3.1.3.

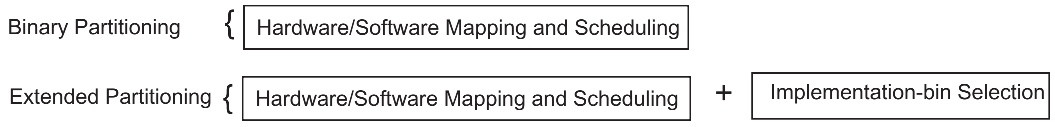

Binary Partitioning
Hardware/Software Mapping and Scheduling
Extended Partitioning
Hardware/Software Mapping and Scheduling
+
Implementation-bin Selection

Figure 3.1. The Extended Partitioning Problem

3.1.1 Assumptions

The partitioning techniques discussed in this thesis are based on the following assumptions:

1. The precedences between the tasks are specified as a DAG ($G = (N, A)$). The throughput constraint on the SDF graph translates to a deadline constraint D, i.e., the execution time of the DAG should not exceed D clock cycles.
2. The target architecture consists of a single programmable processor (which executes the software component) and a custom datapath (the hardware component). The software and hardware components have capacity constraints — the software (program and data) size should not exceed AS (memory capacity) and the hardware size should not exceed AH. The communication costs of the hardware-software interface are represented by three parameters: ah_{comm}, as_{comm}, and t_{comm}. Here, ah_{comm} (as_{comm}) is the hardware (software) area required to communicate one sample of data across the hardware-software interface and t_{comm} is the number of cycles required to transfer the data. The parameter ah_{comm} represents the area of the interface glue logic and as_{comm} represents the size of the code that sends or receives the data. In our implementation we assume a self-timed blocking memory-mapped interface. We neglect the communication costs of software-to-software and hardware-to-hardware interfaces.
3. The area and time estimates for the hardware and software implementation bins of every node are assumed to be known. The specific techniques used to compute these estimates are described in Section 3.4.1.
4. In this work we assume that there is no resource sharing between nodes

mapped to hardware. The issue is discussed further in Chapter 4.

3.1.2 Binary Partitioning

The binary partitioning problem (*P1*): Given a DAG, area and time estimates for software and hardware 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 (M_i) and the start time for the execution of the node (schedule t_i), such that the total area occupied by the nodes mapped to hardware is minimum.

P1 is combinatorial in the number of nodes ($O(2^{|N|})$ by enumeration). *P1* can be formulated exactly as an integer linear program (ILP) as shown in Appendix A1. It is shown to be NP-hard in Appendix A2.

3.1.3 Extended Partitioning

As discussed in Chapter 1, in addition to selecting a hardware or software mapping for the nodes, there is yet another dimension of design flexibility — for a particular mapping, a node can have different implementation alternatives. These implementation alternatives correspond to different algorithms, transformations, and synthesis mechanisms that can be used to implement a node. Figure 3.2 shows the Pareto-optimal points in the area-time trade-off curves for the hardware implementation of typical nodes. The nodes shown include a pulse shaper, a timing recovery block, and an equalizer. The design points on the curves represent different implementations for the node resulting from different synthesis techniques. The sample period is shown on the X axis, and the corresponding hardware area required to implement the node is shown on the Y axis. The left-most point on the X axis for each curve corresponds to the critical path of that node; it represents the fastest possible implementation of the node. As the sample period increases, the same node can be implemented in a smaller hardware area. The right-most point on the X axis for each curve corresponds to the smallest possible area; it represents the point where only one resource of each type is used and the design cannot get any smaller. Thus, the curve represents the design space for the node for a particular algorithm in a hardware mapping. We have generated each of these curves by running a task-level Silage [Hilfinger85] description of the node through Hyper [Rabaey91], a high-level synthesis system. The various design points for a given node were obtained by computing the hardware area corresponding to different sample periods.

Similarly, different software synthesis strategies can be used to implement a given node in software, giving rise to similar area-time trade-off curves for software implementations. For instance, inlined code is faster than code using subroutine calls, but has a larger code size. Thus, there is a trade-off between code size

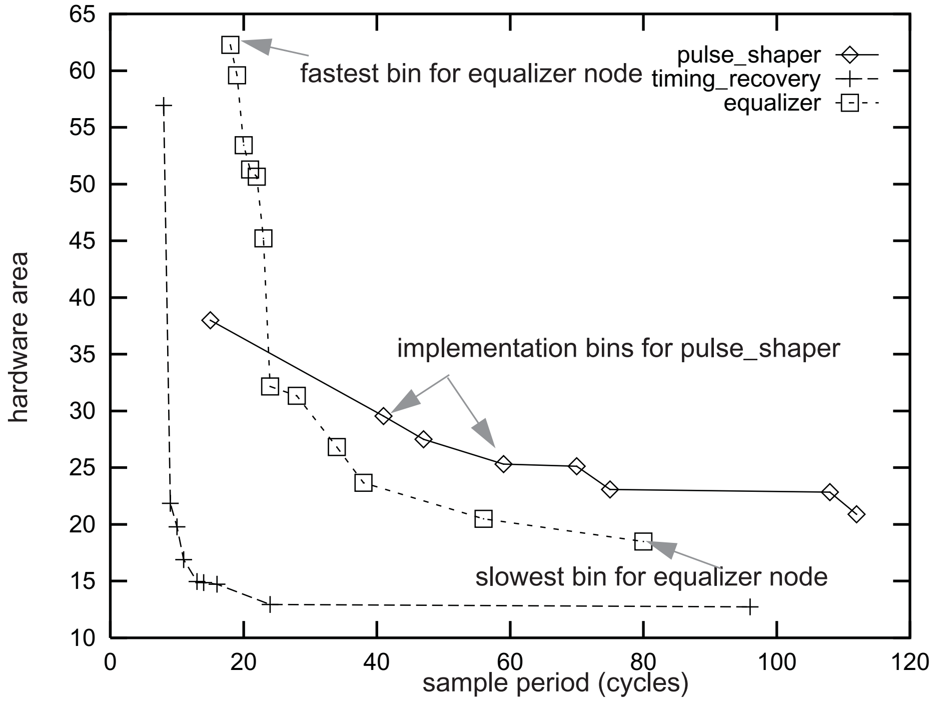


Figure 3.2. Typical hardware area-time trade-off curves, called hardware implementation curves. The design points are called implementation bins.

(program and/or data memory) and execution time. Software for each node can be synthesized using different optimization goals such as throughput, program memory, or data memory [Ritz93][Murthy94].

In Figure 3.2, a curve corresponds to various implementations of a node for a particular algorithm. Each node can also be implemented using different algorithms and transformations. Figure 3.3 generalizes this. Thus, several implementation alternatives (or implementation bins) exist for a node, within each mapping.

We assume that associated with every node i is a hardware implementation curve CH_i, and a software implementation curve CS_i. The implementation curve plots all the possible design alternatives for the node. $CH_i = \{(ah_i^j, th_i^j), j \in NH_i\}$, where ah_i^j and th_i^j represent the area and execution time when node i is implemented in hardware bin j, and NH_i is the set of all the hardware implementation bins (Figure 3.4). $CS_i = \{(as_i^j, ts_i^j), j \in NS_i\}$, where as_i^j and ts_i^j represent the program size and execution time when node i is implemented in software bin j, and NS_i is the set of all the software implementation bins. The fastest implementation bin is called L bin, and the slowest implementation bin is called H bin.

The extended partitioning problem (*P2*): Given a DAG, hardware and software implementation curves for all the nodes, communication costs, resource

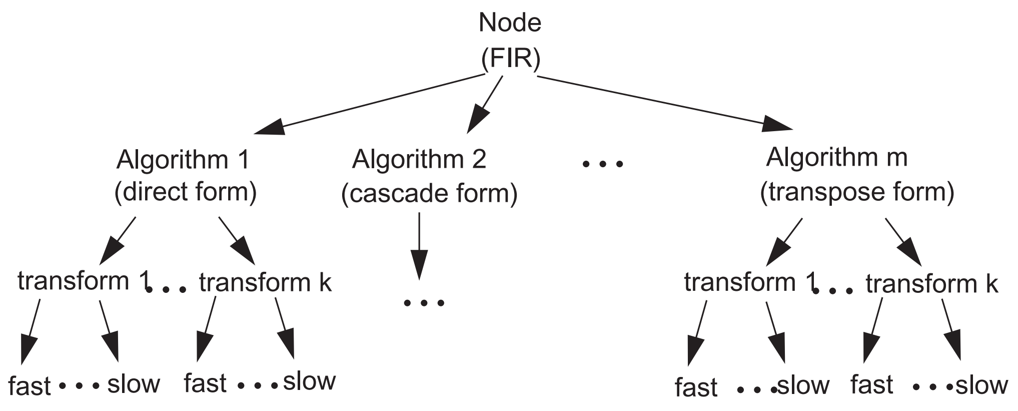


Figure 3.3. Implementation alternatives for a node within a mapping

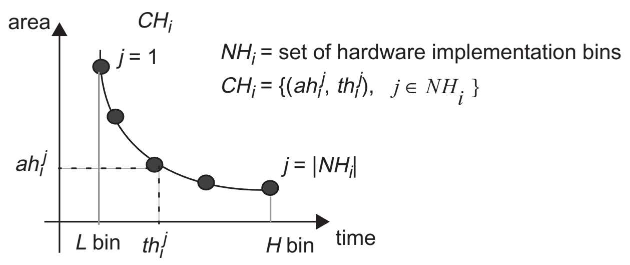


Figure 3.4. CH_i: Hardware implementation curve for node *i*. Each bin corresponds to a possible design option for the node.

capacity constraints, and a required deadline D, find a hardware or software mapping (M_i), the implementation bin (B_i*), and the schedule (t_i) for each node *i*, such that the total area occupied by the nodes mapped to hardware is minimum.

It is obvious that *P2* is a much harder problem than *P1*. It has $(2B)^{|N|}$ alternatives, given B implementation bins per mapping. *P2* can be formulated exactly as an integer linear program, similar to *P1*. An ILP formulation for *P2* is given in Appendix A3.

The motivation for solving the extended partitioning problem is two-fold. First, the flexibility of selecting an appropriate implementation bin for a node, instead of assuming a fixed implementation, is likely to reduce the overall hardware area. In this chapter, we investigate, with examples, the pay-off in using extended partitioning over just mapping (binary partitioning). Secondly, from the practical perspective of hardware (or software) synthesis, solution to *P2* provides us with the best sample period or algorithm to use for the synthesis of a given node.

3.2 Related Work

The binary partitioning problem has received some attention recently. In Section 3.2.1, we will discuss some of the other approaches used to solve the binary partitioning problem.

As of this writing, we are not aware of any work that formulates or solves the extended partitioning problem in system-level design. In Section 3.2.2, we will briefly summarize the related work in physical CAD and highlevel synthesis.

3.2.1 Binary Partitioning: Related Work

The hardware/software mapping and scheduling problem can be formulated as an integer linear program (ILP) and solved exactly. This formulation becomes intractable even for moderately-sized applications (Appendix A1). Some heuristics have been reported to solve the variants of this problem[2].

Gupta *et al.* [Gupta92] discuss a scheme where all nodes (except the data-dependent tasks) are initially mapped to hardware. Nodes are at an instruction level of granularity. Nodes are progressively moved from hardware to software subject to timing constraints in a manner described next. A hardware-mapped node is selected (this node is an immediate successor of the node previously moved to software). The 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.

The scheme proposed by Henkel *et al.* [Henkel94] also assumes an instruction level of granularity. All the nodes are mapped to software at the start. Nodes are then moved to hardware (using simulated annealing) until timing constraints are met. 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.

2. Unless stated differently, the target architecture assumed in these approaches consists of a programmable processor and custom hardware modules that communicate via shared memory.

Baros *et al.* [Baros92] present a two-stage clustering approach to the mapping problem. Clusters are characterized by attribute values (degree of parallelism, amount of data dependency, degree of repetitive computations, etc.) and are assigned hardware and software mappings based on their attribute values. Clusters with a large number of repetitive computations are assigned to software. This approach ignores precedences and scheduling information; it solves only the mapping problem. Scheduling the clusters *after* mapping them to hardware or software leaves little flexibility in trying to achieve a desired throughput. The mapping process does not try to minimize hardware area or maximize software utilization.

D'Ambrosio *et al.* [Ambrosio94] describe 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 moderate-sized problems can become computationally infeasible.

In the approach proposed by Eles *et al.* [Eles94], each node and arc in the graph is annotated with a weight. The weight of a node is a convex combination of four terms: computation load (number of operations executed), uniformity (the ratio of the number of operations in the node to the number of distinct types of operations in the node), parallelism (the ratio of the number of operations in the node to the length of the critical path of the node), and software-suitability (the ratio of the number of software-suitable operations in the node to the number of operations in the node. The software-suitable instructions include floating point operations, file accesses, etc.). The coefficients of these factors are decided by the designer. They claim that a node with a high weight is better suited to hardware. The weights on arcs correspond to the communication cost. The hardware-software partitioning problem is then formulated as the problem of partitioning the graph into two sets such that the sum of the communication costs across the partition is minimized, subject to the constraints that no node with a weight less (greater) than w_l (w_h) is assigned to hardware (software). The parameters w_l and w_h are set by the designer. The problem is solved by simulated annealing. Their scheme ignores throughput constraints; it does not solve the scheduling problem.

Jantsch *et al.* [Jantsch94] present an approach for improving the overall performance of an application by moving parts to hardware, subject to a constraint on the size (number of gates) of the allowed hardware. Initially, all the nodes are mapped to software and are assumed to execute according to a sequential schedule. They restrict themselves to the problem of finding the set of nodes that can be moved to hardware so as to get the maximum speedup, subject to the allowed hardware capacity. The problem translates to the knapsack packing problem (given k items, each with a utility and size, select the set of items that can be packed into a knapsack of a given capacity such that the total utility is maximized) and is solved using dynamic programming. Since they assume a schedule to be available, their approach does not consider overlapped hardware and software execution and hence could lead to poor resource utilization.

Thomas *et al.* [Thomas93] propose a manual partitioning approach for task-level specifications. They discuss the properties of tasks that render them suitable to either hardware or software mapping, such as types of arithmetic operations used, data parallelism, and existence of control structures. In their approach, the designer has to qualitatively evaluate these properties and make a mapping decision. As will be shown later, our approach not only quantifies such properties, but also incorporates mapping decisions based on algorithmic properties into an automated partitioning approach.

We will now briefly discuss some work in related areas such as software partitioning, hardware partitioning, and highlevel synthesis to evaluate the possibility of extending it to the binary partitioning problem.

The heterogeneous multiprocessor scheduling problem is to partition an application into multiple processors. The processors could be heterogeneous, i.e., the execution times on the different processors vary. Approaches used in the literature [Hamada92][Sih93] to solve this problem ignore the area dimension while selecting the mapping; they cannot be directly applied to the hardware/software mapping and scheduling problem.

Considerable attention has been directed towards the hardware partitioning problem in the highlevel hardware synthesis community. The goal in most cases is to meet the chip capacity constraints; timing constraints are not considered. Most of the proposed schemes (for example, [McFarland90b], [Lagnese91], and [Vahid92]) use a clustering-based approach first presented by Camposano *et al.* [Camposano87].

The approaches used to solve the throughput-constrained scheduling problem in highlevel hardware synthesis, (such as force directed scheduling by Paulin *et al.* [Paulin89]), do not extend to the hardware/software mapping and scheduling problem directly. Such approaches do not consider the area and time heterogeneity that is offered by the possibility of hardware or software mappings. Also, they solve only the scheduling problem — mapping (to a particular type of hardware) is assumed to be known. Decoupling the mapping and scheduling problems leaves little flexibility in trying to achieve a desired throughput. Further, most of these approaches use the notion of “slack” or “mobility”, which is the difference between the ASAP and ALAP schedules, for scheduling nodes. In the hardware-software partitioning case, such ASAP and ALAP times cannot be determined exactly since each node has two possible values of execution time (corresponding to hardware and software mappings).

In Section 3.3, we will present an algorithm for automated hardware-software partitioning that overcomes most of the drawbacks of the approaches discussed here. It operates on task-level specifications and attempts to find a partition that meets the throughput constraints, while approximately minimizing the hardware area and maximizing the software utilization. It is an efficient ($O|N|^2$) heuristic that uses a combination of global and local measures while making the mapping and scheduling decisions. It also takes into account the inherent suitability of a node to either a hardware or a software realization based on intrinsic algorithmic properties. For the examples tested, the solutions generated by the algorithm are found to be reasonably close to the corresponding optimal solutions.

3.2.2 Extended Partitioning: Related Work

The authors are not aware of any published work that *formulates or solves* the extended hardware-software partitioning problem in system-level design. The problem of selecting an appropriate bin from the area-time trade-off curve is reminiscent of the technology mapping problem in physical CAD [Brayton90], and the module selection (also called resource-type selection) problem in highlevel synthesis [DeMicheli94], both of which are known to be NP-complete problems.

The technology mapping problem is to bind nodes in a Boolean network, representing a combinational logic circuit, to *gates* in the library such that the area of the circuit is minimized while meeting timing constraints. Gates with different area and delay values are available in the library. Several approaches ([Chaudhary92], among others) have been presented to solve this problem.

The module selection problem is the search for the best resource type for each *operation*. For instance, a multiply operation can be realized by different implementations, e.g., fully parallel, serially parallel, or fully serial. These resource types differ in area and execution times. A number of heuristics ([Ishikawa91], among others) have been proposed to solve this problem.

3.3 The Binary Partitioning Problem: GCLP Algorithm

In this section, we present the Global Criticality/Local Phase (GCLP) algorithm to solve the hardware/software mapping and scheduling problem (*P1*). The notation used is summarized in Table 1.

3.3.1 Algorithm Foundation

The underlying scheduling framework in the GCLP algorithm is based on *list scheduling* [Hu61]. The general approach in list scheduling is to serially traverse a node list (usually from the source node to the sink node in the DAG[3]) and for each node to select a mapping that minimizes an objective function. In the context of *P1*, two possible objective functions could be used: (1) minimize the *finish time* of the node (i.e., sum of the start time and the execution time), or (2) minimize the *area* of the node (i.e., the hardware area or software size). Neither of these objectives by itself is geared toward solving *P1*, since *P1* aims to minimize

3. Note that we are not restricted to a single source (sink) node. If there are multiple source (sink) nodes in the DAG, they can all be assumed to originate from (terminate into) a dummy source (sink) node.

area and meet timing constraints at the same time. For example, an objective function that minimizes finish time drives the solution towards feasibility from the viewpoint of deadline constraints. This solution is likely to be suboptimal (increased area). On the other hand, if a node is always mapped such that area is minimized, the final solution is quite likely infeasible. Thus a fixed objective function is incapable of solving *P1*, a constrained optimization problem. There is also a

Notation	Interpretation
	Graph Parameters
G	DAG $G = (N, A)$, where N is set of nodes representing computations, and A is set of arcs representing precedences
D	Deadline (or makespan) constraint. The execution time for the graph should not exceed D cycles.
ah_i	Hardware area estimate for node i
as_i	Software size estimate for node i
th_i	Hardware execution time estimate for node i
ts_i	Software execution time estimate for node i
$size_i$	Size of node i (number of atomic operations)
	Architectural Constraints
AH	Hardware capacity constraint (total area of nodes mapped to hardware cannot exceed AH)
AS	Software capacity constraint (total area of nodes mapped to software cannot exceed AS)
ah_{comm}	Hardware required for the transfer of one data sample between hardware and software
as_{comm}	Software required for the transfer of one data sample between hardware and software
t_{comm}	Number of cycles required to transfer one sample of data between hardware and software
Algorithm Outputs	
M_i	Mapping for node i (0 for software, 1 for hardware)
t_i	Start time for the execution of node i (schedule)

Table 1. Summary of notation

limitation with list scheduling; mapping based on serial traversal tends to be greedy, and therefore globally suboptimal.

The GCLP algorithm tries to overcome these drawbacks. It *adaptively* selects an appropriate mapping objective at each step[4] to determine the mapping and the schedule. As shown in Figure 3.5*,* the mapping objective for a particular node is selected in accordance with:

1. *Global Criticality* (*GC*): *GC* is a global look-ahead measure that estimates the time criticality at each step of the algorithm. *GC* is compared to a threshold to determine if time is critical. If time is critical, an objective function that minimizes finish time is selected, otherwise one that minimizes area is selected. *GC* may change at every step of the algorithm. The adaptive selection of the mapping objective overcomes the problem associated with a hardwired objective function. The "global" time criticality measure also helps overcome the limitation of serial traversal.

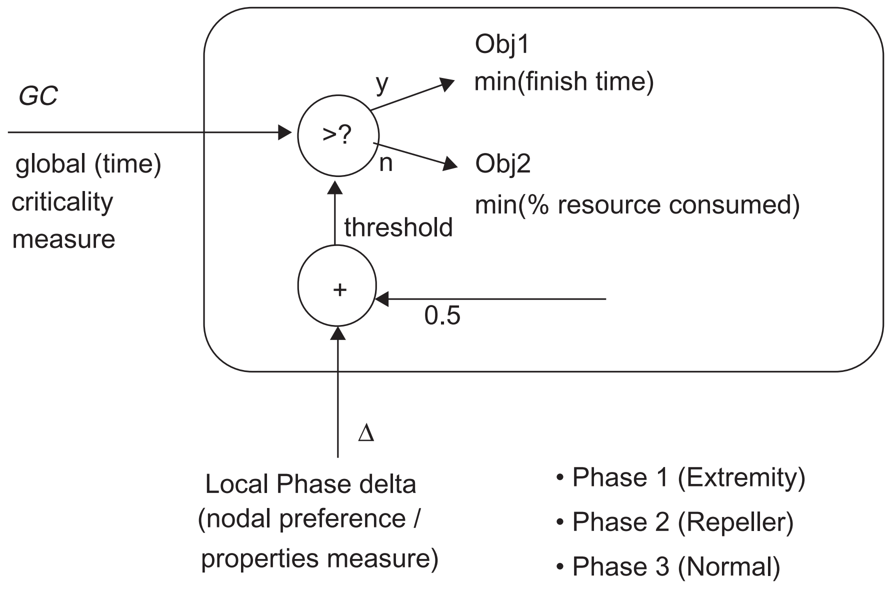


Figure 3.5. Selection of the mapping objective at each step of GCLP.

4. A *step* corresponds to the mapping of a particular node in the DAG.

2. *Local Phase* (LP): LP is a classification of nodes based on their heterogeneity and intrinsic properties. Each node is classified as an extremity (local phase 1), repeller (local phase 2), or normal (local phase 3) node. A measure called *local phase delta* quantifies the local mapping preferences of the node under consideration and accordingly modifies the threshold used in *GC* comparison.

The flow of the GCLP algorithm is shown in Figure 3.6. *N* represents the set of nodes in the graph. $N_U(N_M)$ is the set of unmapped(mapped) nodes at the current step. N_U is initialized to *N*. The algorithm maps one node per step. At the beginning of each step, the global time criticality measure *GC* is computed. *GC* is a global measure of time criticality at each step of the algorithm, based on the currently mapped and unmapped nodes and the deadline requirements. The details of this computation will be given in Section 3.3.2. Unmapped nodes whose predecessors have already been mapped and scheduled are called *ready* nodes. A node is

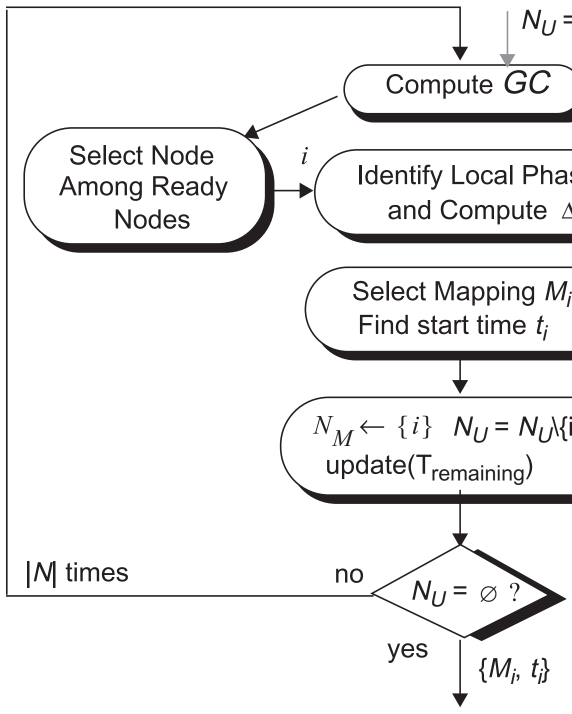

$N_U = N, N_M = \varnothing$
Compute GC
Select Node Among Ready Nodes
i
Identify Local Phase and Compute Δ
Select Objective
Select Mapping M_i Find start time t_i
$N_M \leftarrow \{i\}$ $N_U = N_U \backslash \{i\}$ update($T_{remaining}$)
$|N|$ times
no
$N_U = \varnothing$?
yes
$\{M_i, t_i\}$

Figure 3.6. The GCLP Algorithm.

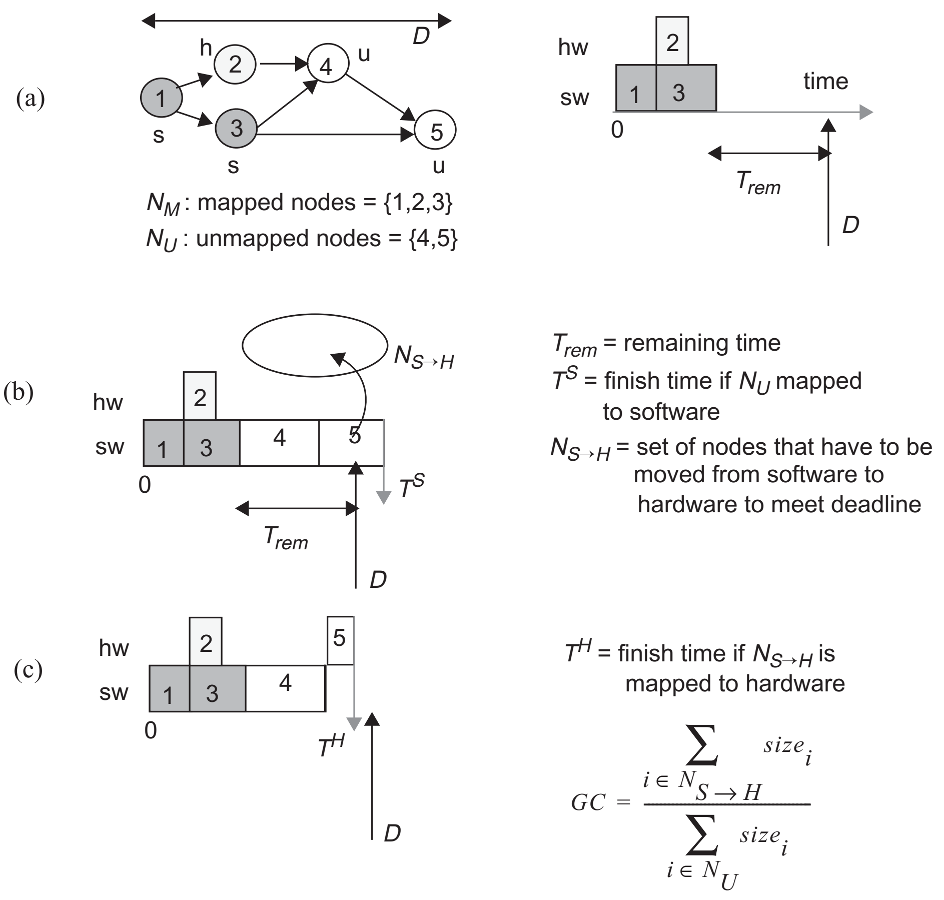
selected for mapping from the set of ready nodes using an urgency criterion, i.e., a ready node that lies on the critical path is selected for mapping. Details of this selection are given in Section 3.3.4. The local phase of the selected node is identified and the corresponding local phase delta is computed. The details of this computation will be given in Section 3.3.3. *GC* and the local phase delta are then used to select the mapping objective. Using this objective, the selected node is assigned a mapping (M_i). The mapping is also used to determine the start time for the node (t_i). The process is repeated $|N|$ times until no nodes are left unmapped.

Next, we describe the computation of the global time criticality.

3.3.2 Global Criticality (*GC*)

GC is a global lookahead measure that estimates time criticality at each step of the algorithm. Figure 3.7 illustrates an example used for computing *GC*. At a given step, the hardware/software mapping and schedule for the already mapped nodes is known (Figure 3.7-a). Using this schedule and the required deadline *D*, the remaining time T_{rem} is first determined. Next, all the unmapped nodes (nodes 4 and 5 in this example) are mapped to software and the corresponding finish time T^S is computed, as shown in Figure 3.7-b. Suppose that T^S exceeds the allowed deadline *D*. Some of the unmapped nodes have to be moved from software to hardware to meet the deadline[5]. Define this to be the set $N_{S \to H}$. Suppose that in the example, $N_{S \to H} = \{5\}$. The finish time (T^H) is then recomputed as shown in Figure 3.7-c. *GC* at this step of the algorithm is defined as the fraction of unmapped nodes that have to be moved from software to hardware, so as to meet feasibility. A high value of *GC* indicates that many as-yet unmapped nodes need to be mapped to hardware so as to get a feasible solution, or in other words, time as a resource is more critical.

5. Assuming there is at least one feasible solution to *P1* satisfying the deadline constraint.



(a)
D
h
u
s
s
u
N_M: mapped nodes = {1,2,3}
N_U: unmapped nodes = {4,5}
hw
sw
time
0
T_{rem}
D
(b)
$N_{S \to H}$
hw
sw
0
T^S
T_{rem}
D
T_{rem} = remaining time
T^S = finish time if N_U mapped to software
$N_{S \to H}$ = set of nodes that have to be moved from software to hardware to meet deadline
(c)
hw
sw
0
T^H
D
T^H = finish time if $N_{S \to H}$ is mapped to hardware
$GC = \frac{\sum_{i \in N_{S \to H}} size_i}{\sum_{i \in N_U} size_i}$

Figure 3.7. Computation of Global Criticality

GC is thus a measure of global time criticality at each step. The following procedure summarizes the computation of *GC*.

Procedure: **Compute_GC**
Input: Mapped (N_M) and Unmapped (N_U) nodes, D, ts_i, th_i, $size_i$, $\forall i \in N$
Output: *GC*

S1. Find the set $N_{S \to H}$ of unmapped nodes that have to be moved from software to hardware to meet deadline D

S1.1. Select a set of nodes in N_U, using a priority function Pf, to move from software to hardware

S1.2. Compute the actual finish time (T^H) based on these $N_{S \to H}$ nodes being mapped to hardware

S1.3. If $T^H > D$ go to S1.1

S2. $$GC = \frac{\sum_{i \in N_{S \to H}} size_i}{\sum_{i \in N_U} size_i}, \ 0 \le GC \le 1$$

In S1.1, the set of nodes to be moved to hardware is selected on the basis of a priority function *Pf*. One obvious *Pf* is to rank the nodes in the order of decreasing software execution times ts_i. A second possibility is to use (ts_i/th_i) as the function to rank the nodes. This has the effect of first moving nodes with the greatest relative gain in time when moved to hardware. A third possibility is to rank the nodes in increasing order of ah_i; nodes with smaller hardware area are moved out of software first. Our experiments indicate that the (ts_i/th_i) ordering gives the best results.

S1.2 determines whether moving this set $N_{S \to H}$ to hardware meets feasibility by computing the actual finish time T^H. The finish time can be computed by an $O(|A| + |N|)$ algorithm, as shown in Appendix A4. If the result is infeasible, additional nodes are moved by repeating steps S1.1 to S1.3.

GC is computed in S2 as a ratio of the sum of the sizes of the nodes in $N_{S \to H}$ to the sum of the sizes of the nodes in N_U. The size of a node is taken to be the number of elementary operations (add, multiply, etc.) in the node. Each node is represented by its size since nodes can be heterogeneous in general.

As indicated earlier, *GC* is a measure of global time criticality; a high *GC* indicates a high global time criticality. *GC* has yet another interpretation; it is a measure of the probability (simplistically speaking) that any unmapped node is mapped to hardware. This probability may change at each step of the algorithm.

3.3.3 Local Phase (LP)

GC is an averaged measure over all the unmapped nodes at each step. This desensitizes *GC* to the local properties of the node being mapped. To emphasize the local characteristics of nodes, we classify nodes as *extremities* (or local phase 1 nodes), *repellers* (or local phase 2 nodes), or *normal* nodes (or local phase 3 nodes).

3.3.3.1 Motivation for Local Phase Classification

Since nodes are at a *task* level of granularity, they are likely to exhibit area and time heterogeneity in hardware and software mappings. Nodes that consume a disproportionately large amount of resource on one mapping as compared to the other mapping are called *extremities* or local phase 1 nodes. For instance, a hardware extremity requires a large area when mapped to hardware, but could be implemented inexpensively in software. The mapping preference of such nodes, quantified by an *extremity measure*, modifies the threshold used in *GC* comparison.

Once a feasible solution is obtained, it is usually possible to further swap nodes between hardware and software so as to reduce the allocated hardware area. GCLP uses the concept of *repellers* or local phase 2 nodes to perform *on-line swaps* (as opposed to post-mapping swaps) of similar nodes between hardware and software. To do this, we identify certain intrinsic nodal properties (called repeller properties) that reflect the inherent suitability of a node to either a hardware or a software mapping. For instance, bit operations are handled better in hardware, while memory operations are better suited to software. As a result, a node with many bit manipulations, relative to other nodes, is a software repeller, while a node with a lot of memory operations, relative to other nodes, is a hardware repeller. Moving a node with many bit manipulation operations out of software is thought of as generating a *repelling force* from the software mapping. Hence the proportion of bit manipulation operations in a node is a software repeller property. Similarly, the proportion of memory operations in a node is a hardware repeller property. A repeller property is quantified by a *repeller value*. The combined effect of all the repeller properties in a node is expressed as the *repeller measure* of the node. All nodes are ranked according to their repeller measures. Given two nodes N1 and N2 with similar software characteristics, if N1 has a higher software repeller measure than N2, and given the choice of mapping one of them to hardware, N1 is preferred. The details of the computation of the repeller measure are deferred to Section 3.3.3.3.

Let us see how the repeller measures effect an *on-line swap* in GCLP. Suppose that the deadline constraint demands that only one of nodes N1 and N2 be mapped to hardware. In this case, the node with a smaller hardware area (say N1) should be selected for hardware mapping. Consider the scenario when N1 is mapped at an earlier step than N2 (due to the serial traversal of the graph). If time is not critical early on when N1 is mapped, mapping based on *GC* alone might map N1 to software. Later as time becomes critical, N2 is mapped to hardware. N1 is, however, a better candidate for hardware mapping than N2. In general, the preferences of all the nodes get modified by serial traversal and possibly suboptimal mapping choices are made. One way to address this is to swap nodes across mappings after the algorithm is done. Alternatively, we use the repeller measure of the node being mapped as an *on-line bias* to modify the threshold used in *GC* comparison. To use our example, the software repeller measure of N1 is used to bias its mapping out of software (towards hardware), thereby making it possible for the yet-unassigned node N2 to get mapped to software. The net effect of such on-line swaps is a reduction in the total hardware area.

Normal nodes (local phase 3) do not modify the default threshold value; their mapping is determined by *GC* consideration only.

In the following sections, we outline procedures to classify nodes into local phases and quantify their local phase measures.

3.3.3.2 Local Phase 1 or Extremity nodes

The bottleneck resource in hardware is area, while the bottleneck resource in software is time. Extremities are nodes that consume a disproportionately large amount of the bottleneck resource on a particular mapping (relative to the other mapping).

A *hardware extremity* node is defined as a node that consumes a large area in hardware, but a relatively small amount of time in software. A *software extremity* node is defined as a node that takes up a large amount of time in software, but a relatively small amount of area when mapped to hardware. The rationale in moving a hardware (software) extremity node to software (hardware) is obvious.

The disparity in the resources consumed by an extremity node i is quantified by an *extremity measure* E_i. The extremity measure is used to modify the threshold to which *GC* is compared when selecting the mapping objective (as shown in Figure 3.5); i.e., E_i is the local phase delta for an extremity node.

Extremity Measure

We now describe a procedure to identify extremity nodes in a graph and compute the extremity measure E_i for all such nodes.

```
Procedure:  Compute_Extremity_Measure
Input:      $ts_i$, $ah_i$, $\forall i \in N$, $\alpha$, $\beta$ percentiles
Output:     $E_i$, $\forall i \in N$, $-0.5 \le E_i \le 0.5$

S1. Compute the histograms of all the nodes with respect to their software execution times ($ts_i$) and hardware areas ($ah_i$)
S2. Determine $ts(\alpha)$ and $ah(\beta)$ that correspond to $\alpha$ and $\beta$ percentiles of the $ts$ and $ah$ histograms respectively
S3. Classify nodes into software and hardware extremity sets $EX_s$ and $EX_h$ respectively:
    If ($ts_i \ge ts(\alpha)$ and $ah_i < ah(\beta)$), $i \in EX_s$ (software extremity)
    If ($ah_i \ge ah(\beta)$ and $ts_i < ts(\alpha)$), $i \in EX_h$ (hardware extremity)
S4. Determine the extremity value $x_i$ for node $i$:
    If $i \in EX_s$, $x_i = \frac{ts_i/ts_{max}}{ah_i/ah_{max}}$, else $x_i = \frac{ah_i/ah_{max}}{ts_i/ts_{max}}$
    where $ts_{max} = max_i\{ts_i\}$ and $ah_{max} = max_i\{ah_i\}$
S5. Order the nodes in $EX_s$ ($EH_h$) by $x$. Denote the maximum and minimum extremity values as $xs_{max}$ ($xh_{max}$) and $xs_{min}$ ($xh_{min}$) respectively.
S6. Compute the extremity measure $E_i$ for node $i$:
    If $i \in EX_s$, $E_i = -0.5 \times \frac{x_i - xs_{min}}{xs_{max} - xs_{min}}$, $-0.5 \le E_i \le 0$
    else if $i \in EX_h$, $E_i = 0.5 \times \frac{x_i - xh_{min}}{xh_{max} - xh_{min}}$, $0 \le E_i \le 0.5$
```

In S1, we compute a distribution of the nodes with respect to their software execution times ts_i and hardware areas ah_i. Parameters α and β represent percentile cut-offs for these distributions. For instance, in S3, a node i is classified as a software extremity node if it lies above α percentile in the ts histogram ($ts_i > ts(\alpha)$) and below β percentile in the ah histogram ($ah_i < ah(\beta)$). Similarly, a node i is classified as a hardware extremity if it lies above β percentile in the ah histogram ($ah_i > ah(\beta)$) and below α percentile in the ts histogram ($ts_i < ts(\alpha)$). Figure 3.8 shows typical histograms and the identification of

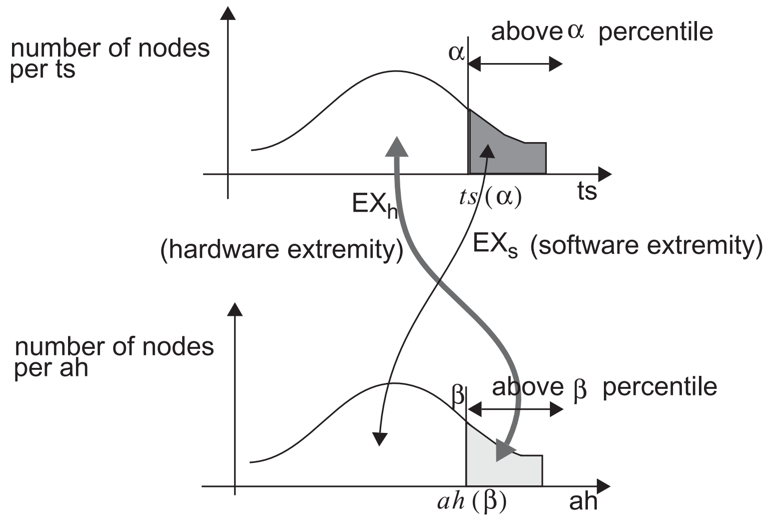


Figure 3.8. Hardware (EX_h) and software (EX_s) extremity sets

extremities. For the examples that we have considered, values of α and β in the range (0.5, 0.75) are used. A value outside this range tends to reduce the number of nodes that fit this behavior and consequently the extremities do not play a significant role in biasing the local preferences of nodes. The extremity value of a node is computed in S4. The extremity measure E_i of a node i is computed in S6, $-0.5 \le E_i \le 0.5$.

Threshold Modification using the Extremity Measure

Let GC_k denote the value of GC at step k when an extremity node i is to be mapped. If E_i is ignored, the threshold assumes its set value of 0.5. Since GC_k is averaged over all unmapped nodes, mapping of node i in this case is based just on GC_k. This leads to:

1. Poor mapping: Suppose node i is a hardware extremity. If $GC_k \ge 0.5$, Obj1 is selected in Figure 3.5 (minimize time), and i could get mapped to hardware based on time-criticality. However, i is a hardware extremity and mapping it to hardware is an obviously poor choice for *P1*.
2. Infeasible mapping: Suppose node i is a software extremity. If $GC_k < 0.5$,

Obj2 is selected in Figure 3.5 (minimize area) and *i* could get mapped to software. Node *i* is a software extremity, however, and mapping it to software could exceed the deadline.

To overcome these problems, the extremity measure E_i is used to modify the default threshold in the direction of the preferred mapping. The new threshold is $0.5 + E_i$. GC_k is compared to this modified threshold. For software extremities, $-0.5 \le E_i \le 0$, so that $0 \le \text{Threshold} \le 0.5$, and for hardware extremities, $0 \le E_i \le 0.5$, so that $0.5 \le \text{Threshold} \le 1$.

3.3.3.3 Local Phase 2 or Repeller Nodes

The use of repellers to effect on-line swaps and reduce the overall hardware area was discussed in Section 3.3.3.1. In this section, we quantify a repeller node with a repeller measure and describe its use in GCLP.

Several repeller properties can be identified for each node. Bit-level instruction mix and precision level are examples of software repeller properties; while memory-intensive instruction mix and table-lookup instruction mix are possible hardware repeller properties. Each property is quantified by a property value. The cumulative effect of all the properties of a node is expressed by a repeller measure.

