EARLY SYSTEM ARCHITECTURE OPTIMIZATION FOR MULTI-CHIP SYSTEMS

by

Dong-Hyun Heo

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Computer Engineering)

December 1997

Copyright 1997 Dong-Hyun Heo

Dedication

To my mother, wife, and children

Acknowledgements

I take this opportunity to express my gratitude to those who have made this thesis possible. I am grateful to my advisor Prof. Alice Parker for her constant guidance and inspiration during my dissertation work. She has been a great source of encouragement and support. I also wish to thank Profs. Peter Beerel and Cliff Neuman for being on my dissertation committee, and Profs. Melvin Breuer, C.P. Ravikumar, and Douglas Ierardi for being on my guidance committee.

I would like to thank my colleagues Diogenes Silva, Suhrid Wadekar, J.C. Batista-Desouza, Chih-Tung Chen, and Pravil Gupta for their friendship and help. I benefitted greatly from interacting with them.

There are many other friends who made my years at USC pleasant. Among them, I especially thank for Kang Woo Lee, Jae Won Oh, Jung Hyun Han, Jae Heon Jung, Jong Woo Bae, and Jong Hyun Kahng.

My family have been a continuous source of love, support and sacrifice. I dedicate this thesis to them. I especially thank my wife Hye-Kyung for her patience through these years.

This research was supported by the Advanced Research Projects Agency (Contract No. JFBI90092) and National Science Foundation (Contract No. GER-9023979). I would like to thank these organizations for their support.

Contents

Dedication ii

Acknowledgements iii

List Of Figures viii

List Of Tables x

Abstract xi

1 Introduction 1

1.1 Background 1
1.2 The System-level Design Problem 3
1.3 System-level Design Automation 6
1.4 System-level Design Methodologies 10
1.5 Problem Approach 13
1.6 Thesis Organization 15

2 Related Research 16

2.1 System-level Synthesis 16
2.2 The USC project 17
2.3 Design Methodologies and Design Process Model 19
2.4 System Specification 21
2.5 Transformation 23
2.6 Simulation and Verification 23
2.7 Design Space Exploration 24
2.8 Hardware/software codesign and partitioning 24
2.9 Multichip Design 25
2.10 Comparison with Previous Research 28

3 The Multi-Chip Design Problem and Model **30**
3.1 The Multi-Chip Architecture Design Problem with an Example 30
3.1.1 Physical Design Style Selection 33
3.1.2 Datapath Architecture Selection 36
3.1.3 Task-Level System Partitioning 38
3.1.4 Die Selection 39
3.1.5 Die Clustering and Package Selection 40
3.1.6 Substrate Technology Selection 42
3.1.7 Channel Allocation and Bus Selection 43
3.1.8 Task scheduling 45
3.2 A Multi-Chip Design Problem and A Model 45
3.2.1 Specification and User Specified Constraints 46
3.2.2 Library 46
3.2.3 Target architecture 48
3.2.4 The Problem Statement 50
3.2.5 The Problem Approach 51
3.3 A First-Order Analytical Model for Multi-Chip System Design . . . 52
3.3.1 Modeling a Design Step and Its Relationship to Other Steps 52
3.3.2 Physical Design Style and Datapath Architecture Selection Subproblem 55
3.3.3 Task-Level System Partitioning Subproblem 57
3.3.4 Die Selection Subproblem 60
3.3.5 Die Cluster Subproblem 62
3.3.6 Substrate Technology Selection Subproblem 65
3.3.7 Package Selection Subproblem 66
3.3.8 Bus Selection Subproblem 66
3.3.9 Task Scheduling 68
3.3.10 System Metric Functions 69
3.3.10.1 The System Cost Metric Function 69
3.3.10.2 System Performance Metric Function 70
3.3.10.3 System Time-To-Market Metric Function 71

4 EDEN: MILP-Based Optimization Tool **74**
4.1 MILP and Linearization 74
4.2 Problem Instantiation and the M-Language 75
4.3 Experimental Results 77
4.3.1 Examples 77
4.3.2 Library Setup for Experiments 79
4.3.3 Experimental Results for Rapidprototyping 80
4.4 Conclusion 84

5 GARDEN: Genetic Algorithm-Based Optimization Tool **86**
5.1 Genetic Algorithm . . . 87
5.2 GARDEN Implementation . . . 88
5.2.1 Encoding and Data Structure of a Solution . . . 89
5.2.2 Initialization . . . 91
5.2.3 Selection . . . 95
5.2.4 Crossover . . . 95
5.2.5 Mutation . . . 100
5.2.6 Evaluation and Fitness Functions . . . 102
5.3 Termination Conditions . . . 107
5.4 Experimental Results . . . 109
5.4.1 Examples . . . 109
5.4.2 The Effect of Physical Style Selections . . . 109
5.4.3 The Effect of the Number of Task Partitions . . . 112
5.4.4 The Influence of MCMs . . . 117
5.4.5 The Example Designs Optimized by GARDEN . . . 117
5.4.6 The Run Time Distribution of GARDEN . . . 120
5.5 Conclusion . . . 120

6 Conclusions and Future Work **124**
6.1 Conclusion . . . 124
6.2 Future Work . . . 126

Appendix A
Linearization for MILP . . . 136
A.1 Lemmas for Linearization . . . 136
A.2 Linearization of Non-linear Formulations for Task-Level System Partitioning . . . 138
A.3 Linearization of Non-linear Die Selection Formulation . . . 144
A.4 Linearization of System Metric Functions . . . 147

Appendix B
Complete List of Linearized Formulae . . . 150
B.1 Definition of Sets . . . 150
B.2 Physical Design Style and Architecture Selection Subproblem . . . 150
B.3 Task-Level System Partitioning Subproblem . . . 151
B.4 Die Selection Subproblem . . . 153
B.5 Package Selection Subproblem . . . 155
B.6 Bus Selection Subproblem . . . 155
B.7 User Constraints on the System Metrics and the Objective Function 156
B.7.1 The System Cost Constraint . . . 156
B.7.2 System Performance Metric Function . . . 156