Let us consider the **bit-level instruction mix**, a software repeller property, in some detail. This property is quantified through its property value called *BLIM*. $BLIM_i$ is defined as the ratio of bit-level instructions to the total instructions in a node *i* ($0 \le BLIM_i \le 1$). For instance, consider the DAG shown in Figure 3.9-a. Suppose that node 2 in the graph is an IIR filter and node 5 is a scrambler. Figure 3.9-b shows the hypothetical *BLIM* values plotted for all the nodes in the DAG in Figure 3.9-a. Node 5, the scrambler, has a high *BLIM* value. Node 2, the IIR filter,

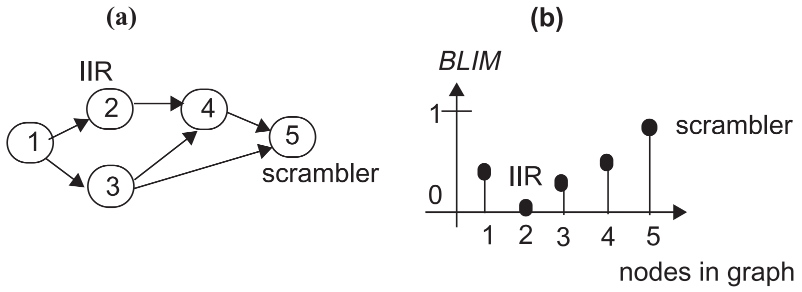


Figure 3.9. An example repeller property.

does not have any bit manipulations, and hence its *BLIM* value is 0. The higher the *BLIM* value, the worse is the suitability of a node to software mapping.

Consider two nodes N_1 and N_2, with software (hardware) areas as_1 (ah_1) and as_2 (ah_2) respectively. Suppose $BLIM_1 > BLIM_2$. Now, if $as_1 \approx as_2$, then $ah_1 < ah_2$ (because bit-level operations can be typically done in a smaller area in hardware). Thus N_1 is a software repeller relative to N_2, based on the bit-level instruction mix property. Based on the discussion in Section 3.3.1, given the choice of mapping one of N1 or N2 to hardware, N1 is preferred for hardware mapping on the basis of the *BLIM* property.

Other repeller properties mentioned earlier are similarly quantified through their property values. The cumulative effect of all the repeller properties in a node is considered when mapping a node. The *repeller measure* R_i of a node captures this aggregate effect. It is expressed as a convex combination of all the repeller property values of the node. The repeller measure is used to modify the threshold against which *GC* is compared when selecting the mapping objective (as shown in Figure 3.5); i.e., R_i is the local phase delta for repeller nodes.

Repeller Measure

The procedure outlined below describes the computation of the repeller measure (R_i) for each node *i*. Let *RH* be the set of hardware repeller properties, and *RS* be the set of software repeller properties. Let $P = RH \cup RS$ be the complete set of repeller properties.

```
Procedure:  Compute_Repeller_Measure
Input:      $v_{i,p}$ = value of repeller property $p$ for node $i$, $i \in N$, $p \in P$
Output:     Repeller measure $R_i$, $\forall i \in N$, $-0.5 \le R_i \le 0.5$
S1. Compute for each property $p$:
    $\sigma^2(v_{i,p})$ = variance of $v_{i,p}$ over all $i$
    $min(v_{i,p})$ = minimum of $v_{i,p}$ over all $i$
    $max(v_{i,p})$ = maximum of $v_{i,p}$ over all $i$
    Let $RX = RH$ if $p \in RH$ or $RX = RS$ if $p \in RS$
    $a_p = \frac{\sigma^2(v_{i,p})}{\sum_{p \in RX} \sigma^2(v_{i,p})}$ = weight of repeller property $p$, $\sum_{p \in RX} a_p = 1$,
S2. Compute the normalized property value $nv_{i,p}$ for each property $p$, of node $i$
    $nv_{i,p} = \frac{v_{i,p} - min(v_{i,p})}{max(v_{i,p}) - min(v_{i,p})}$, $0 \le nv_{i,p} \le 1$.
S3. Compute the repeller measure $R_i$ for each node $i$
    $R_i = \frac{1}{2} \cdot (\sum_{p \in RH} a_p \cdot nv_{i,p} - \sum_{p \in RS} a_p \cdot nv_{i,p})$, $-0.5 \le R_i \le 0.5$.
```

The value $v_{i,p}$ of each repeller property p for node i is obtained by analyzing the node. For instance, consider the bit-level instruction mix property. The bit-level operations (such as OR, AND, EXOR) are first identified in the node. The *BLIM* value of a node i is simply the ratio of the number of bit-level operations to the total number of operations in that node. Other repeller property values are similarly computed.[6]

In S1 of the above procedure, the variance, minimum, and maximum of each repeller property value are computed. The property values are normalized in S2. In S3, the repeller measure for each node is computed as a convex combination

6. Guerra *et al.* [Guerra94] present techniques for the quantification of algorithmic properties such as operator concurrency, temporal density, and regularity.

of the normalized repeller property values. The weight a_p of a property p is proportional to the variance of its value. This deemphasizes properties with small variances in their values. When the repeller measure is used to swap repeller nodes with comparable property values, there is hardly any area reduction; they are not worth swapping. The variance weight ensures this.

Threshold Modification using the Repeller Measure

As described in Section 3.3.3.1, repellers constitute an *on-line swap* to reduce the overall hardware area — a post-mapping swap is avoided. Recall that the repeller measure is a measure of the swapping gain of two similar local phase 2 nodes between hardware and software mappings. Given a choice of mapping just one of two nodes to hardware, the node with a higher software repeller measure is chosen.

Given a phase 2 node i with a sufficiently high software repeller measure, the algorithm tries to achieve a relative shift of i out of software. It modifies the threshold such that the objective selected will favor the complementary mapping. This swap frees up the current resource for an as-yet unscheduled node with a lower repeller property measure, thus reducing the overall allocated hardware area.

The repeller measure R_i is used to modify the threshold so that the new threshold is $0.5 + R_i$. For software repellers, $-0.5 \le R_i \le 0$, so that $0 \le \text{Threshold} \le 0.5$, and for hardware repellers, $0 \le R_i \le 0.5$ so that $0.5 \le \text{Threshold} \le 1$.

3.3.3.4 Local Phase 3 or Normal Nodes

A node that is neither an extremity nor a repeller is defined to be a **local phase 3** node or a **normal** node. The threshold is set to its default value (0.5) when a normal node is mapped. Thus the mapping objective is governed by *GC* alone.

In summary, nodes are classified into three disjoint sets: extremity nodes, repeller nodes, and normal nodes. The local preference of each node is quantified by its measure, represented by a *local phase delta* (Δ). In particular, $\Delta = E_i$ for extremity nodes, $\Delta = R_i$ for repeller nodes, and $\Delta = 0$ for normal nodes. This local phase delta is used to compute the modified threshold: threshold = $0.5 + \Delta$.

3.3.4 GCLP Algorithm

```
Algorithm:    GCLP
Input:        $ah_i$, $as_i$, $th_i$, $ts_i$, $E_i$ (extremity measure), and $R_i$, (repeller measure)
              $\forall i \in N$
              Communication costs: $ah_{comm}$, $as_{comm}$, and $t_{comm}$, and constraints:
              $AH$, $AS$, and $D$.
Output:       Mapping $M_i$ ($M_i \in$ {hardware, software}), start time $t_i$, $\forall i \in N$
Initialize:   $N_U$ = {unmapped nodes} = $N$, $N_M$ = {mapped nodes} = $\phi$.
Procedure:
while {$|N_U| > 0$} {
S1. Compute $GC$
S2. Determine $N_R$, the set of ready nodes
S3. Compute the effective execution time $t_{exec}(i)$ for each node $i$
    If $i \in N_U$            $t_{exec}(i) = GC \cdot th_i + (1-GC) \cdot ts_i$
    else if $i \in N_M$       $t_{exec}(i) = th_i \cdot I(M_i == \text{hw}) + ts_i \cdot I(M_i == \text{sw})$[7]
S4. Compute the longest path $longestPath(i)$, $\forall i \in N_R$ using $t_{exec}(i)$
S5. Select node $i$, $i \in N_R$, for mapping: max($longestPath(i)$)
S6. Determine mapping $M_i$ for $i$:
    S6.1. if ($E_i \neq 0$)        $\Delta = \gamma \cdot E_i$ (local phase 1)
                                   where $\gamma$ is extremity measure weight, $0 \le \gamma \le 1$
          else if($R_i \neq 0$)    $\Delta = \nu \cdot R_i$ (local phase 2)
                                   where $\nu$ is repeller measure weight, $0 \le \nu \le 1$
          else                     $\Delta = 0$; (local phase 3)
```

7. *I*(*expr*) is an indicator function that evaluates to 1 when *expr* is true, 0 else.

```
    S6.2.  $\text{Threshold} = 0.5 + \Delta$, $0 \le \text{Threshold} \le 1$
    S6.3.  If $(GC \ge \text{Threshold})$          $m$: minimize($Obj1$);
           else                                $m$: minimize($Obj2$);
    S6.4.  $M_i = m$; Set($t_i$); $N_U = N_U \backslash \{i\}$; $N_M \leftarrow \{i\}$ ,[8]
         Update($T_{remaining}$, $AH_{remaining}$, $AS_{remaining}$);
}
```

The algorithm maps one node per step. In S1, *GC* is computed using the procedure described in Section 3.3.2. In S2, we determine the set of ready nodes, i.e., the set of unmapped nodes whose predecessors have been mapped. One of these ready nodes is selected for mapping in S5. In particular, we select the node on the maximum longest path, the critical path of the graph. The critical path generally involves unmapped nodes; computing it can present problems since the execution times of such nodes is not known at the current step. To overcome this difficulty, we define the effective execution time ($t_{exec}(i)$) of an unmapped node i as the mean execution time of the node, assuming it is mapped to hardware with probability *GC* and to software with probability (1-*GC*). Here we use the notion of *GC* as a node-invariant hardware mapping probability (see Section 3.3.2). In S6, the mapping and schedule are determined for the selected node. If the node is an extremity (or a repeller) its extremity (or repeller) measure is used to modify the threshold. The contribution of the extremity and repeller measures can be varied by weighting factors γ and ν. In Section 3.8.4, we discuss the tuning of these weights. The mapping objective is selected in S6.3 by comparing *GC* against the threshold. If time is critical, an objective that minimizes the finish time is selected, otherwise one that minimizes resource consumption is selected. The objective functions are:

8. Using set-theoretic notation, $N_U = N_U \backslash \{i\}$ means element i is deleted from set N_U, and $N_M \leftarrow \{i\}$ means element i is added to set N_M.

Obj1: $t_{fin}(i, m)$, where $m \in \{\text{software, hardware}\}$

$$t_{fin}(i,m) = max(max_{P(i)}(t_{fin}(p) + t_c(p,i)),\ tf_{last}(m)) + t(i,m)$$

where

$P(i)$ = set of predecessors of node i, $p \in P(i)$

$t_{fin}(p)$ = finish time of predecessor p

$t_c(p,i)$ = communication time between predecessor p and node i

$tf_{last}(m)$ = finish time of the last node assigned to mapping m

= 0 if m corresponds to hardware

$t(i,m)$ = execution time of node i on mapping m

Obj2: $$\frac{(as_i + as_{comm}{}^{tot})}{AS} \cdot I(m = sw) + \frac{(ah_i + ah_{comm}{}^{tot})}{AH_{remaining}} \cdot I(m = hw)$$

Obj1 selects a mapping that minimizes the finish time of the node. A node can begin execution only after all of its predecessors have finished execution and the data has been transferred to it from its predecessors. Also, a node cannot begin execution on the software resource until the last node mapped to software has finished execution.

Obj2 uses a “percentage resource consumption” measure. This measure is the fraction of the resource area of a node (nodal area plus communication area) to the total resource area. The area $ah_{comm}{}^{tot}$ ($as_{comm}{}^{tot}$) takes into account the total cost of communication (glue logic in hardware and code in software) between node i in hardware (software) and all its predecessors. For the hardware resource, the resource area required by the node is divided by the available hardware area ($AH_{remaining}$). Obj2 thus favors software allocation as the algorithm proceeds.

As shown in Appendix A4, GCLP has a quadratic complexity in the number of nodes. The performance of the algorithm is analyzed in the next section.

3.4 Performance of the GCLP Algorithm

The examples used to analyze the algorithm performance are described in Section 3.4.1. We present two sets of experiments. The first experiment (Section 3.4.2) is a comparison of the solution obtained with GCLP to the optimal solution generated by an ILP formulation (described in Appendix A1). The second experiment (Section 3.4.3) demonstrates the effectiveness of classifying nodes into extremities and repellers. Section 3.4.4 discusses the algorithm behavior with the help of an example trace.

3.4.1 Examples

Two classes of examples are used to analyze the performance of the algorithm: practical examples, and random graphs.

Practical Examples

We consider practical signal processing applications with periodic timing constraints. Two examples are used: 32KHz 2-PSK modem, and 8 KHz bidirectional telephone channel simulator (TCS). These applications are specified as SDF graphs in the Ptolemy [Buck94a] environment. Figure 3.10 shows the receiver section of the modem example. A DAG is generated from the SDF graph representation. Nodes in the DAG are at a task level of granularity. Typical nodes in the modem include carrier recovery, timing recovery, equalizer, descrambler, etc. in the receiver section, and pulse shaper, scrambler etc. in the transmitter section. The nodes in the TCS include linear distortion filter, Gaussian noise generator, harmonic generator, etc. In the modem and TCS examples considered, the DAGs consist of 27 and 15 nodes respectively[9].

9. Much larger examples have been easily solved with the GCLP algorithm (as will be shown in Section 3.4.3), Here, we consider these relatively small examples that can be solved by ILP as well. The intent is to compare the GCLP solution to the optimal ILP solution to evaluate the quality of the heuristic.

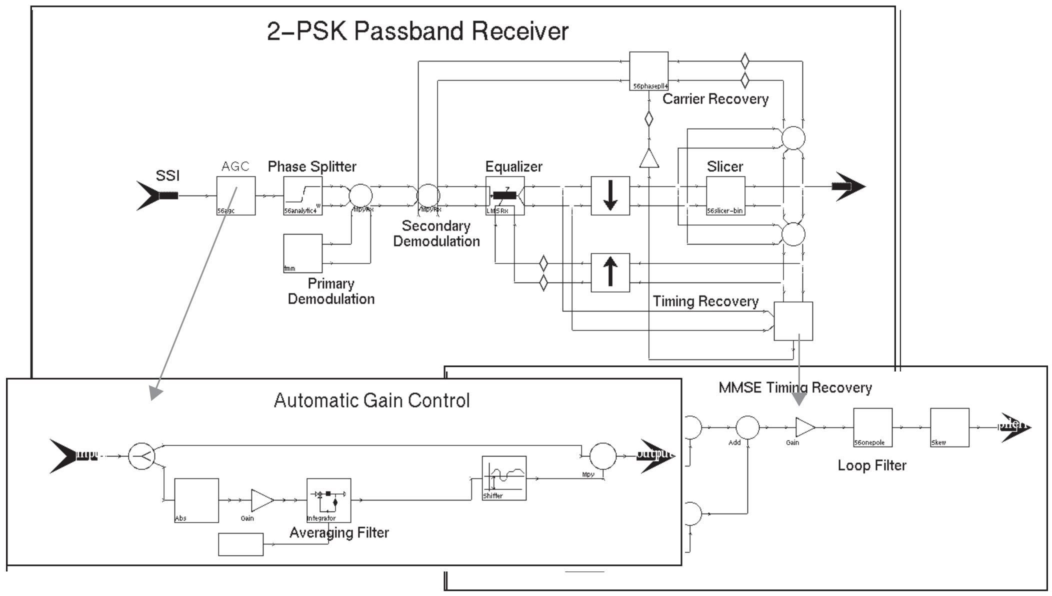


Figure 3.10. Receiver section of a modem, described in Ptolemy. Hierarchical descriptions of the AGC and timing recovery blocks are shown.

The area and time estimates (in hardware and software mappings) for each node in these DAGs are obtained by using the Ptolemy and Hyper environments (Figure 3.11). We assume a target architecture consisting of: (1) Motorola DSP 56000 [Motorola] for the software component, (2) standard-cell based custom hardware generated by Hyper [Rabaey91], and (3) self-timed memory-mapped I/O

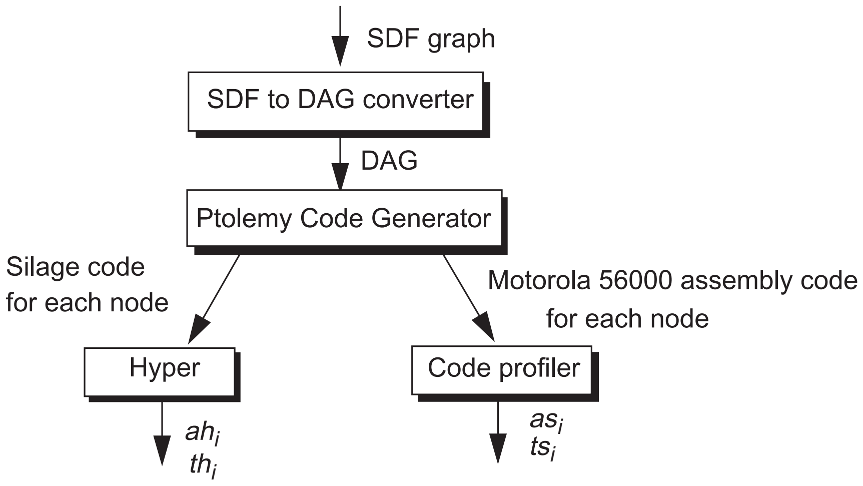


Figure 3.11. Estimation of area and time values in hardware and software.

for hardware-software communication. The code generation feature of Ptolemy is used to synthesize 56000 assembly code [Pino95a] and Silage code [Kalavade93] for each node in the DAG. Estimates of the software area (as_i) and software execution time (ts_i) for each node *i* are obtained by using simple scripts that analyze the generated DSP 56000 assembly code. The Silage code for each node *i* is input to Hyper, which generates estimates of the hardware execution time (th_i) and hardware area (ah_i) for the node. The hardware execution time is computed as the best-case execution time (corresponding to the critical path of the control-dataflow graph associated with the node[10]). The hardware area is computed by setting the sample period to the critical path. Further details of the estimation mechanisms can be found in Appendix A8.

Random Examples

A random graph generator is used to generate a graph with a random topology for a given number of nodes (graph size). The hardware-software area and time estimates of the nodes in the random graph are generated by taking into account the trend observed in real examples. Details of the techniques used to generate the random graphs are given in Appendix A7. For each size, we generate 10 random graphs differing in topology and area and time metrics. The heuristic is applied for each random graph and the average value of the result is reported for that size.

3.4.2 Experiment 1: GCLP vs. ILP

Examples are first partitioned using the GCLP algorithm. The ILP formulation for these examples is then solved using the ILP solver CPLEX[11]. Table 2 lists the GCLP and ILP solutions for the modem and TCS examples. The total hard-

10. This control-dataflow graph is generated by Hyper during the hardware synthesis process.
11. CPLEX is a commercially available optimization software.

example	size	algorithm	**total hardware area** (normalized with respect to hardware area required in ILP solution)	**DSP utilization** (1 - idle_time/D)*100	**solution time**
modem	27	ILP	1.0	93.8%	19190 s
		GCLP	1.1935	84.89%	0.535 s
TCS	15	ILP	1.0	73.5%	6656 s
		GCLP	1.0	73.5%	0.387 s

Table 2. Results from ILP and GCLP algorithm.

ware area (normalized with respect to the optimal solution), DSP utilization, and solution time for all the cases are compared. The solution time represents the CPU time required to generate the solution on a SPARCstation 10. The total hardware area obtained with the GCLP algorithm is quite close to the ILP solution. In the modem example, the GCLP mapping has all but one node identical with the ILP mapping. The GCLP mapping for the TCS is identical to the ILP mapping.

Figure 3.12 compares the total hardware area obtained with the GCLP algorithm to the optimal solution obtained with ILP formulation for a number of random examples. For all tested examples the generated GCLP solution is within **30%** of the optimal solution. Examples larger than 20 nodes could not be solved by ILP in reasonable time[12]. GCLP has been used to solve examples with up to

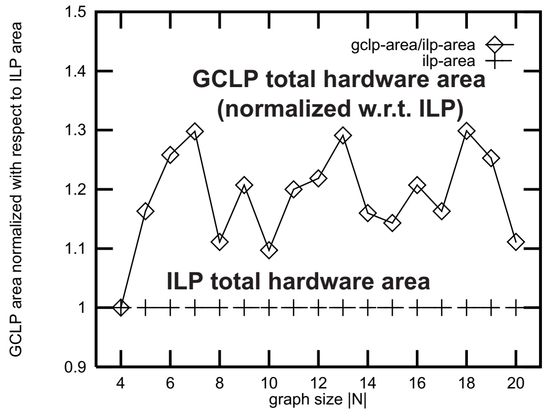


Figure 3.12. GCLP vs. ILP: Random examples

500 nodes with relative ease.

3.4.3 Experiment 2: GCLP with and without local phase nodes

To examine the effect of local phase nodes on the GCLP performance, the GCLP algorithm is applied under three cases:

Case 1. The local phase classification is not used — all nodes are normal nodes with $\Delta = 0$. The objective function is selected by comparing *GC* with the default threshold = 0.5.

Case 2. Nodes are classified as either repellers or normal nodes. For repeller nodes $\Delta = R_i$, otherwise $\Delta = 0$.

Case 3. Complete local phase classification, using extremities, repellers, and normal nodes. For extremity nodes $\Delta = E_i$, for repeller nodes $\Delta = R_i$, and for normal nodes $\Delta = 0$.

In the modem example, it was found that:

1. Repellers reduce the hardware area through on-line swaps (the solution obtained in case 2 is 13% smaller than the solution obtained in case 1).
2. Extremities are seen to match their expected mappings (ex: *Pulse Shaper,* a hardware extremity node, is mapped to software. *Carrier Recovery,* a software extremity node, is mapped to hardware).
3. Repeller nodes are also mapped to their intuitively expected mappings (ex: *Scrambler*, a high *BLIM* software repeller, is mapped to hardware).

Figure 3.13 plots the results of these three cases when applied to random examples. The total hardware areas are normalized with respect to the total hardware area obtained in case 1. It is seen that the use of repellers (case 2) significantly reduces the hardware area as compared to a purely *GC*-based selection in

12. Some fine-tuning of the ILP formulation could improve the ILP solution time slightly.

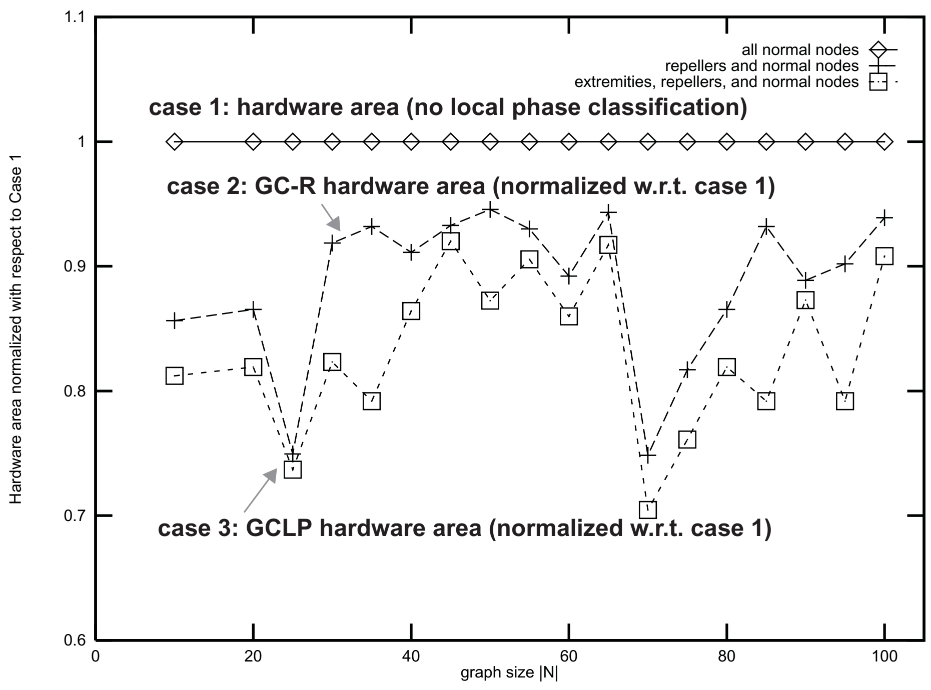


Figure 3.13. Effect of local phase classification on GCLP performance: case 1: mapping based on GC only, case 2: threshold modified for repellers, case 3: complete local phase classification.

case 1. This verifies our premise that repellers effect on-line swaps to reduce the total hardware area. Using both extremity and repeller nodes further improves the quality of the solution. On an average, the complete classification of local phase nodes reduces the total hardware area by **16.82%**, when compared with case 1.

3.4.4 Algorithm Trace

The behavior of GCLP with complete local phase classification of nodes is quite complex. To understand the relation between node classification, threshold, *GC*, and the actual mapping at each step of the algorithm, we illustrate these key parameters in algorithm traces for specific examples.

Figure 3.14 illustrates the *GC* variation and the mapping at each step, ignoring the local phase classification. When *GC* > 0.5, time is critical. Almost always, nodes get mapped to hardware. Eventually, this reduces the time criticality

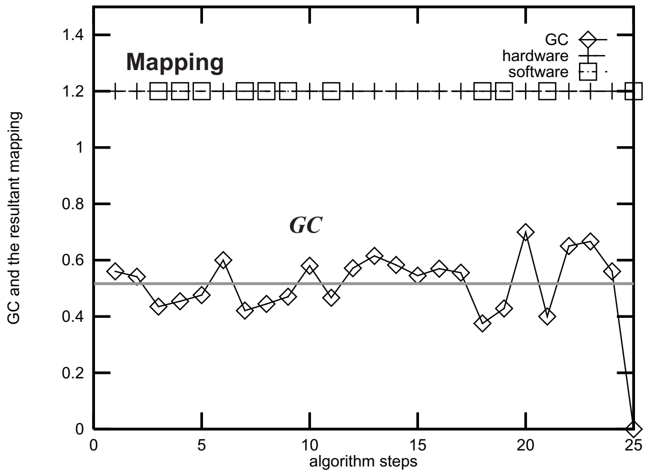


Figure 3.14. Algorithm trace: GC and the Mapping. All normal nodes, threshold = 0.5.

and *GC* reduces. When it drops below 0.5, area minimization is selected as the objective. In most cases, this objective selects software mapping. Subsequently, time becomes critical, *GC* increases, and further nodes get mapped to hardware. Thus *GC* (and the mapping) adapts continually.

Figure 3.15 illustrates the effect of adding extremity nodes to the above example. Nodes mapped in steps 8 and 10 are software extremities. In Figure 3.15-a, the extremity measure is not considered, in Figure 3.15-b, it is. We assume that nodes are mapped in the same order in both these cases. Mapping marked by a cross means software mapping and by a square means hardware mapping. In Figure 3.15-a, the extremity measures are not considered; the threshold assumes its default value (0.5). The software extremity node mapped in step 8 (marked *e1*) gets mapped to software, hence, time criticality increases. When compared to the corresponding mapping in Figure 3.15-b, nodes mapped subsequently in steps 15 and 16 (marked *h1*) get mapped to hardware. A similar effect is observed while mapping nodes in steps 22, 23, and 24 (marked *h2* in Figure 3.15-a). The generated solution has a total hardware area of 1253, and 14 nodes are mapped to hardware. Also, the sample mean of the *GC* over all steps is 0.50448. In Figure 3.15-b, extremity measures are used to change the default threshold. The extremity measure for nodes mapped in steps 8 and 10 lowers the threshold at the points marked *e1* and *e2* in Figure 3.15-b. This induces a hardware mapping. Mapping these software extremities to hardware reduces time criticality. Subsequently, nodes mapped in steps 15 and 16 (marked *s1*) get mapped to software. The total hardware area is 756, and 10 nodes are mapped to hardware. Thus by taking into account the local preference of nodes 8 and 10, the quality of the solution is improved by 40%. Also,

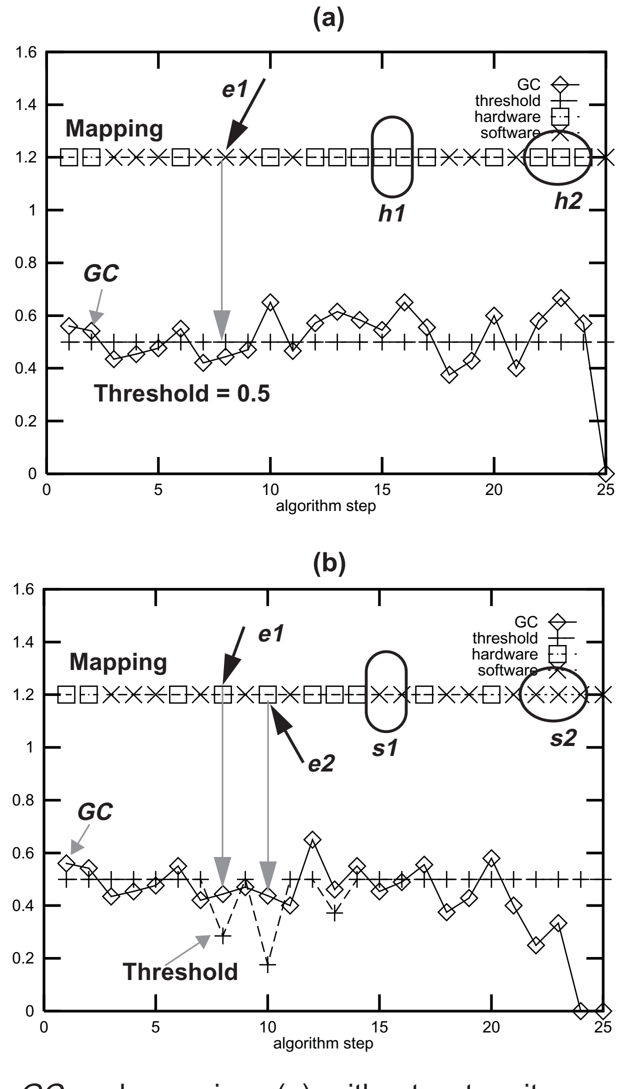


Figure 3.15. *GC* and mapping: (a) without extremity measures (b) with extremity measures.

the sample mean of the *GC* over all the steps is 0.4288.

Figure 3.16 illustrates the effect of repellers on *GC* and mapping at each step of the algorithm applied to a particular example. In this example, nodes mapped in steps 6 and 10 are software repellers; the node mapped in step 6 has a larger repeller measure than the node mapped in step 10. In Figure 3.16-a repellers are not considered, in Figure 3.16-b, they are. We assume that nodes are mapped in the same order in both the cases. In Figure 3.16-a, the repeller measures are not considered when mapping — the threshold assumes its default value of 0.5. In this

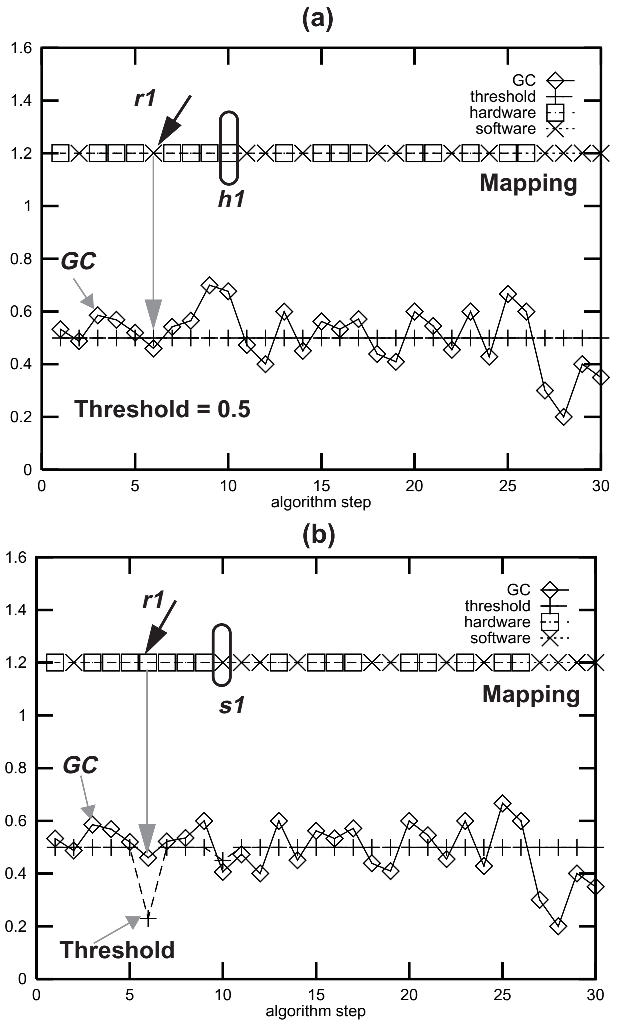


Figure 3.16. GC and mapping: (a) without repeller measures (b) with repeller measures to change the threshold.

case, node mapped in step 6 is mapped to software and the node mapped in step 10 is mapped to hardware. Clearly, this mapping can be improved. In Figure 3.16-b, repeller measures are taken into consideration; they modify the default threshold and hence the mapping. The repeller measure for the node mapped in step 6 lowers the threshold at the point marked *r1* in Figure 3.16-b. This forces the node to get mapped to hardware. The threshold for the node in step 10 is not lowered as much and it gets mapped to software. The mapping of nodes in steps 6 and 10 is thus exactly opposite to that in Figure 3.16-a. Thus nodes in steps 6 and 10 were in effect *swapped on-line* in Figure 3.16-b — the node with a larger repeller measure displaced the one with a smaller measure and this reduced the overall area (1803 in Figure 3.16-a vs. 1677 in Figure 3.16-b, 17 nodes are mapped to hardware in both cases).

3.4.5 Summary of the GCLP algorithm

We have so far discussed the Global Criticality/Local Phase algorithm to solve the binary partitioning problem (*P1*). The key features of the algorithm can be summarized as follows.

Global criticality is a global lookahead measure that quantifies the time criticality at each step of the algorithm, taking into account the currently unmapped nodes and the desired throughput requirements. *GC* is compared with a threshold to select a mapping objective at each step of the algorithm. Nodes that consume disproportionate amounts of resources in hardware and software mappings are classified as extremities. The threshold used for mapping such nodes is modified to account for their mapping preference. The total hardware area is further reduced by using the concept of on-line swaps between repeller nodes. Repeller nodes are classified and quantified on the basis of intrinsic nodal properties.

The threshold used for mapping repeller nodes is modified in accordance with their relative “repulsion” for a mapping. The GCLP algorithm is computationally efficient ($O(|N|^2)$) with a solution quality comparable to the optimal.

3.5 Algorithm for Extended Partitioning: Design Objectives

The GCLP algorithm described so far solves the binary partitioning problem *P1*. The extended partitioning problem *P2* is to jointly optimize the mapping as well as implementation bin for each node. Consider the implementation-bin curve of a node as shown in Figure 3.4. Denote *L* to be the fastest (left-most) implementation bin, and *H* to be the slowest (right-most) implementation bin. As the implementation-bin curve is traversed from bins *L* to *H*, the hardware area required to implement the node decreases. From the viewpoint of minimizing hardware area, each node mapped to hardware can be set at its *H* bin (lowest area). This might, however, be infeasible since the *H* bins correspond to the slowest implementations. The extended partitioning problem is to select an “appropriate” implementation bin and mapping for each node such that the total hardware is minimized, subject to a deadline and resource constraints. This problem is obviously far more complex than the mapping (binary partitioning) problem. Our goal is to design an efficient algorithm to solve the extended partitioning problem. There are two guiding objectives used in the design of this algorithm.

1. Design Objective 1: Complexity that scales reasonably: The binary partitioning problem has $2^{|N|}$ mapping possibilities for $|N|$ nodes in the graph. Given B implementation bins within a mapping, the extended partitioning problem has $(2B)^{|N|}$ possibilities in the worst-case. The algorithm complexity should not scale with the dimensionality (number of design alternatives

per node) of the partitioning process, i.e., if a binary partitioning algorithm has complexity O($|N|^2$), the extended partitioning algorithm should not have complexity O($|N|^{2B}$), since B is typically in the range 5 to 10. Obviously the binary partitioning algorithm cannot be extended directly to solve the extended partitioning problem, since the implementation possibilities explode.

2. Design Objective 2: Reuse of GCLP: Since we already have an efficient algorithm for binary partitioning, the algorithm for extended partitioning should reuse it. This suggests that extended partitioning can be decomposed into two blocks: mapping and implementation-bin selection. GCLP can be used for mapping.

It is not enough, however, to decompose the extended partitioning problem into two isolated steps, namely that of mapping followed by implementation-bin selection. The serial traversal of nodes in a graph means that the implementation bin of a particular node affects the mapping of as-yet unmapped nodes. Since there is a correlation between mapping and implementation-bin selection, they cannot be optimized in isolation. This dependence has to be captured in the algorithm.

Our approach to solving the extended partitioning problem is summarized in Figure 3.17. The heuristic is called MIBS. In the final solution, each node in the graph is characterized by three attributes: mapping, implementation bin, and schedule. As the algorithm progresses, depending on the extent of information that has been generated, each node in the DAG passes through a sequence of three states: (1) free, (2) tagged, and (3) fixed. Before the algorithm begins, all three attributes are unknown. Such nodes are called *free* nodes. Assuming median area and time values, GCLP is first applied to get a mapping and schedule for all the free nodes in the graph. A particular free node (called a *tagged* node) is then

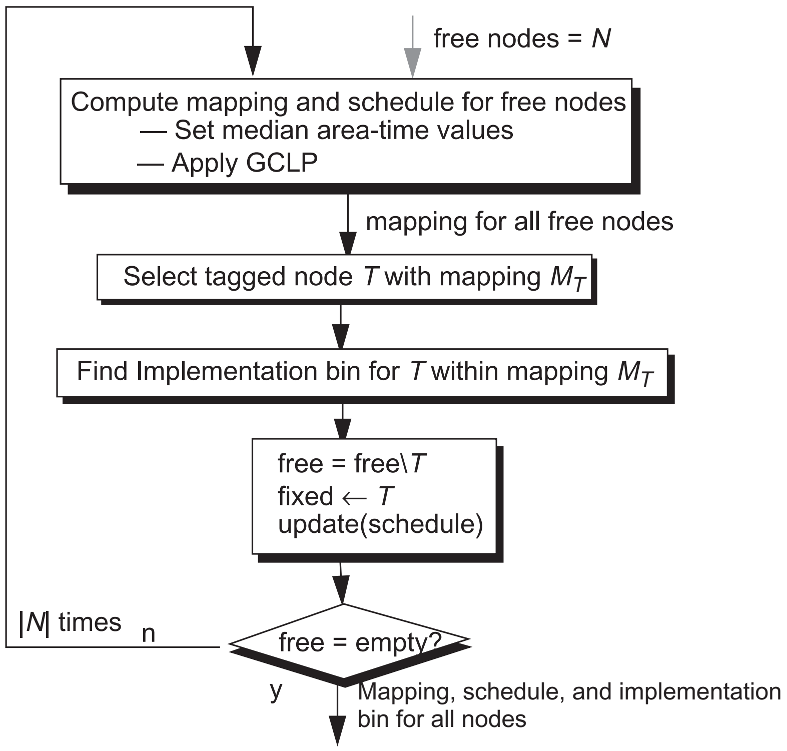

free nodes = N
Compute mapping and schedule for free nodes
— Set median area-time values
— Apply GCLP
mapping for all free nodes
Select tagged node T with mapping M_T
Find Implementation bin for T within mapping M_T
free = free\T
fixed ← T
update(schedule)
|N| times
n
free = empty?
y
Mapping, schedule, and implementation bin for all nodes

Figure 3.17. MIBS approach to solving extended partitioning

selected. Assuming its mapping to be that determined by GCLP, an appropriate implementation bin is then chosen for the tagged node. In the following section, we describe a bin selection procedure that determines the implementation bin for the tagged node. Once the mapping and implementation bin are known, the tagged node becomes a *fixed* node. GCLP is the applied on the remaining nodes and this process is repeated until all nodes in the DAG become fixed; the MIBS algorithm has $|N|$ steps[13] for $|N|$ nodes in the DAG.

The MIBS approach subscribes closely to the design objectives outlined. GCLP is used for mapping (according to Design Objective 2). Since GCLP and bin selection are applied alternately within each step of the MIBS algorithm, there is continuous feedback between the mapping and implementation-bin selection

13. Each *step* of the MIBS algorithm constitutes the determination of the mapping, implementation bin, and schedule of a node.

stages. The MIBS algorithm will be shown to be reasonably efficient ($O(|N|^3 + B \cdot |N|^2)$, where B is the number of implementation bins per mapping. Thus it scales polynomially with the dimensionality of the problem (Design Objective 1).

In the next section, we describe the bin selection procedure to solve the implementation-bin selection problem.

3.6 Implementation-bin Selection

3.6.1 Overview

In the following, we restrict ourselves to the problem of selecting the implementation bin for hardware-mapped nodes only. The concepts introduced here can be extended to software implementation-bin selection as well.

Recall from Figure 3.17 that, in each step of the MIBS algorithm, GCLP is first applied to determine the revised mapping of free nodes. Let the free nodes mapped to hardware at the current step be called $free^h$ nodes. A tagged node is selected from the set of free nodes. Assuming its mapping to be that determined by GCLP, the bin selection procedure is applied to select an implementation bin for the tagged node.

Figure 3.18 shows the flow of the bin selection procedure. The key idea is to use a lookahead measure to correlate the implementation bin of the tagged node with the hardware area required for the $free^h$ nodes. It selects the most *responsive* bin in this respect as the implementation bin for the tagged node.

Computing the lookahead measure can be very complex since the final implementation bins of the $free^h$ nodes are not known at this step. To simplify matters, we assume that $free^h$ nodes can be in either L or H bins[14]. All $free^h$ nodes are

14. A $free^h$ node loses this restriction when it becomes tagged later on.

assumed to be in their *H* bins initially. The lookahead measure (called *bin fraction* BF_T^j) computes, for each bin *j* of the tagged node *T*, the fraction of $free^h$ nodes that need to be moved from *H* bins to *L* bins in order to meet timing constraints. A high value of BF_T^j indicates that if the tagged node *T* were to be implemented in bin *j*, a large fraction of $free^h$ nodes would likely get mapped to their fast implementations (*L* bins), hence increasing the overall area. The *bin fraction curve* (BFC_T) is the collection of the all bin fraction values of the tagged node *T*.

Bin sensitivity is the gradient of BFC_T. It reflects the responsiveness of the bin fraction to the bin motion of node *T*. Suppose that the maximum slope of the bin fraction curve is between bins *k*-1 and *k* (Figure 3.18). Moving the tagged node from bin *k*-1 to *k* shifts the largest fraction of $free^h$ nodes to their *L* bins. Equivalently, the *k* to *k*-1 motion for the tagged node results in the largest reduction of the area of $free^h$ nodes. Hence the (*k*-1)th bin is selected as the implementation bin for the tagged node (B_T*). The computation of the *BFC* and bin sensitivity is described next.

The notation used in the bin selection procedure is summarized Table 3.

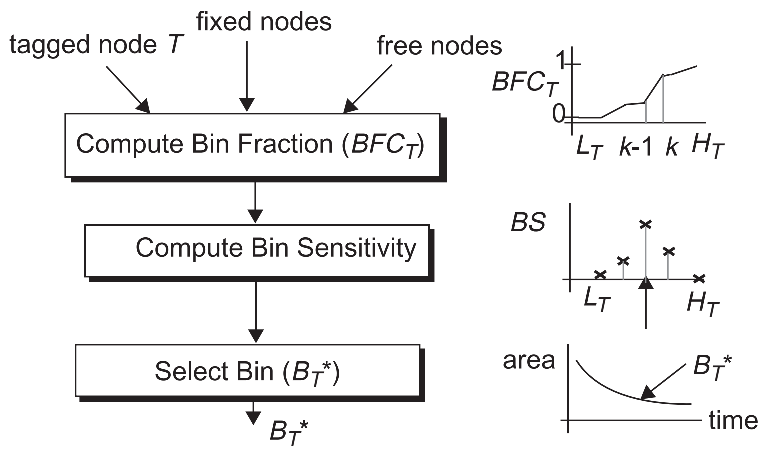


Figure 3.18. The bin selection procedure

Notation	Interpretation
T	Tagged node
fixed nodes	Nodes whose mapping as well as implementation bin has been fixed
free nodes	Nodes whose mapping and implementation bin have not been fixed
$free^h$ nodes	Free nodes mapped to hardware by GCLP at any step in the MIBS algorithm
CH_T (CS_T)	Hardware (software) implementation curve for node T
NH_T (NS_T)	Set of hardware (software) implementation bins for node T
L_T (H_T)	L (H) bin for node T. L (H) is the fastest (slowest) bin.
B_T*	Implementation bin finally selected for node T
BF_T^j	Bin fraction computed when node T is implemented in bin j
BFC_T	Bin fraction curve for node T
BS_{max}	Maximum value of bin sensitivity

Table 3. Summary of notation used in bin selection procedure.

3.6.2 Bin Fraction Curve (BFC)

Assuming node T is implemented in bin j, BF_T^j is computed as the fraction of $free^h$ nodes that have to be moved from their H bins to their L bins in order to meet feasibility. The bin fraction curve BFC_T is the plot of the bin fraction BF_T^j for each bin j of the tagged node T. The procedure to compute the *BFC* is described next. The underlying concept is similar to that used in *GC* calculation (Section 3.3.2). For simplicity, we apply the bin selection procedure only for a tagged node mapped to hardware by GCLP. A single implementation bin is assumed when the tagged node is mapped to software.

```
Procedure:  Compute_BFC
Input:      $N_{fixed}$ = {fixed nodes}, $N_{free}^h$ = {$free^h$ nodes},
            $T$ = tagged node, with mapping $M_T$ (assumed hardware),
            hardware implementation curve $CH_T$
Output:     $BFC_T = \{(BF_T^j, j), \forall j \in NH_T\}$
```

```
Initialize:    $N_{H \to L} = \phi$, $t_{exec}(p)$ known for all fixed nodes $p$, $p \in N_{fixed}$.

for ($j = 1$; $j \le |NH_T|$, $j{+}{+}$) {
S1.  Set $t_{exec}(T) = th_T^j$
S2.  For all $k \in N_{free}^{h}$, set $t_{exec}(k) = th_k^H$ (all $free^h$ nodes at $H$ bins)
S3.  Compute $T_{finish}$, given the mapping and $t_{exec}$ for all nodes
S4.  Find the set $N_{H \to L}$ of $free^h$ nodes that need to be moved to their $L$ bins in
     order to meet deadline
    S4.1.  $N_{H \to L} \leftarrow next(N_{free}^{h})$
    S4.2.  $t_{exec}(f) = th_f^L$, $\forall f \in N_{H \to L}$ (set to $L$ bins)
    S4.3.  Update($T_{finish}$)
    S4.4.  If $T_{finish} > D$ go to S4.1
S5.  $BF_T^j = \dfrac{\sum_{i \in N_{H \to L}} size_i}{\sum_{i \in N_{free}^{h}} size_i}$, $0 \le BF_T^j \le 1$.
    }
```

The sequence S1 to S5 outlines the procedure used to compute the bin fraction BF_T^j for a particular bin j. N_{fixed} is the set of fixed nodes and N_{free}^{h} is the set of free nodes that has been mapped to hardware by GCLP at the current step of the MIBS algorithm. In S1, the execution time of the tagged node is set to the execution time for the jth bin. In S2, the execution times for all the $free^h$ nodes are set corresponding to their respective H bins. The finish time for the DAG (T_{finish}) is computed in S3. In S4, we compute $N_{H \to L}$, the set of $free^h$ nodes that need to be moved to their L bins in order to meet the timing constraints. Various ranking functions can be used to order the $free^h$ nodes. One obvious choice is to rank the nodes in the order of decreasing H bin execution times th_i^H. A second possibility is to use (th_i^H / th_i^L) as the function to rank the nodes. This has the effect of moving nodes with the greatest relative gain in time when moved from *H* to *L* bin.

BF_T^j is computed in S5 as a ratio of the sum of the sizes of the nodes in $N_{H \to L}$ to the sum of the sizes of the nodes in $N_{free}{}^h$. Recall that the *size* of a node is the number of elementary operations (add, multiply, etc.) in the node.

In summary, a high value of the bin fraction for node *T* indicates that selecting the *j*th implementation bin is likely to result in a large fraction of $free^h$ nodes being subsequently assigned to their *L* bins.

3.6.3 Implementation-bin Selection

Figure 3.19-a plots a typical bin fraction curve for a tagged node *T*. Let $L_T(H_T)$ denote the $L(H)$ bin for node *T*. How is the desired bin $B_T{}^*$ to be selected for this node? An intuitive choice is to set $B_T{}^* = H_T$, since this corresponds to the smallest hardware area for node *T*. At H_T, however, $BF_T{}^H$ is high, i.e., a large fraction of the $free^h$ nodes are in *L* bins so that the total hardware area might be unnecessarily large. As the tagged node shifts from bin H_T downwards, the resulting decrease in *BF* implies that the fraction of $free^h$ nodes at their *L* bin decreases, and consequently the allocated hardware area of $free^h$ nodes reduces. The slope of BFC_T represents how fast the $free^h$ node area reduces with the (leftward) bin motion of node *T*. This slope is called bin sensitivity *BS*; it reflects the correlation between bin motion of the tagged node and the overall area reduction of the $free^h$ nodes. That is, $BS_T{}^j = BF_T{}^{(j+1)} - BF_T{}^j$, $L \leq j \leq H - 1$, where $BS_T{}^H = 0$.

Let the maximum bin sensitivity be BS_{max} (Figure 3.19-b). The implementation bin ($B_T{}^*$) for the tagged node *T* is selected to be the bin with bin sensitivity equal to BS_{max}, if $BS_{max} > 0$. If BFC_T is constant (Figure 3.19-c), then $BS_{max} = 0$, and the tagged node is mapped to its *H* bin, since moving it from its slowest to fastest implementations does not affect the $free^h$ nodes.

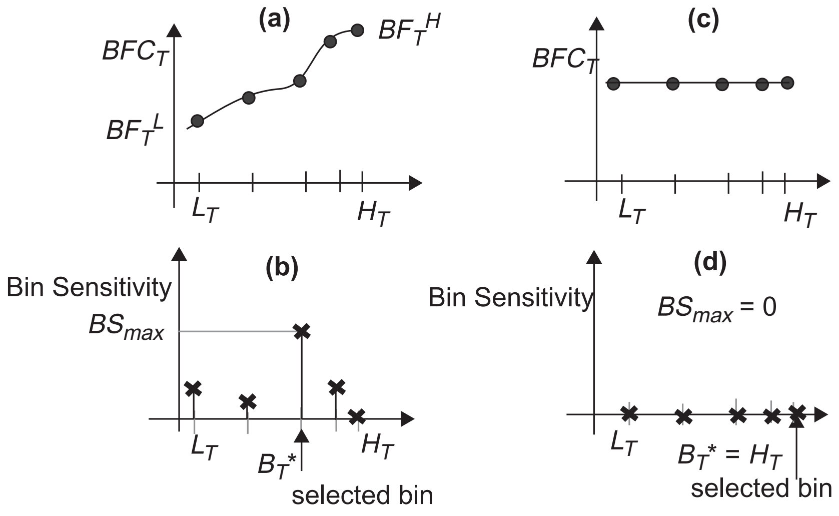


Figure 3.19. Bin fraction and bin sensitivity for implementation-bin selection

Consider the plot of bin sensitivity in Figure 3.20-a, where the regions marked S1 and S2 have identical slopes, i.e., same bin sensitivity. In this case, bin B1 which is closer to the H_T bin is preferred over bin B2 since it corresponds to a smaller area of node *T*. To incorporate this effect in general, the bin sensitivity values are weighted by the area of node *T*. In particular, the weighted bin sensitivity is plotted by multiplying the bin sensitivity at each bin *j* by ah_T^H/ah_T^j (Figure 3.20-c). B_T* is then selected to be the bin with the maximum weighted bin sensitivity. In case of a tie for the maximum weighted bin sensitivity, the bin closer to H_T bin is

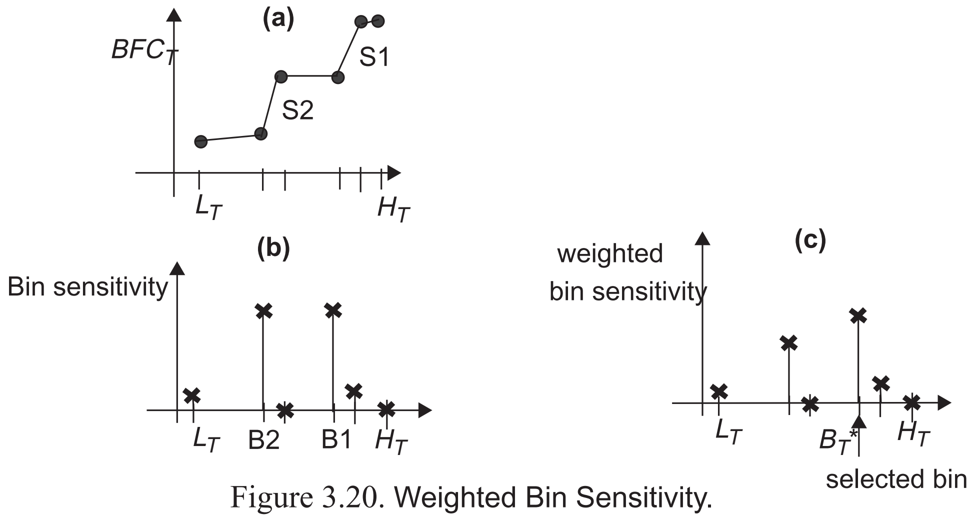


Figure 3.20. Weighted Bin Sensitivity.

selected.

In summary, the strategy for implementation-bin selection is to plot the weighted bin sensitivity and set B_T* to be the bin with the maximum bin sensitivity that is closest to the H_T bin. The bin selection procedure has complexity O($B \cdot$ (|N| + |A|)), as shown in Appendix A5. The procedure is outlined below:

Procedure **bin_selection**

Input N_{fixed} = {fixed nodes}, $N_{free}{}^h$ = {$free^h$ nodes}
T = tagged node, with mapping M_T (assumed hardware),
hardware implementation curve CH_T

Output B_T*

S1. Compute BFC_T (Section 3.6.2)
S2. Compute bin sensitivity
S3. Compute weighted bin sensitivity
S4. Determine bin B_T* corresponding to the bin with the maximum weighted bin sensitivity

---

In the next section, we present the Mapping and Implementation Bin Selection (MIBS) algorithm to solve the extended partitioning problem *P2*.

3.7 The Extended Partitioning Problem: MIBS Algorithm

Algorithm: **MIBS**

Input: $\forall i \in N$: CH_i, CS_i, E_i (extremity measure), and R_i (repeller measure).
Software-hardware interface communication costs: ah_{comm}, as_{comm}, and t_{comm}. Constraints: *AH*, *AS*, and *D*.

Output $\forall i \in N$: mapping M_i ($M_i \in$ {hardware, software}), implementation bin B_i*, and start time t_i.

Initialization N_{fixed} = {fixed nodes} = ϕ, N_{free} = {free nodes} = {N}.
Compute median area and time values for all nodes in software and hardware.

Procedure

```
while {$|N_{free}| > 0$} {
S1.  Determine $M_i$ and $t_i$ for all $i \in N_{free}$
     S1.1.  For all $i \in N_{free}$, set area and time values to their median values
     S1.2.  Use GCLP to compute $M_i$ and $t_i$ for $i \in N_{free}$. (Section 3.3)
S2.  Determine the set of ready nodes $N_R$
S3.  Select tagged node $T$ ($T \in N_R$) using urgency measures
S4.  Determine the implementation bin $B_T^*$ for node $T$ assuming mapping $M_T$
     S4.1.  Use the bin selection procedure to determine bin $B_T^*$ (Section 3.6)
S5.  $N_{free} = N_{free} \backslash \{T\}$; $N_{fixed} \leftarrow \{T\}$ , Update $t_T$ based on the selected implemen-
     tation bin $B_T^*$.
     }
```

N represents the set of nodes in the graph. N_{free} is the set of free nodes; it is initialized to N. N_{fixed} is the set of fixed nodes and is empty at start. The median values of the area and time on hardware and software mappings are computed in the initialization phase. For each step, the MIBS algorithm computes the mapping, the implementation bin, and the schedule of one node. In S1 of each step, the mapping and schedule for all the free nodes is first computed. This is done by applying GCLP over the set of free nodes assuming median area and time values. The set of ready nodes is determined in S2. This represents the set of nodes whose predecessors are fixed nodes. One of these ready nodes is selected as a tagged node in S3. In particular, we select a ready node on the critical path. In S4, the bin selection procedure is applied to determine the implementation bin for this tagged node. Finally, in S5, the schedule of the tagged node is updated depending on the implementation bin selected. The tagged node then becomes fixed. The sequence S1-S5 is repeated $|N|$ times until all the nodes in the graph become fixed.

Note that the mapping of all the nodes is not finalized at one shot in MIBS; future mappings of the remaining free nodes are allowed to change depending on the implementation bin selected for a tagged node. At any step, the known mappings and implementation bins of the fixed nodes affect the mappings of the free nodes. The complexity of the MIBS algorithm is $O(|N|^3 + B \cdot |N|^2)$, where B is the number of implementation bins per mapping (Appendix A6).

3.8 Performance of the MIBS Algorithm

The performance of the MIBS algorithm is examined in this section. As in Section 3.4, we will use both practical examples (the modem and TCS) as well as random graphs to evaluate the performance.

The procedure used to generate the hardware implementation curve for each node in the DAG is described in Section 3.8.1. In Section 3.8.2, the solutions obtained with the MIBS algorithm are compared to the optimal solutions obtained with the ILP formulation. In Section 3.8.3, we demonstrate the effectiveness of the MIBS algorithm in reducing the hardware area relative to the GCLP algorithm.

3.8.1 Estimation of the Hardware Implementation Curve

Silage code is generated for all the nodes in the DAG using the Silage code generation feature of Ptolemy. The Hyper environment is used to generate the hardware implementation curves as follows (see Figure 3.21). For each node, the critical path T_c associated with its control-dataflow graph is first computed. The hardware area required to implement the node at a sample period d equal to the critical path is estimated (Appendix A8.1 discusses the estimation techniques in detail). This sample period corresponds to the L implementation bin for the node. The sample period is then increased in multiples of the sample period until the required hardware reduces to just one resource of each type. This sample period corresponds to the H implementation bin for the node; the hardware area cannot

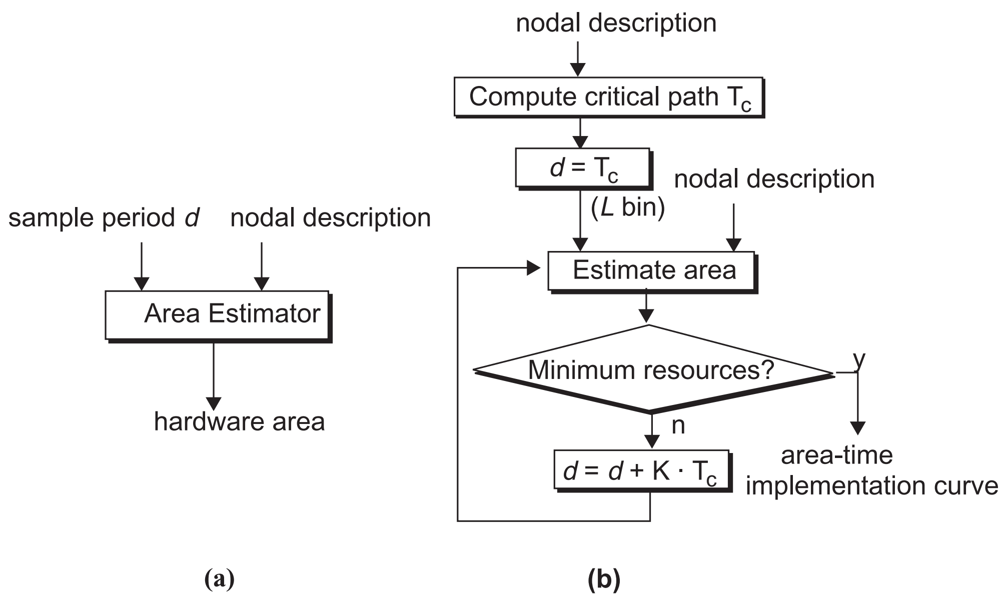


Figure 3.21. (a) Estimation of the hardware area for a node, for a given sample period (b) Computation of the hardware implementation curve for a node.

get any smaller using the particular synthesis mechanism. Sample periods between *L* and *H* bins correspond to the remaining implementation bins for the node.

The generation of the implementation curve for the random graphs is discussed in Appendix A7.3.

3.8.2 Experiment 1: MIBS vs. ILP

ILP formulations of the modem and TCS examples become impossible to solve in a reasonable time. A simplified version of the modem example with 15 nodes and 5 hardware implementation bins per node is considered here. The ILP formulation for this example requires 718 constraints and 396 variables. Table 4 summarizes the solutions obtained with ILP and with MIBS algorithm. The closeness of the solutions is encouraging, especially since ILP becomes formidable for even slightly larger problems.

Figure 3.22 plots the MIBS and ILP hardware areas for a number of random examples. For the examples tested, the MIBS solution is within 18% of the

Scenario	hardware area	solution time
ILP	158	3.5 hours
MIBS	181	3 minutes
Comparison	1.1456 times bigger	70 times faster

Table 4. Comparison of ILP and MIBS solutions.

optimal solution obtained by ILP. Larger examples could not be solved by ILP in reasonable time. In these examples, ILP failed to give even a single feasible integer solution.

3.8.3 Experiment 2: Binary Partitioning vs. Extended Partitioning

Our next objective is to evaluate the effectiveness of the extended partitioning approach in reducing the total hardware area compared with binary partitioning. Three cases are considered. In the first case, mapping is done based on GCLP, assuming that the execution times and areas for the nodes mapped to hardware are set to the values corresponding to their *L* bins. In the second case, this mapping is recomputed, now with the area and execution time values corresponding to the median implementation bins. In the third case, extended partitioning is done based on the MIBS algorithm. Table 5 shows the results for the three cases

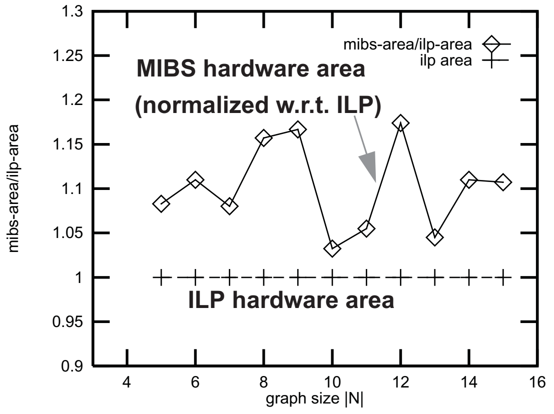


Figure 3.22. Comparison of MIBS and ILP solutions.

case	Scenario	hardware area	area reduction normalized with respect to case 1	solution time
1	GCLP, *L* implementation bin	736	1.0	0.0525s
2	GCLP, median implementation bin	530	0.7201	0.0525s
3	MIBS	362	0.4918	0.7974s

Table 5. Area improvement using MIBS vs. GCLP

applied to the modem example. The MIBS solution is observed to be much superior to both the GCLP solutions (50% less hardware compared to case 1, and 32% less than case 2). This strengthens our premise that implementation flexibility can be used at the partitioning level to reduce the overall hardware area.

In Figure 3.23, we compare, for random graphs, the hardware area obtained with MIBS to that obtained with GCLP (median area and time values). On an average, the area generated by MIBS is 26.4% smaller than that generated by GCLP.

Figure 3.24-a shows the distribution of the nodes among the implementation bins selected by the MIBS algorithm. This distribution is averaged over a number of random examples for a fixed graph size of 25 nodes. The bins are classified into 5 categories: *L* bin, *L* to median bin, median bin, median to *H* bin, and the *H* bin. It is seen that the nodes in hardware are distributed among all the implementation bins. This flexibility reduces time criticality at every mapping decision

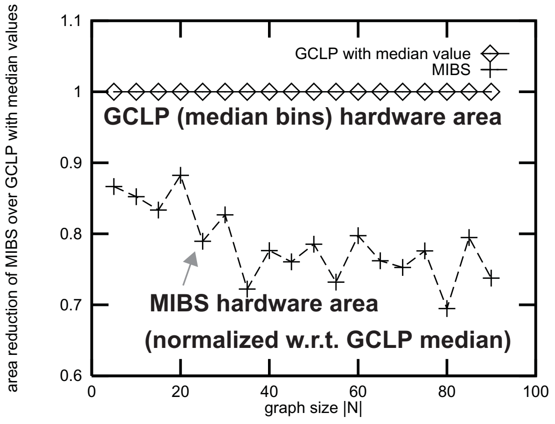


Figure 3.23. MIBS vs. GCLP (Extended Partitioning vs. Binary Partitioning)

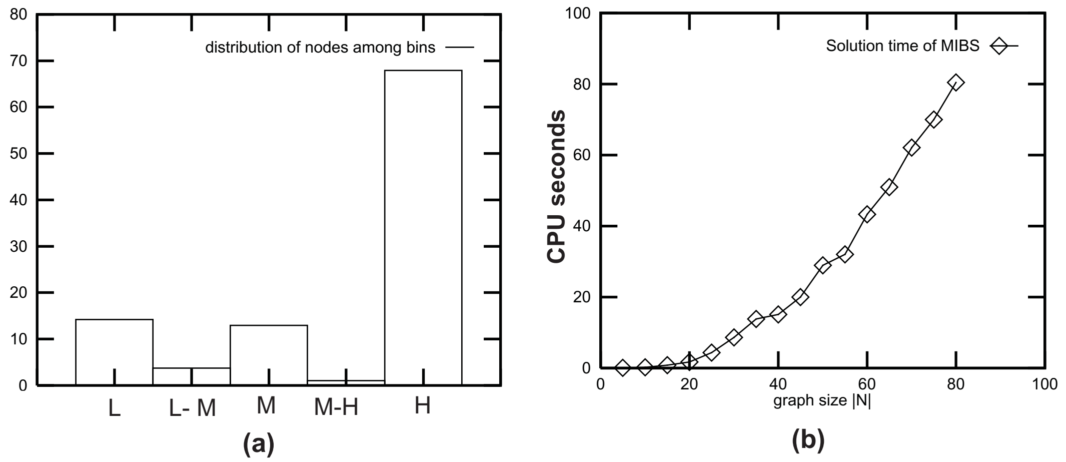


Figure 3.24. (a) Node distribution among implementation bins. (b) Solution time of MIBS algorithm averaged over random graphs per graph size.

and improves DSP utilization, i.e., the number of nodes mapped to software increases. This combined effect (reduced number of hardware nodes and their distribution over several bins) reduces the total hardware area. The MIBS solution time is plotted as a function of the graph size in Figure 3.24-b. The algorithm has complexity of O($|N|^3 + B.\ N|^2$).

3.8.4 Parameter Tuning

Several user-settable parameters come into play in the MIBS algorithm. These include: (1) the cut-off percentiles (α, β) used for classifying extremities in GCLP, (2) the extremity measure weight (γ) and the repeller measure weight (ν) in GCLP, (3) the ranking function for *GC* calculation (*ts*, *ts/th*, or *ah*), and (4) the ranking function for *BF* calculation (th^H, th^H/th^L, or ah^L).

Parameters α, β, γ, and ν are tuned by a simple binary search between 0 and 1. We have incorporated this automated search mechanism in our algorithm implementation. Since the MIBS algorithm is extremely fast, such an exploration is computationally viable. The *ts*/*th* and th^H/th^L ranking functions have been found to perform best for *GC* and *BF* calculations respectively.

3.9 Summary

At the system-level, designs are typically represented modularly, with moderate to large granularity. Each node can be implemented using a variety of algorithms and/or synthesis mechanisms in hardware or software. These implementations typically differ in area and execution time. We define *extended partitioning* as the joint problem of mapping nodes in a precedence graph to hardware or software, scheduling, and selecting a particular implementation (called implementation bin) for each node. The end-objective is to minimize the total hardware area subject to throughput and resource constraints. The extended partitioning problem is NP-hard; we proposed an efficient heuristic called MIBS to approximately solve it. The MIBS algorithm has a complexity of $O(|N|^3 + B \cdot |N|^2)$, where $|N|$ is the number of nodes, and B is the number of implementation options per node, per mapping.

In this chapter, we first presented the GCLP algorithm to solve the *binary partitioning* (mapping and scheduling) problem. It uses a global time criticality measure to adaptively select a mapping objective at each step — if time is critical, it selects a mapping that minimizes the finish time of the node, otherwise it minimizes the resource consumption. This time criticality measure overcomes the inherent drawback with serial traversal. In addition to global consideration, local optimality is sought by taking into account the preferences of nodes that consume disproportionate amounts of resources in hardware and software mappings. This effect is quantified by classifying nodes as extremities. The hardware area is further reduced by using a concept of repellers to effect on-line swaps between nodes. Repellers take into account the relative preferences of nodes, based on intrinsic algorithmic properties that dictate a preferred hardware or software mapping. The

GCLP algorithm is computationally efficient (O($|N|^2$)). For the examples tested, the GCLP solution was found to be no more than 30% larger than the optimal solution. The effectiveness of local phase nodes (extremities and repellers) in reducing the overall hardware area was experimentally verified. On an average, the use of local phase nodes reduces the hardware area by 17%, relative to solutions obtained without using local phase classification of nodes.

The philosophy of the MIBS algorithm is to extend the GCLP heuristic for extended partitioning without the associated complexity buildup. The strategy is to classify nodes in the graph as free, tagged, and fixed. Initially all nodes in the graph are free — their mappings and implementation bins are unknown. GCLP is applied over the set of free nodes. A tagged node is then selected from this set; its mapping is assumed to be that determined by GCLP. A bin selection procedure is used to compute an appropriate implementation bin for the tagged node. The procedure uses a lookahead measure, called bin fraction, which estimates for each bin of the node, the fraction of unmapped nodes that need to move to their fastest implementations so that timing constraints are met. The bin fraction is used to compute a bin sensitivity measure that correlates the implementation bin with the overall hardware area reduction. The procedure selects the bin with maximum bin sensitivity. The procedure simplifies this computation by assuming that the remaining free nodes are either in their slowest or fastest implementations. The tagged node becomes a fixed node once its implementation bin is determined. GCLP is then applied over the remaining free nodes and the sequence is repeated until all nodes in the graph become fixed. In the examples tested, the MIBS solution is found to be within 18% of the optimal solution. Experimental results also indicate that implementation bins can be used effectively to reduce the overall area by as much as 27% over solutions generated using binary partitioning.

4

COSYNTHESIS AND COSIMULATION

In Chapter 3, algorithms to partition an application into hardware and software were described. Partitioning makes three attributes available for each node in the application: a hardware or software mapping, the type of implementation for this mapping, and a schedule.

The next step in the design process is to *synthesize* the mixed hardware-software system. The cosynthesis problem is to synthesize the hardware, software, and interface components for the final implementation. In Section 4.1 we discuss this problem. Based on the assumed target architecture, we develop an architectural model in Section 4.1.1. Our approach to synthesis is discussed in Section 4.1.2. We describe the particular mechanisms used to synthesize the hardware, the software, and the interface in Sections 4.1.3 to 4.1.5. Some of the other approaches to synthesis are discussed in Section 4.1.6.

Hardware-software *cosimulation* is the process of simulating the hardware and software components of a mixed hardware-software system within a *unified* environment. This includes simulation of the hardware modules, the processor, and the software that the processor executes. The cosimulation problem is discussed in Section 4.2. The requirements of a cosimulation environment are outlined. Ptolemy has an infrastructure that supports most of these requirements. The relevant details of the Ptolemy environment are discussed in Section 4.2.1. Our approach to hardware-software cosimulation within the Ptolemy environment is presented with the help of an example in Section 4.2.2.

4.1 Cosynthesis

After partitioning, each node of the DAG is annotated with a mapping, implementation bin, and schedule. We define cosynthesis as the problem of synthesizing the final implementation (hardware, software, and interface) from this annotated DAG.

4.1.1 Architectural Model

A particular target architecture for the mixed hardware-software system has been assumed in Chapter 2. As shown in Figure 4.1, the architecture consists of a single programmable processor and multiple hardware modules connected to a single system bus. Each node mapped to hardware is synthesized as a hardware module[1]. A hardware module consists of a datapath and controller, and input and

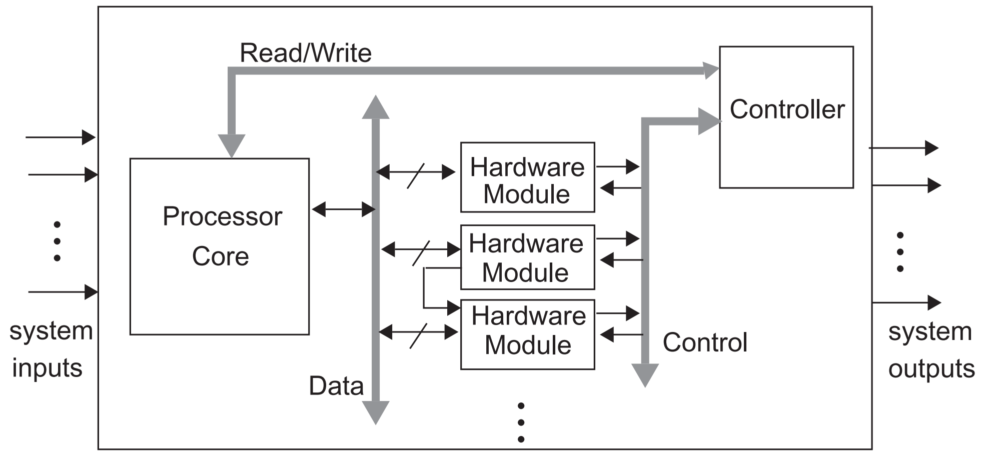


Figure 4.1. The target architecture.

output interfaces. The input and output interfaces are responsible for the communication between the hardware modules and the processor. The software component of the architecture is the program that runs on the programmable processor. This includes the communication code, i.e., the device drivers that manage communication of data between hardware and software. The architecture is assumed to be non-pipelined, i.e., one set of input data is processed completely before the second set arrives into the system. We assume that nodes in hardware and software communicate via a memory-mapped, asynchronous, blocking communication mechanism[2]. By *memory-mapped* communication between the software and hardware, we mean that data is communicated across the hardware-software boundary by writing to and reading from a shared address space. The various components in the architecture are configured as a *globally asynchronous locally synchronous* model. This means that each individual hardware module operates internally as a synchronous circuit, but the modules themselves communicate with each other and the processor asynchronously. In other words, within a hardware module, a module controller activates the various components of the datapath (ALUs, registers, multiplexers, etc.) at predetermined clock cycles[3]. Communication between a hardware module and the processor or between hardware modules is *asynchronous*, though. The reason is obvious. Since the schedule generated by partitioning is based on execution-time estimates, it is not guaranteed to be cycle-accurate. That is, we cannot exactly determine *when* a processor or hardware module should

1. We assume no hardware reuse *between* modules. This issue is further discussed in Section 4.1.3.
2. A system may contain several types of interfaces, such as serial ports or custom-designed peripherals. All nodes need not communicate via the same mechanism. An appropriate communication mechanism may be selected for the different data transfers, depending on various factors such as the size of the data (number of words and the bitwidth) and whether the data is communicated on-chip or off-chip. In this work we simplify matters by restricting all nodes to use the same type of communication mechanism.
3. The datapath and controller will be discussed in detail in Section 4.1.3.

clock. The generated schedule is valuable, however, in that it can reliably indicate the *order* in which nodes execute. A global controller activates the hardware modules in this order.

Architecture of a hardware module

The architecture of a hardware module is shown in Figure 4.2. We refer to the datapath and controller collectively as the *kernel* of a hardware module. Each kernel has two handshaking signals, *ready* and *completion,* in addition to the input and output data signals. Each input and output of the kernel is connected to a data latch. A latch has an input enable (IE) and an output enable (OE) control. The kernel begins its computation after it receives a *ready* signal, which indicates that valid data is available on all of its inputs. On finishing computation, the kernel raises a *completion* flag. The *ready* signal is a function of the input enable signals and the *completion* signal and is generated by combinational logic.

The details of the hardware-software, software-software, and hardware-hardware interface are described next.

Hardware-software interface

In a memory-mapped scheme, each input and output of a hardware module corresponds to a unique address in the shared address space of the processor. A traditional implementation of memory-mapped communication requires an explicit

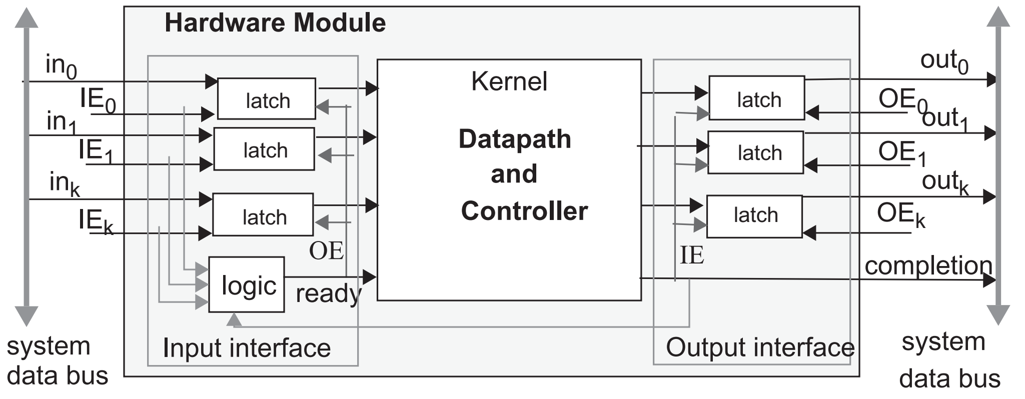


Figure 4.2. Architecture of a hardware module.

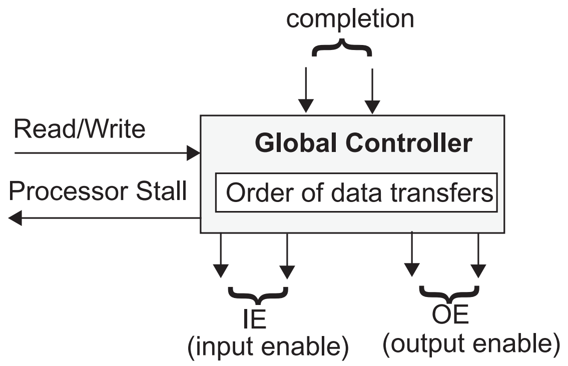


Figure 4.3. The global controller for the hardware-software interface.

address decoder that enables the appropriate input (output) of a hardware module when a write (read) to that address arrives. As the number of modules increases, the decoder tends to get quite large. To reduce the time overhead associated with semaphore checking and area overhead associated with explicit address decoders, we apply the *ordered transactions principle*, proposed by Lee *et al.* [Lee90] [Sriram93][4]. The hardware-software communication scheme works as follows. The schedule (determined by partitioning) is analyzed to determine the order of data transfers across the hardware-software interface. Each data transfer is assigned a unique memory address. Thus, the sequence of hardware module-addresses to which the processor sends read and write signals, is known apriori from the schedule. Each of these addresses corresponds uniquely to an input or output latch of a particular hardware module. A global controller, shown in Figure 4.3, uses the order information to activate the input and output latches of all the hardware modules. When the processor issues a *write* request, the global controller knows the corresponding input latch based on the order of transfers; it does not need to explicitly decode the address. Accordingly, it enables the corresponding input latch and the data from the processor is latched into the hardware mod-

4. They propose this principle in the context of shared-memory multiprocessor machines running SDF applications.

ule. Note that a hardware module's input data is always read before the next input arrives, since the architecture is non-pipelined. When all the inputs to a kernel are available, the combinational logic associated with the hardware module generates a *ready* signal that notifies the kernel to begin execution. When the processor issues a *read* request, the controller checks whether the corresponding hardware module has set its *completion* signal. If not, the controller stalls the processor until *completion* gets set. When *completion* is set, the controller enables the corresponding output latch and the data from the hardware module is made available to the processor. If the execution-time estimates are reasonably accurate, the ordered transactions approach does not introduce a significant overhead (the *ready* or *completion* signals are already set when the processor request arrives).

Software-software interface

Data transfers between two software nodes are assigned unique memory addresses in the internal data memory of the processor. Communication between two software nodes is achieved by writing the results to the corresponding locations in the internal data memory. Since the software executes sequentially according to the schedule, there is no need for semaphore checking.

Hardware-hardware interface

Communication between two hardware modules is achieved by directly connecting the output latch of the sending hardware module to the corresponding input latch of the receiving hardware module. After the sending hardware module finishes sending data, its *completion* signal enables the input latch of the receiving module. The *completion* signal of the sending module is also used in the combinational logic that generates the *ready* signal for the receiving module.

4.1.2 The Cosynthesis Approach

The cosynthesis problem involves synthesizing the following components:

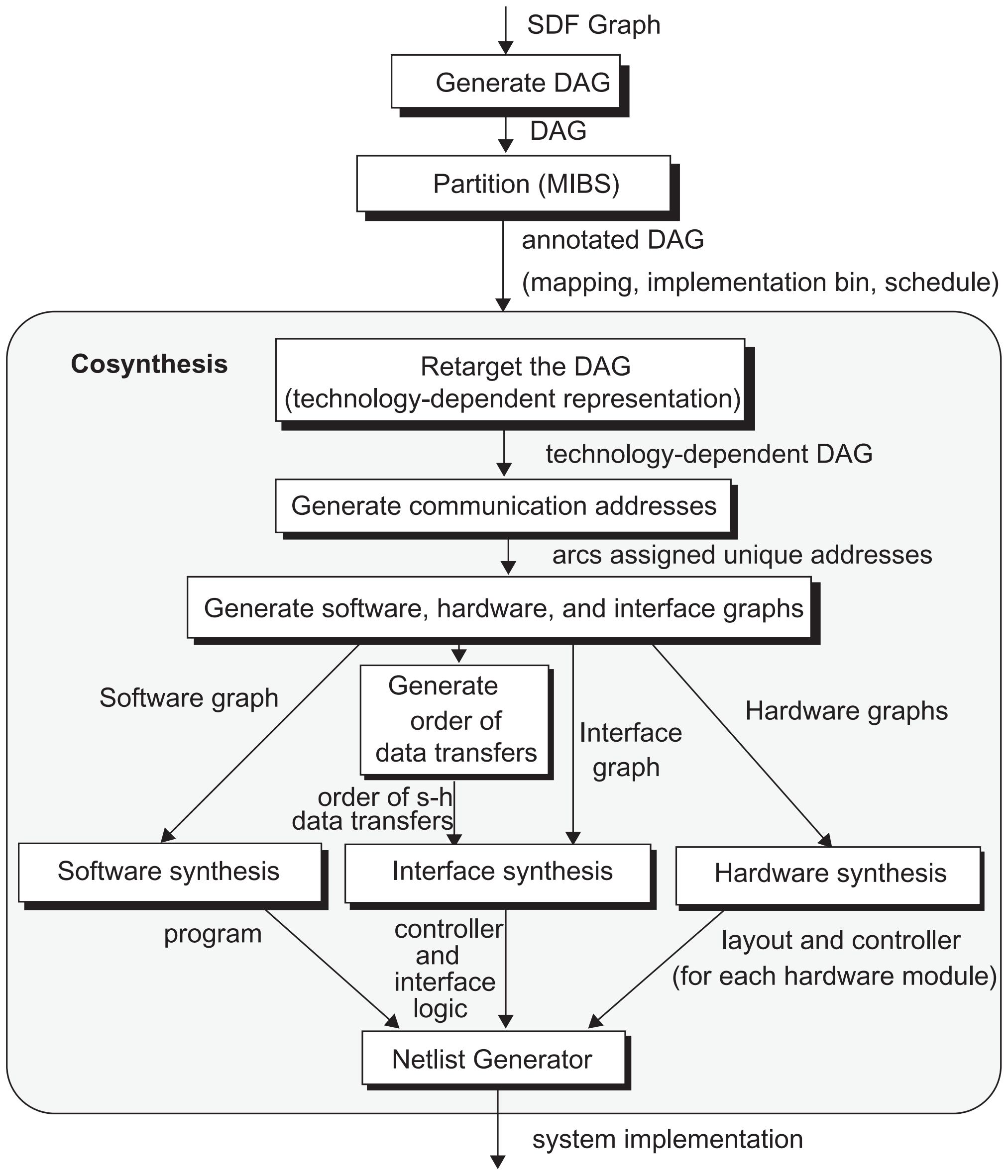
1. the datapath and controller for each hardware module.
2. the program running on the programmable processor. This includes the code to read from and write to the shared memory locations for communication with the hardware modules.
3. the global controller, the input and output interfaces of the hardware modules, and the netlist connecting the controller to the various hardware modules and the processor.