B.7.3 Objective Function:Modified System Time-To-Market Metric Function 156

Appendix C

MILP Formulation in the *M* Language 157

Appendix D

The VHDL Descriptions of Tasks in the JPEG and MPEG examples . . 163
D.1 JPEG Package 163
D.2 Forward DCT Specification 174
D.3 Quantizer Specification 180
D.4 Huffman Encoding Specification 182
D.5 Inverse DCT Specification 187
D.6 Dequantizer Specification 192
D.7 Huffman Decoder Specification 194
D.8 Motion Estimation Specification 197

Appendix E

The Libraries Used in the Experiments 200

List Of Figures

1.1 Characteristics of the design problems at different abstraction levels. 3
1.2 Development cycle of a digital system and concurrent engineering . 4
1.3 Four domains and steps of system-level design 8
1.4 Breaking the cyclic dependencies among design steps 11
1.5 Our system architecture design methodology 14

3.1 A specification for the MPEG encoder. 31
3.2 Comparison of different physical design styles. (a) Prototyping time (b) System clock frequency and cost 34
3.3 Area-delay trade-off of data path architectures 37
3.4 The utilization of die resources of different physical design styles . . 40
3.5 Estimated board sizes and clock cycles for 8-chip set of the RS6000 processor[OHK92] 41
3.6 A hypothetical example of a multi-chip system architecture 49
3.7 A tree representation of a multi-chip system 50
3.8 A model for a design step as a mapping with binary decision variables 53
3.9 Generic design flow for multi-chip system development 71

4.1 Software architecture of EDEN 76
4.2 Task flow graph of the JPEG codec 78
4.3 One of solutions found by EDEN for the JPEG example (EX2) . . . 82
4.4 One of solutions found by EDEN for the MPEG example (EX3) . . 84

5.1 A generic genetic algorithm[Mic92] 87
5.2 Chromosome representation of a system architecture 90
5.3 Building the tree representation from the chromosome 90
5.4 Oscillation of solution state by the random repair 93
5.5 The ordering of design steps used in GARDEN 94
5.6 Illustration for the problem of partitioning a chromosome at a fixed point 96
5.7 Illustration of crossover 98
5.8 The probability of finding a compatible solution as a function of m 98
5.9 Compatibility checking algorithm 99

5.10 Algorithm for the crossover operation 99
5.11 Illustration of crossover operations based on compatibility 100
5.12 Illustration of a mutation operation for task-level system partitioning101
5.13 Optimization sequence in GARDEN 107
5.14 Normalized fitness functions 108
5.15 Examples used in GARDEN experiments. (a) parallel example (b) serial example 110
5.16 (c) Example 2 111
5.17 Design space reachable by different physical design styles 111
5.18 The variation of system cost per unit with the volume of production 113
5.19 The variation of system cost by the change in the number of task partitions 115
5.20 The variation of system performance with the number of task partitions. 116
5.21 The variation of system cost per unit with the number of task partitions when MCMs are used. 118
5.22 Example designs optimized by GARDEN for the MPEG encoder . . 121
5.23 The run time vs. the number of generations of GARDEN 122

A.1 Piece-wise approximation of die cost vs. die size for the standard cell implementation 141

List Of Tables

1.1 MPEG encoders implemented in different target architectures . . . 4
1.2 Comparison of library elements at different levels of design abstraction 6
1.3 Design entities in different abstraction levels 8

3.1 Characteristics of different MCM technologies 43

4.1 The number of datapath architectures for tasks in the JPEG codec predicted by BEST 80
4.2 Constraints used for experiments on JPEG example 82
4.3 Summary of JPEG experiment results 83
4.4 Constraints used for experiments on MPEG example 83
4.5 Summary of MPEG experiment results 83

5.1 The number of task partitions (No. of TP) and the prototyping time (PTime) for the optimized architectures for the MPEG encoder. 112

E.1 Package library 200
E.2 Die library 201
E.3 Bus Library 202
E.4 Substrate Technology Library 202

Abstract

In this research, we developed a model for multi-chip implementation of a digital system that captures the complex process of system-level design decisions. We also designed tools that automate the system architecture optimization process, producing trade-off informations on design alternatives. A set of system-level decisions, namely physical design style selection, datapath architecture selection, task-level system partitioning, die selection, die clustering, substrate technology selection, package selection, and bus selection are automated. These decisions are modeled as binary decisions and a set of functions is designed which quickly computes metrics for design entities based on the first-order effect of a given set of design decisions. Among metrics, the prototyping time of a system is defined as a metric for a digital system and the cost of a die is computed by considering the effect of yield. The interrelationships among the system-level design decisions are captured as validity constraints which allows the software to detect the inconsistency among design decisions and therefore reduce the risk of incorrect design decisions. The design model can be used for the purpose of both automatic optimization or interactive optimization.

Based on the model developed, two automatic optimization tools have been developed. The first tool, EDEN, is an MILP-based optimization tool constructed by linearizing the model specifically for designs that can be rapidly prototyped. As a part of EDEN, a language called M and a constraint generator called GEM were developed to automate the instantiation of the linearized formulation for a specific design problem. The second tool, GARDEN, is based on a Genetic Algorithm. Novel schemes for solution encoding, population initialization, crossover, mutation, and fitness function are presented to apply a genetic algorithm framework to

the complex optimization problem considered in this research. GARDEN is implemented in an object-oriented manner and includes many heuristics developed from our understanding of the multi-chip system design problem.

With this research, the possibility of developing a more sophisticated model and automation tools for system-level design decisions is shown, which help designers to handle more complex systems, reduce the risk of development process iterations and design a better system in a shorter time.

Chapter 1

Introduction

1.1 Background

The growing popularity of internet, multimedia, and telecommunication applications has increased the demand for complex electronic systems. Several conflicting requirements are often associated with these systems, such as low cost, high performance, low prototyping time, short time-to-market, high testability, and low power dissipation. Reducing the time-to-market is acknowledged by all industries as the most important objective in system design, due to the fierce competition in the market and narrowing market windows. The cost of a unit is an important factor in consumer electronics applications. Power dissipation and energy consumption are important in battery-operated mobile computing and communication equipment. Performance is a prime consideration in many applications which involve real-time processing of image data sets. Designing systems considering all the factors mentioned above is understandably a difficult task.