Our approach to synthesis is shown in Figure 4.4. The application is specified as an SDF graph. As discussed in Chapter 2, the SDF graph is first translated to a directed acyclic graph representing data precedences. The DAG is an implementation-independent specification of the application, i.e., nodes are at a task level of granularity and have no commitment to a particular hardware or software implementation. The partitioning tool (discussed in Chapter 3) annotates each node of the DAG with a mapping, implementation bin, and schedule.

This annotated graph is then passed to a technology-dependent **retargeting tool**. This tool replaces each node in the DAG (and all the nodes within its hierarchy) with a technology-dependent representation corresponding to the mapping and implementation bin selected by the partitioning tool. By technology-dependent representation, we mean an assembly or C code representation for nodes mapped to software, and a VHDL or Silage representation for nodes mapped to hardware[5].

5. Note that the technology-dependent hardware description is still not the final implementation; the hardware implementation (at the architecture or layout-level) for a node is generated by highlevel synthesis tools that operate on this description. We assume such a two-tiered approach because it is unreasonable to expect full implementations of task-level elements to be available in a library, while library elements for Silage and VHDL descriptions are readily available within Ptolemy.

Furthermore, if different implementations are available for a node (for instance, cascade and transpose form IIR filters, or inlined and subroutine-based software representations), the retargeting tool selects the one corresponding to the implementation bin selected by the partitioning tool. In the case of hardware-mapped nodes, transformation and resource-level implementation options are also possible, as discussed in Chapter 1. Since these come into play only at a lower level, the corresponding technology-dependent description of a hardware-mapped node is anno-



SDF Graph
Generate DAG
DAG
Partition (MIBS)
annotated DAG
(mapping, implementation bin, schedule)
Cosynthesis
Retarget the DAG
(technology-dependent representation)
technology-dependent DAG
Generate communication addresses
arcs assigned unique addresses
Generate software, hardware, and interface graphs
Software graph
Generate order of data transfers
Interface graph
Hardware graphs
order of s-h data transfers
Software synthesis
Interface synthesis
Hardware synthesis
program
controller and interface logic
layout and controller (for each hardware module)
Netlist Generator
system implementation

Figure 4.4. Cosynthesis approach

tated with the selected transformation and sample period. This information is then used by the hardware synthesis tools.

This retargeting approach assumes the existence of a library for each technology. For instance, suppose a node is an IIR filter. We assume descriptions of the IIR filter in C, VHDL, Motorola 56000 assembly code, etc. to be available. For instance, within the Ptolemy environment, extensive libraries for the various technologies, such as C, assembly code for various DSPs, VHDL, and Silage are available. Also, multiple representations (corresponding to different values of the implementation metrics) could be available for the same node within a given technology.

An alternative to the library-based retargeting approach is to *compile* the technology-independent representation of every node into its desired representation. In this approach, the node is described in a highlevel language, such as C. The C description for the node is then translated to an intermediate representation, which is typically a variant of a control-dataflow graph [McFarland90a]. The intermediate representation is *compiled* to either VHDL, or assembly code, depending on the desired synthesis technology. This compilation approach can be simplistically thought of as retargeting at the instruction level, accompanied by some traditional compiler-like optimizations. Such a compilation approach is used in the DSPStation [MentorGraphics1] for synthesizing hardware or software from a highlevel description based on Silage.

We have chosen the library-based retargeting approach. There are two advantages to this approach. First, it is computationally efficient and retains modularity. If a node (or its mapping or implementation) changes, the corresponding representation for the node is re-synthesized by simply selecting the new library element. In the compilation approach, the intermediate graph would have to be regenerated and compiled. Secondly, each module can be individually optimized (when added to the library), thereby improving the quality of the implementation. The disadvantage of this approach is that it depends on the richness of the library. If a corresponding element does not exist in the library, the user is forced to develop it.

After the retargeting step, each node in the DAG has been replaced by an equivalent technology-dependent representation. The **address generator** assigns a unique address to every data transfer involving a software and a hardware node, i.e., every arc connecting a software node to a hardware node is assigned an address in the processor's external address space.

The next step in the synthesis process is to generate the **hardware, software,** and **interface graphs**. Figure 4.5 shows typical hardware, software, and interface graphs for a simple example. These graphs are fed to the hardware, software, and interface synthesis tools respectively. A separate **hardware graph** is generated for each node mapped to hardware. Since a node could be represented hierarchically in the original description, the hardware graph can, in general, contain several subnodes. The hardware graph is annotated with the transformation and sample period corresponding to the implementation bin selected for this node. The hardware synthesis tool generates a datapath and controller for each hardware graph as required by its implementation bin. For software nodes, the synthesis technique is slightly different. All nodes mapped to software are combined in a single **software graph**. *Send* and *receive* nodes are added wherever a hardware-software communication occurs. Recall that the partitioning algorithm generates a global schedule that determines the order in which all the nodes in the DAG begin execution. The ordering between the nodes of the software graph is derived from this global schedule. The software synthesis tool generates a single program from the software graph. The program contains code for all the nodes in the software graph, where the code is concatenated in the predetermined ordering.

The **order generator** determines the order of transfers between nodes mapped to software and hardware. The interface synthesis tool generates the glo-

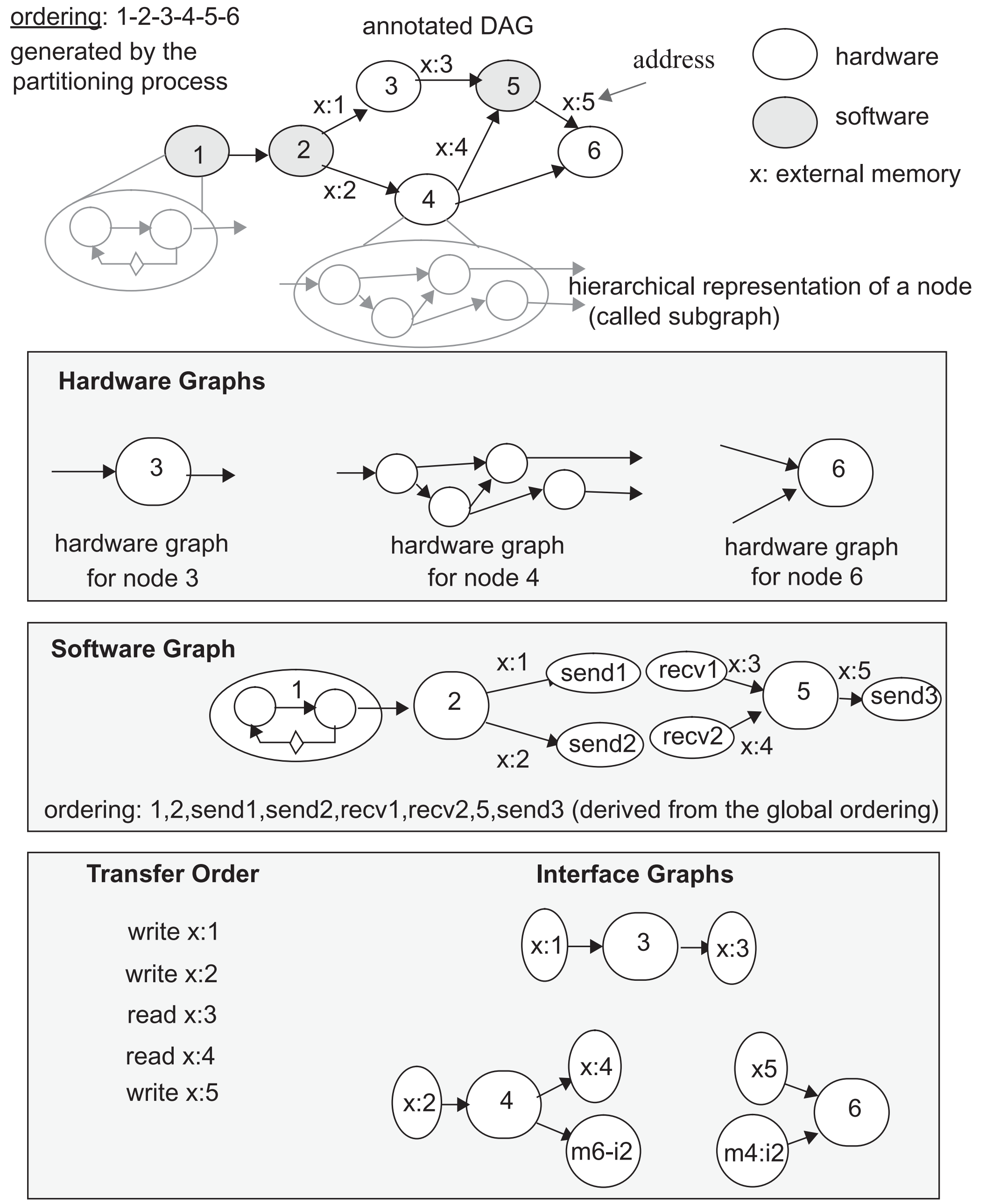

ordering: 1-2-3-4-5-6
generated by the partitioning process
annotated DAG
address
hardware
software
x: external memory
x:1
x:2
x:3
x:4
x:5
hierarchical representation of a node (called subgraph)
Hardware Graphs
hardware graph for node 3
hardware graph for node 4
hardware graph for node 6
Software Graph
send1
send2
recv1
recv2
send3
ordering: 1,2,send1,send2,recv1,recv2,5,send3 (derived from the global ordering)
Transfer Order
write x:1
write x:2
read x:3
read x:4
write x:5
Interface Graphs
x5
m6-i2
m4:i2

Figure 4.5. Hardware, software, and interface graphs. Each node in the hardware (software) graph contains a Silage or VHDL (C or assembly) description.

bal controller using this order of transfers. It also generates the interface glue logic (latches, combinational logic for generating the *ready* signal, etc.) for the hardware modules.

A netlist that describes the connectivity between the hardware modules, the processor, and the global controller is next generated. Standard placement and routing tools can then be used to synthesize the layout for the complete system.

In the following sections we discuss the hardware and software synthesis tools in more detail.

4.1.3 Hardware Synthesis

Given a hardware graph (for a node), we wish to synthesize its datapath and controller. The general approach for hardware synthesis is to:

1. Generate a synthesizeable description of the hardware graph. A synthesizeable description is typically in a language such as Silage or VHDL. Such a description is generated by combining the technology-dependent description of all the subnodes in the hardware graph.

2. Feed this description to highlevel hardware synthesis tools to obtain the implementation for the node. Several highlevel synthesis tools that operate on such synthesizeable descriptions already exist — our approach is to use them directly.

Figure 4.6 summarizes the hardware synthesis mechanism. We restrict ourselves to Silage[6] descriptions for the hardware graph, and assume that the Hyper

6. Silage [Hilfinger85] is an applicative language that is well-suited to expressing DSP applications. Its key features include: dataflow semantics, multirate operators, manifest iterations, signal precision specification, and single-assignment. Several research and commercial synthesis systems use Silage as the specification language (for instance, Hyper from UC Berkeley [Rabaey91], Cathedral from IMEC [Laneer91], and DSPStation from Mentor Graphics [MentorGraphics1]).

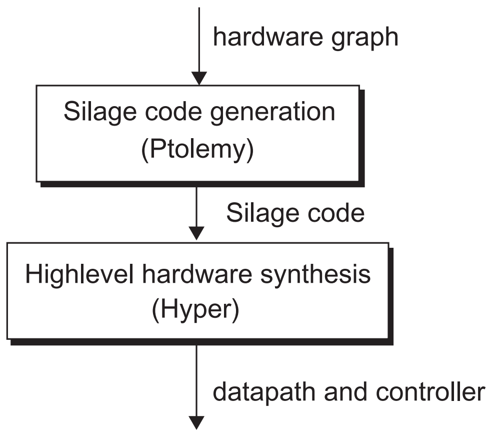

hardware graph
Silage code generation (Ptolemy)
Silage code
Highlevel hardware synthesis (Hyper)
datapath and controller

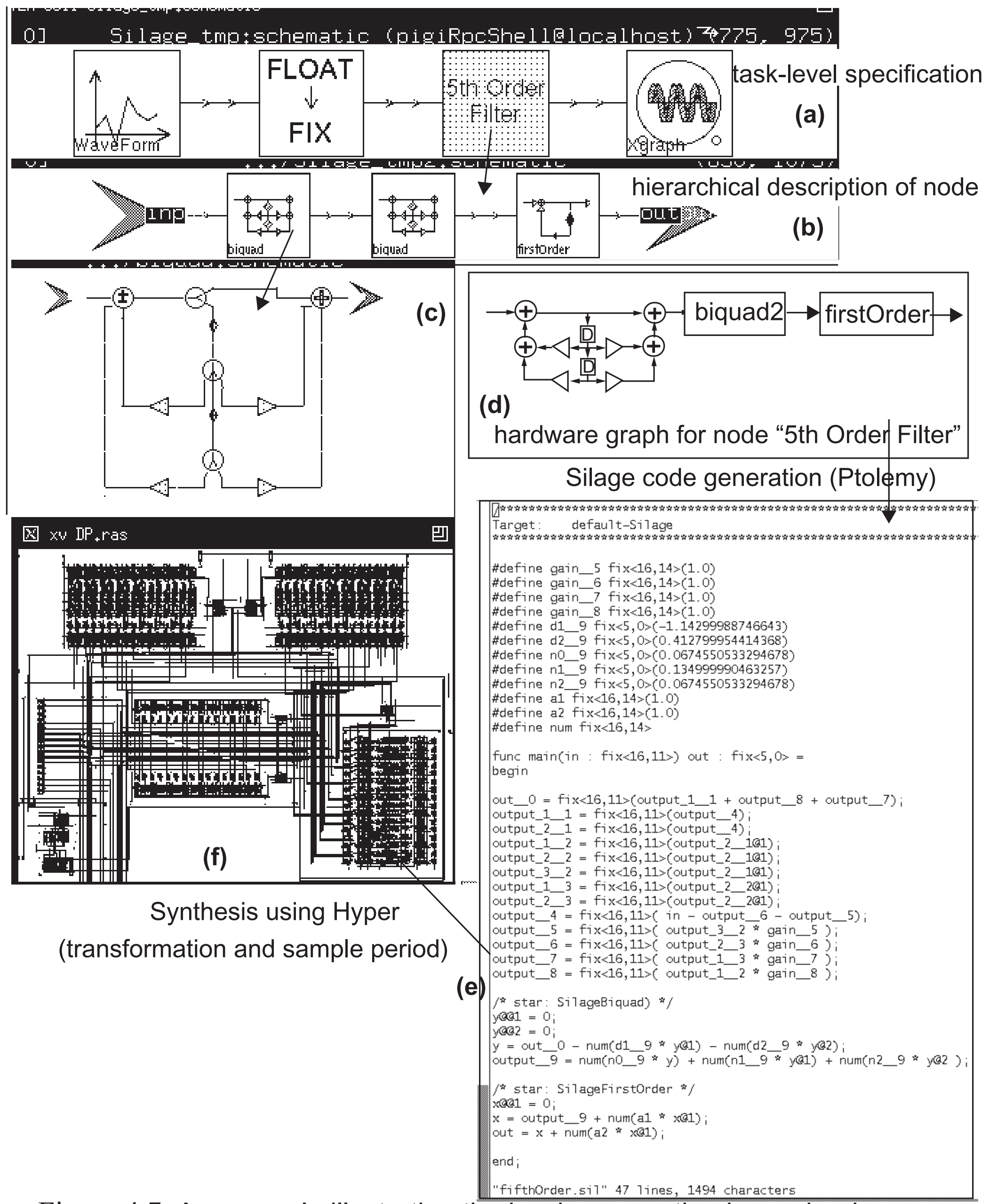
Figure 4.6. Mechanism for hardware synthesis for a single node

system [Rabaey91] is used to synthesize the datapath and controller from this Silage code.

Figure 4.7 illustrates the hardware synthesis mechanism with an example. Figure 4.7-a shows a simple task-level description (specified in Ptolemy), where one of the nodes is a 5th order IIR filter. Suppose that this node is mapped to hardware. Let us consider the synthesis of its implementation. Figure 4.7-b shows the hierarchical description of this node (a cascade of two biquads and a first order section). The first biquad is specified hierarchically in terms of elementary operations as shown in Figure 4.7-c, and the second biquad and the first order section are assumed to be monolithic. Figure 4.7-d shows the hardware graph for the node. Each subnode in the hardware graph contains Silage code. A single Silage program is generated by combining the Silage descriptions of the individual subnodes in the hardware graph. We have developed a Silage code generation mechanism, within Ptolemy, that generates Silage code starting from such a hardware graph (see [Kalavade93] and [Ptolemy95, Chapter 17] for details on Silage code generation[7]).

7. The Silage code generation process includes: (1) generating a schedule for the subnodes in the hardware graph (2) stitching together the code from the individual subnodes in the order specified by the schedule (this includes buffer management between the modules).

The generated Silage description for the IIR node in our example is shown in Figure 4.7-e. This description, along with the transformation and sample period values of the node (corresponding to its implementation bin) are used by Hyper to generate the implementation. Figure 4.7-f shows the final layout of the datapath and controller for the IIR filter.



0] Silage_tmp:schematic (pigiRpcShell@localhost) (775, 975)
WaveForm
FLOAT
FIX
5th Order Filter
Xgraph
task-level specification
(a)
inp
biquad
biquad
firstOrder
out
hierarchical description of node
(b)
(c)
biquad2
firstOrder
D
D
(d)
hardware graph for node “5th Order Filter”
Silage code generation (Ptolemy)
xv DP.ras
(f)
Synthesis using Hyper
(transformation and sample period)
(e)

```
/***********************************************************************
Target:    default-Silage
***********************************************************************

#define gain__5 fix<16,14>(1.0)
#define gain__6 fix<16,14>(1.0)
#define gain__7 fix<16,14>(1.0)
#define gain__8 fix<16,14>(1.0)
#define d1__9 fix<5,0>(-1.14299988746643)
#define d2__9 fix<5,0>(0.412799954414368)
#define n0__9 fix<5,0>(0.0674550533294678)
#define n1__9 fix<5,0>(0.134999990463257)
#define n2__9 fix<5,0>(0.0674550533294678)
#define a1 fix<16,14>(1.0)
#define a2 fix<16,14>(1.0)
#define num fix<16,14>

func main(in : fix<16,11>) out : fix<5,0> =
begin

out__0 = fix<16,11>(output_1__1 + output__8 + output__7);
output_1__1 = fix<16,11>(output__4);
output_2__1 = fix<16,11>(output__4);
output_1__2 = fix<16,11>(output_2__1@1);
output_2__2 = fix<16,11>(output_2__1@1);
output_3__2 = fix<16,11>(output_2__1@1);
output_1__3 = fix<16,11>(output_2__2@1);
output_2__3 = fix<16,11>(output_2__2@1);
output__4 = fix<16,11>( in - output__6 - output__5);
output__5 = fix<16,11>( output_3__2 * gain__5 );
output__6 = fix<16,11>( output_2__3 * gain__6 );
output__7 = fix<16,11>( output_1__3 * gain__7 );
output__8 = fix<16,11>( output_1__2 * gain__8 );

/* star: SilageBiquad) */
y@@1 = 0;
y@@2 = 0;
y = out__0 - num(d1__9 * y@1) - num(d2__9 * y@2);
output__9 = num(n0__9 * y) + num(n1__9 * y@1) + num(n2__9 * y@2 );

/* star: SilageFirstOrder */
x@@1 = 0;
x = output__9 + num(a1 * x@1);
out = x + num(a2 * x@1);

end;

"fifthOrder.sil" 47 lines, 1494 characters
```

Figure 4.7. An example illustrating the hardware synthesis mechanism.

Hyper-generated Architecture

Figure 4.8 shows the generic architecture of the datapath and controller generated for a single hardware graph. The datapath comprises one or more execution units (ALUs, multipliers, adders, etc.). Inputs to the execution units are latched in registers. If an execution unit receives inputs from several sources, a multiplexor is added. Outputs of the execution unit are latched in tristate buffers. Data is communicated internally via a crossbar network. The datapath is controlled by a module controller. Associated with each execution unit is a local controller; the module controller controls the local controllers.

Note that the module controller generated by Hyper does not normally include the *ready* and *completion* signals. It can be modified slightly to incorporate

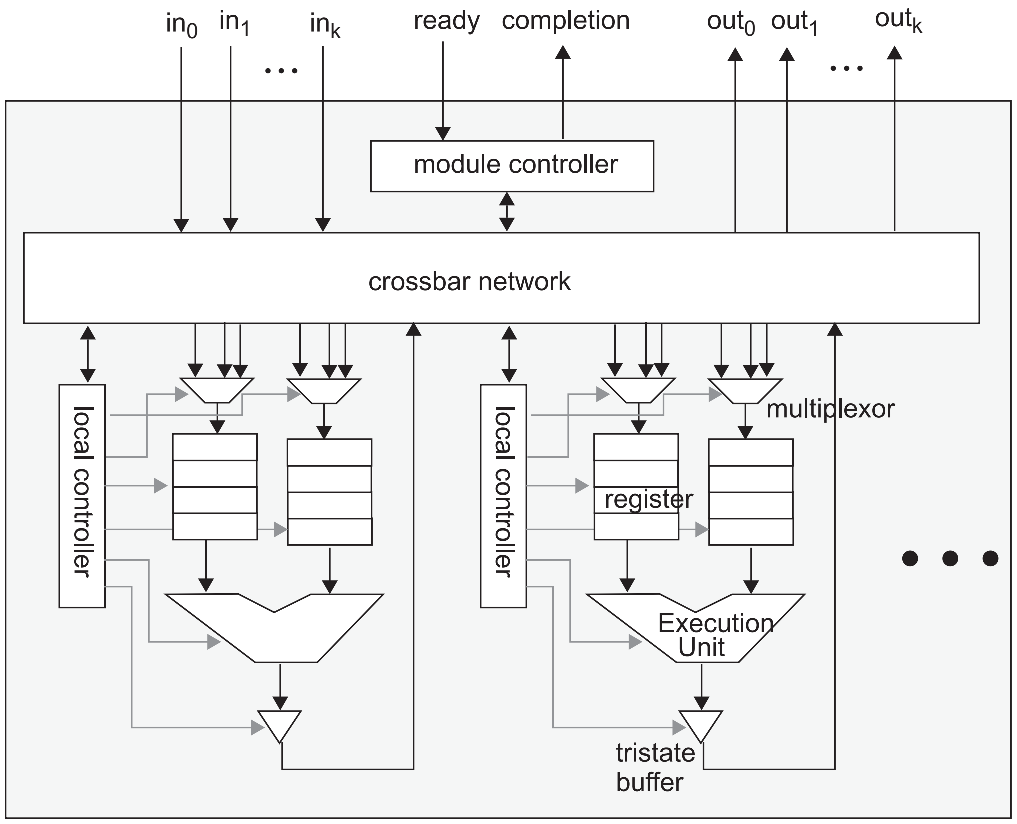


Figure 4.8. Architecture of the datapath and controller in a hardware module. This is the underlying architecture generated by Hyper, augmented with the *ready* and *completion* signals.

these signals. The module controller activates the execution units and the local controllers only after receiving the *ready* signal. It sets the *completion* signal after processing one set of inputs.

Hardware Reuse

A limitation of our current partitioning and synthesis mechanism is that it does not allow reuse of hardware *between* nodes. This is limiting especially in the case of multirate applications, where the DAG may contain several instances of a particular node. In such a case, the partitioning algorithm can be modified to take reuse into account[8]. The synthesis process would then have to take this reuse into account while generating the interface.

4.1.4 Software Synthesis

The software synthesis problem is to generate a software implementation (the program running on the programmable processor) corresponding to a software graph. Although the software synthesis technique described here applies to any target processor, our discussion assumes that the programmable processor used is a Motorola DSP 56000.

Recall that a software graph contains all the nodes mapped to software, augmented by *send* and *receive* nodes for communication. Each node in this graph contains a technology-dependent representation, i.e., a *codeblock* representing the functionality of the node. The software synthesis process essentially involves *stitching* together these codeblocks to generate a single program for the entire software graph. To preserve the functionality of the system, the codeblocks need to be put together according to the sequence generated by the partitioning tool. Note,

8. A simplistic approach is to consider the extremity and repeller measures of a node with multiple instances in the DAG and explicitly assign all the instances of the node to its preferred mapping.

however, that the schedule generated by the partitioning tool is at the level of nodes in the DAG; it does not contain any information on the sequence in which the subnodes *within* a hierarchical node should be executed. As a result, we first need to determine this sequence before we can stitch together the codeblocks. Such a sequence is generated by standard list scheduling techniques available within Ptolemy. The ordering of the nodes in the software graph is updated using this sequence. The final ordering is referred to as the *flattened ordering*. The individual pieces of code are then stitched together in the order specified by the flattened ordering. Note that this includes the code for the *send* and *receive* nodes too. The code for a *send* node contains a “write” instruction; it writes the results generated by the node connected to its input to the memory location assigned to its input arc. Similarly, a *receive* node contains a “read” instruction to read from the memory location assigned to its output arc. The end product is a program corresponding to the software graph. Our approach for software synthesis is summarized in Figure 4.9.

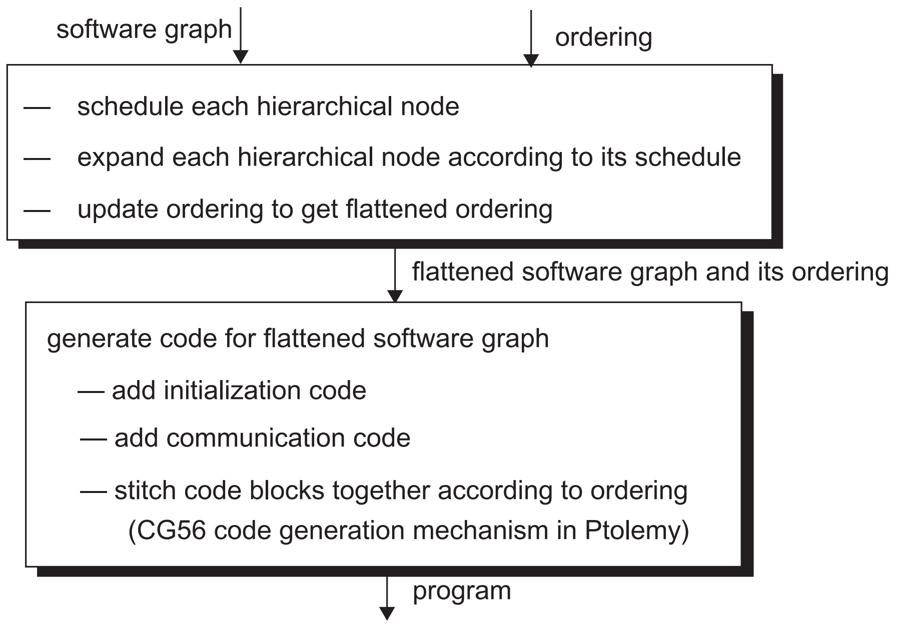

software graph
ordering
— schedule each hierarchical node
— expand each hierarchical node according to its schedule
— update ordering to get flattened ordering
flattened software graph and its ordering
generate code for flattened software graph
— add initialization code
— add communication code
— stitch code blocks together according to ordering (CG56 code generation mechanism in Ptolemy)
program

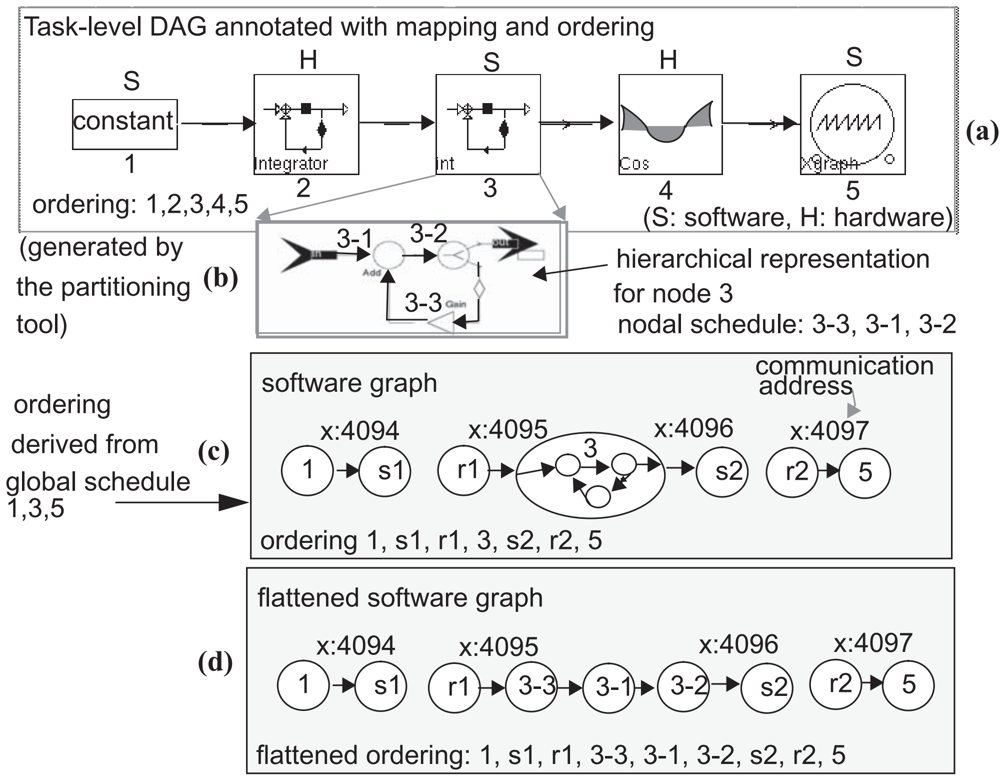
Figure 4.9. Mechanism for software synthesis from the software graph.

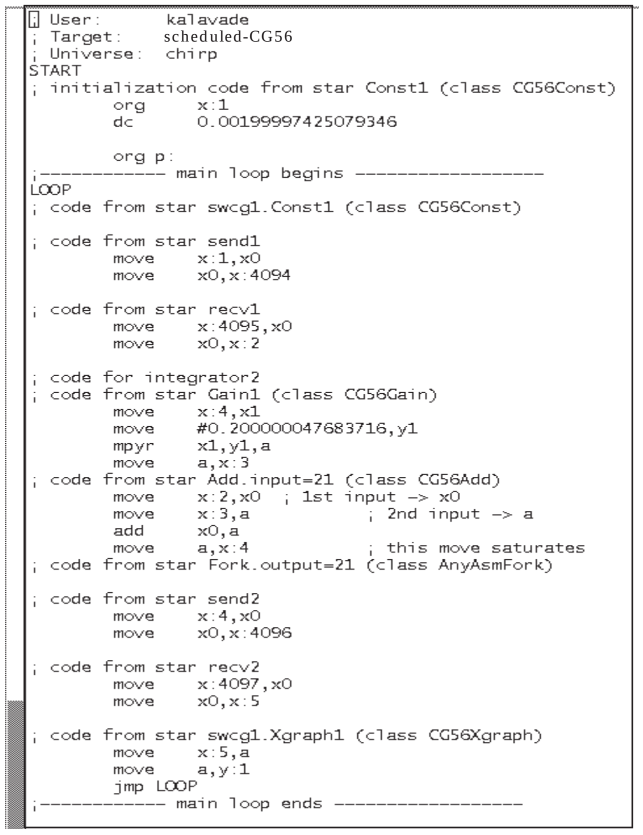
We have augmented the DSP 56000 code generation capability within Ptolemy ([Pino95a] and [Ptolemy95, Chapter 14]) to stitch together the code according to a *predefined* schedule. This predefined schedule is the order derived from the global schedule generated by the partitioning tool. The generated code is repeatedly executed by the DSP.

Figure 4.10 shows the software synthesis mechanism with the help of an example. Figure 4.10-a shows the task-level DAG annotated with the mapping and the ordering for a simple application (chirp signal generation). Suppose that nodes 1 (constant generator), 3 (integrator2), and 5 (xgraph) are mapped to software. Suppose further that node 3 is represented hierarchically as shown in Figure 4.10-b, while nodes 1 and 5 are monolithic. The software graph and the derived ordering is shown in Figure 4.10-c. The flattened software graph and its ordering are shown in Figure 4.10-d. The generated code (according to the flattened ordering) is listed in Figure 4.10-e.

4.1.5 Interface Synthesis

Interface synthesis involves synthesizing the global controller as well as the interface glue logic. A finite state machine description for the global controller can be generated using the order of transfers. An implementation for the controller can be synthesized using logic synthesis tools. The latches and other glue logic surrounding a hardware module can be synthesized using a template-based approach, such as that proposed by Sun *et al.* [Sun92]. The interface graphs can be used to generate a netlist connecting the various hardware modules. Currently, the interface synthesis is done manually, although the approach we have presented is relatively simple to automate.


Task-level DAG annotated with mapping and ordering
S
H
S
H
S
constant
Integrator
Int
Cos
Xgraph
1
2
3
4
5
ordering: 1,2,3,4,5
(generated by the partitioning tool)
(S: software, H: hardware)
(a)
(b)
3-1
3-2
3-3
In
Add
Gain
out
hierarchical representation for node 3
nodal schedule: 3-3, 3-1, 3-2
(c)
ordering derived from global schedule 1,3,5
software graph
communication address
x:4094
x:4095
x:4096
x:4097
1
s1
r1
3
s2
r2
5
ordering 1, s1, r1, 3, s2, r2, 5
(d)
flattened software graph
x:4094
x:4095
x:4096
x:4097
1
s1
r1
3-3
3-1
3-2
s2
r2
5
flattened ordering: 1, s1, r1, 3-3, 3-1, 3-2, s2, r2, 5
(e)



```
; User:       kalavade
; Target:     scheduled-CG56
; Universe:   chirp
START
; initialization code from star Const1 (class CG56Const)
        org     x:1
        dc      0.00199997425079346

        org p:
;------------ main loop begins ------------------
LOOP
; code from star swcg1.Const1 (class CG56Const)

; code from star send1
        move    x:1,x0
        move    x0,x:4094

; code from star recv1
        move    x:4095,x0
        move    x0,x:2

; code for integrator2
; code from star Gain1 (class CG56Gain)
        move    x:4,x1
        move    #0.200000047683716,y1
        mpyr    x1,y1,a
        move    a,x:3
; code from star Add.input=21 (class CG56Add)
        move    x:2,x0  ; 1st input -> x0
        move    x:3,a           ; 2nd input -> a
        add     x0,a
        move    a,x:4           ; this move saturates
; code from star Fork.output=21 (class AnyAsmFork)

; code from star send2
        move    x:4,x0
        move    x0,x:4096

; code from star recv2
        move    x:4097,x0
        move    x0,x:5

; code from star swcg1.Xgraph1 (class CG56Xgraph)
        move    x:5,a
        move    a,y:1
        jmp LOOP
;------------ main loop ends ------------------
```

Figure 4.10. Software synthesis for chirp signal generator.

4.1.6 Other Synthesis Approaches

In this section we discuss some of the other approaches to cosynthesis. Systems that synthesize just hardware or just software from synthesizeable descriptions have been around for a while. A comprehensive summary of these tools exists elsewhere [Camposano91][Bier95a] and we will not discuss them here. Instead, we will discuss a few representative systems that focus on the higher-level cosynthesis problem.

Chiodo *et al.* [Chiodo94] focus on the cosynthesis of control-dominated applications. An application is typically described in Esterel or VHDL. This is translated to an internal representation consisting of a network of CFSMs (codesign finite state machines). CFSMs are similar to hierarchical concurrent FSMs. In addition, they communicate by broadcast. Events can have arbitrary propagation times, and do not necessarily follow the synchronous hypothesis. Each CFSM corresponds to a component of the system being modeled and is manually assigned to either hardware or software. Hardware is generated using logic synthesis tools. For software synthesis, each node in the CFSM is translated to a thread and a run-time scheduler controls the execution of threads. This is different from our approach, where a run-time scheduler is not required since the schedule is known at compile time. The key strength of the semantics of the CFSMs is that these systems can be verified using formal verification techniques.

In the VULCAN system [Gupta93], an application is specified in HardwareC and translated into a control-dataflow based internal representation. During software synthesis, the internal representation is compiled to C code. Hardware is synthesized by passing the internal representation through highlevel synthesis tools.

The COSYMA system [Henkel94] starts with a C-like description of the application, which gets translated to an internal syntax graph. After partitioning, the internal graph is compiled to HardwareC and C representations for hardware and software synthesis respectively.

Srivastava *et al.* [Srivastava91] have developed a framework, called SIERRA, for synthesizing the hardware and software components for multi-board systems. An application is described as a hierarchical network of communicating sequential processes. The user maps the application onto a system architecture, consisting of multiple boards with dedicated hardware modules and programmable processors. They have developed techniques that automatically generate the netlist for the system architecture. Software synthesis involves generating code for each process; a real-time kernel on the processors manages the software execution. The interface is synthesized using a template-based approach [Sun92].

Chou *et al.* [Chou92] address the problem of synthesizing the device drivers and glue logic for the hardware-software interface between microcontrollers and hardware devices. Given a list of device ports that need to be connected to the microcontroller and a list of available microcontroller ports, they present a procedure for assigning device ports to controller ports.

Coelho *et al.* [Coelho94] present an interesting variant of the cosynthesis problem. They address the problem of resynthesizing the system when the specifications change, while ensuring that hardware does not have to be resynthesized. They present techniques to resynthesize software such that the timing constraints imposed by the hardware are still met.

4.1.7 Summary

Cosynthesis is the problem of synthesizing the hardware, software, and interface components of the system, starting with a partitioned DAG. Our synthesis techniques target an architecture consisting of a single programmable processor and multiple hardware modules. The architecture is assumed to be non-pipelined and nodes mapped to hardware and software communicate using a memory-mapped, self-timed, blocking mechanism. Hardware-software communication is managed efficiently using the ordered transactions principle.

Our approach to cosynthesis is to decompose the partitioned DAG into hardware, software, and interface graphs, where each node in the hardware and software graphs is a technology-dependent representation of the original node. Pre-existing synthesis tools are used to generate the final implementation from these graphs. The emphasis in our work has been on generating synthesizeable representations for the hardware and software components. We use preexisting synthesis tools (such as Hyper and Ptolemy) to generate the final implementation.

4.2 Cosimulation

Hardware-software cosimulation is the process of simulating the hardware and software components of a mixed hardware-software system within a *unified* environment. This includes simulation of the hardware modules, the processor, and the software that the processor executes.

Hardware is typically simulated using discrete-event or cycle-driven simulators. The processor that executes the software can be modeled at various levels of detail, depending on the desired accuracy and simulation speed. In general, there is a spectrum of approaches for modeling a processor [Rowson94][Chang95][Becker92][Shanmugan94]. Possible approaches include:

1. <u>Detailed processor models</u>: In principle, the processor components could be modeled using a discrete-event model of their internal hardware archi-

tectures (datapath, instruction decoder, busses, memory management unit, etc.) as they execute the embedded software. The processor internals are modeled in a way typical of hardware systems, often using VHDL or Verilog. The interaction between models of individual processors and other components is captured using the native event-driven simulation capability supported by a hardware simulator. Unfortunately, most processor vendors are reluctant to make such models available, because they reveal a great deal about the internal processor design. Moreover, such models are extremely slow to simulate.

2. Bus models: These are discrete-event shells that simulate the activity on the periphery of a processor without executing the software associated with the processor. This is useful for verifying very low-level interactions, such as bus and memory interactions, but it is difficult to guarantee that the model of activity on the periphery is accurate; it is also difficult to simulate the interaction of the software with the hardware.

3. Instruction-set architecture models: The instruction set architecture can be simulated efficiently by a C program. The C program is an interpreter for the embedded software. It updates a representation of the processor state and generates events to model the activities on the periphery of the processor when appropriate. This type of modeling can be much more efficient than detailed processor modeling because the internals of the processor do not suffer the expense of discrete-event scheduling.

4. Compiled Simulation: Very fast processor models are achievable in principle by translating the executable embedded software specification into native code for the processor doing the simulating. For example, code for a

programmable DSP could be translated into Sparc assembly code for execution on a workstation. This is called *binary-to-binary translation*. The translated code has to include code segments that generate the events associated with the external interactions of the processor. In principle, such processor simulations can be extremely fast. The dominant cost of the simulation becomes the discrete-event or cycle-based simulation of the interaction between the components.

5. Hardware Models: If the processor exists in hardware form, the physical hardware can often be used to model the processor in a simulation. Alternatively, the processor could be modeled using an FPGA prototype, for instance, using Quickturn. The advantage of this sort of processor model is the simulation speed, while the disadvantage is that the physical processor must be available.

Thus, depending on the desired simulation accuracy and speed, one of these models can be used for simulating the processor in the mixed hardware-software system. A cosimulation environment should allow the processor to be modeled using any of these approaches.

It is often useful to be able to simulate the system at different levels of abstraction, throughout the entire design process. For instance, at the start of the design process, the hardware components may not have been synthesized completely. At this point, to study the interaction of the hardware and software, the hardware could be modeled functionally. As the design evolves, and more implementation-level details of the hardware become available, the functional model of the hardware can be replaced by a more detailed model at the netlist level. On the other hand, once the detailed operation of the hardware has been verified, one could go back and replace the hardware components by a highlevel model to increase the speed of the simulation. A cosimulation environment should support this migration across levels of abstraction.

Some of the components in the mixed hardware-software system may be off-the-shelf components, whose design is not part of the current design process. Such components can be modeled by a functional representation through the entire design process. It is not required to model the internal architectural details since they can be assumed to be correct, besides, we have no control over changing the model of such a component anyway.

In addition to the digital components, a system may also contain analog components. The cosimulation environment should also be able to support the modeling of such components and their interaction with the rest of the system. Figure 4.11 summarizes these aspects of cosimulation.

Thus it is clear that a single simulator, such as a discrete-event hardware simulator, cannot support all these requirements of hardware-software cosimulation. A framework that allows these different models to be mixed and matched is required for effective cosimulation.

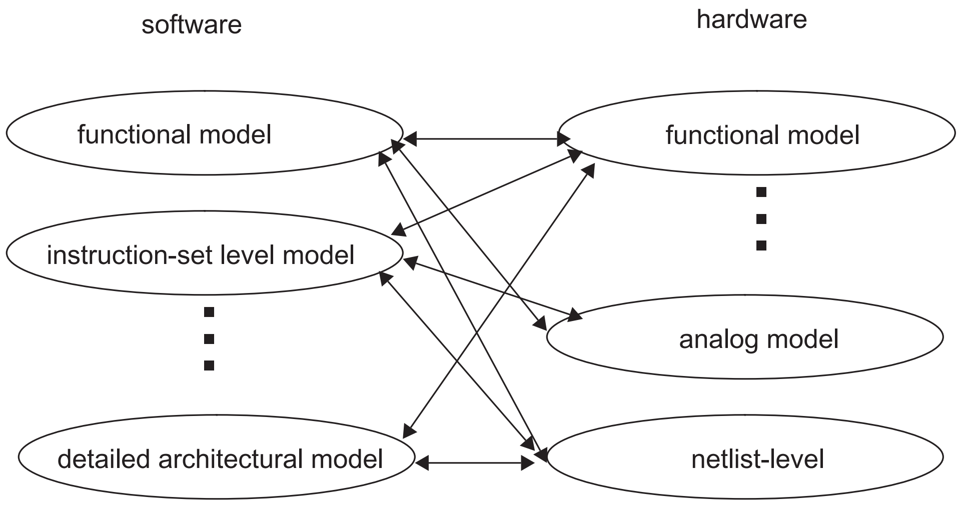


Figure 4.11. Cosimulation.

Ptolemy [Buck94a] is a heterogeneous simulation and prototyping environment, developed by the DSP Group at the University of California at Berkeley[9]. Ptolemy has the unique capability of supporting the integrated simulation of different models of computation. It meets most of the above-mentioned requirements of a cosimulation environment. We use Ptolemy for the cosimulation of mixed hardware-software systems.

In Section 4.2.1, we describe the features of Ptolemy that make it suitable for simulating systems with such diverse components. The use of Ptolemy for cosimulation is demonstrated with the help of an example in Section 4.2.2.

4.2.1 The Ptolemy Framework

Ptolemy is an environment for simulation and prototyping of heterogeneous systems. It uses object-oriented software technology to model each subsystem in its most natural and efficient manner, and has mechanisms to integrate heterogeneous subsystems into a whole.

Figure 4.12 shows the structural components of Ptolemy. The basic unit of modularity in Ptolemy is the Block. The system being simulated (or designed) is described as a network of Blocks. A Scheduler determines the operational semantics of a network of Blocks, i.e., it determines the order in which the Blocks are executed. The operation of the Scheduler is governed by the semantics of the underlying description. The lowest level (atomic) objects in Ptolemy are of the type Star, derived from Block. A Star is described by a C++ description. The description includes a function that gets invoked when a Block is initialized ("setup()"), a function to describe the run-time behavior of the Block ("go()"), and

9. While Ptolemy supports both simulation and synthesis of systems, we restrict the discussion in this section to the simulation aspects only. Complete details of Ptolemy can be found in [Ptolemy95]

a function that gets called at the termination of the simulation (“wrapup()”). A Galaxy, also derived from Block, contains other Blocks internally. A Galaxy may internally contain both Galaxies and Stars.

Ptolemy accomplishes the goal of multiparadigm simulation by supporting a plethora of different design styles encapsulated in objects called Domains. A Domain realizes a computational model appropriate for a particular type of subsystem. Some of the simulation Domains that are currently supported include ‘Synchronous Data Flow’ (SDF), ‘Dynamic Dataflow’ (DDF), ‘Discrete Event’ (DE), and the ‘Digital Hardware Modeling Environment’ (Thor).

A Domain in Ptolemy consists of a set of Blocks and Schedulers that conform to a common computational model — the operational semantics that govern how Blocks interact with one another. The Domain class by itself gives Ptolemy the ability to model *subsystems* differently, using a model appropriate for each subsystem. It also supports mixing these models at the system level to develop a heterogeneous system with different levels of abstraction. The mixture is hierarchical.

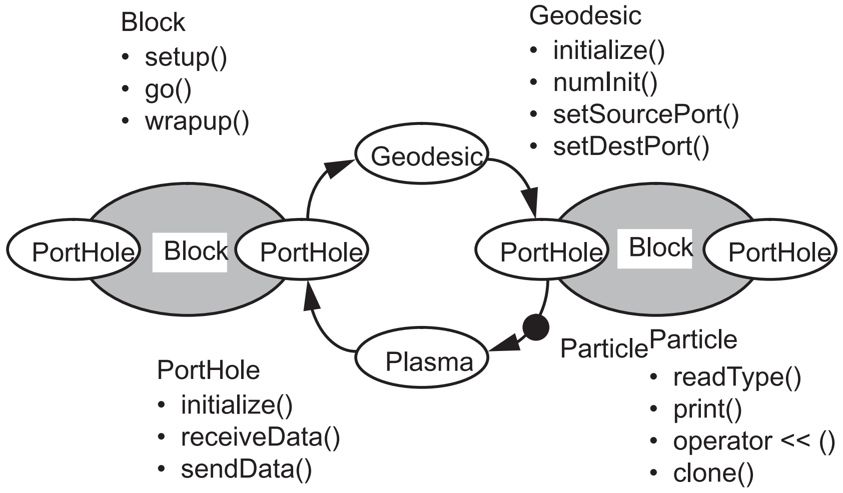


Figure 4.12. Block objects in Ptolemy send and receive data encapsulated in Particles to the outside world through Portholes. Buffering and transport is handled by the Geodesic and garbage collection by the Plasma.

Any model of computation can be used at the top level of the hierarchy. Within each level of the hierarchy, it is possible to have Blocks containing foreign Domains. This hierarchical heterogeneity is quite different from the concept of a *simulation backplane*, implemented for example in Viewlogic's SimBus system. A simulation backplane imposes a top-level model of computation through which all subsystems interact.

Figure 4.13 shows the top view of a system associated with a certain Domain called "XXX", associated with which are XXXStars and a XXXScheduler. A foreign subsystem that belongs to Domain YYY, and has its own set of YYYStars and a YYYScheduler is embedded in this XXXDomain system. This foreign subsystem is contained entirely within an object called an XXXWormhole. An XXXWormhole is a type of XXXStar — so at its external interface it obeys the operational semantics of the external Domain, but internally it consists of an entire foreign subsystem. A Wormhole can be introduced into the XXX Domain without any need for the XXXScheduler to know of the existence of the YYY Domain. The key to this interoperability is the interface between the internal structure of a

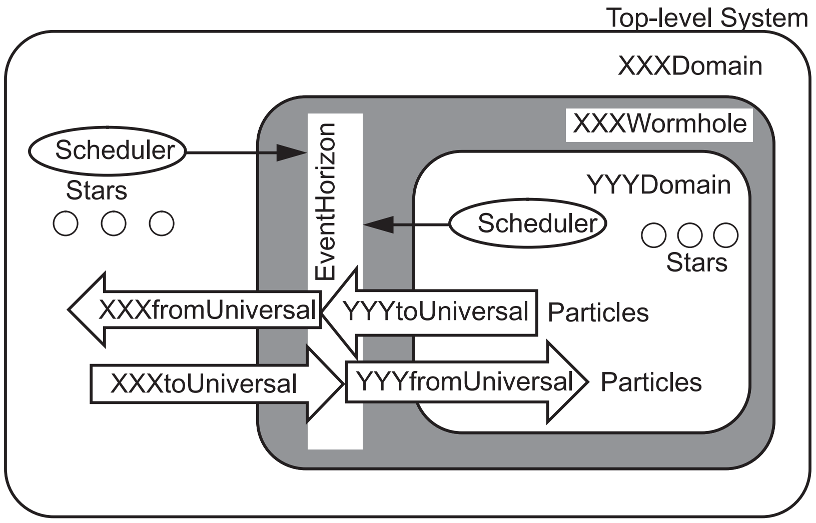


Figure 4.13. The universal EventHorizon provides an interface between the external and internal Domains.

Wormhole and its external environment. This interface is called the EventHorizon. The EventHorizon is a minimal interface that just supports exchange of data and permits rudimentary standardized interaction between schedulers. Each Domain provides an interface to the EventHorizon, and thus gains an interface to any other Domain.

The Domain and the mechanism of coexistence of Domains are the primary features that distinguish Ptolemy from otherwise comparable systems such as Comdisco's SPW and Bones, Mentor Graphic's DSPstation, and the University of New Mexico Khoros System [Rasure91].

We use the SDF and Thor simulation Domains for hardware-software cosimulation. Some details of these Domains are discussed next.

Synchronous Data Flow (SDF) Domain: The SDF Domain is a data-driven, statically scheduled Domain. "Data-driven" means that the availability of data on the inputs of a Star enables it. Stars with no inputs ("sources") are always enabled. "Statically scheduled" implies that the firing order of the Stars is determined only once during the start-up phase and this schedule is periodic. The SDF Domain supports simulation of algorithms, and also allows functional modeling of components such as filters and signal generators.

Thor Domain: The Thor Domain implements the Thor simulator [Thor86], which is a cycle-based register-transfer-level simulator for digital hardware. It supports the simulation of circuits from the gate level to the behavioral level. The Thor Domain makes it possible to simulate digital components ranging in complexity from simple logic gates to programmable DSP chips[10].

10. The Thor domain might likely be replaced in the future by a VHDL domain.

4.2.2 Cosimulation Using Ptolemy

Figure 4.14 shows an example of a mixed hardware-software system being simulated in Ptolemy. Figure 4.14-a shows the task-level SDF description of an application (telephone channel simulator) that is running on this system. Using the partitioning techniques described in Chapter 3, this application can be partitioned into hardware and software. The software synthesis mechanism described in Sec-

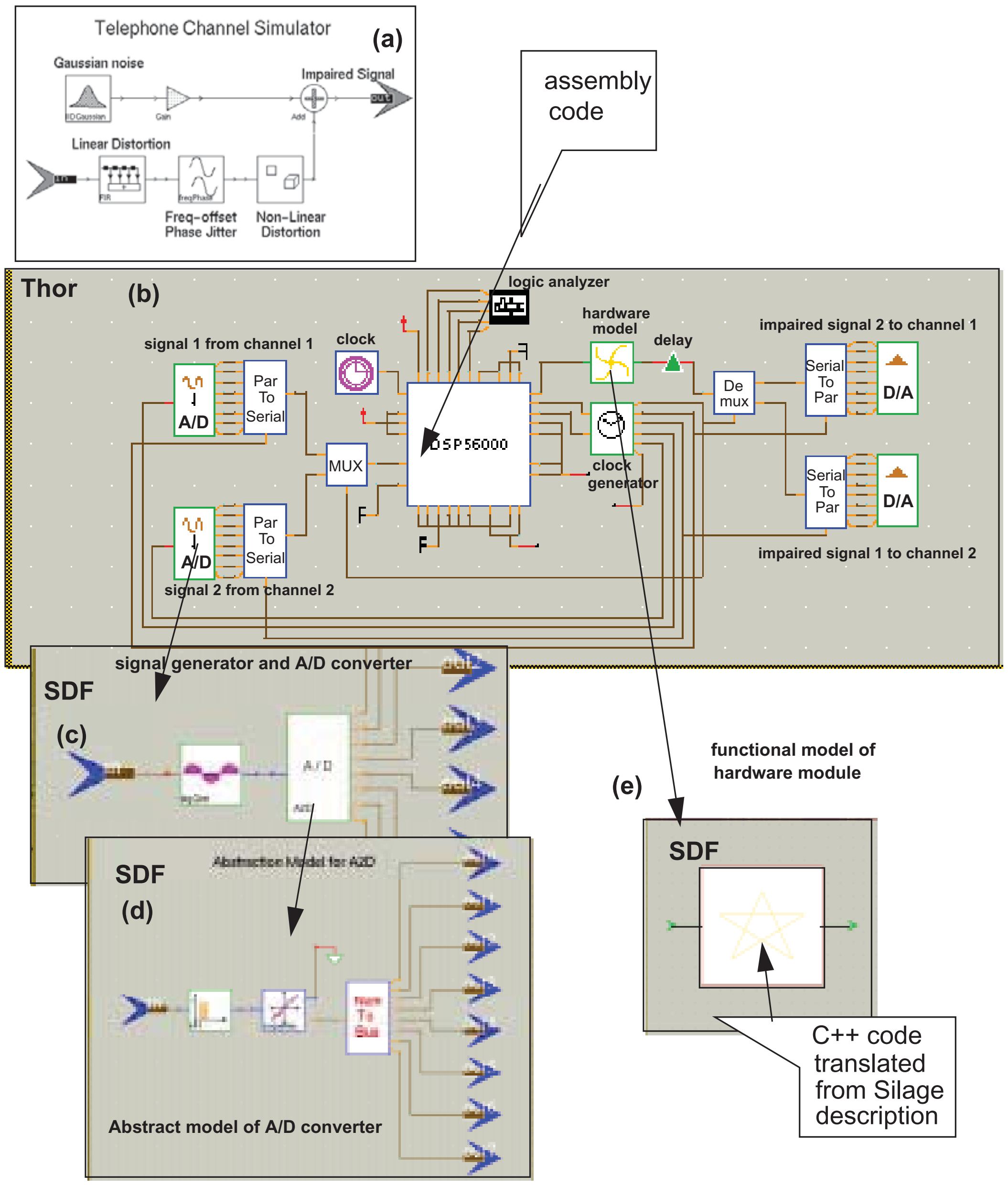


Figure 4.14. Hardware-software cosimulation using Ptolemy: system modeling

tion 4.1.4 is used to generate DSP assembly code corresponding to the nodes that are mapped to software. Hardware modules are synthesized for the nodes mapped to hardware using techniques described in Section 4.1.3. The next step in the design process is to simulate the interaction between the DSP (running the generated code) and the hardware modules.

4.2.2.1 System-level Modeling

The top-level model of the system is represented in the Thor Domain in Figure 4.14-b. The key components of the target architecture include the DSP, custom hardware, and the glue logic.

Modeling the DSP

The DSP56000 is modeled as a Star in the Thor Domain[11]. The “setup()” method of the DSP Star establishes a socket connection with *Sim56000*, Motorola’s stand-alone simulator for the DSP56000. *Sim56000* is an instruction-set simulator; it accurately models the behavior of each of the processor’s signal pins, while executing the code. During its “go()” method, this Star translates the logic values present at the processor’s pins into values meaningful to the simulator, transfers them to *Sim56000*, and commands the simulator to advance the simulation by one step. It waits for the simulator to transmit the new logic values back to the processor pins and continues with the rest of the simulation.

Functional model of the hardware

Nodes mapped to hardware can be modeled at different levels of abstraction. In this example, we model the hardware at a *functional* level. The Silage code for the hardware module is translated to C++ code. This code is nested inside an

11. Early work on encapsulation of the Motorola simulator for interfacing with a stand-alone Thor simulation was done by Bier *et al.* [Bier89]. We have extended their techniques to encapsulate the simulator into a Ptolemy Star.

SDF Star and simulated as a Wormhole within the parent Thor Domain, as shown in Figure 4.14-e[12]. When this wormhole is run, it models the functional behavior of the hardware module. At the output of the hardware module is a “delay” block that emulates the execution time of this hardware module[13]. Such a model for the hardware module allows us to model the timing associated with the interaction between the hardware module and the rest of the system without having to go through a slow architectural simulation of the module.

Alternatively, the hardware could be simulated at a *structural* level within the same framework by replacing the functional model and the delay by a structural VHDL description of the hardware module.

Modeling the glue logic and the system I/O

The glue logic consisting of clock generators, decoders, and latches is easily modeled in the Thor Domain.

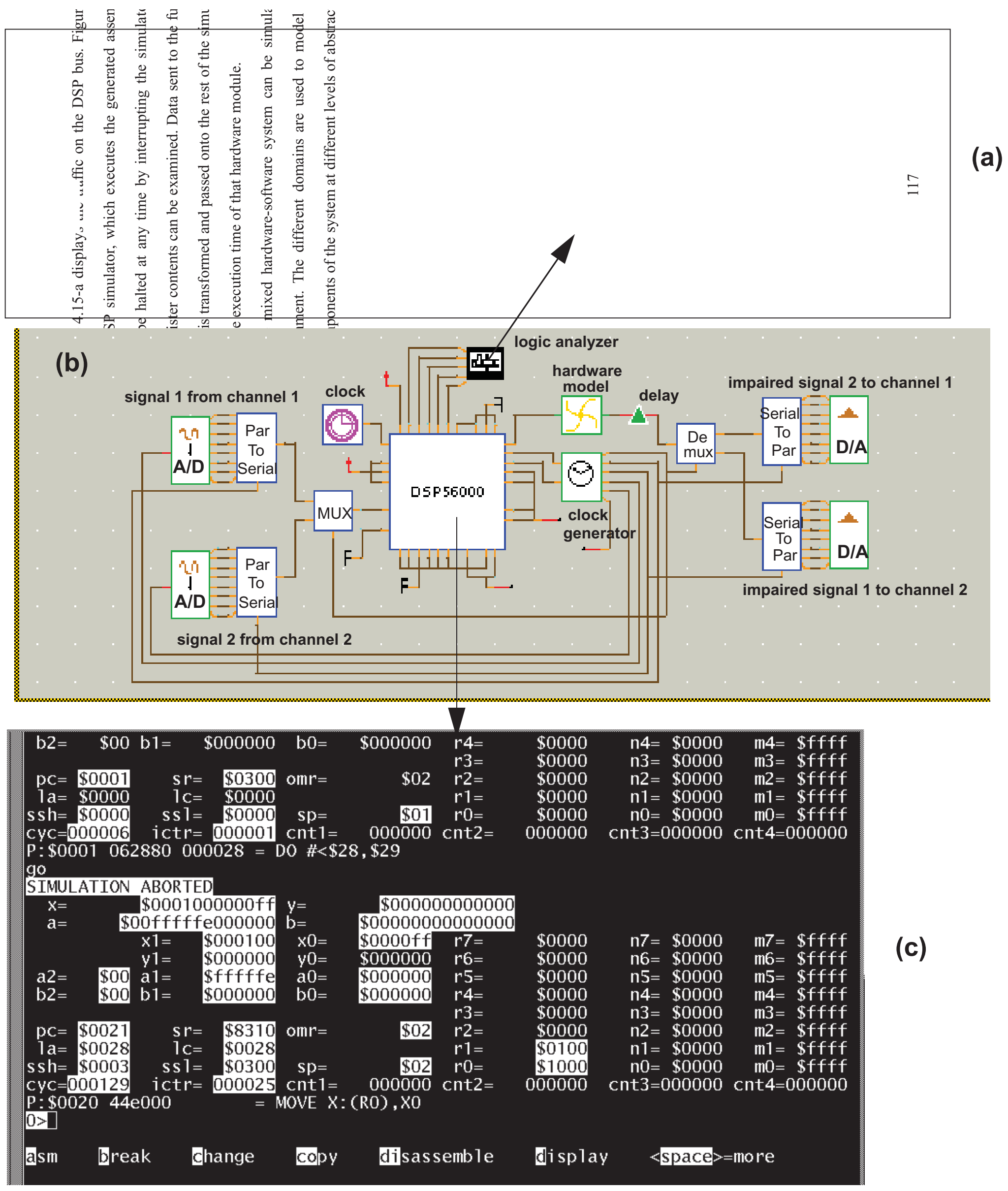
Besides processors and digital logic, it is often necessary to model analog components such as A/D and D/A converters, and filters that operate in conjunction with this digital hardware. These analog components can most conveniently be represented by their “functional models” using the SDF Domain. Often abstract functional modeling of components such as filters is sufficient — detailed behavioral modeling is not needed — particularly if an off-the-shelf component with well-understood behavior will be used in the final implementation. The Wormhole mechanism discussed earlier is used to mix the data-driven, statically-scheduled SDF models of analog components with event-driven, logic-valued Thor models of digital components within a single simulation. Figure 4.14-c shows a signal source being modeled as a sine generator followed by an analog to digital con-

12. Chapter 17 of [Ptolemy95] describes this Silage to C++ translation in some detail.
13. The execution time for a hardware module is determined by synthesizing an implementation for the node using hardware synthesis techniques discussed in Section 4.1.3.

verter. The A/D converter is modeled functionally as a bandpass filter followed by a quantizer, as shown in Figure 4.14-d. A functional model of this kind can be easily generated from the available device specifications.

4.2.2.2 System Simulation

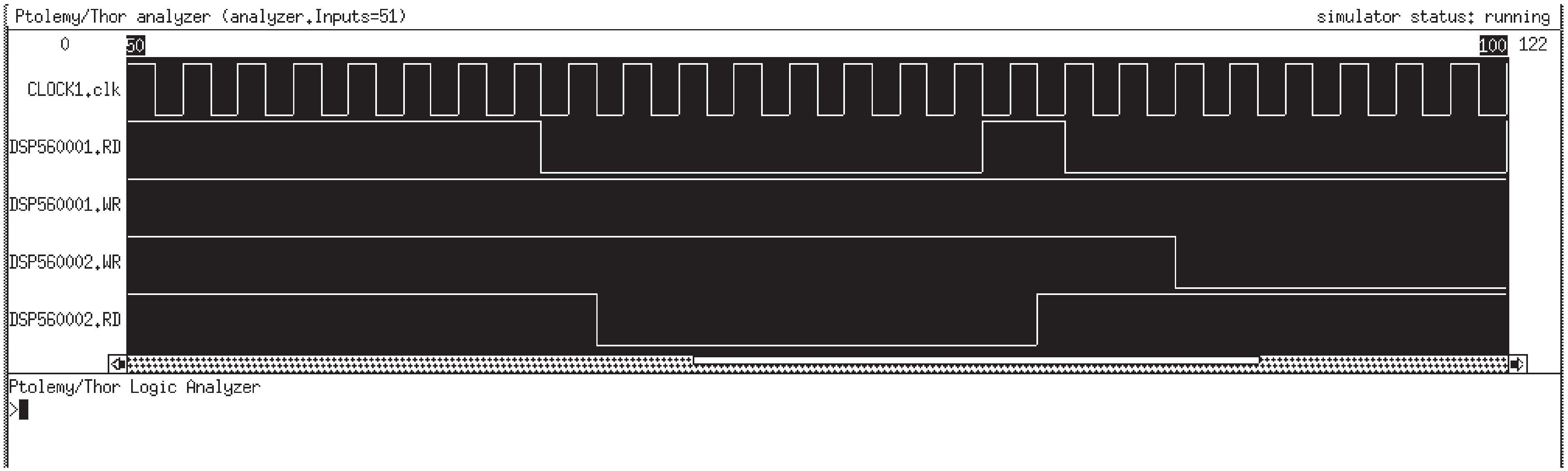
Figure 4.15 shows the run-time behavior of the system. The logic analyzer



(a)
4.15-a display... ...ffic on the DSP bus. Figur
SP simulator, which executes the generated assen
be halted at any time by interrupting the simulat
ister contents can be examined. Data sent to the fu
is transformed and passed onto the rest of the simu
e execution time of that hardware module.
mixed hardware-software system can be simul
ment. The different domains are used to model
ponents of the system at different levels of abstrac
117
(b)
logic analyzer
signal 1 from channel 1
clock
hardware model
delay
impaired signal 2 to channel 1
A/D
Par To Serial
MUX
DSP56000
clock generator
De mux
Serial To Par
D/A
impaired signal 1 to channel 2
signal 2 from channel 2
(c)

```
 b2=    $00 b1=    $000000   b0=    $000000  r4=      $0000    n4= $0000    m4= $ffff
                                             r3=      $0000    n3= $0000    m3= $ffff
 pc= $0001     sr=  $0300 omr=          $02  r2=      $0000    n2= $0000    m2= $ffff
 la= $0000     lc=  $0000                    r1=      $0000    n1= $0000    m1= $ffff
ssh= $0000    ssl=  $0000   sp=         $01  r0=      $0000    n0= $0000    m0= $ffff
cyc=000006   ictr= 000001 cnt1=    000000 cnt2=    000000  cnt3=000000 cnt4=000000
P:$0001 062880 000028 = DO #<$28,$29
go
SIMULATION ABORTED
  x=        $0001000000ff y=        $000000000000
  a=      $00fffffe000000 b=      $00000000000000
            x1=    $000100  x0=    $0000ff   r7=      $0000    n7= $0000    m7= $ffff
            y1=    $000000  y0=    $000000   r6=      $0000    n6= $0000    m6= $ffff
 a2=    $00 a1=    $fffffe  a0=    $000000   r5=      $0000    n5= $0000    m5= $ffff
 b2=    $00 b1=    $000000  b0=    $000000   r4=      $0000    n4= $0000    m4= $ffff
                                             r3=      $0000    n3= $0000    m3= $ffff
 pc= $0021     sr=  $8310 omr=          $02  r2=      $0000    n2= $0000    m2= $ffff
 la= $0028     lc=  $0028                    r1=      $0100    n1= $0000    m1= $ffff
ssh= $0003    ssl=  $0300   sp=         $02  r0=      $1000    n0= $0000    m0= $ffff
cyc=000129   ictr= 000025 cnt1=    000000 cnt2=    000000  cnt3=000000 cnt4=000000
P:$0020 44e000          = MOVE X:(R0),X0
0>

asm    break    change    copy    disassemble    display    <space>=more
```

Figure 4.15. Hardware-software cosimulation using Ptolemy.


Ptolemy/Thor analyzer (analyzer.Inputs=51)

simulator status: running

Ptolemy/Thor Logic Analyzer

```
>
```

5

DESIGN SPACE EXPLORATION

In the previous chapters we have discussed specific solutions for automated partitioning and synthesis. There we assumed a particular design objective and target architecture. In general, the system-level design problem cannot be posed as a single well-defined optimization problem from the designer's perspective. Typically, the designer needs to explore the possible options, tools, and architectures, choosing either automated tools or manually selecting his/her choices. This design space is quite large. As shown in Figure 5.1, a large number of target architectures and implementation technologies could be used to implement a system. Several optimization objectives and constraints are possible. The user also has access to a large number of design tools. The user might experiment with the design parameters, the target architectures, the optimization criteria, the tools used, or the sequence in which these tools are applied. As such, there is no *hardwired* design methodology. System-level design requires an *infrastructure* that *supports* efficient design space exploration — such an infrastructure permits considerable flexibility and at the same time relieves the user of the book-keeping.

As discussed in Chapter 1, tools that aid in design space exploration fall into two categories: estimation and management. Estimation tools are primarily used for *what-if* analysis, i.e., they give quick predictions on the outcome of applying certain synthesis or transformation tools. Management tools are required to orchestrate the design process, i.e., for systematic control of the design data, tools, and flow. In this chapter we focus on the management aspects of design space exploration; this is often referred to as *design methodology management*[1].

In Section 5.1, some of the related work in this area is discussed. In Section 5.2, we identify some of the requirements of a design methodology management framework. An infrastructure that supports these requirements is proposed in Section 5.3. We have implemented this infrastructure within the Ptolemy environment.

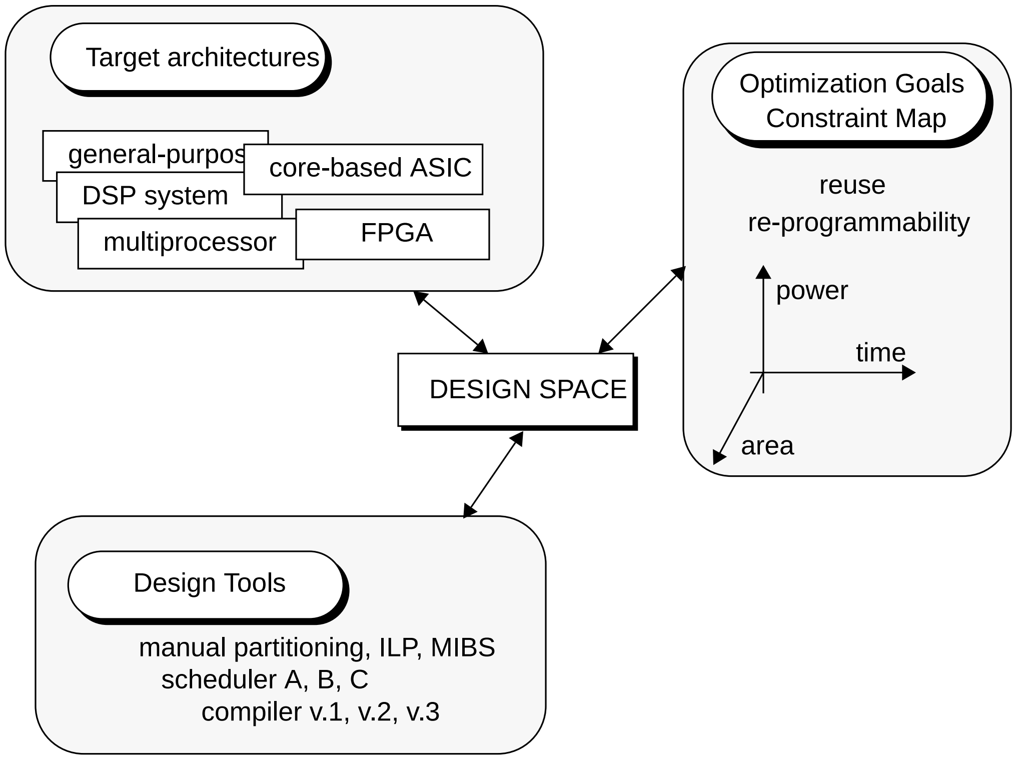


Figure 5.1. The design space.

1. Design methodology is defined as “the processes, techniques, or approaches employed in the solution of a problem”. Design methodology management (DMM) is formally defined as “definition, execution, and control of design methodologies in a flexible and configurable way” [Kleinfelft94].

Some details of the implementation are presented in Section 5.4. In Section 5.5, we discuss the operation of the design methodology management framework with the help of two representative design flows.

5.1 Related Work

Design methodology management (DMM) as such is not new; traditional DMM systems (often referred to as "frameworks") are used quite extensively in the physical VLSI design process. Kleinfelft *et al.* [Kleinfelft94] provide an exhaustive summary of the various research activities in this area. We mention only a few representative systems here.

The three main problems in design methodology management involve data management, tool management, and flow management.

The DMM systems used in the physical VLSI design process focus primarily on data management (i.e., maintaining consistent versions of data) [Harrison86] and tool management (i.e., invoking a user-specified tool after ensuring that the preconditions for enabling it are satisfied) [Chiuch90][Bosch91] [Brockman91]. Commercial CAD frameworks such as the Falcon framework [MentorGraphics2] also assist in tool and data management. The NELSIS framework [Bosch91][VanDerWolf93] provides a systematic representation and management mechanism for data and tools within a semantic database. CFI [CFI] defines standards for tool encapsulation and data models. As of this writing, no standard for flow management has been proposed by CFI.

The MMS framework [Allen91] focusses on distributed tool execution and multi-user environments. Recent efforts address the flow management problem. Some efforts approach the flow management problem from an AI angle [Knapp86]

[Bushness89], where the methodology and firing rules are stored in a knowledge-base and an inference engine determines the tool execution sequence. Yoda [Dewey89], a filter design system, has a knowledge-base of predictors (estimators). Predictors are used to determine the outcome of applying a particular tool and the results from such an exploration are used to construct a design plan (similar to a script), with feedback from the designer. The generated design plan is then automatically executed, where the actual tools are run. A trace-driven approach is proposed in [Casotto90], where a sample design session (the sequence of tools run by the user) is saved and future design sessions can be automatically controlled by following this trace.

CODES [Buchenrieder92] is a framework for codesign. It provides an open architecture for the integration of commercial and proprietary tools. A graphical representation of a design flow is translated to an internal Petri net representation. This is analyzed to determine firing rules. A codesign manager invokes the tools based on these firing rules.

Bentz et al. [Bentz95] present an information-based approach to design. They combine some of the traditional design methodology management features (such as data and tool management) with novel features for design space exploration via a tool called *Design Agent*. The *Design Agent* assists the user in collecting and managing information about a design. The designer queries for specific feedback (example: “what is the area of this FIR filter when implemented as an ASIC in a particular technology?”, or “what is the code size when implemented on a particular DSP?”), rather than explicitly performing tasks to obtain this information. The *Design Agent* computes the information based on *actions* (sequence of tools) specified its database. These actions are context-specific, that is, different sets of actions apply to different design *domains* (such as ASIC design and DSP design).

5.2 Design Methodology Management: Requirements

Our focus is primarily on design flow specification and management. We do not address issues of database management, multi-user operation, and distributed tool execution. Our goal is to develop a framework that simplifies the designer's tasks by (1) providing mechanisms that allow the design flow to be specified in an intuitive way, (2) automatically invoking the tools in the design flow whenever possible, and (3) managing the infrastructure when the user decides the sequence of tool execution. The key features required for this are discussed next.

Flow Specification

Since tools involved in the system-level design process are often computationally intensive, it is important to avoid unnecessary invocation of tools. This requires that the design flow be specified in a *modular* way so that only the desired tools may be invoked. It should also be possible to specify the design flow hierarchically, in order to retain design modularity.

Specification of the flow also requires *iterative* and *conditional* constructs. Consider an example where an application is to be mapped to a multiprocessor system. Assume that the number of processors is not known apriori. The design sequence usually is to (1) estimate the number of processors required, (2) schedule the application onto these processors and compute the resultant throughput, (3) if the throughput does not meet the desired throughput, repeat (1), else continue with the remaining parts of the design such as code generation and netlist generation. To express such a design flow, the flow specification mechanism should support constructs such as conditionals and iterations.

A number of design tools are available for each step in the design process.

Consider hardware-software partitioning for example. If the application involves few components that have obvious mappings, partitioning could be done manually. If the mappings are not obvious, an exact and time-consuming integer-linear programming approach could be used to determine the optimal mappings. On the other hand, if the application is quite complex and there are several options available for implementing the different components, an efficient heuristic (such as the MIBS algorithm) could be used. If the design flow is *parameterizeable*, a new flow need not be developed for each type of tool used. Depending on the design size, the available design time, and the desired accuracy, one of the tools can be selected. This selection can be done either by the user, or by embedding this design choice within the flow.

Flow Execution

A designer should not have to keep track of the tools that have already run and those that need to be run. When the parameters or the data associated with a tool change, the entire design flow need not be re-run; only the affected tools should be run again. Keeping track of the tools that need to be run is quite cumbersome. A mechanism that automatically determines the sequence of tool invocations is needed. This calls for a mechanism much like a graphical "make" utility [Feldman79].

Flow Management

Different types of tools, with varying input and output formats, are used in the system-level design process. In the very least, a mechanism to automatically detect incompatibilities between tools is required. Data translators could also be invoked automatically.

Versions of tools and design flows also need to be maintained; it is not sufficient to just keep track of versions of data.

5.3 Infrastructure

We have developed an infrastructure that tries to support most of the requirements discussed in the previous section. Details of the infrastructure are discussed in this section. In Section 5.3.1, we present the underlying models used to specify the design flow, tools, and data. We use the term *dependency* to qualify the conditions that require a tool to be invoked for execution. In Section 5.3.2, we identify different types of dependencies. The flow execution mechanism analyzes the dependencies and automatically invokes tools within a design flow. Details of the flow execution mechanism are presented in Section 5.3.3.

5.3.1 Flow, Tool, and Data Model

Figure 5.2 illustrates the user's view of the design flow and tools. The design *flow* is specified as a directed graph, where nodes represent tools, and arcs specify the ordering between tools[2].

Tools encapsulate actual programs. Tool parameters specify the arguments for these programs. A tool can have multiple input and output ports. The ports are used to transfer filenames between tools. A "source" tool (such as a signal genera-

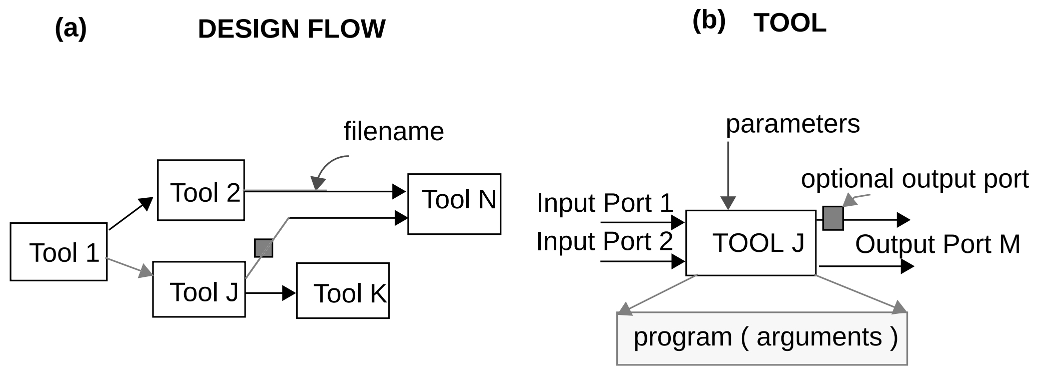


Figure 5.2. User's view of the design flow and tools.

2. Certain restrictions are imposed on the design flow in our implementation so as to avoid possible nondeterminacy. Refer to Section 5.4.

tor) has no input ports, while a “sink” tool (such as a display tool) has no output ports.

Ports can be either *required* or *optional*. The difference between required and optional input ports is that a tool cannot run unless it has data (valid filenames) on all its required input ports. A tool can run even if data is absent on an optional input port. When a tool is run, it generates data on all its required output ports, but not necessarily on optional output ports. Optional ports facilitate the use of conditionals and iterations in flows[3]. To understand this, consider the multiprocessor synthesis example mentioned earlier. The design problem is to synthesize a multiprocessor system that meets a desired throughout with a minimum number of processors. A possible design flow is shown in Figure 5.3. The *proc_estimator* determines the number of processors required and is described hierarchically as shown in the figure. The *estimator* tool estimates the number of processors needed. The estimated number of processors *num_procs_est* is given to a *scheduler* that computes the actual throughput for this number of processors. The *comparator* compares the actual throughput to the desired throughput and sends feedback to the *estimator.* The *estimator* has both a required input (port *x*), and an optional

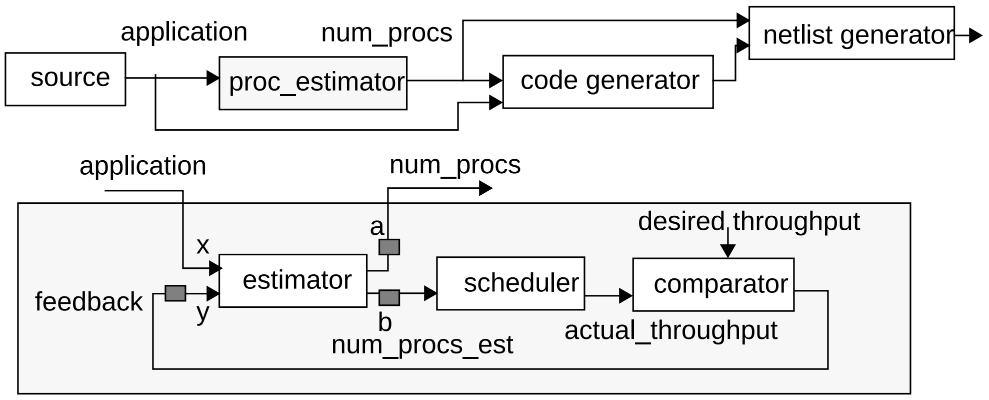


Figure 5.3. A design flow for the multiprocessor synthesis example.

3. Note that initial tokens can be used on arcs to equivalently model the functionality offered by optional ports.

input (port *y*). The *feedback* from the *comparator* is fed to the optional port of the *estimator.* If there is no feedback, the *estimator* estimates the number of processors based on its own information. If there is feedback, it updates its earlier estimate. Both output ports of the *estimator* are optional. When the estimation loop converges, the *estimator* generates data on the optional output port *a,* otherwise it generates the data for the revised estimate on the other optional output port *b*.

While optional ports permit modeling of a wide variety of applications, their use can potentially lead to nondeterminate behavior. In our implementation, we handle nondeterminacy by imposing strict restrictions on the allowed design flows for fully automated design flow execution. The flow scheduler identifies a potential nondeterminacy in a design flow and alerts the user. In most cases, the user knows what he/she had in mind when designing the design flow and can guide the system accordingly. The details of our implementation are described in Section 5.4.

Figure 5.4 shows the internal model of the tools and the data associated with a tool's input and output ports. Associated with each tool is a flag, called *Param_Changed_Flag,* which gets set when parameters of a tool are changed. Associated with each port of a tool are several attributes: $File\_Name_{last}$, $File\_Name_{new}$, $Time\_Stamp_{last}$, $Time\_Stamp_{new}$, and *Optional_Flag*. $File\_Name_{last}$ and $Time\_Stamp_{last}$ attributes store the filename and the timestamp[4] of the data on a port as of the earlier invocation of the tool. $File\_Name_{new}$ (and $Time\_Stamp_{new}$) represent the filename and the timestamp of the data updated on a port in the *current* invocation of the tool. *Optional_Flag* indicates whether the port is required or optional. The infrastructure stores the internal representation of the design flow as shown in Figure 5.4-c.