On the technology side of digital system development, new technologies keep emerging and old technologies become economically feasible. Decreasing feature size, high-density programmable devices, and new packaging techniques are a few examples of such changes. Designers can consider employing such newly-available technologies for improving system characteristics. However, increasing the number of design options further complicates the job of system design, which is already unbearably complex.

Today's system designers are forced to think at an even higher level of abstraction and rely heavily on the design automation tools so as to be able to deal with the complexity of modern-day electronic systems under increasing time-to-market pressure. However, current design automation tools lag the demands of designers of such complex systems. Though employing the existing tools for system design steps reduces the design time, many system-level design issues cannot be handled with such tools. For example, while a combination of high-level synthesis tools, logic synthesis tools, and layout tools are sufficient to synthesize application-specific *integrated circuits*, the same is insufficient when the designer want to set up an overall implementation strategy of a specification into a system of multiple chips. To handle the system-level issues, new types of design automation tools are called for.

Figure 1.1 summarizes the general characteristics of the design problems at different abstraction levels. In general, at a higher abstraction level, a design problem has a smaller number of design entities and therefore, takes less time to solve. Also design decisions have a bigger impact on system quality because of more degrees of freedom. However, making correct higher-level design decisions is very difficult because of the degrees of freedom and "distance" to a final implementation. Making the right decisions in the early system development phase is a challenging task. Introducing a systematic way of analyzing the effect of different alternatives and optimizing a system architecture can reduce the number of possible design iterations and optimize a system better.

In this section, we introduce the notion of design at the *system-level* of abstraction. In the next section, we explain how system-level design is different from designing at other levels. In Section 1.3, we define the system-level design steps. In Section 1.4, we define a *system design methodology* as an ordered set of design steps and investigate the properties of these design steps. We review the system design methodologies that are currently in use and point out the problems associated with the current practices. Finally, we briefly state the problem and approach which we propose in order to automate system-level design tasks.

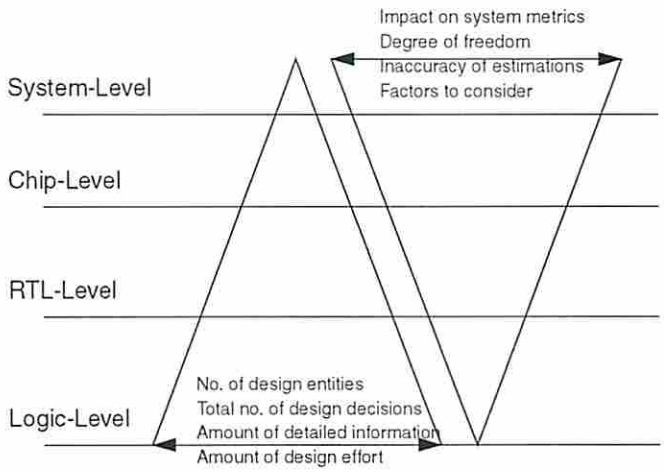


Figure 1.1: Characteristics of the design problems at different abstraction levels.

1.2 The System-level Design Problem

What are system-level design tasks? Since a system can be implemented in different target architectures as shown in Table 1.1, specific design steps are different for different target architectures. However, there are common characteristics shared by different system-level design tasks that differentiate these tasks from those of lower-level design steps. Figure 1.2 shows a schematic view of the development cycle of a digital system. The high level of abstraction of system design places the system-level design problem in an early phase of development where an overall system implementation strategy is set up. Peculiar characteristics of the system-level design problem come from its position in the development cycle. First, design steps encompass a broader range of development issues outside the design process itself. Second, the designer must make the right decisions based on rough information about lower level implementations.

Although across the whole development process any design decision brings a change in final system characteristics, the amount of change caused by a design decision is different across abstraction levels. In the literature, many authors have

Target Architecture	Comments	Reference
Full software	Both single and multiple processor implementation	[GR94]
Mixed HW/SW	Two video RISC processors are used	Optbase, Dallas, TX
Full hardware (ASIC)	H.261 video conferencing standard	[FLS+92]
Full hardware (custom)	A custom designed single chip	[RHU+93]

Table 1.1: MPEG encoders implemented in different target architectures

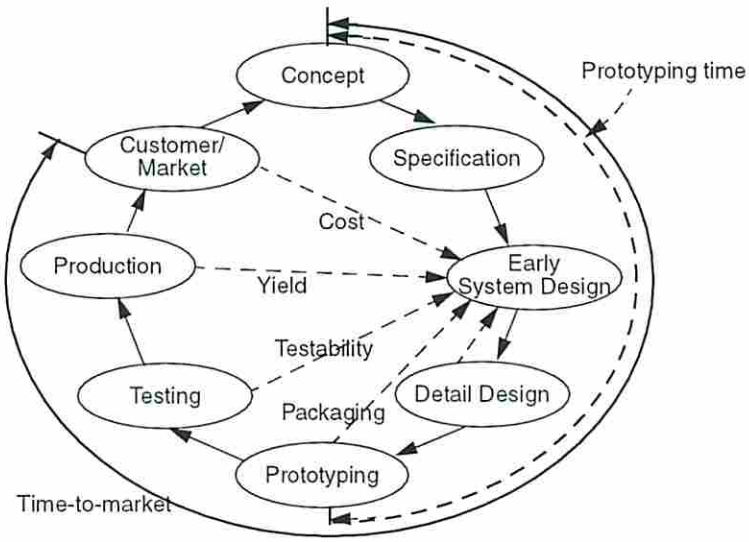


Figure 1.2: Development cycle of a digital system and concurrent engineering

pointed out the importance of making design decisions at the system level because the overall characteristics of a system are determined by decisions made at an early stage, although the time for making such design decisions does not occupy a significant part of the whole development process. "Requirements and architecture development together represent only 10% of total cost. However, once an architecture is selected, much of the development and life cycle cost of the system, as well as achievable performance, are determined" [SAM95, SA95]. In short, the importance of system-level design decisions is similar to that of an opening game of a chess match or making the sketch for a painting. Elaborate work on the details of a portion of a painting will not compensate for a bad overall sketch.

Since the impact by system-level decisions is so significant, other development issues cannot be ignored in making such decisions. For example, one of the important steps in designing a digital system is to select implementation styles for components in a system. A system can be implemented in hardware or software and a hardware module can be realized using a programmable device, an off-the-shelf component, or as an application-specific integrated circuit. In making design decisions on implementation styles of components, not only the performance and size of chips or software modules in a system must be considered but also other factors such as yield effect, packaging, and time-to-market should be considered.

In order to make good system-level design decisions, extensive trade-off analysis is necessary without knowing implementation details. Only by setting up a correct implementation strategy and *resource budget* based on such trade-off analysis among design alternatives a good system would be resulted in. While synthesizing each component of a system and integrating them together is itself a formidable task, without correct decisions in the beginning, it could be wasted. For example, for a system which is composed of a number of functional tasks, a system-level issue is to determine how to budget silicon area for each functional task such that the performance requirements are met, the cost is minimized, and the power dissipations is constrained. Without the right budgeting, the effort and time to optimize the design of each functional task separately could be in vain because the resulting system might be of superior performance but too expensive or there might be no economically feasible packaging method for the given design. This emphasizes the

Abstraction	Library Elements		
	Processing	Storage	I/O
System-level	DCT, MMU, CPU, Software module	RAM, ROM	bus and I/O circuits
High-level	Adder, Comparator, ALU	Register	Bus, Mux
Logic-level	NAND, NOR, NOT	F/F, Latch	Wire
Gate-level	Transistors	Capacitance	Metal, Poly

Table 1.2: Comparison of library elements at different levels of design abstraction

approach that making the right architectural trade-offs is usually more important than optimizing components.

1.3 System-level Design Automation

We have enumerated the general characteristics of the system-level design problem in the previous section without explaining why they have such characteristics. In this section, we view our problem from the design automation point to understand from where the characteristics of the system-level design problem are derived as well as to form a basis for automating system-level design tasks. For this purpose, we investigate how the design automation model of lower abstraction design tasks can be extended to include our system-level design problem.

The design of electronic systems and design automation tools at the physical level, logic level and register-transfer level of abstraction are well documented in the literature. At the physical design level, software packages are available for automatic generation and optimization of floor plans, placement of building blocks, and routing of interconnect. At the logic level, software tools have been reported for automatic generation of gate-level netlists starting from Boolean expressions. Tools have also been reported for technology mapping, which takes a generic gate-level netlist and maps it to a component-level netlist making use of components from a specified cell library. High-level synthesis tools which begin with a behavioral-level description of the circuit and synthesize an optimal register-transfer level architecture have also been developed.

Design problems at different levels of abstraction are characterized by the types of library elements (see Table 1.2). Besides library elements, different operational models, specification methods, and possible architectures are developed and used for different levels of abstraction. For the lower design levels such as logic design, the operation of a design is often modeled as an FSM, specified with a truth table or state diagram and the data are exchanged in the form of signals or bits. Such a logic function can be implemented with random logic, programmable arrays, or ROMs[McC86]. At the RTL level, the operation is modeled as a control data-flow graph and the data are represented as variable, integer, or floating-point numbers. An RTL design can be implemented in a number of different classes of architectures. Typical examples are pipelined and non-pipelined[PP88], and synchronous and asynchronous[MC80]. At the system level, a design can be viewed as communicating multiprocesses executing concurrently. Processes exchange more complicated forms of data such as arrays, aggregated data structures, and bit streams. Possible architectures to implement a system specification can be multi-chip systems, embedded systems, or reconfigurable systems.

The relationships among different abstraction levels and different representations of a digital system were noted and formalized by Knapp, Parker and Granacki[KP85]. They observed that there are four different domains in which a digital system can be represented, namely *data flow, timing and sequencing, structural, and physical.* A design problem is defined as a translation process from the behavioral domain to the physical domain. The system-level design problem can also be modeled similarly. As shown in Figure 1.3, we added another outer layer for system-level design to extend their model.

An example of design steps as mappings between different domains is as follows: Specification is a process to create a design entity. Synthesis is a process in which a behavioral representation of a system (e.g. truth table) is translated into the structural representation (e.g. schematic). Verification is the process of comparing the structural representation and the behavioral representation. Layout is a process of translating a structural representation into a physical representation. Different types of design entities are involved in steps at different abstraction levels. In Table 1.3, we compared design entities at different abstraction levels.