4. The timestamp indicates when the data associated with the file was last modified.

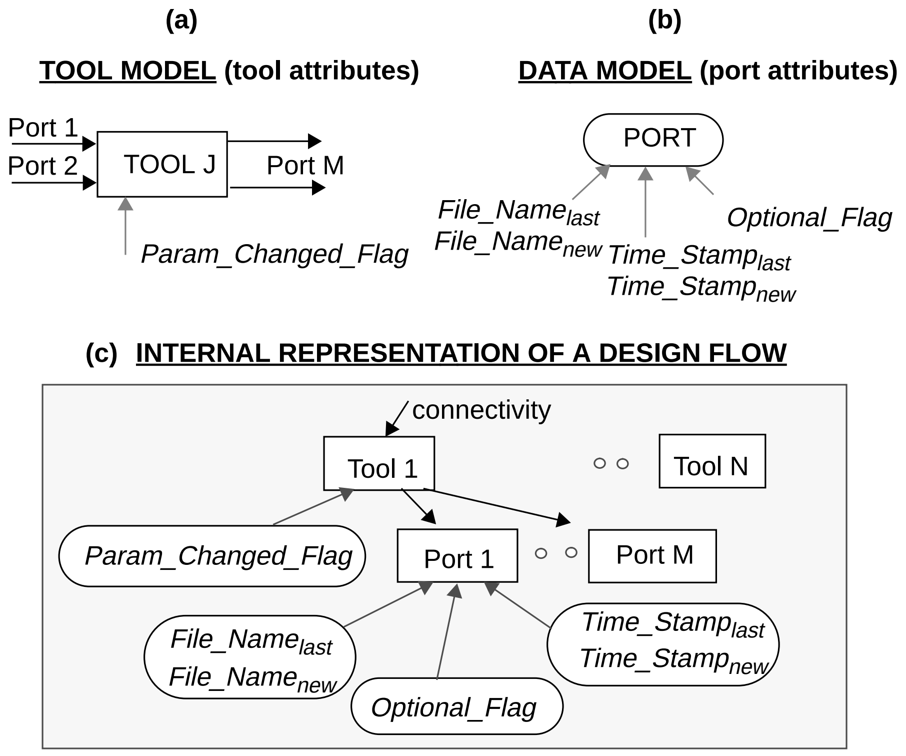


Figure 5.4. Internal representation of the tools and data.

5.3.2 Dependencies

The *Param_Changed_Flag*, and the filename and timestamp attributes are used to determine whether a tool needs to be run. For example, if the parameters of a tool have changed since its last invocation, the tool needs to be run. Similarly, if the filename associated with the current invocation of a tool is different from the filename associated with its previous invocation, this indicates a change in data, and requires the tool to be run again. We use the term *dependency* to qualify this behavior. Figure 5.5 shows three types of dependencies that are supported.

A *temporal* dependency tracks the timestamps associated with the data on the input ports of a tool. A tool needs to be run if its temporal dependency is alive, i.e., if the timestamp on any of its inputs is *newer* than the timestamp associated with the previous invocation of the tool, or if $Time\_Stamp_{new} > Time\_Stamp_{last}$. A

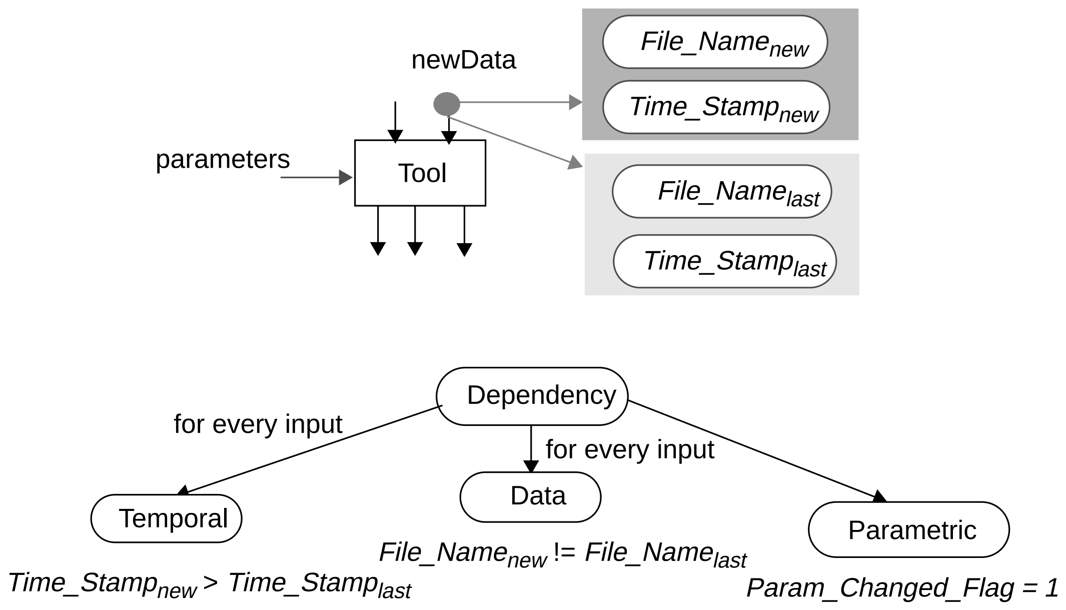


Figure 5.5. Dependencies used for tool management.

data dependency tracks changes in the filenames associated with the input ports of the tool. A tool needs to be run if its data dependency is alive, i.e., if the filename of the data associated with any input port is *different* from the filename associated with the previous invocation of the tool, or equivalently if $File\_Name_{last}$!= $File\_Name_{new}$. A *parametric* dependency tracks parameter changes; a tool is run if any parameter changes.

5.3.3 Flow Management

Automatic flow invocation is based on analyzing the tool dependencies and executing tools as required. A tool is said to be *enabled* when all of its required input ports have data. Absence of data on the optional input ports does not affect enabling. Once enabled, a tool is checked for dependencies. A tool is *invoked* (*run*) when at least one of its dependencies is alive. On execution, a tool generates data on its required output ports, and possibly on its optional output ports.

As shown in Figure 5.6, two types of flow invocation mechanisms are desired: data-driven, and demand-driven. In the data-driven approach, the flow

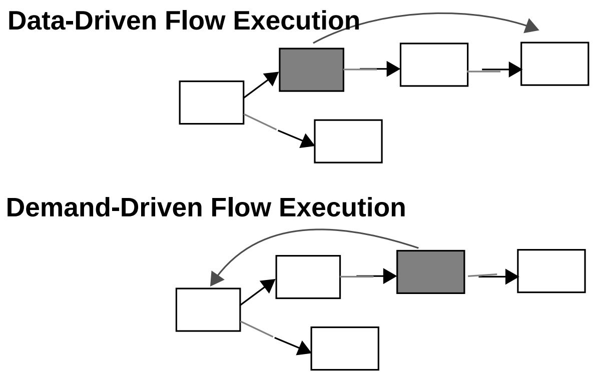


Figure 5.6. Flow execution mechanisms.

scheduler traverses the flow according to precedences. The process halts when all tools with live dependencies have been exhausted. In the demand-driven mode, the user selects a tool for execution. The scheduler traverses the predecessors and executes all tools with live dependencies on the path. The details of the scheduler are given in the next section and in Appendix A9.

5.4 Implementation: DMM Domain within Ptolemy

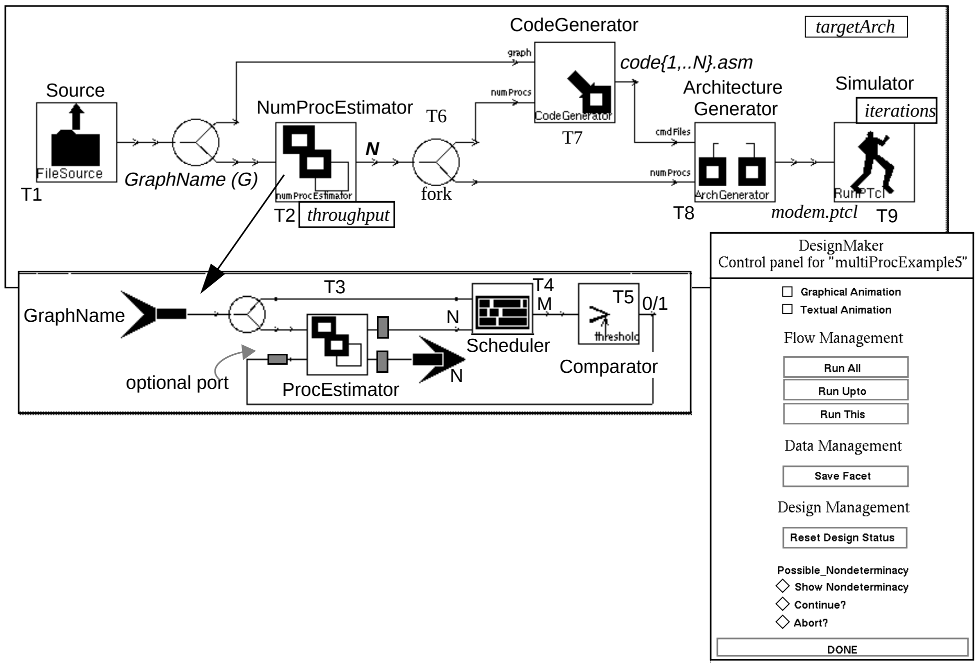
We have implemented the design methodology management framework within the Ptolemy environment as a separate *domain* (called DMM domain). The details of the flow specification and execution mechanisms are described next.

Flow Specification

The design flow is specified as a graphical netlist (through the Ptolemy user interface, VEM). The netlist shows the connectivity between tools. In the current implementation, the flow is restricted to having a single "source" tool; multiple sources are not allowed. The flow can be described hierarchically. Figure 5.7 shows the design flow for the multiprocessor synthesis example described earlier. This flow will be discussed in more detail in Section 5.5.1. Optional ports can be used to define conditionals and iterations in the flow definition. However, this could lead to nondeterminate flows. A following section on the flow scheduler discusses the operation of the scheduler when this happens.

Tool Encapsulation

Tools are encapsulated within Stars (the atomic elements in Ptolemy, described in Section 4.2.1). Tool encapsulation involves writing scripts that call various programs. These programs could be either other Ptolemy functions or stand-alone executables. Figure 5.8 shows an example of tool encapsulation. It shows the *Code Generator* tool from the multiprocessor synthesis example. The *Code Generator* generates a multiprocessor implementation (multiple programs) for the input application. It uses a code generation routine within Ptolemy (*ptkGenCode*) to generate the code. The *Code Generator* has two inputs (both required): the application description (*graph*) and the number of processors (*numProcs*). The tool generates programs for all the processors. The output of the tool is specified as a file (*codeFileNames*) containing the names of the generated pro-



DesignMaker
Control panel for "multiProcExample5"

- Graphical Animation
- Textual Animation

Flow Management

Run All | Run Upto | Run This

Data Management

Save Facet

Design Management

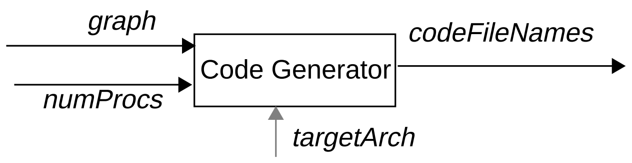
Reset Design Status

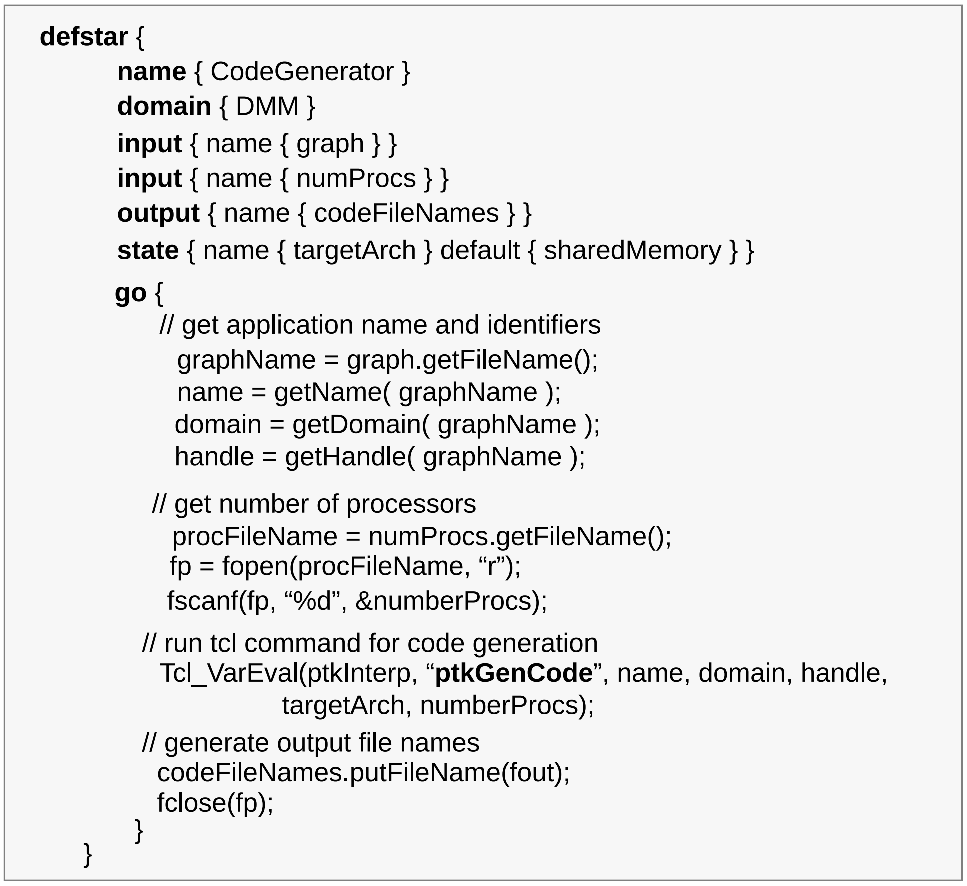
Possible_Nondeterminacy
◇ Show Nondeterminacy
◇ Continue?
◇ Abort?

DONE

Figure 5.7. The design flow for the multiprocessor synthesis example, specified within the DMM domain in Ptolemy.

grams. The *Code Generator* has a parameter (*targetArch*) that specifies the assumed target architecture. The function "*go*()" contains the code that gets executed when the tool is invoked. When invoked, the tool first reads in the name of the file containing the graph (*graphName*). The functions *getName*(), *getDomain*() and *getHandle*() obtain the identifiers for the application pointed to by *graphName*. The number of processors is read into the variable *numberProcs* by scanning the input file *procFileName*. The Ptolemy routine that generates the code





```
defstar {
        name { CodeGenerator }
        domain { DMM }
        input { name { graph } }
        input { name { numProcs } }
        output { name { codeFileNames } }
        state { name { targetArch } default { sharedMemory } }
        go {
            // get application name and identifiers
              graphName = graph.getFileName();
              name = getName( graphName );
              domain = getDomain( graphName );
              handle = getHandle( graphName );
            // get number of processors
              procFileName = numProcs.getFileName();
              fp = fopen(procFileName, "r");
              fscanf(fp, "%d", &numberProcs);
            // run tcl command for code generation
              Tcl_VarEval(ptkInterp, "ptkGenCode", name, domain, handle,
                          targetArch, numberProcs);
            // generate output file names
              codeFileNames.putFileName(fout);
              fclose(fp);
        }
    }
```

Figure 5.8. An example of Tool encapsulation: *CodeGenerator* tool.

(*ptkGenCode*) is then called with the application identifiers, the target architecture, and the number of processors. The result is written into the file *codeFileNames*.

Flow Scheduler (*DesignMaker*)

The tool writer need not worry about the underlying timestamps and filenames. The DMM attributes (*Param_Changed_Flag*, $Time\_Stamp_{last}$, $File\_Name_{last}$, $Time\_Stamp_{new}$, $File\_Name_{new}$, and *Optional_Flag*) and flow netlists are stored within the Oct database [Harrison86]. The flow manager called *DesignMaker* analyzes these attributes and automatically invokes the tools. Figure 5.9 shows the control panel associated with *DesignMaker.* There are three possible approaches to executing the flow: (1) The user opts to run the entire design flow (*Run All*), or (2) The user selects a certain tool upto which the flow should be executed (*Run Upto*), or (3) The user asks for a specific tool to be invoked (*Run This*).

In the *Run All* mode, the flow scheduler traverses the design flow, starting with the source tool, executing tools as necessary. To do this, it maintains a list of enabled tools. An enabled tool is checked for its dependencies and invoked for execution if any of its dependencies is alive. On execution, the appropriate descen-

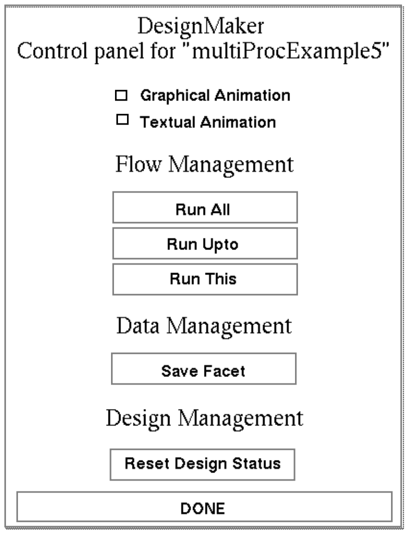


Figure 5.9. Control panel in the DMM domain.

dents of the tool are identified, and their corresponding input port is marked. If all the required input ports of a descendent tool are marked, the descendent tool is added to the list of enabled tools. Further details are presented in Appendix A9.2.

As mentioned earlier, it is possible that a flow may have a nondeterminate behavior[5] due to the presence of optional ports. Currently, the scheduler does very strict checking and conservatively flags flows that are possibly nondeterminate. The pseudocode for the algorithm for detecting nondeterminacy is given in Appendix A9.2. Some of these flagged flows can be determinate if the user has taken appropriate care in designing the flow. In such a case, the user can choose to use our flow scheduler at his/her risk. Alternatively, the user can schedule the tools manually in the sequence desired, by using the *Run This* mode.

In the *Run Upto* mode, the user selects a certain tool upto and including which the flow should be executed. All the predecessors of the selected tool are identified and tagged. The design flow is then traversed as in the *Run All* mode; only the tagged tools are executed. In the current implementation, we support this mode for acyclic flows only. The details of the scheduling algorithm are given in Appendix A9.3.

As mentioned earlier, the user can choose to manually schedule the flow by using the *Run This* mode. Details are given in Appendix A9.4.

5.5 Examples

The design methodology management domain and the operation of *DesignMaker* are described next with the help of two examples.

5. The generated data might not be independent of the sequence used to invoke the tools.

5.5.1 Multiprocessor System Design

Consider the design flow for the multiprocessor synthesis example, shown in Figure 5.7. A dataflow graph *G* (*GraphName*) describing the application (ex: music synthesis) and a specific interprocessor communication mechanism (*targetArch*, for example shared memory) are specified by the user. The goal is to synthesize this multiprocessor system with the minimum number of processors such that a required *throughput* is met. The synthesis process generates the hardware components (netlist) and the software components (programs running on the processors).

We now run through a typical flow execution sequence. Suppose that the required *makespan* (or 1/*throughput*) is 320 cycles.

When the *Run All* command is issued, the scheduler detects a possible nondeterminacy between tools *T4* and *T6* [6]and alerts the user accordingly. The control panel in Figure 5.7 shows the query for continuing or aborting the run. We know, however (by the way we have designed it), that the estimator tool *T3* (*ProcEstimator*) generates the output going to *T6* only after *T4* has finished. Hence, there is no nondeterminacy. We can then select the "continue" option to use the automated scheduling. The operation of the system under this selection is discussed next.

Suppose that the flow is run the very first time using *Run All*. Tools are examined for active dependencies by traversing the flow according to precedence ordering between tools. *Source (T1)* outputs the dataflow graph *G* specified by *GraphName*. *NumProcEstimator* (*T2*) and *Code Generator* (*T7*) are dependent on *T1*. Note that *T2* is enabled, while *T7* is not (the second input *N* has not yet been generated). *T2* is a hierarchical description of the estimation process, which deter-

6. Since the nondeterminacy condition is (using the notation defined in Appendix A9) $T4 = P(T6)$, $T6 \neq P(T4)$, and $T4 \neq OP(T6)$.

mines the minimum number of processors required to implement *G* at the desired throughput by an iterative computation. The operation of the estimator is described next.

ProcEstimator (T3) estimates the number of processors (*N*) required to implement *G*. *T3* has an *optional* input that receives an indication as to whether or not the current *N* satisfies the throughput requirements. When this input is available, *T3* uses it to improve the estimate for *N,* and when it is not, *T3* computes the estimate from scratch. Note that the loop (*T3*-*T4*-*T5*) does not deadlock because this input is optional. At the start the feedback input is not available since *T5* has not run. In this case, *T3* computes the estimate based on its estimation algorithm. Estimators of different accuracy can be selected by changing parameters of this block. Consider a simple estimator that computes the estimate assuming maximum parallelism. Suppose that the sum of execution times of all the nodes in *G* is 900. *T3* estimates a lower bound of 3 (=900/320) processors. The *Scheduler* (*T4)* then schedules *G* onto *N* (*N*=3) processors and determines the actual time required (*makespan M*) to implement *G*. Different scheduling algorithms can be selected by changing parameters of *T4*. Suppose it computes the makespan to be 350. The *Comparator (T5)* compares the required makespan (320) and *M* (350) to generate the control signal for *T3*. *T3* is enabled by the input received on its optional input, and refines its estimate of *N* to 4. Note that up to this point, as the system is being run the very first time, data dependencies are alive for all the blocks. For *T4* however, the data dependency is no longer active (since the same filename is retained), but temporal dependency becomes alive. Recall that temporal dependencies detect out-of-date data. As the timestamp on the input to *T4* is newer than the timestamp associated with the previous invocation of *T4,* T4 is re-run and it recomputes *M* (say, 290). *T5* sends a output of zero to *T3*. *T3* detects convergence of the estimation loop when the feedback from *T5* is zero, and sends the computed number of processors *N* to the fork *T6*. *T6* merely copies its input data to its two outputs.

Since its second input is available, the *Code Generator* (*T7*) is now invoked. *T7* synthesizes software for the *N* processors. *T8* generates the netlist for the system. The generated architectural model, where the processors run the synthesized software, is then simulated by the *Simulator* (*T9*). *T9* has a parameter (*iterations*) that indicates the number of cycles to simulate the design for. After *T9* runs, no live dependencies exist and the flow execution stops.

Subsequent changes to the flow or data could render parts of the flow invalid. For instance, suppose that the parameter "*iterations*" to *T9* is modified and the *Run All* command is issued. A parametric dependency is activated for *T9*. As no other dependencies are alive, only *T9* is invoked. If the *Run All* command is issued again, since no data files or parameters have changed, no tool is invoked. Next, suppose that *Run All* is issued after *GraphName* is changed. *T1* is first invoked due to a parametric dependency. This activates a data dependency for *T2* as the input data (*GraphName)* changes, causing it to be invoked. Other tools on the downstream flow that are data-dependent on *T2* (the complete flow) are then invoked by the scheduler.

5.5.2 Design Assistant

The Design Assistant, described in Chapter 2, is a framework for system-level codesign. Figure 5.10 shows the Design Assistant implemented in the DMM domain.

Source (T1) outputs the SDF graph *G* specified by *GraphName* (say *simple.sdf*). *G* is then partitioned into hardware and software. Currently the Design Assistant supports manual partitioning.

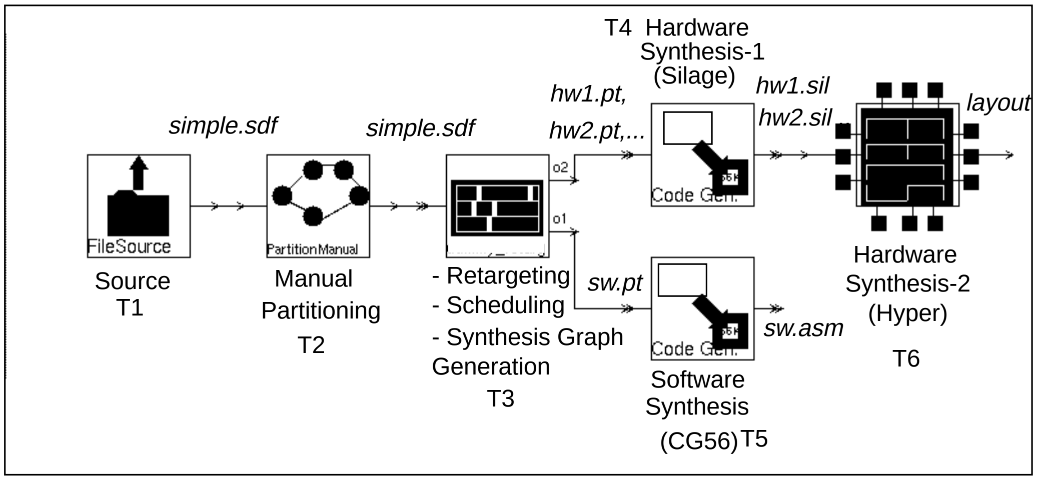


Figure 5.10. The Design Assistant implemented in the DMM domain.

Figure 5.11 shows the behavior of *T2* when configured for manual partitioning. The application, which is to be partitioned, is automatically displayed by the partitioning tool (Figure 5.11-a). The partitioning tool also brings up a selection panel to aid in manual partitioning (Figure 5.11-b). The user can then select parts of the application and assign them to hardware or software.

The next tool in the design flow is *T3*. *T3* first retargets the nodes in the SDF description to technology-dependent nodes (CG56 for software-mapped nodes, and Silage for hardware-mapped nodes). *T3* then inserts the *Send* and *Receive* nodes for communication across the hardware-software boundary. Since we have done manual partitioning, a global schedule needs to be generated. Given

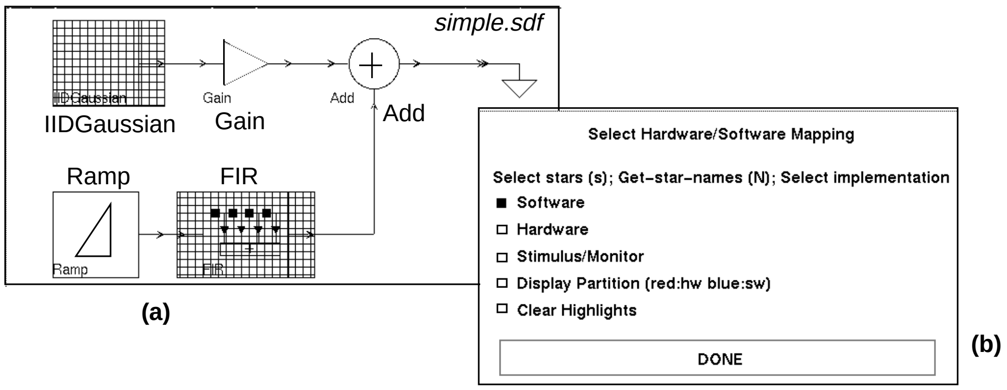
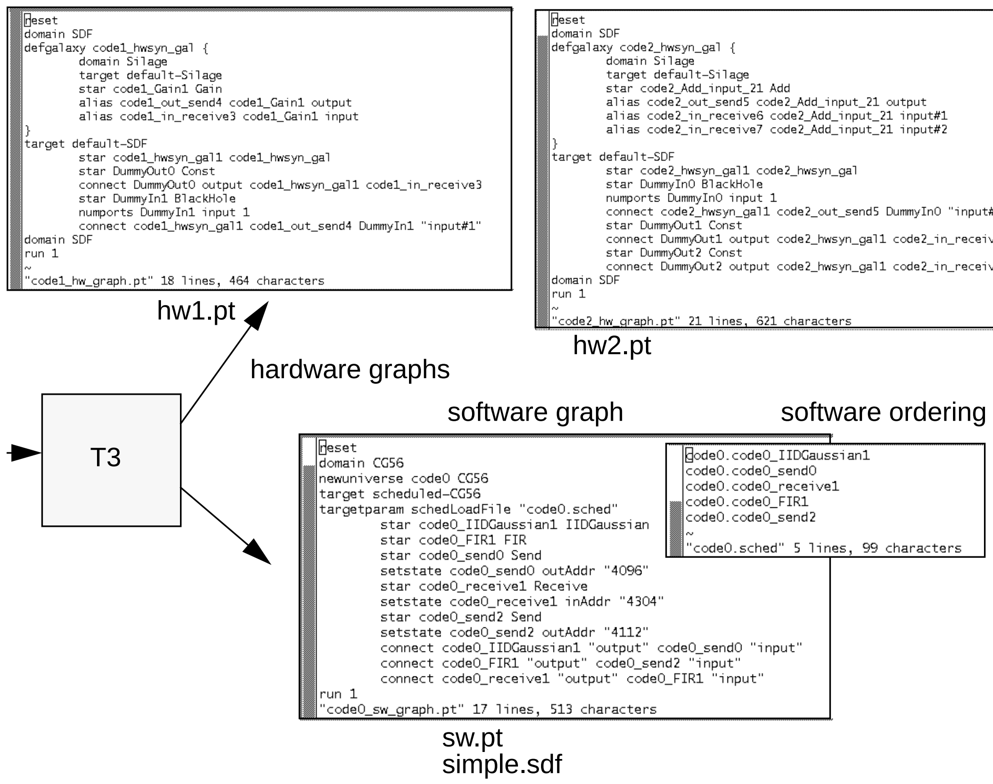

simple.sdf
IIDGaussian
Gain
Add
IIDGaussian
Gain
Add
Ramp
FIR
Ramp
FIR
(a)
(b)
Select Hardware/Software Mapping
Select stars (s); Get-star-names (N); Select implementation
■ Software
□ Hardware
□ Stimulus/Monitor
□ Display Partition (red:hw blue:sw)
□ Clear Highlights
DONE

Figure 5.11. Run-time behavior of the Design Assistant with manual partitioning.

the assignment to hardware and software, *T3* generates a global schedule[7]. Finally, *T3* generates the hardware and software graphs using techniques described in Section 4.1.2. A separate hardware graph is generated for each node mapped to hardware. A single software graph is generated for all the nodes mapped to software. The software graph includes the *Send* and *Receive* nodes. *T3* uses the global schedule to derive a schedule for the nodes mapped to software. Figure 5.12 shows the outputs generated by *T3* for the graph *simple.sdf* shown in Figure 5.11. Suppose that the nodes *IIDGaussian* and *FIR* are mapped to software, and the nodes *Gain* and *Add* are mapped to hardware. The nodes *Ramp* and *Blackhole* correspond to the stimulus and monitor nodes which are not actually synthesized. Figure 5.12 (a) and (b) show the hardware graphs for the two hardware-mapped nodes.



```
reset
domain SDF
defgalaxy code1_hwsyn_gal {
        domain Silage
        target default-Silage
        star code1_Gain1 Gain
        alias code1_out_send4 code1_Gain1 output
        alias code1_in_receive3 code1_Gain1 input
}
target default-SDF
        star code1_hwsyn_gal1 code1_hwsyn_gal
        star DummyOut0 Const
        connect DummyOut0 output code1_hwsyn_gal1 code1_in_receive3
        star DummyIn1 BlackHole
        numports DummyIn1 input 1
        connect code1_hwsyn_gal1 code1_out_send4 DummyIn1 "input#1"
domain SDF
run 1
~
"code1_hw_graph.pt" 18 lines, 464 characters
```

```
reset
domain SDF
defgalaxy code2_hwsyn_gal {
        domain Silage
        target default-Silage
        star code2_Add_input_21 Add
        alias code2_out_send5 code2_Add_input_21 output
        alias code2_in_receive6 code2_Add_input_21 input#1
        alias code2_in_receive7 code2_Add_input_21 input#2
}
target default-SDF
        star code2_hwsyn_gal1 code2_hwsyn_gal
        star DummyIn0 BlackHole
        numports DummyIn0 input 1
        connect code2_hwsyn_gal1 code2_out_send5 DummyIn0 "input#1"
        star DummyOut1 Const
        connect DummyOut1 output code2_hwsyn_gal1 code2_in_receive6
        star DummyOut2 Const
        connect DummyOut2 output code2_hwsyn_gal1 code2_in_receive7
domain SDF
run 1
~
"code2_hw_graph.pt" 21 lines, 621 characters
```

```
reset
domain CG56
newuniverse code0 CG56
target scheduled-CG56
targetparam schedLoadFile "code0.sched"
        star code0_IIDGaussian1 IIDGaussian
        star code0_FIR1 FIR
        star code0_send0 Send
        setstate code0_send0 outAddr "4096"
        star code0_receive1 Receive
        setstate code0_receive1 inAddr "4304"
        star code0_send2 Send
        setstate code0_send2 outAddr "4112"
        connect code0_IIDGaussian1 "output" code0_send0 "input"
        connect code0_FIR1 "output" code0_send2 "input"
        connect code0_receive1 "output" code0_FIR1 "input"
run 1
"code0_sw_graph.pt" 17 lines, 513 characters
```

```
code0.code0_IIDGaussian1
code0.code0_send0
code0.code0_receive1
code0.code0_FIR1
code0.code0_send2
~
"code0.sched" 5 lines, 99 characters
```

Figure 5.12. Hardware and software graphs generated by *T3*.

7. If the automated partitioning mechanism described in Chapter 3 were used, this step would not be needed since the automated partitioning algorithm generates the schedule.

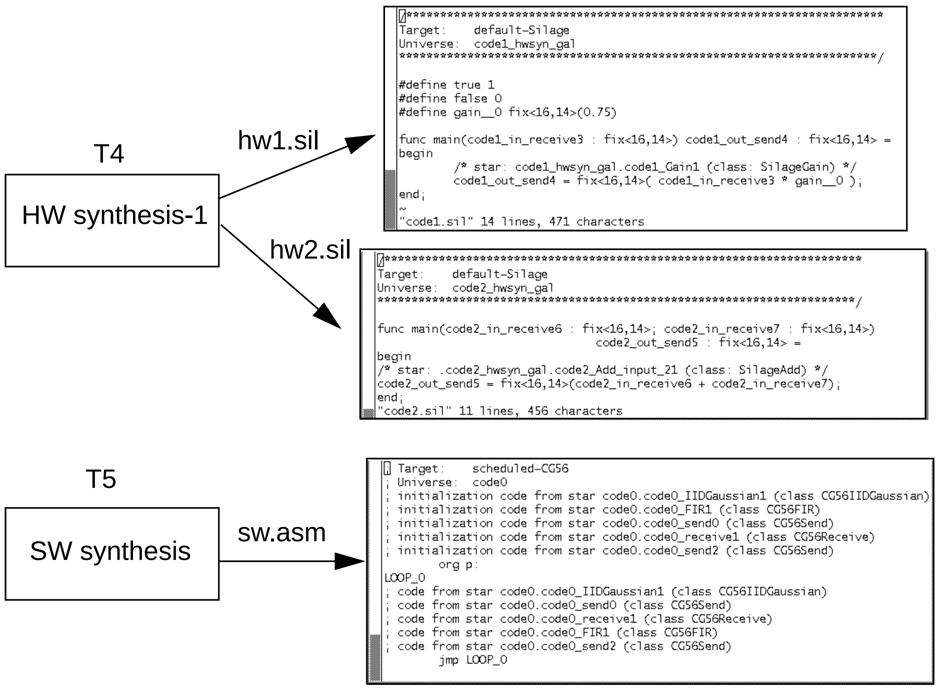
Figure 5.12-(c) and (d) show the software graph and the software ordering respectively. Note the added *Send* and *Receive* nodes in the software graph and ordering.

T4 and *T5* are the hardware and software synthesis tools. *T4* generates Silage code for each hardware graph. *T5* generates a single assembly code file corresponding to all the software-mapped nodes. It uses the schedule generated by *T3* to order the nodes in the software graph. The outputs of *T4* and *T5* are shown in Figure 5.13.

T6 uses the generated Silage code to generated the final layout for the hardware components by running Hyper.

The current implementation of the Design Assistant has some restrictions:

1. It supports a predefined target architecture consisting of a single programmable processor and multiple hardware units. The current implementation assumes a Motorola 56000 target processor and Silage-based hardware

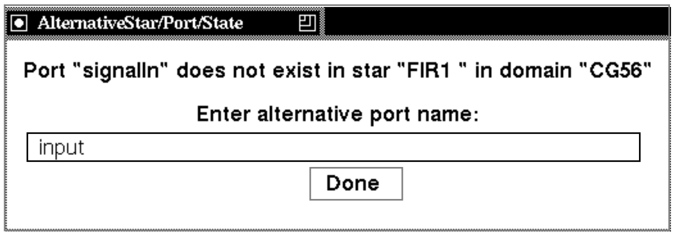


```
/************************************************************************
Target:    default-Silage
Universe:  code1_hwsyn_gal
************************************************************************/

#define true 1
#define false 0
#define gain__0 fix<16,14>(0.75)

func main(code1_in_receive3 : fix<16,14>) code1_out_send4 : fix<16,14> =
begin
        /* star: code1_hwsyn_gal.code1_Gain1 (class: SilageGain) */
        code1_out_send4 = fix<16,14>( code1_in_receive3 * gain__0 );
end;
~
"code1.sil" 14 lines, 471 characters
```

```
/************************************************************************
Target:    default-Silage
Universe:  code2_hwsyn_gal
************************************************************************/

func main(code2_in_receive6 : fix<16,14>; code2_in_receive7 : fix<16,14>)
                                code2_out_send5 : fix<16,14> =
begin
/* star: .code2_hwsyn_gal.code2_Add_input_21 (class: SilageAdd) */
code2_out_send5 = fix<16,14>(code2_in_receive6 + code2_in_receive7);
end;
"code2.sil" 11 lines, 456 characters
```

```
; Target:    scheduled-CG56
; Universe:  code0
; initialization code from star code0.code0_IIDGaussian1 (class CG56IIDGaussian)
; initialization code from star code0.code0_FIR1 (class CG56FIR)
; initialization code from star code0.code0_send0 (class CG56Send)
; initialization code from star code0.code0_receive1 (class CG56Receive)
; initialization code from star code0.code0_send2 (class CG56Send)
        org p:
LOOP_0
; code from star code0.code0_IIDGaussian1 (class CG56IIDGaussian)
; code from star code0.code0_send0 (class CG56Send)
; code from star code0.code0_receive1 (class CG56Receive)
; code from star code0.code0_FIR1 (class CG56FIR)
; code from star code0.code0_send2 (class CG56Send)
        jmp LOOP_0
```

Figure 5.13. Silage and CG56 assembly code generated by the hardware and software synthesis tools.


AlternativeStar/Port/State

Port "signalIn" does not exist in star "FIR1 " in domain "CG56"

Enter alternative port name:

input

Figure 5.14. Support for retargeting.

synthesis, although it can be easily extended to support other processors or VHDL-based synthesis. Also, as mentioned in Chapter 4, we do not consider hardware reuse.

2. *T3* operates on a directed acyclic graph. It can be extended quite easily to support the full SDF model by simply adding a tool that converts a SDF graph to a directed acyclic graph.

3. The hardware-software interface is not automatically generated, although the techniques described in Section 4.1.5 can be automated quite easily.

4. The retargeting mechanism in *T3* assumes the existence of Silage and CG56 library elements for all the nodes in the SDF graph. We have implemented a rudimentary mechanism to query the user for alternative elements if a certain element does not exist in the corresponding implementation. The alternative element should, however, have the same number (names can be different) of inputs and outputs as the SDF (technology-independent) model. Figure 5.14 shows a simple scenario where the FIR node in *simple.sdf* is mapped to software. The names of the ports in the technology-independent model (SDF) and the technology-dependent model (CG56) are different (called *signalIn* and *signalOut* in the SDF domain and *input* and *output* in the CG56 domain). The system queries for an alternative port name to look for when doing the retargeting. A similar mechanism queries

for an alternative star if one doesn’t exist for the desired implementation.

5.6 Summary

The system-level design space is quite large. As the number of tools and design possibilities increases, the design space explodes quite rapidly. Although a number of CAD systems for system-level design are now emerging [Chou94] [Kumar93][Theissinger94], most of them do not provide any support for managing the complexity of the design process; they contain point tools and leave the management aspects to the designer. We believe that managing the design process plays an equally important role in system-level design, as do the tools used for different aspects of the design. To this end, we have developed a framework that supports design methodology management. The design flow is considered a part of the design itself.

In this chapter, we have presented an infrastructure that supports efficient management of the design process, with emphasis on design flow management. This infrastructure contains powerful constructs for flow definition, dependency analysis, and automated flow execution. The infrastructure is developed as the DMM domain within Ptolemy. A tool called *DesignMaker* is responsible for automated flow management. Although *DesignMaker* derives its name from being a “*make*” utility for designs, it is much more powerful than a “graphical” *make* utility. Specification of iterations, hierarchy, and conditionals in the design flow, allowing optional inputs and outputs for tools, ensuring tool compatibility, and detecting parameter changes are some of the additional features.

We have integrated a number of point tools (such as tools for estimation, hardware-software partitioning, cosynthesis, and cosimulation) into this framework to form a complete codesign environment, which we call the Design Assistant.

6

CONCLUSION

This thesis studies a systematic approach to the system-level design of embedded signal processing systems. The key ideas presented are summarized in Section 6.1. In Section 6.2, we conclude with a discussion of some of the future directions to this research.

6.1 Contribution

In this thesis we develop techniques for the design of embedded signal processing systems. The design of such systems is challenged by stringent cost, performance, and time-to-market constraints. Pure hardware or software implementations often cannot meet these constraints; besides, some parts of these applications are inherently better-suited to either hardware or software. Hence, these applications tend to have mixed hardware and software implementations. Due to their algorithmic complexity and also to avoid early commitment to a particular hardware or software implementation, these applications are best specified at a *task* level of granularity.

Our approach to designing such systems is to *codesign* the hardware and software components. This allows the hardware and software designs to proceed 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 options. Thus, the system can be optimized in its entirety.

Four key problems are identified in the context of system-level codesign of embedded systems: partitioning, synthesis, simulation, and design methodology management. We attempt to provide solutions to all these problems. To enable meeting this ambitious objective, we have chosen to focus on a class of signal processing systems that is specified using synchronous dataflow semantics.

At the system-level, designs are typically represented as task graphs, where tasks have moderate to large granularity. Each task can be synthesized by a multiplicity of algorithms. Each algorithm selection, can, in turn, be synthesized using different transformations and resource-level options. The resulting implementations 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. The system-level designer is hence faced with the question of selecting the best design option for each task from a considerably large number of design options. We present a systematic approach to solving this problem by formulating the binary partitioning and extended partitioning problems. The key contribution is summarized in Section 6.1.1.

In the context of mixed hardware-software systems, system-level design offers even greater challenges. How do we take advantage of the inherently distinct properties of hardware and software and still tightly couple their design processes? We present approaches to cosynthesis and cosimulation that address this problem. The key ideas are summarized in Section 6.1.2.