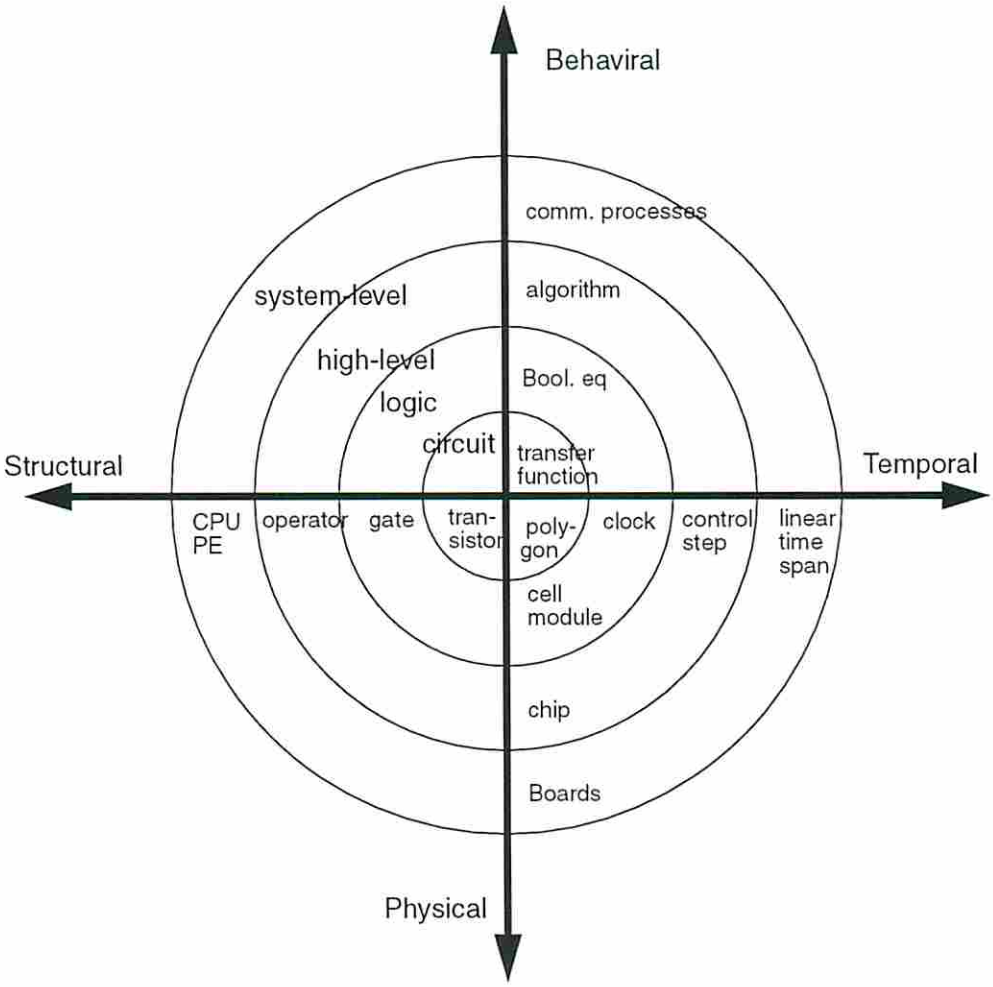


Figure 1.3: Four domains and steps of system-level design

Typical step	Abstraction Levels			
	Gate-level	Logic-level	High-level	System-level
Specification	Transfer Function	Truth Table State-diagram	Data-f-low Language	Task-flow
Verification	Circuit Simulation	Logic Simulation	HDL simulation	-
Synthesis	Schematic	Logic netlist	RTL netlist	Partitioning
Physical Design	Layout	Layout	Layout	Packaging Plan

Table 1.3: Design entities in different abstraction levels

Another common characteristic of most design steps is that they are constrained optimization problems. For example, synthesizing a schematic from a truth table can be viewed as a constrained optimization problem if the designer tries to construct a schematic with a given library of gates that meets performance constraints and minimizes the module size. For the system-level, there are more metrics simultaneously involved to create a more complicated constrained optimization problem.

Across abstraction levels, the characteristics of design activities as described above are shared among design steps. On the other hand, design issues involving each step are different at different levels and the amount of impact which each design step has on a system is different across levels. Synthesizing a logic circuit and an RTL netlist are both constrained optimization problems and there is a translation process from the behavioral domain to the structural domain. But the impact of reducing the size of a large module is bigger than the impact of reducing the number of gates; similarly, satisfying the timing constraints in an RTL netlist is more important than satisfying the delay constraints of a data path logic circuit unless the logic circuit will be used many times in a system. Steps at the system level are also different from design steps at lower levels because the result of a step may not be something that can be directly simulated or laid out. The outputs mostly take an abstract form reflecting overall characteristics and a global plan about a possible implementation.

At a lower level of abstraction, the metrics associated with design entities are well defined. For instance, the cost and performance of logic are well-defined metrics e.g. area and the maximum delay from inputs to outputs, although delay definitions vary enormously. But at the system-level, such metrics are often complicated or hard to define e.g. the cost of a system not only includes the cost of fabricating a piece of silicon but many other costs such as amortized development cost, the cost of testing, and so on. Therefore, at a lower level of abstraction, designers can focus relatively easily on satisfying and/or optimizing well-defined metrics. System-level design problems demand that designers evaluate complicated and not well-defined metrics such as cost, manufacturing, development time, and user satisfiability.

In general, the lower a level is, the easier it is to assess the effect of a design decision because "distance" from a decision to a final implementation is shorter and the number of possible implementation choices is smaller. For example, the effect on size of adding a gate to a module can be predicted more accurately than the change of the size of a chip resulting from moving a task from one chip to another.

1.4 System-level Design Methodologies

In the previous section, we described a design problem as a translation from the behavioral to the physical domain and did not explicitly mention the ordering of design tasks in a certain way. The ordering of design tasks is important in the development process. There are a few reasons for ordering. The characteristics of design problem itself is one and the logistics of a design house is the other. Design steps cannot be executed in any order because of the dependencies among them. A design process can be represented a directed graph in which each node represents steps and each arc represents the necessary information from a source step to a destination step for the destination step to be performed. In order for a design to be completed, all steps should be visited in a certain order at least once while not violating dependencies among them. Such an ordering of steps is called a *design methodology*. For example, using one methodology of designing a combinational logic with gates in a library, a truth table representation is created (specification). The designer creates a schematic (synthesis). The schematic is simulated (verification) and laid out into a layout (layout).

The dependencies among design steps are often cyclic. When there is a cyclic dependency between two steps, ordering them becomes difficult. It is not possible to optimize the design by optimally solving steps in sequence because design decisions for one step can be made only after the other step have been solved. Such cyclic interdependencies among steps exist not only in one translation process but across different translation processes. For example, it is well known that the two physical design steps, namely *placement* and *routing*, have such a cyclic

dependency. *Scheduling*, *allocation*, and *binding* also exhibit a cyclic interdependency as part of the RTL synthesis process. An example of cyclic dependencies across processes is the relationship between floor-planning and scheduling. The designer should consider floor-planning during the scheduling of operations since the delay of the critical path is influenced by the wiring length between modules while floorplanning can be influenced by scheduling[WP91]. Solving the interrelated subproblems across processes simultaneously is difficult. McFarland, Parker and Camposano called this as "integrating levels of design"[MMC88].

Based on our understanding of the design process model, we can understand existing design methodologies. The objectives of a design methodology are two fold. One objective is how to handle the complexity of design problems and the other is how to handle cyclic dependencies among design steps. Traditionally, a *top-down* design methodology is used to manage the complexity of a design, while the cyclic dependency problem is approached with *construct-improve* methodology or through the use of estimators.