The system-level design space is quite large. In general, the system-level design problem cannot be posed as a single well-defined optimization problem from the designer's perspective. Typically, the designer needs to explore the possible options, tools, and architectures, choosing either automated tools or manually selecting his/her choices. We believe that managing the design process plays an equally important role in system-level design, as do the tools used for different aspects of the design. To this end, we develop a framework that supports design methodology management. The key aspects of this infrastructure are summarized in Section 6.1.3.

6.1.1 Partitioning

Binary partitioning is the problem of determining, for each node in the application, a hardware or software mapping and schedule for execution. We define *extended partitioning* as the joint problem of mapping nodes in a precedence graph to hardware or software, scheduling, and selecting a particular implementation bin for each node. The end-objective in both cases is to minimize the total hardware area subject to throughput and resource constraints. Since both of these problems are NP-hard, we develop efficient heuristics.

We present the GCLP heuristic to solve the **binary partitioning problem**. This algorithm has several unique features:

1. Recognizing that binary partitioning is a constrained optimization problem, GCLP uses a *global time-criticality measure GC* to adaptively select a mapping objective at each step — if time is critical, it selects a mapping that minimizes the finish time of the node, otherwise it minimizes the resource consumption. The global time criticality measure attempts to overcome the inherent drawback of serial traversal.
2. In addition to global consideration, local preference is taken into account

in the case of a node that consumes disproportionate amounts of resources in hardware and software mappings. Such a node is classified as an extremity and is quantified by an *extremity measure*. The extremity measure is used to bias the mapping selected by *GC* alone. The motivation in identifying extremities is to avoid infeasible or poor mappings that may arise if these preferences are not taken into account.

3. We also identify several intrinsic properties (such as bit manipulation operations and memory intensive operations) that make a node suited to either hardware or software. The effect of these properties is quantified by a *repeller measure*. The repeller measure is used in a novel way to effect *on-line* swaps between nodes to reduce the hardware area. This avoids post mapping swaps.

The GCLP algorithm is computationally efficient (O$|N|^2$). For the examples tested, the solution from GCLP is reasonably close to the optimal solution. We verify with examples that quantifying the local phase of nodes (i.e., extremities and repellers) helps improve the quality of the solution.

The formulation and solution of the **extended partitioning problem** is an original contribution of this thesis. We are not aware of any other work that attempts to do this at the system level. We present the MIBS heuristic to solve the extended partitioning problem.

The philosophy of the MIBS algorithm is to extend the GCLP heuristic for extended partitioning without the associated complexity buildup. The strategy is to classify nodes in the graph as *free*, *tagged*, and *fixed*. Initially all nodes in the graph are free — their mappings and implementation bins are unknown. GCLP is applied over the set of free nodes. A tagged node is then selected from this set; its mapping is assumed to be that determined by GCLP. A *bin selection procedure* is used to compute an appropriate implementation bin for the tagged node. It uses a lookahead measure, called the *bin fraction*, which estimates, for each bin of the node, the fraction of unmapped nodes that need to move to their fastest implementations so that timing constraints are met. The bin fraction is used to compute a *bin sensitivity* measure that correlates the implementation bin with the overall hardware area reduction. The bin selection procedure selects the bin with maximum bin sensitivity. The computation of bin sensitivity is simplified by assuming that the remaining free nodes are either in their slowest or fastest implementations. The tagged node becomes a fixed node once its implementation bin is determined. GCLP is then applied over the remaining free nodes and the sequence is repeated until all nodes in the graph become fixed.

The MIBS algorithm is reasonably efficient and has a complexity of $O(|N|^3 + B \cdot |N|^2)$, where $|N|$ is the number of nodes, and B is the number of implementation options per node, per mapping. In the examples tested, the MIBS solution is reasonably close to the optimal solution. We also illustrate that implementation bins can be used effectively to *reduce the overall area* over solutions generated using binary partitioning.

6.1.2 Cosynthesis and Cosimulation

Cosynthesis is the problem of synthesizing the hardware, software, and interface components of the system, starting with a partitioned DAG. Our synthesis techniques target an architecture consisting of a single programmable processor and multiple hardware modules. The architecture is assumed to be non-pipelined and nodes mapped to hardware and software communicate using a memory-mapped, self-timed, blocking mechanism. An architecture for efficiently managing hardware-software communication is described.

Our approach to cosynthesis is to decompose the partitioned DAG into hardware, software, and interface graphs, where each node in the hardware and software graphs is a technology-dependent representation of the original node. Synthesis tools are used to generate the final implementation from these graphs. The emphasis in our work is on generating synthesizeable representations for the hardware and software components. We use pre-existing synthesis tools (such as Hyper and Ptolemy) to generate the final implementation.

Hardware-software *cosimulation* is the process of simulating the hardware and software components of a mixed hardware-software system within a *unified* environment. This includes simulation of the hardware modules, the processor, and the software that the processor executes.

We identify some of the key requirements of a cosimulation framework and show, with the help of an example, how the Ptolemy environment is effectively used for cosimulation.

6.1.3 Design Methodology Management

We present an infrastructure that supports efficient management of the design process, with emphasis on design flow management. This infrastructure contains powerful constructs for flow definition, dependency analysis, and automated flow execution. The infrastructure is developed as the DMM domain within Ptolemy. A tool called *DesignMaker* is responsible for automated flow management.

Design Assistant

The Design Assistant is a framework for unified system-level design. Figure 6.1 illustrates the Design Assistant. It contains specific tools for partitioning, synthesis, and simulation, which are configured by the user to create a design flow.

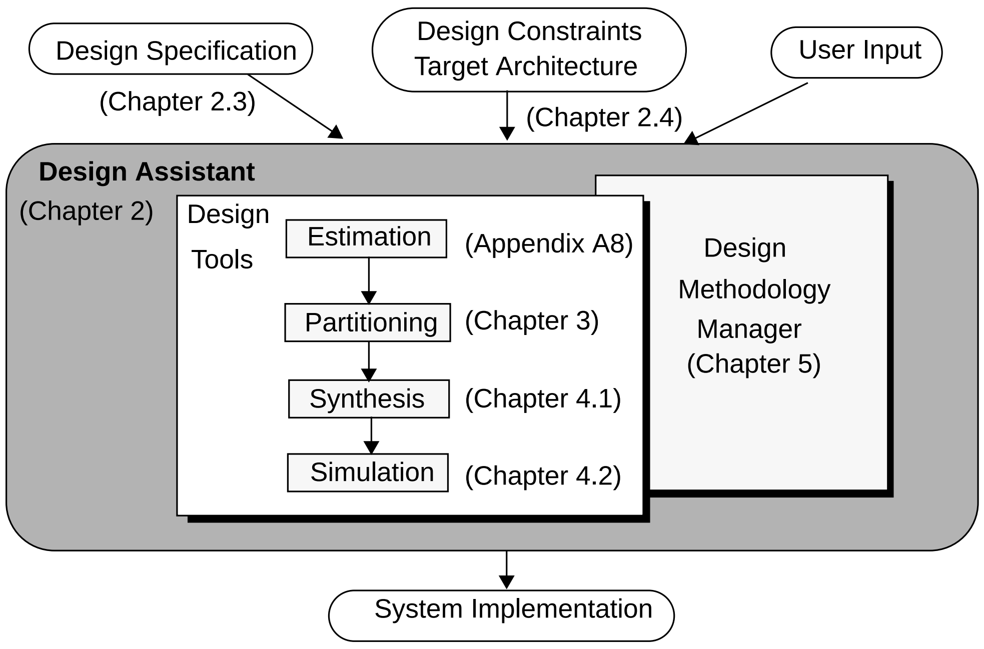


Figure 6.1. The Design Assistant

The Design Assistant operates in the DMM framework.

6.2 Future directions

6.2.1 Synthesis of Mixed Control-Dataflow Systems

The partitioning and synthesis techniques developed in this thesis apply to a class of applications that can be described using the synchronous dataflow model of computation. The SDF model is limiting in describing the dynamic behavior and control structures that are required to model typical embedded systems. The next obvious step is to extend our design techniques to a larger domain of applications. A good starting point is mixed control and dataflow systems.

Several questions need to be answered. 1. How should these systems be specified? Two opposite philosophies are emerging. One is a *unified* approach that seeks a consistent semantics for specification of the complete system (for example, boolean dataflow semantics [Buck94b], which allows specification of some control structures within a primarily dataflow model). The other is the *heterogeneous* approach that seeks to systematically combine disjoint semantics (for example, a combination of dataflow and FSM semantics [Chang95], where each component of the system is specified in a model best suited for it). 2. Once specified, how can these systems be partitioned into hardware and software? 3. What is the best approach for simulating such systems? Chang *et al*. discuss some of the issues related to cosimulation in [Chang95].

6.2.2 System-level Low Power Synthesis

Our partitioning techniques have focussed on two metrics: area and time. With the growing importance of low power systems, it is important to introduce a third dimension of power consumption in the system-level design process. Low power design techniques at the circuit and architecture level are gaining some maturity [Chandrakasan95]. Can we extend them to the system level? This would include developing techniques for mapping nodes into different hardware/software implementations with the additional goal of minimizing the power consumption. This would also involve developing new techniques for analyzing power consumption from high-level system specifications. Some research is underway in the area of high-level power estimation [Mehra94] [Tiwari94]. The related hardware/software interface also contributes to the power and needs to be carefully designed. Some insight into the issues involved in designing power-sensible CAD tools is given by Singh *et al*. [Singh95].

6.2.3 Smart Frameworks

The design space exploration process can be further systematized to develop “smart” frameworks. For instance, given a user-specified performance constraint, the design methodology management system could automatically configure a design flow. One way to achieve this is by using high-level performance estimates to determine appropriate design flows.

A starting point in this direction is the “information-based” design approach proposed by Bentz *et al*. [Bentz95]. Currently, this approach attempts to provide the user with estimates. The next step is to develop techniques to allow the system to use these estimates for configuring design flows.

REFERENCES

[Allen91]
W. Allen, D. Rosenthal, K. Fidule, “The MCC CAD Framework Methodology Management System”, *Proc. of the 28th Design Automation Conference,* San Francisco, CA, USA, June 1991, pp. 694-698.

[Ambrosio94]
J. G. D’Ambrosio, X. Hu, “Configuration-level Hardware/Software Partitioning for Real-Time Embedded Systems”, *Proc. of CODES/CASHE, Third International Workshop on Hardware/Software Codesign*, Grenoble, France, Sept. 1994, pp. 34-41.

[Baros92]
E. Baros, W. Rosential, “A Method for Hardware/Software Partitioning”, *Proc. of COMPEURO’92, IEEE Intl. Conference on Computer and Software Engineering,* The Hague, The Netherlands, May 1992, pp. 580-585.

[Becker92]
D. Becker, R. K. Singh, S. G. Tell, “An Engineering Environment for hardware/software co-simulation”, *Proc. of the 29th Design Automation Conference,* Anaheim, CA, USA, June 1992, pp. 129-134.

[Bentz95]
Ole Bentz, “Information Based Design”, Presented at the *Industrial Liasion Program (ILP) Conference,* Berkeley, CA, USA, March 8, 1995.

[Bier89]
J. C. Bier, E. A. Lee, “Frigg: A Simulation Environment for Multiple-Processor DSP System Development”, *Proc. of the International Conference on Computer Design*, Cambridge, MA, USA, Oct. 1989, pp. 280-3.

[Bier95a]
J. Bier, P. Lapsley, E. A. Lee, F. Weller, “DSP Design Tools and Methodologies,” *Technical Report,* Berkeley Design Technology, 39355 California St., Suite 206, Fremont, CA 94538, USA, 1995.

[Bier95b]
J. C. Bier, “DSP Processors and Cores: The Options Multiply”, *Integrated System Design,* June 1995, pp. 56-67.

[Bosch91]
K. O. ten Bosch, P. Bingley, P. van der Wolf, “Design Flow Management in the Nelsis CAD Framework”, *Proc. of the 28th Design Automation Conference,* San Francisco, CA, USA, June 1991, pp. 711-716.

[Brayton90]
R. K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, “Multilevel Logic Synthesis”, *Proceedings of the IEEE,* Feb. 1990, vol. 78, no. 2, pp. 264-300.

[Brockman91]
Brockman, S. W. Director, “The Hercules CAD Task Management Sys-

tem”, *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, Santa Clara, CA, USA, Nov. 1991, pp. 254-247.

[Buchenrieder92]
K. Buchenrieder, C. Veith, “CoDES: A Practical Concurrent Design Environment”, *Handouts of the 1st Intl. Workshop on Hardware/Software Codesign*, Estes Park, Colorado, USA, Sept. 1992.

[Buck94a]
J. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, “Ptolemy: a Framework for Simulating and Prototyping Heterogeneous Systems”, *International Journal of Computer Simulation*, special issue on “Simulation Software Development,” Apr. 1994, vol. 4, pp. 155-182.

[Buck94b]
J. T. Buck, “A Dynamic Dataflow Model Suitable for Efficient Mixed Hardware and Software Implementations of DSP Applications”, *Proceedings of CODES/CASHE, Third International Workshop on Hardware/Software Codesign*, Grenoble, France, Sept. 1994, pp. 165-172.

[Bushness89]
M. Bushnell, S. W. Director, “Automated Design Tool Execution in the Ulysses Design Environment”, *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, March 1989, pp. 279-287.

[Camposano87]
R. Camposano, R. K. Brayton, “Partitioning before Logic Synthesis”, *Proc. of the Intl. Conference on Computer Aided Design (ICCAD)*, Santa Clara, CA, Nov. 1987, pp. 324-326.

[Camposano91]
High-level Synthesis, Edited by: R. Camposano, W. Wolf, Dordrecht, Kluwer Academic Publishers, 1991.

[Casotto90]
Andrea Casotto, A. R. Newton, “Design Management Based on Design Traces”, *Proceedings of the 27th Design Automation Conference*, Orlando, Florida, USA, June 1990, pp. 136-141.

[CFI]
CAD Framework Initiative.

[Chandrakasan95]
A. P. Chandrakasan, R. W. Broderson, “Minimizing Power Consumption in Digital CMOS Circuits”, *Proceedings of the IEEE*, Apr. 1995, vol. 83, no. 4, pp. 498-523.

[Chang95]
W.-T. Chang, A. Kalavade, E. A. Lee, “Effective Heterogeneous Design and Cosimulation”, Proc. of NATO Advanced Study Institute Workshop on Hardware/Software Codesign, Lake Como, Italy, June 1995.

[Chaudhari94]
S. Chaudhuri, R. Walker, “Computing Lower Bounds on Functional Units before Scheduling”, *Proceedings of the 7th Intl. Symposium on High-level*

Synthesis, Niagara-on-the-lake, Ontario, Canada, May 1994, pp. 36-41.

[Chaudhary92]

K. Chaudhary, M. Pedram, “A Near Optimal Algorithm for Technology Mapping Minimizing Area under Delay Constraints”, *Proc. of 29th Design Automation Conference*, Anaheim, CA, USA, June 1992, pp. 492-498.

[Chiodo94]

M. Chiodo, P. Giusto, A. Jurecska, H. C. Hsieh, A. Sangiovanni-Vincentelli, L. Lavagno, “Hardware-Software Codesign of Embedded Systems”, *IEEE Micro*, Aug. 1994, vol.14, no.4, pp. 26-36.

[Chiuch90]

T. Chiuch, R. Katz, “A History Model for Managing the VLSI Design Process”, *Proceedings of the International Conference on Computer Aided Design (ICCAD),* Santa Clara, CA, Nov. 1990, pp. 358-361.

[Chou92]

P. Chou, R. Ortega, G. Borriello, “Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems”, *Proc. 1992 IEEE/ACM International Conference on Computer-Aided Design*, Santa Clara, CA, USA, Nov. 1992, pp. 488-95.

[Chou94]

P.Chou, E. A. Walkup, G. Borriello, “Scheduling for Reactive Real-Time Systems”, *IEEE Micro*, Aug. 1994, pp. 37-47.

[Coelho94]

C.N. Coelho, C.-Y.J. Yang, V. Mooney, G. De. Micheli, “Redesigning Hardware-Software Systems”, *Proceedings of CODES/CASHE, Third International Workshop on Hardware/Software Codesign*, Grenoble, France, Sept. 1994, pp. 116-123.

[CPLEX]

CPLEX Optimization, Inc., “Using the CPLEX Callable Library”, CPLEX Optimization, Inc., Suite 279, 930 Tahoe Blvd., Bldg. 802, Incline Village, NV 89451-9436.

[DeMicheli86]

Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation, edited by G. De Micheli, A. Sangiovanni-Vincentelly, P. Antognetti, NATO Advanced Study Institute Series on Logic Synthesis and Silicon Compilation for VLSI Design, Series E, Alllied Sciences, v.136, 1986.

[DeMicheli94]

G. De Micheli, “Synthesis and Optimization of Digital Circuits”, New York, McGraw-Hill, 1994.

[Dewey89]

Allen Dewey, S. W. Director, “Yoda: A Framework for the Conceptual Design of VLSI Systems”, *Proceedings of the International Conference on Computer Aided Design (ICCAD),* Santa Clara, CA, USA, Nov. 1989, pp 380-383.

[Eles94]

P. Eles, Z. Peng, A. Daboli, "VHDL System-Level Specification and Partitioning in a Hardware/Software Co-Synthesis Environment", *Proceedings of CODES/CASHE, Third International Workshop on Hardware/Software Codesign*, Grenoble, France, Sept. 1994, pp. 49-55.

[Feldman79]

S. Feldman, "Make — A Program for Maintaining Computer Programs", *Software Practice and Experience*, 1979, Vol. 9, pp. 255-265.

[Franke91]

D. W. Franke, Martin K. Purvis, "Hardware/Software Codesign: A Perspective", *Proc. of 13th Intl. Conference on Software Engineering*, Austin, Texas, USA, May 1991, pp. 344-352.

[Garey79]

M. Garey, D. Johnson, "Computers and Intractability", W. H. Freeman and Company, New York, 1979.

[Gong94]

J. Gong, D. D. Gajski, S. Narayan, "Software Estimation from Executable Specifications", *Journal of Computer and Software Engineering*, 1994, vol.2, no.3, pp. 239-58.

[Guerra94]

L. Guerra, M. Potkonjak, J. Rabaey, "System-level Design Guidance Using Algorithm Properties", *VLSI Signal Processing VII*, Oct. 1994, Edited by Jan Rabaey, Paul M. Chau, John Eldon, IEEE, New York, pp. ??

[Gupta92]

R. Gupta, G. DeMicheli, "System-level Synthesis Using Re-programmable Components", *Proceedings of the European Conference on Design Automation*, Brussels, Belgium, Feb.1992, pp. 2-7.

[Gupta93]

R. K. Gupta, "Co-synthesis of Hardware and Software for Digital Embedded Systems", Ph. D. Dissertation, Stanford University, Dec. 1993.

[Hamada92]

T. Hamada, S. Banerjee, P. M. Chau, R. D. Fellman, "Macropipelining Based Heterogeneous Multiprocessor Scheduling", *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, San Francisco, CA, USA, March 1992, vol 5, pp. 597-600.

[Harrison86]

D. Harrison, P. Moore, R. Spickelmier, A. R. Newton, "Data Management and Graphics Editing in the Berkeley Design Environment", *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, Santa Clara, CA, USA, Nov. 1986, pp. 24-27.

[Henkel94]

J. Henkel, T. Benner, R. Ernst, W. Ye, *et al.*, "COSYMA: A Software-oriented Approach to Hardware/Software Codesign", *Journal of Computer and Software Engineering*, 1994, vol. 2, no. 3, pp. 293-314.

[Herrmann94]
J. Herrmann, Henkel, R. Ernst, “An Approach to the Adaptation of Estimated Cost Parameters in the COSYMA System”, *Proceedings of CODES/CASHE, Third International Workshop on Hardware/Software Codesign*, Grenoble, France, Sept. 1994, pp. 100-107.

[Hilfinger85]
P. Hilfinger, “A High-level Language and Silicon Compiler for Digital Signal Processing”, *Proc. of IEEE 1985 Custom Integrated Circuits Conference*, Portland, OR, USA, May 1985, pp. 213-216.

[Hu61]
T. C. Hu, “Parallel Sequencing and Assembly Line Problems”, *Operations Research* 9(6), Nov. 1961, pp. 841-848.

[Ishikawa91]
M. Ishikawa, G. De Micheli, “A Module Selection Algorithm for High-level Synthesis”, *Proc. of 1991 IEEE International Symposium on Circuits and Systems*, Singapore, June 1991, vol.3, pp. 1777-80.

[Ismail94]
T. B. Ismail, K. O’Brien, A. Jerraya, “Interactive System-level Partitioning with PARTIF”, *Proceedings of EDAC94*, Paris, France, Feb. 1994, pp. 464-468.

[Jantsch94]
A. Jantsch, P. Ellervee, J. Oberg, A. Hemani, H. Tenhunen, “Hardware/ Software Partitioning and Minimizing Memory Interface”, *Proceedings of EuroDAC’94*, Grenoble, France, Sept. 1994, pp. 226-31.

[Johnsonbaugh91]
R. Johnsonbaugh, M. Kalin, “A Graph Generation Software Package”, *Proc. of 22nd SIGCSE Technical Symposium on Computer Science Education*, San Antonio, TX, USA, March 1991, vol. 23, no 1, pp. 151-154.

[Kalavade92]
A. Kalavade, E. A. Lee, “Hardware/Software Codesign Using Ptolemy — A Case Study”, M. S. Report, Dec. 1991, Dept. of EECS, Univ. of California, Berkeley, CA, 94720.

[Kalavade93]
A. Kalavade, E. A. Lee, “A Hardware/Software Codesign Methodology for DSP applications”, *IEEE Design and Test of Computers*, Sept. 1993, pp. 16-28.

[Kalavade94a]
A. Kalavade, E. A. Lee, “Manifestations of Heterogeneity in Hardware/ Software Codesign”, *Proc. of the 31st Design Automation Conference*, San Diego, CA, USA, June 1994, pp. 437-438.

[Kalavade94b]
A. Kalavade, E. A. Lee, “A Global Criticality/ Local Phase Driven Algorithm for the Constrained Hardware/Software Partitioning Problem”, *Proc. of CODES/CASHE, Third International Workshop on Hardware/Software*

Codesign, Grenoble, France, Sept. 1994, pp. 42-48.

[Kalavade95a]
A. Kalavade, Jose Pino, E. A. Lee, “Managing Complexity in Heterogeneous System Specification, Simulation, and Synthesis”, *Proc. of International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Detroit, Michigan, USA, May 1995, vol. 5, pp. 2833-2836.

[Kalavade95b]
A. Kalavade, E. A. Lee, “The Extended Partitioning Problem: Hardware/ Software Mapping and Implementation-bin Selection”, *Proc. of Sixth International Workshop on Rapid System Prototyping*, Chapel Hill, North Carolina, USA, June, 1995, pp. 12-18.

[Keutzer94]
K. Keutzer, “Hardware-Software Co-Design and ESDA”, Proc. of 31st Design Automation Conference, San Diego, CA, USA, June 1994, pp. 435-6.

[Kleinfelft94]
S. Kleinfelft, M. Guiney, J. K. Miller, and M. Barnes, “Design Methodology Management”, *Proc. of the IEEE,* vol. 82, no. 2, Feb. 1994, pp. 231-250.

[Knapp86]
D. W. Knapp, A. Parker, “A Design Utility Manager: the ADAM Planning Engine”, *Proceedings of the 23rd Design Automation Conference*, Las Vegas, Nevada, USA, June 1986, pp. 48-54.

[Kumar93]
S. Kumar, J. H. Aylor, B. W. Johnson, W. A. Wulf, “A Framework for Hardware/Software Codesign”, *Computer*, Dec. 1993, vol. 26, no. 12, pp. 39-45.

[Kurdahi89]
F. J. Kurdahi, A. C. Parker, “Techniques for area estimation of VLSI Circuits”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Jan 1989, vol. 8, no. 1, pp. 81-92.

[Lagnese91]
E. D. Lagnese, D. E. Thomas, “Architectural Partitioning for System-level Synthesis of ICs”, *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, July 1991, vol. 10, no. 7, pp. 847-860.

[Laneer91]
D. Laneer, *et al.*, “Architectural Synthesis for Medium and High Throughput Signal Processing with the New CATHEDRAL Environment”, *High-level VLSI Synthesis*, edited by R. Camposano, W. Wolf, W. Dordrecht, Kluwer Academic Publishers, 1991, pp 27-54. ?*

[Lee87]
E. A. Lee, D. G. Messerschmitt, “Synchronous Data Flow”, *Proceedings of the IEEE,* September 1987, vol. 75, no. 9, pp. 1235-1245.

[Lee90]

E. A. Lee, J. C. Bier, “Architectures for Statically Scheduled Dataflow”, *Journal of Parallel and Distributed Computing*, Dec. 1990, pp. 333-348.

[McFarland89]

M. C. McFarland, “A Fast Floorplanning Algorithm for Architectural Evaluation”, *Proceedings of International Conference on Computer Design*, Santa Clara, CA, USA, Oct. 1989, pp. 96-99.

[McFarland90a]

M. C. McFarland, A. C. Parker, R. Camposano, “The High-level Synthesis of Digital Systems.”, *Proceedings of the IEEE*, Feb. 1990, vol.78, no. 2, pp. 301-18.

[McFarland90b]

M. C. McFarland, T. J. Kowalski, “Incorporating Bottom-up Design into Hardware Synthesis”, *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems,* Sept. 1990, vol. 9, no. 9, pp. 938-950.

[Mehra94]

R. Mehra, J. Rabaey, “Behavioral Level Power Estimation and Exploration”*, Proc. of International workshop on Low Power Design*, Napa Valley, CA, USA, April 1994, pp. 197-202

[MentorGraphics1]

Mentor Graphics Tools, Mentor Graphics Corp., 1001 Ridder Park Drive, San Jose, CA 95131-2314.

[MentorGraphics2]

Falcon Framework Reference Manual, Mentor Graphics Corp., 1001 Ridder Park Drive, San Jose, CA 95131-2314.

[Motorola]

DSP56000/DSP56001 Digital Signal Processor, User’s Manual, Motorola Inc., 1990.

[Murthy94]

P. K. Murthy, S. S. Bhattacharyya, E. A. Lee, “Minimizing Memory Requirements for Chain-structured Synchronous Dataflow Programs”, *Proc. of International Conference on Acoustics, Speech, and Signal Processing (ICASSP 94)*, Adelaide, Australia, April 1994, vol. 2, pp. 453-456.

[Papadamitriou82]

C. H. Papadamitriou, K. Steiglitz, “Combinatorial Optimization: Algorithms and Complexity”, Prentice-Hall, Inc. 1982.

[Paulin89]

P. G. Paulin, J. P. Knight, “Force-Directed Scheduling for the Behavioral Synthesis of ASICs”, *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems,* June 1989, vol. 8, no. 6, pp. 661-679.

[Paulin95]

P. G. Paulin, *et al.*, “DSP Design Tool Requirements for Embedded Systems: A Telecommunications Industrial Perspective”, *Journal of VLSI Signal Processing*, Jan. 1995, vol. 9, no. 1-2, pp. 23-47.

[Pino95a]

J. L. Pino, S. Ha, E. A. Lee, J. T. Buck, “Software Synthesis for DSP using Ptolemy”, *Journal of VLSI Signal Processing*, Jan. 1995, vol. 9, no. 1-2, pp. 7-21.

[Pino95b]

J. L. Pino, E. A. Lee, “Hierarchical Static Scheduling of Dataflow Graphs onto Multiple Processors”, *Proc. of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 95)*, Detroit, MI, USA, May 1995, vol. 4, pp 2643-2646.

[Potkonjak94]

M. Potkonjak, J. Rabaey, “Optimizing Resource Utilization using Transformations”, *IEEE Transactions of Computer-Aided Design of Integrated Circuits and Systems*, March 1994, vol. 13, no. 3, pp. 277-292.

[Press88]

W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, “Numerical Recipes in C: The Art of Scientific Computing”, Cambridge University Press, 1988.

[Ptolemy95]

The Almagest: Ptolemy User’s and Programmer’s Manual. (http://ptolemy.eecs.berkeley.edu/Almagest.html)

[Quantify]

Quantify manual pages, Pure Software Inc.

[Rabaey91]

J. M. Rabaey, C. Chu, P. Hoang, M. Potkonjak, “Fast Prototyping of datapath-intensive Architectures”, *IEEE Design and Test of Computers*, pp. 40-51, June 1991.

[Rabaey94]

J. Rabaey, M. Potkonjak, “Estimating Implementation Bounds for Real Time DSP Application Specific Circuits”, *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 13, no. 6, June 1994, pp. 669-683.

[Rabaey95]

J. Rabaey, L. Guerra, R. Mehra, “Design Guidance in the Power Dimension”, *Proc. of International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Detroit, Michigan, USA, May 1995, vol. 5, pp. 2836-2839.

[Rasure91]

J. R. Rasure, C. S. Williams, “An Integrated Data Flow Visual Language and Software Development Environment”, *Journal of Visual Languages and Computing*, Sept. 1991, vol. 2, no. 3, pp 217-246.

[Ritz93]

S. Ritz, M. Pankert, V. Zivojinovic, H. Meyr, “High-level Software Synthesis for the Design of Communication Systems.”, *IEEE Journal on Selected Areas in Communications*, April 1993, vol.11, no.3, pp. 348-58.

[Rowson94]

J A. Rowson, “Hardware/Software Co-Simulation”, *Proceedings of the 31st Design Automation Conference*, San Diego, CA, USA, June 1994, pp. 439-440.

[Sharma93]

A. Sharma, R. Jain, “Estimating Architectural Resources and Performance for High-level Synthesis Applications”, *IEEE Transactions on VLSI Systems*, June 1993, vol. 1, no. 2, pp. 175-190.

[Shanmugan94]

K. Sam Shanmugan, “Simulation and Implementation Tools for Signal Processing and Communication Systems”, *IEEE Communications Magazine*, July 1994, pp. 36-40.

[Sih91]

G. C. Sih, “Multiprocessor Scheduling to Account for Interprocessor Communication”, Ph. D. Thesis, Electronics Research Laboratory, University of California, Berkeley, UCB/ERL M91/29, April 1991.

[Sih93]

G. Sih, E. A. Lee, “A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures”, *IEEE Transactions on Parallel and Distributed Systems*, Feb. 1993, vol. 4, no. 2, pp. 175-187.

[Singh95]

D. Singh, J. M. Rabaey, M. Pedram, *et al.*, “Power Conscious CAD Tools and Methodologies: A Perspective”, *Proceedings of the IEEE*, Apr. 1995, vol. 83, no. 4, pp. 570-94.

[Sriram93]

S. Sriram, E. A. Lee, “Design and Implementation of an Ordered Memory Access Architecture”, *Proc. of IEEE Intl. Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Minneapolis, MN, April 1993, vol 1., pp. 345-348.

[Srivastava91]

M. B. Srivastava, R. W. Broderson, “Rapid-Prototyping of Hardware and Software in a Unified Framework”, *Proc. IEEE Conference on Computer Aided Design (ICCAD)*, Santa Clara, CA, Nov. 1991, pp. 152-155.

[Sun92]

J. S. Sun, R. W. Brodersen, “Design of System Interface Modules”, *Proc. of Intl. Conference on Computer Aided Design (ICCAD)*, Santa Clara, CA, USA, Nov. 1992, pp 478-481.

[Synopsys]

Synopsys Tools, Synopsys Inc., 700 East Middlefield Rd., Mountain View, CA 94043.

[Theissinger94]

M. Theissinger, P. Stravers, H. Veit, “Castle: An Interactive Environment for HW-SW Co-Design”,*Proceedings of CODES/CASHE, Third Interna-*

tional Workshop on Hardware/Software Codesign, Grenoble, France, Sept. 1994, pp. 203-9.

[Thomas93]
D. E. Thomas, J. K. Adams, H. Schmit, “A Model and Methodology for Hardware/Software Codesign”, *IEEE Design and Test of Computers*, Sept. 1993, pp. 6-15.

[Thor86]
“*Thor Tutorial*”, VLSI/CAD Group, Stanford University, 1986.

[Tiwari94]
V. Tiwari, S. Malik, A. Wolfe, “Power Analysis of Embedded Software: A First Step Towards Software Power Minimization”, IEEE Transactions on VLSI Systems, Vol. 2, no. 4, Dec. 1994, pp 437-445.

[Vahid92]
F. Vahid, D. Gajski, “Specification Partitioning for System Design”, *Proceedings of the 29th Design Automation Conference*, June 1992, Anaheim, CA, USA, pp. 219-224.

[VanDerWolf93]
Pieter van der Wolf, “Architecture of an Open and Efficient CAD Framework”, Ph.D. Thesis, Delft University of Technology, May 1993.

[Verbauwhede94]
I. Verbauwhede, C. Sheers, J. Rabaey, “Specification and Support for Multi-dimensional DSP in the SILAGE Language”, *Proc. of the Intl. Conference on Acoustics, Speech, and Signal Processing (ICASSP-94)*, Adelaide, Australia, April 1994, pp. ??

[Wolf94]
W. H. Wolf, “Hardware-software Co-design of Embedded Systems”, *Proceedings of the IEEE*, July 1994, vol.82, no.7, pp. 965-89.

[Zimmerman88]
G. Zimmerman, “A New Area and Shape Function Estimation Technique for VLSI Layouts”, *Proceedings 25th Design Automation Conference*, June 1988, pp. 60-65.

7

APPENDIX

A1 ILP Formulation for Problem *P1*

The hardware/software mapping and scheduling problem *P1* is formally described by the following integer linear program:

Given:

1. A set of nodes $N = \{1, 2, \ldots, \}$, representing the tasks in the algorithm.

2. Associated with each node i in N, are four non-negative integers: ah_i, as_i, th_i, and ts_i, representing the area and execution time corresponding to the in hardware and software mappings of node i.

3. A directed acyclic graph $G = (N,A)$. A is a set of arcs (i,j), where $i, j \in N$. An arc (i,j) in A implies that task i must complete its execution before task j can start its own execution.

4. Associated with each arc $(i,j) \in A$ is a non-negative number p_{ij}, which represents the number of samples sent from node i to node j.

5. The source and sink nodes in the graph G are identified as S and K respectively.

6. Six non-negative integer-valued parameters AH, AS, T, ah_{comm}, as_{comm}, and t_{comm} are specified. AH and AS are the capacities of the hardware and the software resource respectively. The total area of all nodes mapped to hardware (software) cannot exceed AH (AS). D is the deadline constraint, i.e., the time between the start time of the source S and the finish time of the sink K should be at most D. Parameters ah_{comm}, as_{comm}, and t_{comm} represent the hardware area required, the software area required, and the delay incurred when one sample of data is sent between two nodes mapped to different resources respectively.

Variables:

1. For each $i \in N$, define binary variables h_i and s_i such that $h_i = 1$ if task i is mapped to hardware, 0 otherwise; and $s_i = 1$ if task i is mapped to software, 0 otherwise. Obviously, $h_i + s_i = 1$. We use the two variables to make the formulation easier to read; only h_i contributes to the variable count. (N variables)

2. For each $i \in N$, define a non-negative integer variable t_i, the starting time of task i. (N variables)

3. For each $(i,j) \in A$, define a binary variable b_{ij} such that $b_{ij} = 1$ if both i and j are in hardware, 0 otherwise. (A variables)

4. For each $(i,j) \in A$, define a binary variable c_{ij}, such that $c_{ij} = 1$ if nodes i and j are on different mappings, and 0 if on the same mapping[1].

5. For each $i, j \in N$, define a binary variable d_{ij} such that $d_{ij} = 1$ if $t_i \le t_j$, otherwise $d_{ij} = 0$. (N(N-1) variables)

1. c_{ij} will be defined in terms of b_{ij} and hence it is not included in the variable count.

<u>Objective function</u> F: minimize hardware area

$$\text{minimize}\left(\sum_{i \in N} h_i \times ah_i + \sum_{(i,j) \in A} c_{ij} \times p_{ij} \times ah_{comm}\right)$$

<u>Subject to</u>:

1. Hardware capacity constraint:

$$\sum_{i \in N} h_i \times ah_i + \sum_{(i,j) \in A} c_{ij} \times p_{ij} \times ah_{comm} \le AH \text{ (1 constraint)}$$

2. Software capacity constraint:

$$\sum_{i \in N} s_i \times as_i + \sum_{(i,j) \in A} c_{ij} \times p_{ij} \times as_{comm} \le AS \text{ (1 constraint)}$$

3. Deadline constraint:

$$t_K + (h_K \times th_K) + (s_K \times ts_K) \le D \text{ (1 constraint)}$$

4. Precedence constraints:

Regardless of the mappings, if $(i,j) \in A$, then j starts executing at the end of i.

$t_i + (h_i \times th_i) + (s_i \times ts_i) + (c_{ij} \times p_{ij} \times t_{comm}) \le t_j$,

where $(i,j) \in A$ (A constraints)

$d_{ij} = 1$ for $(i,j) \in A$ (A constraints)

5. Processor overlap constraint:

No two jobs can be executed simultaneously on the software processor. We want to express the fact that if i and j are both on the software processor, and j executes before i (i.e., $(j, i) \in A$), then t_j + execution time of j $\le t_i$. The variable d_{ij} is 1 if $t_i \le t_j$, and 0 otherwise. Define $M = \sum_i ts_i$. The meaning of d_{ij} is expressed by the following constraints:

$(d_{ij} + d_{ji}) = 1$ for $i < j$, $i, j = 1, \ldots,$ N. (N(N-1)/2 constraints)

$t_j - t_i - (M \cdot d_{ij}) \le 0$ for $i \ne j$, $i, j = 1, \ldots,$ N. (N(N-1) constraints)

Since $t_j - t_i < M$ for an optimal solution, the expression is restrictive only for d_{ij} = 0. If d_{ij} = 0, then j starts no later than i. The overlap constraint can now be specified as:

$$t_j + (h_j \times th_j) + (s_j \times ts_j) + \sum_k c_{jk} \times p_{jk} \times t_{comm} - t_i \le M \cdot (3 - d_{ji} - s_i - s_j)$$

for $i \ne j$, i,j = 1, …, N, k such that $(j,k) \in A$ (N(N-1) constraints)

The expression is restrictive only for d_{ji} = 1 and $s_i = s_j$ = 1, i.e., i and j are both on software, and j starts no later than i. In this case it guarantees that t_j + execution time of i $\le t_i$, as it should be.

6. Additional constraints: To account for costs incurred when two communicating nodes i and j are on different mappings, we use c_{ij}, where c_{ij} is expressed as:

$$c_{ij} = 1 - (h_i \times h_j) - ((1 - h_i) \times (1 - h_j))$$, where $(i,j) \in A$

$$c_{ij} = h_i + h_j - 2h_ih_j$$

Note that this is a quadratic constraint in h_i, and it can be linearized by defining a binary variable b_{ij} for all (i,j) in A such that $b_{ij} = h_ih_j$, where b_{ij} can be expressed by:

$$b_{ij} \le h_i$$

$$b_{ij} \le h_j$$

$$b_{ij} \ge (h_i + h_j - 1)$$

The original expression for c_{ij} can now be replaced by:

$$c_{ij} = h_i + h_j - 2b_{ij}$$

Note that only b_{ij} adds to the variable count (3A constraints); c_{ij} is used only to enhance readability; in the actual formulation c_{ij} is replaced by $(h_i + h_j - 2b_{ij})$. (3A constraints)

A solution to the mapping and scheduling problem (*P1*) is specified completely by h_i (mapping) and t_i (schedule). The formulation has ($N^2 + N + A$) variables and $(\frac{5}{2} \cdot N^2 - \frac{5}{2} \cdot N + 5A + 3)$ constraints, in addition to the integrality constraints on the variables.

A2 Proof of NP-Completeness for Problem *P1*

Let $\overline{P1}$ be the decision (or recognition) version of *P1*, i.e., $\overline{P1}$ is the problem of finding a *feasible* mapping and schedule. We will first show that $\overline{P1}$ is NP-complete. Since *P1* is at least as hard as $\overline{P1}$, it follows that *P1* is NP-hard.

Claim: $\overline{P1}$ is NP-complete.

Proof:

1. $\overline{P1}$ is in NP

   A given solution (schedule and mapping) can be verified in polynomial time.

2. $\overline{P1}$ can be restricted to the Precedence Constrained Scheduling problem (*PCS*)

   The precedence constrained scheduling problem [Garey79] is as follows.

   Instance: Set *Y* of tasks, each having length $l(y) = 1$, *m* processors, a partial order < on *Y*, and a deadline *D*.

   Question: Is there an *m*-processor schedule *s* for *Y* that meets the overall deadline *D* and obeys the precedence constraints, i.e., such that $y1 < y2$ implies $s(y2) \geq s(y1) + l(y1) = s(y1) + 1$?

   $\overline{P1}$ can be restricted to *PCS*, by setting the execution times in hardware and software to be equal (i.e., the tasks take the same execution time on all processors — hardware and software in this case).

   The precedence constrained scheduling problem is NP-complete

([Garey79], pg. 239) for two processors and task lengths not all equal to 1[2]. Thus, a known NP-complete problem is a special case of $\overline{P1}$, and hence, by restriction ([Garey79], pg 63), $\overline{P1}$ is NP-complete.

Obviously, *P1*, the optimization version of $\overline{P1}$ (find a feasible mapping and schedule such that the hardware area is minimum), is at least as hard as the decision version[3]. Thus, *P1* is NP-hard.

A3 ILP formulation for Problem *P2*

The extended partitioning problem can be formulated as an integer linear problem. The formulation is similar to that of the binary partitioning problem.

Given:

1. A set of nodes $N = \{1, 2, \ldots,\}$, representing the tasks in the algorithm.
2. Associated with each node $i \in N$, are non-negative integers, ah_{ij}, th_{ij}, where $j = 1, \ldots, |NH_i|$, and as_{ij}, ts_{ij}, where $j = 1, \ldots, |NS_i|$, representing the hardware and software implementation curves.
3. A directed acyclic graph $G = (N,A)$. A is a set of arcs (i,j), where $i, j \in N$. An arc (i,j) in A implies that task i must complete its execution before task j can start its own execution.
4. Associated with each arc $(i,j) \in A$ is a non-negative number p_{ij}, which represents the number of samples sent from node i to node j.
5. The source and sink nodes in the graph G are identified as S and K respectively.

2. For the class of heterogeneous task-level graphs that we are interested in, the execution times are obviously greater than 1.
3. *P1* is not in NP, since a given solution for *P1* cannot be verified in polynomial time to be the minimum solution.

6. Six non-negative integer valued parameters *AH, AS, D,* ah_{comm}, as_{comm}, and t_{comm} are specified. *AH* and *AS* are the capacities of the hardware and the software resource respectively. The total area of all nodes mapped to hardware (software) cannot exceed *AH* (*AS*). *D* is the deadline constraint, i.e., the time between the start time of the source *S* and the finish time of the sink *K* should be at most *D*. Parameters ah_{comm}, as_{comm}, and t_{comm} represent the hardware area required, the software area required, and the delay incurred when one sample of data is sent between two nodes mapped to different resources.

Variables:

1. For each *i* in *N*, define binary variables h_i and s_i such that $h_i = 1$ if task *i* is mapped to hardware, 0 otherwise; and $s_i = 1$ if task *i* is mapped to software, 0 otherwise. Obviously, $h_i + s_i = 1$. We use the two variables to make the formulation easier to read; only h_i contributes to the variable count. (N variables)

2. In addition to the variables indicating the mapping, additional variables for selecting the optimal implementation bin are needed. Define a set of binary variables bh_{ij} (and bs_{ij}), where $i \in N$, and $j \in NH_i$ (and $j \in NS_i$). In particular, $bh_{ij} = 1$, if node *i* is mapped to hardware implemented bin *j*. In general, let *B* be the number of implementation bins per node per mapping. (*2BN* variables)

3. For each $i \in N$, define a non-negative integer variable t_i, the starting time of task *i*. (N variables)

4. For each $(i,j) \in A$, define a binary variable b_{ij} such that $b_{ij} = 1$ if both *i* and *j* are in hardware, 0 otherwise. (A variables)

5. For each $(i,j) \in A$, define a binary variable c_{ij}, such that $c_{ij} = 1$ if nodes i and j are on different mappings, and 0 if on the same mapping. c_{ij} will be defined in terms of b_{ij} and is not included in the variable count.

6. For each $i, j \in N$, define a binary variable d_{ij} such that $d_{ij} = 1$ if $t_i \le t_j$, otherwise $d_{ij} = 0$.

   (N(N-1) variables)

   <u>Objective function</u> F: minimize hardware area.

   $$\text{minimize}\left(\sum_{i \in N} \sum_{j \in NH_i} bh_{ij} \times ah_{ij} + \sum_{(i,j) \in A} c_{ij} \times p_{ij} \times ah_{comm}\right)$$

   <u>Subject to:</u>

1. To ensure that only one implementation bin is selected for every node:

   $$\sum_{j \in NH_i} bh_{ij} + \sum_{j \in NS_i} bs_{ij} = 1$$. (N constraints)

   Define $h_i = \sum_{j \in NH_i} bh_{ij}$, $s_i = \sum_{j \in NS_i} bs_{ij}$. where h_i (s_i) is 1 if i is in hardware (software).

2. Hardware capacity constraint:

   $$\sum_{i \in N} \sum_{j \in NH_i} bh_{ij} \times ah_{ij} + \sum_{(i,j) \in A} c_{ij} \times p_{ij} \times ah_{comm} \le AH$$

   (1 constraint)

3. Software capacity constraint:

   $$\sum_{i \in N} \sum_{j \in NS_i} bs_{ij} \times as_{ij} + \sum_{(i,j) \in A} c_{ij} \times p_{ij} \times as_{comm} \le AS$$

   (1 constraint)

4. Deadline constraint:

$$t_K + \sum_{j \in NH_K} (bh_{Kj} \times th_{Kj}) + \sum_{j \in NH_K} (bs_{Kj} \times ts_{Kj}) \le D$$

(1 constraint)

5. Precedence constraints:

$$t_i + \sum_{k \in NH_i} (bh_{ik} \times th_{ik}) + \sum_{k \in NH_i} (bs_{ik} \times ts_{ik}) + (c_{ij} \times p_{ij} \times t_{comm}) \le t_j$$

where $(i,j) \in A$ (A constraints)

$d_{ij} = 1$ for $(i,j) \in A$ (A constraints)

6. Processor overlap constraint:

No two jobs can be executed simultaneously on the software processor. We want to express the fact that if *i* and *j* are both on the software processor, and *j* executes before *i* (i.e., $(j, i) \in A$), then t_j + execution time of j $\le t_i$.

The variable d_{ij} is 1 if $t_i \le t_j$, and 0 otherwise. Define $M = \sum_i ts_i$. The meaning of d_{ij} is expressed by the following constraints:

$(d_{ij} + d_{ji}) = 1$ for $i < j$, $i, j = 1, \ldots, N$. (N(N-1)/2 constraints)

$t_j - t_i - (M \cdot d_{ij}) \le 0$ for $i \ne j$, $i, j = 1, \ldots, N$ (N(N-1) constraints)

Since $t_j - t_i < M$ for an optimal solution, the expression is restrictive only for $d_{ij} = 0$. If $d_{ij} = 0$, then j starts no later than i. The overlap constraint can now be specified as:

$$t_j + time_h + time_s + \sum_k c_{jk} \times p_{jk} \times t_{comm} - t_i \le M \cdot (3 - d_{ji} - s_i - s_j),$$

where, $time_h = \sum_{k \in NH_j} (bh_{jk} \times th_{jk})$ $time_s = \sum_{k \in NH_j} (bs_{jk} \times ts_{jk})$,

$s_i = \sum_{k \in NS_i} bs_{ik}$, and $s_j = \sum_{k \in NS_j} bs_{jk}$.

for $i \neq j$, $i,j = 1, \ldots, N$, k such that $(j,k) \in A$ (N(N-1) constraints)

The expression is restrictive only for $d_{ji} = 1$ and $s_i = s_j = 1$, i.e., *i* and *j* are both on software, and *j* starts no later than *i*. In this case it guarantees that t_j + execution time of i $\leq t_i$, as it should be.

7. Additional constraints:

To account for costs incurred when two communicating nodes *i* and *j* are on different mappings, we use c_{ij}. c_{ij} is expressed as:

$c_{ij} = 1 - (h_i \times h_j) - ((1 - h_i) \times (1 - h_j))$, where $(i,j) \in A$

$c_{ij} = h_i + h_j - 2h_ih_j$

Note that this is a quadratic constraint in h_i, and it can be linearized by defining a binary variable b_{ij} for all (i,j) in *A*. such that $b_{ij} = h_ih_j$. This can be expressed by:

$b_{ij} \leq h_i$

$b_{ij} \leq h_j$

$b_{ij} \geq (h_i + h_j - 1)$

The original expression for c_{ij} can now be replaced by:

$c_{ij} = h_i + h_j - 2b_{ij}$

Note that only b_{ij} adds to the variable count. c_{ij} is used only to enhance readability; in the actual formulation c_{ij} is replaced by $c_{ij} = h_i + h_j - 2b_{ij}$. The three expressions defining b_{ij} add 3A constraints. (3A constraints)

A solution to the extended partitioning problem is specified completely by h_i (mapping), bh_{ij} or bs_{ij} (implementation bin depending on the mapping selected), and t_i (schedule). The formulation has $(N^2 + N \cdot (2B + 1) + A)$ variables and

$(\frac{5}{2} \cdot N^2 - \frac{3}{2} \cdot N + 5A + 3)$ constraints, in addition to the integrality constraints on the variables.

A4 Complexity Analysis of the GCLP Algorithm

The *run-time* complexity for the GCLP algorithm is computed. Estimations of area and time, and computations of the extremity and repeller measures are outside of the main loop and are not included in the complexity calculation. The complexity of each sub-step of the GCLP algorithm (described in Section 3.3) is shown below:

Complexity(GCLP)

S1. Compute GC: O(|A| + |N|)

S1.1. Estimate the set of nodes to move to hardware: O(1)

S1.2. Compute the actual finish time: O(|A| + |N|)

S2. Determine the set of *ready* nodes: O(|N|)

S3. Compute the effective execution time: O(|N|)

S4. Compute the longest paths: O(|A|)

S5. Select a *ready* node with maximum longest path: O(|N|)

S6. Determine the mapping and schedule: O(1)

The GC computation in S1 involves the estimation of a mapping, followed by the computation of the actual finish time for this mapping. The estimation in S1.1 is done by selecting nodes from an ordered list and is done in constant time. The actual finish time is computed in S1.2 by ignoring communication costs, using the following procedure:

```
Procedure:       Compute_actualFinishTime
Input:           DAG with predetermined mapping and corresponding execution
                 time ($t_{exec}(i)$) for each node $i$
Output:          $T_{finish}$ = finish time of the DAG
Initialization:  $dsp\_Finish\_Time = 0$, $T_{finish} = 0$:
S1.  Label all nodes with their indegree
S2.  Mark all nodes with 0 indegree as ready nodes
S3.  while(ready nodes exist){
    S3.1.  Select ready node $i$: O(1)
    S3.2.  Find $t_{start}(i)$: O(1)
    S3.3.  if($i$ in software) $t_{start}(i) = \max(\max_j(t_{avail}(j)), dsp\_Finish\_Time)$
                                           ($j$: inputs of node $i$)
           if($i$ in hardware) $t_{start}(i) = \max_j(t_{avail}(j))$
    S3.4.  Update $t_{finish}(i) = t_{start}(i) + t_{exec}(i)$: O(1)
    S3.5.  If $i$ is in software:
           $dsp\_Finish\_Time = (dsp\_Finish\_Time > t_{finish}(i))$ ?
                                  $dsp\_Finish\_Time : t_{finish}(i)$
    S3.6.  For each output $k$ of node $i$ set $t_{avail}(k) = t_{finish}(i)$
    S3.7.  For each output $k$ of node $i$, access node $p$ connected to it
        S3.7.1.  Decrease indegree of node $p$
        S3.7.2.  If indegree of $p$ is 0, add $p$ to list of ready nodes
    S3.8.  $T_{finish} = (T_{finish} > t_{finish}(i))$ ? $T_{finish} : t_{finish}(i)$
} : (O(|A| + |N|)
```

Since each node and arc is traversed once, the complexity of the algorithm is O($|N| + |A|$). The longest path calculation (S4 of GCLP) is described with the help of the following procedure. Note that as the graph is acyclic, a label setting algorithm can be used.

```
Procedure:       Compute_longestPath
Input:           $G = (N,A)$, $t_{eff}(i)$ for $i \in N$
Output:          longest path $d(i)$ for $i \in N$ ($d(i)$ is the longest path from $i$ to sink)
Initialization:  counter $c = 0$, $d(i) = 0$ for $i \in N$
S1.  Reverse the directions of all the arcs in the graph: (O(|A|))
```

```
S2. Topologically order the graph (an ordering where arc $(i,j)$ means node $i$ is a
  predecessor of node $j$)
    S2.1. Label all nodes with their indegree: $(O(|N|))$
    S2.2. while $(c < |N|)$ {
        S2.2.1. Identify a node $i$ with indegree = 0.
        S2.2.2. Label $i$ with $c$.
        S2.2.3. Decrease indegree of all nodes connected to $i$.
        S2.2.4. Delete node $i$ and all arcs emanating from $i$.
        S2.2.5. $c = c+1$
      } : $(O(|A|))$
S3. Negate effective execution times for all $i \in N$. $(t_{eff}(i) = (-1) \cdot t_{eff}(i))$: $(O(|N|))$
S4. Traverse graph in topological order: $(O(|A|))$
    S4.1. $d(i) = \min(d(j)) + t_{eff}(i)$, where $j \in \{\text{predecessors of i}\}$
    S4.2. Negate $d(i)$ to give longest path for each $i$, $i \in N$ : $(O(|N|))$
```

Note that the longest path is computed as the shortest path on the graph with negated execution times. The complexity of the algorithm is $O(|A|)$, since each edge is visited once.

The actual mapping and schedule (S6 of GCLP) are computed in constant time. Thus, the total complexity for one step of the GCLP algorithm is $O(|N| + |A|)$. The GCLP algorithm runs $|N|$ times, hence total complexity is $O(|N| \cdot (|N| + |A|))$. Typically, for DSP applications, $|A| \approx |N|$, hence the worst case complexity of the GCLP algorithm is $O(|N|^2)$.

A5 Complexity Analysis of the Bin Selection Procedure

The bin selection procedure is applied on a tagged node to select its implementation bin. In general, let B be the number of implementation bins for each node. The complexity of the bin selection procedure is computed as follows:

Complexity(bin selection)

S1. Compute BFC: O($B \times$ ($|N|$ + $|A|$))

  S1.1.For each bin, compute the bin fraction: O($|N|$ + $|A|$)

    S1.1.1.Estimate the nodes to move to L bins

    S1.1.2.Compute actual finish time assuming this mapping and bin selection

S2. Compute bin sensitivity: O(B)

S3. Compute weighted bin sensitivity: O(B)

S4. Select bin: O(B)

The bin fraction computation in S1.1 is similar to that used in the computation of *GC*. Hence, the complexity for computation of the bin fraction for a particular bin is O($|N|$ + $|A|$) (see Section 4). The complexity of S1 is hence O($B \cdot$ ($|N|$ + $|A|$)) for B implementation bins. The complexity of other steps is simply in the order of the number of bins B. The complexity of the bin selection procedure hence O($B \cdot$ ($|N|$ + $|A|$)).

A6 Complexity Analysis of the MIBS Algorithm

The MIBS algorithm applies the GCLP-bin selection sequence $|N|$ times, for all the nodes in the graph. The complexity for each sub-step in the MIBS algorithm is computed as follows:

Complexity(MIBS)

S1. Determine the mapping for all free nodes by running GCLP: O($|N|^2$)

S2. Determine ready nodes: O($|N|$)

S3. Select tagged node: O(|N|)

S4. Compute implementation bin for tagged node using the bin selection procedure: O(B · (| N| + |A|))

The complexity of one step of the MIBS algorithm is O($|N|^2 + B \cdot |N|$). Each step is repeated |N| times for the |N| nodes, hence the complexity of the MIBS algorithm is O($|N|^3 + B \cdot |N|^2$).

A7 Algorithm for Random Graph Generation

To get an estimate on the behavior of an algorithm in *practice*, empirical studies can be performed. This involves devising a set of supposedly "typical" instances, running the algorithm and its competitors on them, and comparing the results. Of course, the usefulness of this technique depends on how "typical" the sample instances are. Ideally we would like to run our algorithm on practical examples. However, due to the absence of system-level benchmarks, and the lack of time to generate a significant number of examples ourselves, we resort to generating random examples. The following procedure presents the method used to generate random DAGs.

A7.1 Random DAG Generation

Procedure:	**Generate_Random_Graph**
Inputs:	size of graph N
Output:	Directed, acyclic graph, without parallel edges.

```
S1.  $A$ = $random\_int(N,N^2)$[4] (number of arcs)
S2.  Generate a random permutation of 1, …, $N$ in the array "$perm$"
     S2.1.  for($i$=0; $i<N$; $i$++) {$perm[i] = i$;}
```

4. The random number generator *random_int*(l,u) is used to generate random integers within the range (l,u) and is adapted from [Press88].

```
    S2.2.  for($i$=0; $i$<$N$; $i$++) {
        S2.2.1.  $j$ = random_int(0,$N$-1);
        S2.2.2.  tmp = perm[$i$];
        S2.2.3.  perm[$i$] = perm[$j$];
        S2.2.4.  perm[$j$] = tmp;}
S3.  Generate "$A$" random edges of the form (perm[$i$], perm[$j$]), for $i < j$
    S3.1.  for($i$=0; $i$<$A$; $i$++) {
    S3.2.  src = random_int(0, $N$-2);
    S3.3.  dest = random_int(src+1, $N$-1);
    S3.4.  if(!existsArc(src, dest)) add_arc(src, dest)}
S4.  Generate nodal information for each node $i$: $size_i$, $ts_i$, $th_i$, $as_i$, $ah_i$.
    S4.1.  Determine if node $i$ is an extremity, repeller, or normal node
        S4.1.1.  $y$ = ran1()[5]
        S4.1.2.  if(($y$ >= 0) && ($y$ < 0.33)) $i$: extremity
        S4.1.3.  if(($y$ >= 0.33) && ($y$ < 0.66)) $i$: repeller
        S4.1.4.  if(($y$ >= 0.66) && ($y$ <= 1)) $i$: normal
    S4.2.  Set area and time values based on the nature of node $i$
S5.  Generate constraints: $T$, $AH$, $AS$.
    S5.1.  $T$ = random_int(sum_th, sum_ts)
    S5.2.  $AH$ = random_int(ah_low, ah_high)
    S5.3.  $AS$ = random_int(as_low, as_high)
```

The number of arcs is first determined in S1. To generate a directed acyclic graph [Johnsonbaugh91] with N nodes, a random permutation perm[0], …, *perm*[N-1] is generated in S2. This random permutation serves as a topological ordering for the graph. In S3, a set of random edges is generated of the form (*perm*[i], *perm*[j]) with $i < j$. This generates a directed acyclic graph.

A7.2 Generation of Hardware-Software Area-Time Values

The nodal information is next generated in S4. Traditionally, random graphs have been used for testing out scheduling heuristics, where the only param-

5. *ran1*() uses a multiplicative congruential method to generate a uniformly distributed random number in the range (0,1) [Press88].

eter associated with a node is its execution time. In that case, the execution time is generated from a uniform distribution (see for example, [Sih91]). In our specific problem, estimates for *area and time* on *hardware and software* are required. It is obvious that there is a correlation between the area and time values for a particular realization. In addition, there is a correlation between the values across hardware and software. Hence generating independent random values for all the four quantities will not model a realistic example. To overcome this problem, we have devised a methodology for generating these values, based on practical observations and realistic correlations. First, to model the nodal heterogeneity, a probabilistic measure is used (S4.1) to mark a node as an extremity node, repeller node, or normal node. Secondly, the area and time values for the node are then set depending on the nature of the node (S4.2). The following procedure illustrates the general methodology for generating the area and time estimates for an extremity node.

```
Procedure:   Generate_Extremity_Node
Input:       $ts_{min}$, $as_{min}$, $th_{min}$, $ah_{min}$, $ts_{max}$, $as_{max}$, $th_{max}$, $ah_{max}$, cutoff threshold
Output:      $ts$, $th$, $ah$, $as$
S1. Compute threshold values $ts_{th}$, $as_{th}$, $th_{th}$, $ah_{th}$ (for ex: $ts_{th} = ts_{min}$ + cutoff * ($ts_{max} - ts_{min}$))
S2. Compute maximum and minimum extremity values
    S2.1. $xs_{max} = ts_{max} / ah_{min}$
    S2.2. $xs_{min} = ts_{th} / ah_{th}$
    S2.3. $xh_{max} = ah_{max} / ts_{min}$
    S2.4. $xh_{min} = ah_{th} / ts_{th}$
    S2.5. $delta_h = xh_{max} - xh_{min}$
    S2.6. $delta_s = xs_{max} - xs_{min}$
S3. Determine if the node is a hardware or software extremity
    S3.1. $z = ran1()$
    S3.2. if($z < 0.5$) hardware_extremity; else software_extremity;
S4. Generate an extremity measure in the range (0, 0.5) ($ex = ran(0, 0.5)$)
S5. If hardware extremity:
    S5.1.  $ah = random\_int(ah_{th}, ah_{max})$
    S5.2.  $ts = ah / ((2 * delta_h * ex) + xh_{min})$
    S5.3.  $as = random\_int(as_{min}, as_{th})$
    S5.4.  $th = random\_int(th_{min}, th_{th})$
S6.  If software extremity:
    S6.1.  $ts = random\_int(ts_{th}, ts_{max})$
    S6.2.  $ah = ts / ((2 * delta_s * ex) + xs_{min})$
    S6.3.  $as = \mathrm{random\_int}(as_{min}, as_{th})$
    S6.4.  $th = random\_int(th_{min}, th_{th})$
```

The mechanism for the calculation of the area and time values for extremities *reverse engineers* the mechanism used for extremity calculation described in Section 3.3.3.2. The range of area and time values in hardware and software is specified by the input parameters. In S1, the cut-off threshold is used to compute the threshold values of area and time in hardware and software, similar to that described in Section 3.3.3.2 for extremity identification. The maximum and minimum extremity values are computed in S2, using the definition of extremities. In S3, the node is probabilistically set to be either a hardware or a software extremity, and in S4 its extremity measure is generated as a random number in the range (0, 0.5). The area and time values are next generated in S5 and S6. If a node is a hardware extremity, its *ah* is set to a random number between (ah_{th}, ah_{max}), and its extremity measure is then used set the *ts* value. The *th* and *as* are set to random numbers in the ranges (th_{min}, th_{th}) and (as_{min}, as_{th}) respectively. The area and time values for a software extremity are similarly generated, as shown in S6.

The procedure for generating area and time estimates for repellers is as follows:

```
Procedure:    Generate_Repeller_Node
Output:       ts, th, as, ah
```

```
S1. Generate $TS$ and $TH$ array values
S2. Determine if the node is a hardware or software repeller
    S2.1. $z = ran1()$
    S2.2. if($z < 0.5$) hardware_repeller; else software_repeller;
S3. Generate a repeller measure in the range (0, 0.5) ($r = ran(0, 0.5)$)
S4. If software repeller:
    S4.1. $ts = TS[random\_int(0, 5)]$
    S4.2. $as = ts / 2$
    S4.3. $ah = (1 - r) * as$
    S4.4. $th = (1 + ran1()) * ah$
S5. If hardware repeller:
    S5.1. $th = TH[random\_int(0, 5)]$
    S5.2. $ah = th / 2$
    S5.3. $as = (1 - r) * ah$
    S5.4. $ts = (1 + ran1()) * th$
```

Recall that repellers identify *relative* preferences, i.e., give two *similar* software nodes, the software repeller measure is the relative area gain in hardware. To take the similarity of nodes into account, in S1, a set of possible *ts* and *th* values is generated. The *ts* (*th*) values for software (hardware) repellers are then selected from this set *TS* (*TH*) so as to ensure that *similar* nodes are considered. In S2, a node is probabilistically assigned to be either a hardware or a software repeller, and in S3 its repeller measure is generated. If the node is a software repeller, its *ts* is selected to be one of the set of possible values from the set *TS*. The *as* is related to this *ts*. The value of *ah* is then generated from the *as* value using the repeller measure. The area and time values for hardware repellers are similarly generated in S5.

```
Procedure:  Generate_Normal_Node
Output:     $ts, th, ah, as$
S1. $ts = random\_int(ts_{min}, ts_{th})$
S2. $as = ts / 2$
```

```
S3.  $th = ts / random\_int(1, 4)$
S4.  $ah = th / 2$
```

For a normal node, *ts* is generated as a random number between ts_{min} and ts_{th}. The value of *th* is a fraction of the software time; this fraction is a probabilistic number from 1 to 4. The areas are computed as a fraction of the times so as to correlate the area and time values on a given mapping.

Finally, in S5 of the graph generation algorithm, the constraints are generated. The deadline is set to a random value between the sum of hardware and software execution times. The hardware capacity is set to a random number between ah_{low} and ah_{high}, where ah_{low} and ah_{high} are fractions of the sum of the hardware sizes of all the nodes. The software capacity constraint is similarly generated.

A7.3 Implementation Curve Generation

To generate the random graphs for the extended partitioning problem, hardware and software implementation curves are required for each node. A *generic* implementation curve was obtained from practical observations of a number of implementation curves. This curve gives the area and time values for various bins, normalized with respect to the L bin values. The generic curve is of the form (tf^k, af^k), where $tf^k > 1$, and $af^k < 1$, for the *k* bins.

The area and time values for a single implementation are generated for a node, as discussed in Section 7.2. The generated values of *ah* and *th* are assumed to correspond to the *L* bin (ah^L, th^L). The generic implementation curve is then used to compute the area and time values for other bins. The generated curve is of the form (th^k, ah^k), where $th^k = tf^k * th^L$, $ah^k = af^k * ah^L$.

A8 Estimation of Area and Execution Time Parameters

The partitioning process relies heavily on the estimates of the area and execution time on different hardware and software implementations. The usefulness of the generated partition depends on how accurate the estimates are; if the actual values obtained after synthesis are very different from the estimates, then the resultant solution could be either infeasible or have an unnecessarily large hardware area. Accuracy of the estimated area and time values is directly proportional to the time spent on generating the estimates; the more accurate the estimate, the longer it takes to derive it. In the limiting case, the best estimate is obtained after actually synthesizing the implementation. Our philosophy in generating the estimates was to get fairly accurate estimates from behavioral specifications, without going through the entire synthesis process.

In Section A8.1, some of the related techniques in hardware estimation are first briefly mentioned, followed by a discussion of the estimation mechanism that we use. In Section A8.2, the software estimation techniques are discussed.

A8.1 Hardware Estimation

The hardware area required to implement a node consists of the active area of the datapath[6] (i.e., execution units, registers, multiplexors, and buffers) as well as the interconnect area. Several methods to estimate the active area have been proposed [Chaudhari94][Kurdahi89][Rabaey94]. Since the interconnect area is usually a significant fraction of the active area, considering only the active area is inaccurate, the interconnect area also needs to be estimated. BUD [McFarland90b], developed by McFarland et al., was one of the first highlevel syn-

6. We assume a standard-cell type hardware architecture; the extension to FPGAs is quite straightforward.

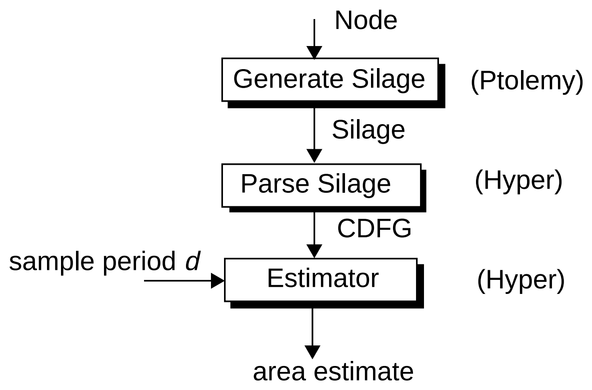


Figure A.1. Methodology for the estimation of hardware area required for a node when implemented at a sample period *d*.

thesis tools that took the interconnect area into account. The approach, however, is based on doing an actual floorplanning of the generated modules to get an estimate of the total area of the design. Zimmerman [Zimmerman88] describes slicing-structure based mechanisms for estimating the floorplan areas. In the approach proposed by Sharma et al. [Sharma93], the number of busses required in the design is first estimated. This number is correlated to the interconnect area to estimate the total area.

Mehra et al. [Mehra94] propose a statistical approach for computing the total chip area (active plus interconnect) for a given sample period *d*. In this scheme, the minimum bounds on the active area (A_{act}) and the number and size of buses (N_{bus}, N_{bits}) are first estimated using techniques proposed by Rabaey *et al.* [Rabaey94]. The total area is then given by a statistical model that uses these values: $A_{total} = \alpha 1 + \alpha 2 \cdot A_{act} + \alpha 3 \cdot N_{bus} \cdot N_{bits} \cdot A_{act}$, where the parameters $\alpha 1$, $\alpha 2$, and $\alpha 3$ have been determined by experimental analysis. These techniques have been implemented in the Hyper highlevel synthesis system [Rabaey91]. We use the estimation techniques implemented in Hyper to generate the hardware area estimates.

The overall methodology for obtaining the hardware area estimate for a node, for a given sample period *d*, is summarized in Figure 8.1. Silage

[Hilfinger85] code is first generated for the node, using the approach described in Chapter 4. This Silage code is then parsed to get a control-dataflow graph (CDFG) [Verbauwhede94]. The estimator, using the estimation techniques implemented in Hyper, operates on this CDFG to give the area estimate for this particular sample period.

A8.2 Software Estimation

Software performance estimation is usually a very difficult problem. Compiler optimizations and operating system interaction make the area and time requirements difficult to predict. Tools such as Quantify [Quantify] can be used to get accurate estimates of the actual time spent in running the code. However, since this requires running the software, code has to be generated, compiled, and run for each node; the estimation times may be unacceptable. Besides, this gives only a single trace, making it hard to estimate execution times when there is data dependency within a node. Other approaches for estimation have also been reported [Gong94] [Tiwari94].

In our approach, the Motorola DSP 56000 is used as the target software processor. Since the software model assumes no operating system, the estimates generated by static profiling of the generated assembly code are fairly accurate. Assembly code is generated for each node. A simple C program parses the generated assembly code to compute the estimates of the software size (program and data memory required) and software execution time.

Herrmann et al. [Herrmann94] describe techniques to move the estimation process into the partitioning loop in order to improve the quality of the solution in the cases where the estimates are inaccurate.

A9 Automated Flow Execution in the DMM Domain

In this section, the details of the flow execution mechanism are provided. Some of the terms used in this section are first defined in A9.1. The scheduling mechanisms used in the "*Run All*", "*Run Upto*", and "*Run This*" modes are described in A9.2, A9.3, and A9.4 respectively.

A9.1 Preliminaries

We define some of the terms used in the rest of this section:

$G_{flow}(N,A)$	The design flow specified as a graph $G_{flow}(N,A)$, where N is the set of nodes representing tools and A is the set of arcs representing the connectivity between the tools.
Required Input Port	The tool cannot run unless data is present on a required input port.
Optional Input Port	The presence of data on an optional input port is not essential for execution of the tool. The behavior of the tool could depend on whether or not data is present on an optional input port.
Required Output Port	Data is always generated on a required output port when the tool executes.
Optional Output Port	Data may or maynot be generated on an optional output port.
Path	A path from tool i to tool j is a sequence of arcs from tool i to tool j.
Predecessor	$i \in P(j)$ (i is a predecessor of j) if there is a path from i to j.
Ordered predecessor	$i \in OP(j)$ (i is an ordered predecessor of j) if there is atleast one path from i to j that traverses all *required* input ports. That is, $i \in OP(j)$ implies that j cannot run without i.
Source	A tool with no inputs.
Nondeterminacy Condition	The generated data is possibly not independent of the sequence used to invoke the tools.

A9.2 Run All mode

In this mode, the graph is traversed, starting with the source tool, executing tools as needed. The following algorithm describes the mechanism.

```
runAll($G_{flow}(N,A)$) {
    if (number_of_sources > 1)
        Error("Multiple sources not allowed");
    for each tool $i$ in $N$ {
        compute_predecessors($i$);              /* set $P(i)$ */
        compute_ordered_predecessors($i$);      /* set $OP(i)$ */
    }
    /* check for possible nondeterminacy condition */
    for each tool $i$ in $N$ {
        for each tool $j$ in $P(i)$ {
            if ($i \in P(j)$) {
                if(($i \in OP(j)$) && ($j \in OP(i)$)
                    Error("Deadlock");              /* Fig. 8.2-a*/
                if(($i \notin OP(j)$) && ($j \notin OP(i)$)
                    Error("Possible Nondeterminacy");
                                                    /* Fig. 8.2-b*/
            }
            else if ($i \notin P(j)$) {
                if($j \notin OP(i)$)
                    Error("Possible Nondeterminacy");
                                                    /* Fig. 8.2-c*/
            }

        }
    }
    if (nondeterminacy)
        Query("continue", "run tool by tool", "abort");
    enabledToolsList.add(source);
    for each tool $i$
        initialize($i$);    /* retrieve last_file_name and last_time_stamp
                               from Oct. Set param_changed_flag if parameters
                               were changed */
    run_enabled_tools();  /* the main procedure */
}
```

As discussed in Section 5.3, to avoid possible nondeterminacy, design flows with multiple sources are not supported for automated scheduling. The pre-

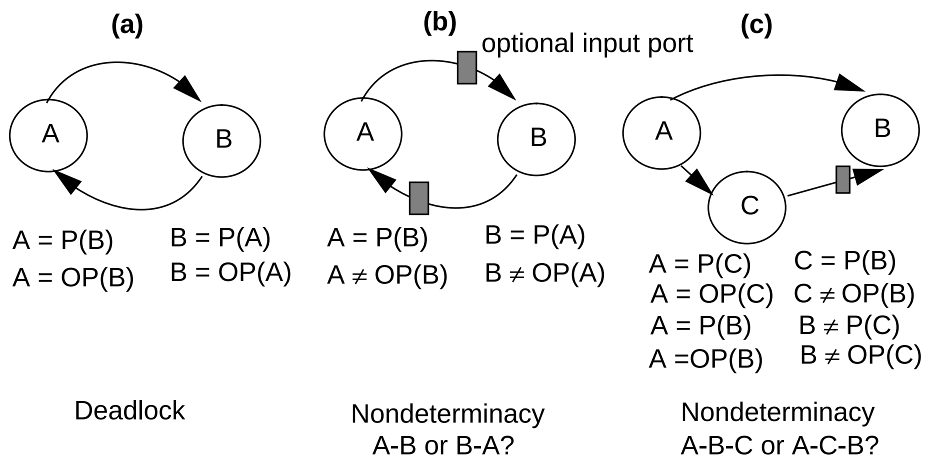


Figure A.2. Illustration of possible nondeterminacy.

decessors and ordered predecessors for all the nodes are first computed. The algorithms for computing the predecessors and ordered predecessors of a node are outlined below and are self-explanatory. The predecessors and ordered predecessors are used to detect possible nondeterminacy and deadlock. Figure 8.2 illustrates some of the cases implying possible nondeterminacy. If the scheduler detects possible nondeterminacy, the user has the option to either abort the run, or control the flow execution by running a tool at a time. If the user has designed the tools so as to avoid nondeterminate behavior, the user can ask the system to run anyway and automated scheduling is used (procedure *run_enabled_tools*()).

Before calling the procedure *run_enabled_tools*(), the tools are first initialized. The filename and timestamp attributes for each port are loaded in from the Oct database. If the tool has never run before, the last filename attribute is initialized to “DUMMY”. The system also checks if a parameter has changed since the previous invocation of this tool. The main routine for running the tools is described in the procedure *run_enabled_tools*().

```
compute_predecessors(i) {
        for all tools { clear_tag(); clean_mark_on_all_ports(); }
        tag_predecessors(i);
```

```
    for each tool j in N
        if(is_tagged(j)) add(P(i),j);
}
```

```
tag_predecessors(i) {
    /* recursively tag predecessors */
    tag(i);
    for each input port p of i {
        if(!is_marked(p)) {
            mark(p);
            j = connected_tool(p);
            tag_predecessors(j);
        }
    }
}
```

```
compute_ordered_predecessors(i) {
    for all tools { clear_tag(); clear_mark_on_all_ports(); }
    tag_required_predecessors(i);
    for each tool j in N
        if(is_tagged(j)) add(OP(i),j);
}
```

```
tag_required_predecessors(i) {
    /* recursively tag required predecessors, i.e., predecessors connected to
required input ports*/
    tag(i);
    for each required input port p of i {
        if( !is_marked(p)) {
            mark(p);
            j = connected_tool(p);
            tag_required_predecessors(j);
        }
    }
}
```

The following procedure shows the mechanism to run enabled tools. A tool *i* is first removed from the head of the list of enabled tools. If there is a tool *j* on the list of enabled tools that is a predecessor to *i*, then *i* is not run, but simply moved to a slot after *j*. This tries to enforce an ordering where a predecessor is always run first. This rule will enforce the A-C-B ordering for the graph shown in Figure 8.2-c. If there is no predecessor on the list of enabled tools, the selected tool is examined for live dependencies and executed if needed.

```
run_enabled_tools() {
    while(enabledToolsList != empty) {
        i = enabledToolsList.head();
        nextTool = 0;
        for each j in enabledToolsList {
            if(($j \in P(i)$) && ($i \notin P(j)$)) {
                enabledToolsList.move(i,j); /* move i to after j */
                /* try to force a behavior by running j, which is a
                 predecessor  to i, before running i. If there is a loop
                 between i and j, then we just run them in the order
                 on the list; we have already flagged to the user that
                 this could be a nondeterminate case. */
                nextTool = 1;
                break;
            }
        }
        if(!nextTool) {
            examine_dependency_and_run_if_needed(i);
            enabledToolsList.remove(i);
        }
    }
}
```

The following procedure examines a tool for live dependencies. If a dependency is alive, the tool is executed, the generated data is passed on to its descendents, and the descendents are enabled, otherwise the old data is propogated to its descendents and the descendents are enabled.

```
examine_dependency_and_run_if_needed(i) {
    if(dependency_alive(i)) {
        execute_tool(i);
        send_new_output_data(i);
```

```
                enable_descendents(i);
                update_status(i);
        }
        else {
                send_old_output_data(i);
                enable_descendents(i);
        }
}
```

The following procedure checks for live dependencies. A parametric dependency is alive if *param_changed_flag* is set. A source needs to run if it has never run before, i.e., a required output port has attribute “DUMMY”.

In general, after a tool finishes execution, the newly generated file name is copied over to the descendent ports. If a tool does not generate data on a certain output (because it is optional), it sends an explicit “DUMMY” to the descendent port. As a result, for non-source tools, a new file name of “DUMMY” on an input port indicates that no new data has been propogated by the predecessor tool to this port. This is the reason for the first check comparing the new filename to DUMMY. If this check holds, then the data and temporal dependencies are evaluated.

```
dependency_alive(i) {
        if(param_changed(i))
                return 1; /* parametric dependency */
        if(is_source(i)) {
                for each required output port p
                        if(strcmp(last_file_name(p),”DUMMY”)==0)
                                return 1;
        }
        else {
                for each input port p of i {
                        if(strcmp(new_file_name(p),”DUMMY”)!=0) {
                                if(strcmp(new_file_name(p),last_file_name(p)) != 0)
                                        return 1; /* data dependency */
                                if(last_time_stamp(p) < new_time_stamp(p))
                                        return 1; /* temporal dependency */
                        }
        }
    }
}
```

The following procedure calls the user-defined code associated with a tool. This operates on the new filenames associated with the input ports. A *data_changed* flag gets set for all the ports for which new data is generated during this execution. The generated data is available in the new filename attribute.

```
execute_tool(i) {
    execute the user-defined code associated with the tool;
        for each output port p of i
        if (data is generated on p)
            set_data_changed_flag(p);
}
```

After a tool finishes execution, the next step in the procedure *examine_dependency_and_run_if_needed* is to send the generated data to the connected ports, as shown in the following procedure *send_new_output_data*. For required output ports, the corresponding descendent port is *marked* (as shown shortly, this mark is used to identify whether a tool is enabled), while for an optional output port *p*, the corresponding descendent port is marked only if *p* generated new data. If the optional output port did not generate data, an explicit "DUMMY" is sent to the descendent port.

```
send_new_output_data(i) {
    /* If new data is generated, send it to the connected input port, otherwise
     send explicit DUMMY.  Mark far port only if new data is sent. */
    for each output port p of i {
        farPort = input port at the other end of the arc originating from p;
        if(has_data_changed(p)) {
            set_new_file_name(farPort, new_file_name(p));
            mark(farPort);
        }
        else {
        /* p is an optional port that did not generate data in this iteration */
            set_new_file_name(farPort, "DUMMY");
```

```
                set_new_file_name(p, “DUMMY”);
                clear_mark(farPort);
            }
        }
}
```

The following procedure examines all the descendents of a tool to check if they are enabled. A tool is enabled only if all its required input ports are marked.

```
enable_descendents(i) {
    for each output port p {
        farPort = input port at the other end of the arc originating from p;
        descendent = tool containing the farPort;
        if(each required input of descendent is marked)
            enabledToolsList.put(descendent);
            /* put only if not already there */
    }
}
```

The following procedure is called to update the filename and timestamp attributes in the Oct database in preparation for the next run.

```
update_status(i) {
    for all ports {
        last_file_name = new_file_name;
        last_time_stamp = new_time_stamp;
    }
    store in Oct database;
}
```

The following procedure is executed by a tool if it has no live dependencies. This procedure propagates the filenames to the descendent ports.

```
send_old_output_data(i) {
    /* Send data generated on the previous invocation.
     Mark only if not DUMMY */
    for each output port p {
        farPort = input port at the other end of the arc originating from p;
        set_new_file_name(farPort, last_file_name(p));
        if(strcmp(last_file_name(p),”DUMMY”) != 0)
            mark(farPort);
        else
```

```
                clear_mark(farPort);
        }
}
```

A9.3 Run Upto mode

In this mode, the user selects a certain tool upto which the flow should be executed. All the predecessors of the selected tool are identified and tagged. The design flow is then traversed as in the *Run All* mode; only the tagged predecessors are executed. In the current implementation, we support this mode for acyclic flows only.

```
runUpto($G_{flow}$, i) {
        /* run upto a tool i in flow $G_{flow}$ */
        if(!is_acyclic($G_{flow}$))
                Error("runUpto supported only on acyclic flows");
        if (number_of_sources > 1)
                Error("Multiple sources not allowed");
        for each tool j in $G_{flow}$
                initialize(i);     /* retrieve last_file_name and last_time_stamp from
                                    Oct. Set param_changed_flag if parameters were
                                    changed */
        compute_predecessors(i);
        for each j in P(i)
                tag(j);
        enabledToolsList.add(source);
        while(enabledToolsList != empty) {
                k = enabledToolsList.head();
                if(is_tagged(k)) {
                        examine_dependency_and_run_if_needed();
                }
        }
}
```

A9.4 Run This mode

In this mode, a single user-selected tool is run. The selected tool is run if it has valid input data. The tool operates on the data generated by its predecessors in their latest invocation.

```
runTool(i) {
    /* Run one iteration of a selected tool.  Operates on the last data generated
     by predecessor tools. Doesn't run other tools */
    for each tool i initialize(i);
    /* read in last_file_name and last_time_stamp for tool i from Oct */
    for each input port p of i {
        farPort = output port at the other end of the arc terminating in p;
        /* read in last_file_name and last_time_stamp of ancestor */
        set_new_file_name(p) = last_file_name(farPort);
        set_new_time_stamp(p) = last_time_stamp(farPort);
    }
    for each required input port p of i {
        if(strcmp(new_file_name(p),"DUMMY")==0)
            Error("Predecessors never run before, can't run i.");
        else
            mark(p);
    }
    for each optional input port p of i {
        if(strcmp(new_file_name(p),"DUMMY")!=0)
            mark(p);
        else
            clear_mark(p);
    }
    if(dependency_alive(i)) {
        execute_tool(i);
        update_status(i);
    }
}
```

System-level Codesign of Mixed Hardware-Software Systems

by

Asawaree Prabhakar Kalavade