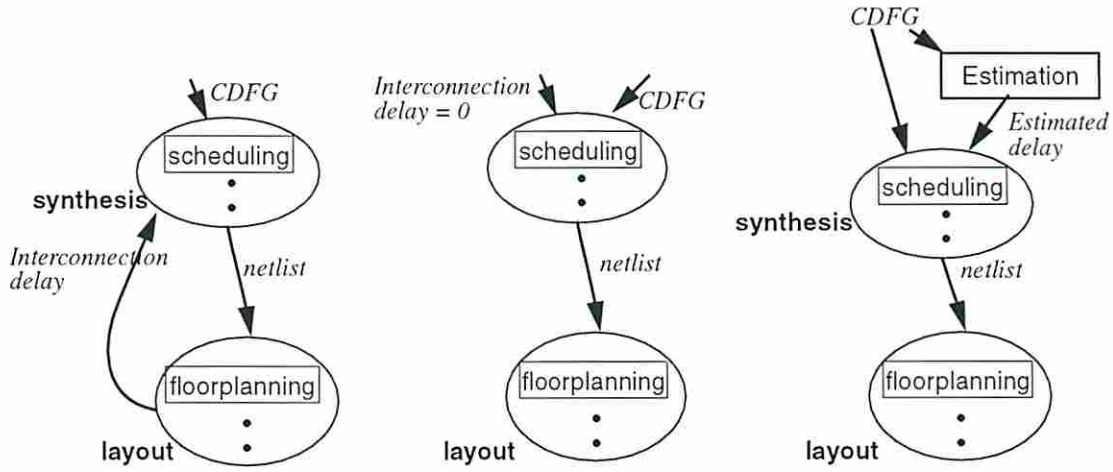


Figure 1.4: Breaking the cyclic dependencies among design steps

A typical top-down design process used for moderately complex ASICs is *capture-simulate-layout* methodology. The design is captured in a schematic, then the

netlist is simulated to verify the functionality of the system and finally the chip is laid out using layout tools. The top-down methodology can be extended to include logic synthesis or high-level synthesis when the complexity of a design problem is very large. This methodology will be called *synthesize-simulate-layout.*

In the *construct-improve* methodology, a cyclic dependency among steps is broken with reasonable assumptions about the results of later steps. When a problem is found after one design step, the problem is fixed in the preceding design step. For the above example of the cyclic dependency between the floorplanning and scheduling, if the interconnection delay due to wires is not significant compared to the operator delays, we can solve the operation scheduling problem early by ignoring wiring delay. If the interconnection delay after floorplanning is found too long to be ignored, we can perform the operation scheduling again with the backannotated wire-delay information. Since such design iterations are very expensive in terms of development time and cost, there has been an effort to reduce the number of iterations by using sophisticated estimation tools which predict the results of subsequent design steps. Such estimations become more important as the coupling of steps becomes stronger. Estimating wiring delay and incorporating it in synthesis steps is critical for the current submicron semiconductor technology in which the interconnection delay is a dominant factor in determining the timing of a system.

Top-down and construct-improve types of design methodologies cannot be simply extended for the system-level design steps. This is because, at the system-level, the "integrating of levels" extends to not only the design process but also to the whole development cycle: manufacturing, customer satisfaction, and time-to-market. Furthermore an iteration in the development cycle is much longer and more expensive. For example, system partitioning is closely related to package selection. System partitioning determines the number of partitions to be packaged and the sizes of partitions. If system partitioning is carried out without considering the packaging issue, a partitioned system might not be able to be packaged due to the huge size of one of the partitions or the required packages are too expensive so as to violate the system cost constraint. Therefore, in a system architecture design, it is important to consider the availability of components and the influence

of selections on system characteristics in the early system-level design stage to reduce the project risk.

Therefore, it is desirable to develop a design methodology and tools for the early stage of the system development which has following properties:

- The methodology concurrently consider issues that encompass the whole development cycle at the beginning of a design process, as shown in Figure 1.2, and
- The designer can view and evaluate quickly as many designs as possible so that the designer can make informed decisions based on a thorough trade-off analysis of design alternatives provided.

Though the importance of such methodologies and tools that perform the above roles has long been emphasized to design better systems in a shorter time, system-level architecture trade-off analysis and optimization remains as an implicit part of the current design process. There are few tools and frameworks compared to the software engineering field[Boe88]. Recently, interest in defining such methodologies for hardware system development is growing. The RASSP project is one such research project that focuses on reducing the development time by improving the current design methodology[Ric94]. In this thesis, we present one approach to build such tools based on a system design model that reflects concurrent engineering issues and can be used to evaluate the effect of possible design alternatives in early development phase.

1.5 Problem Approach

Our approach to achieve the aforementioned goals is shown in Figure 1.5. The design procedure is divided into a system architecture design phase and a component design phase. During the system architecture design phase, the designer generates a small number of good candidate designs using optimization tools that work at a high abstraction level but are incorporated with lower-level estimators. The designer further narrows the candidate designs from designs suggested by optimization tools and manually optimizes the selected designs with an interactive

what-if analysis tool. The final round of candidate designs is further verified with more detailed and specialized estimators. From the detailed information for candidate designs, finally a design is selected and a detailed design process can be started.

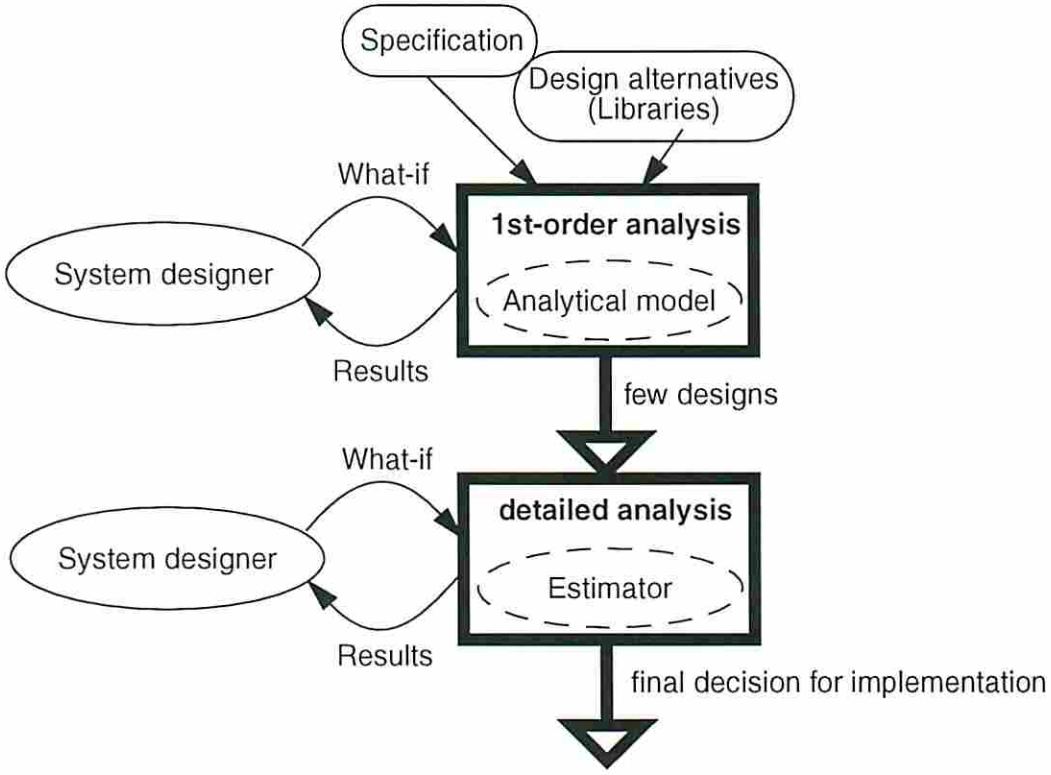


Figure 1.5: Our system architecture design methodology

The focus of our approach is the trade-off between the accuracy and the speed of estimation tools. While higher accuracy of predicted results can be achieved by emulating the design process more closely, such an approach is often impractical in the early stage because of the huge design space and the excessive computation time. On the other hand, a less accurate but faster estimator can evaluate many designs quickly but the predictions can deviate enormously from the final implementation. For example, although a predicted partition size fits into a selected die type, there is still a chance that the selected die type is too small. Therefore,

to reduce the whole system architecture design time while generating a reliable result, gradually detailed estimation and optimization with designer interaction is desirable.

To implement the above design methodology, in this thesis, we define a multi-chip architecture design problem as a subset of the system-level design problem and investigate the possibility of developing a fast estimation method and optimization tools that automate the system-level architectural trade-offs.

1.6 Thesis Organization

The organization of this thesis is as follows:

Chapter 2 surveys related research on system-level design automation.

In Chapter 3, our approach for modeling the multi-chip system design problem is described. The binary variable representation of system design decisions and the first-order analytical model that computes metrics of design entities at different abstraction levels and checks the validity of a solution will be presented.

A Mixed Integer Linear Programming (MILP) approach for optimizing the multi-chip system architecture for rapid-prototyping will be given in Chapter 4. The software architecture for an MILP-based optimization tool called EDEN will be detailed. Experimental results of the system architecture are also presented.

In Chapter 5, another optimization approach for multi-chip system design based on a Genetic Algorithm will be described. The design of GARDEN, the optimization tool using a genetic algorithm will be detailed. Experimental results are given that demonstrate how GARDEN can be used for various trade-off analyses of a system architecture.

Finally, a summary of this thesis and outline of future research are given in Chapter 6.

Chapter 2

Related Research

2.1 System-level Synthesis

Attempts to automate system-level design have only been recent. Design tasks are different from each other depending on target architectures since computing systems can be implemented using hardware, software, or a mix of hardware and software modules. An example of different implementations of the same system behavior was given in 1.1.

Hardware-software codesign is an area of system-level design which has been researched the most. Design of application-specific multiprocessor systems with heterogeneous processor types has been studied at the University of Southern California, and will be discussed later. Hardware implementation can be classified into reconfigurable systems and hardwired systems. Reconfigurable systems implement specifications using arrays of programmable logic devices. Because of their capability of on-the-fly change of system behavior, such systems are used for emulating a complex digital system or a military system in which a high performance light-weight system is required and the system is expected to perform several predetermined functions. Hardwired systems which aim at high performance are typically implemented in multi-chip systems. In this chapter, our aim is to summarize the salient work in the area of system-level design automation.

2.2 The USC project

The Advanced Design Automation System (ADAM) suite of tools [GKP85] was developed at the University of Southern California. ADAM design tools synthesize a low-cost RTL implementation from the given behavioral specification using the modules from a given operator library such that the RTL implementation satisfies the timing constraints provided by the designer. Input specification to the ADAM system is either in the form of a VHDL description or a control data flow graph. The Design Data Structure (DDS) is used for internal representation of designs [KP85]. The datapath generated by ADAM is an RTL netlist of selected library components and the control path is a finite state machine description. Unified System Construction (USC) [PCG93] is a successor of the synthesis part of the ADAM design automation research project, and aims to develop an integrated set of automation and prediction tools to produce system-level designs. The target architectures that the system can produce are (a) heterogeneous multiprocessors [Pra93, DP94], (b) asynchronously communicating multi-chip ASIC systems, or (c) synchronously communicating multiple ASIC systems[HRP96]. The main application domain of USC is that of real-time systems. Tools which are developed up to this point can perform behavioral-level partitioning[Che94], behavioral-level estimation [KP95], task-level system partitioning and package selection [HRP96], and global task scheduling on an application specific heterogeneous multiprocessor [Pra93, DP94]. BEST[Kuc91] is a comprehensive tool for estimating the number of operators, registers, multiplexers, wiring space, and delay for a process, starting from the control-data flow graph description of the process. To synthesize a synchronous multi-chip system of a specification which does not fit on a single chip, CHOP[KP95] performs behavioral partitioning at the operation level based on behavioral estimations done by BEST. MCS[Hun92] schedules the partitioned behavioral specifications across chip boundaries synchronously so as to satisfy the user specified timing constraints as well as pin constraints posed by chips. SOS[Pra93] pioneered the automation of synthesis of application-specific heterogeneous multiprocessor systems. Prakash[Pra93] used a task flow graph

model to represent a system specification. The library for SOS consists of general or special-purpose microprocessors and special hardware processors. Prakash developed a formal model of multiprocessor synthesis using mixed integer linear programming (MILP). By solving the MILP formulation, SOS synthesizes non-inferior designs which satisfy (or minimize) the timing constraints defined on the task flow graph while minimizing (or satisfying) the system cost. The synthesis results of SOS include selecting the number and types of processors in the library which best handle the given tasks and synthesizing a multiprocessor architecture by determining the interconnection style. SOS also statically schedules tasks in the task-flow graph on the chosen processors as well as the data-transfer activities among processors. SOS computes the amount of local memory which is used as storage for the code and temporary computation result for each processor in the synthesized system. Batista[DP94] extended SOS in his work, MEGA, to synthesize heterogeneous multiprocessor systems with task-level pipelining. The genetic programming paradigm was used in MEGA. Batista considered two other computing models in addition to the traditional deterministic computing model: (a) the execution time of a task on a given processor is treated as a random variable rather than a constant, and (b) the execution time of a task is divided into two parts, a mandatory part and an optional part. The first model is useful in situations where the execution time of a task cannot be estimated accurately because the number of iterations of the loops in the algorithm is dependent on the nature of the input data or there are natural statistical variations in design properties due to design and fabrication. The second model is useful in some applications such as image processing. Chen[Che94] proposed process-level partitioning across packages as an alternate way of behavioral partitioning. In his work also, the task flow graph description of a system was used. Different implementations of each task in the task flow graph and package information were supplied in a library. ProPart[Che94] synthesizes a multi-chip system by selecting an implementation for each process, partitioning the processes across multiple chips, and selecting a package type for each chip which minimizes the cost of the system while satisfying the timing constraints given on the task-flow graph. Chen formulated the problem first using ILP and then used a genetic algorithm to speed up the run time of ProPart. Chen also

studied the problem of concurrent scheduling of internal behavior of tasks and I/O transactions with unbounded delay, using the communicating process model of a system.

2.3 Design Methodologies and Design Process Model

In software engineering research, the design methodology for building complex software systems has been studied for decades, and methods for software project planning, estimating development cost, and scheduling system development have long been understood[Boe88, Pre87]. The *waterfall* model is the most influential software process model. In this model, a software development process consists of successive stepwise refinement stages namely, operational plan, operational specification, coding specification, coding, parameter testing, assembly testing, shakedown, and system evaluation. In addition to the stages, feedback loops from one stage to the previous stage exist along with prototyping in parallel with requirement analysis. The waterfall model requires an elaborate specification and documentation step between stages, which is difficult in the early stage of software development. Boehm [Boe88] proposed a *spiral* model in which 4 basic development steps are repeated with gradual refinement on the software requirements and implementation. The 4 basic steps are "determine objectives, alternatives, constraints", "evaluate alternatives, identify, resolve risks", "develop, verify, next-level product", and "plan next phases". As the development process goes through each cycle, more elaborate specification, detailed architecture consideration, and a more refined system are evolved by prototyping, simulation, and benchmarking. Therefore, the spiral model allows incomplete specification in the early phase of design, and reduces risk by evaluating the risk factor and prototyping at each round of the development process.

An important recent change in the hardware system development process is the emphasis on concurrent engineering. The sequential approach of designing a system has an inherent possibility of an iteration in a system development cycle due to failure to address issues of a later development task in early development tasks.

With people of different disciplines such as design, test, manufacturing, marketing, etc. participating in the early phase of system development, concurrent engineering tries to reduce the aforementioned risk of an development iteration. Therefore, a mechanism that ensures the end product will meet constraints posed by different groups of people is necessary. Darr and Birmingham [DB96] developed such a tool called Automated Configuration Design Service (ACDS) for a system that is built with a set of available components. ACDS is composed of catalog agents which maintain the information on available components, a system agent which requests parts that meet the specification and constraints from catalog agents, and constraint agents which monitor the feasibility of a current design. System design by multi-disciplinary people can be coordinated by ACDS to detect infeasible designs and prune the design space.

Gajski *et al.*[GNRV96] classified digital system design methodologies currently in use in the industry, namely *capture-and-simulate* methodology and *describe-and-synthesize* methodology. Capture-and-simulate methodology is a primary design methodology for ASICs in last few decades. In this methodology, an informal specification is translated into a block diagram by the chip architect and functional blocks are refined into logic circuits. Then each circuit is simulated and laid out. With the development of logic synthesis and high-level synthesis, the function of a block diagram is often described at a higher level of abstraction such as a Boolean equation or a state diagram and then translated into a logic circuit. For system-level design, Gajski *et al.* proposed a *specify-explore-refine (SER)* methodology. They pointed out that the design space exploration was an important but informal step in the system design which is often not pursued thoroughly in the design process. In SER methodology, the exploration stage is further divided into allocation and partitioning. In the allocation step, the physical implementation for a class of objects such as channels is selected from libraries based on the constraints defined by designers. Then, the class of objects is partitioned into multiple instances of selected components. For each configured system, system metrics are estimated. By manually changing design decisions, designers further optimize the system configuration.

Rapid-prototyping of Application Specific Signal Processor (RASSP) [SAM95] is concerted effort to develop a design methodology and design automation tools for DSP applications to reduce the cost and development time by four-fold. In order to achieve this, RASSP uses the following aspects in its design methodology:

- requirement traceability,
- virtual prototyping in which a software model of the hardware is developed,
- synthesis and reuse of Hardware/Software modules, (design reuse)
- use of VHDL, and
- system engineering EDA tools (pricing models and design adviser).

The RASSP design methodology starts with a VHDL specification of the system. The VHDL specification is used to simulate the system behavior. Many VHDL models are required for complex modules as well as standard parts because design reuse is an important issue. Then the architectural trade-off is made for a problem like hardware-software partitioning. Each module is refined and implemented at a more detailed level by virtual prototyping. The integration among modules is also tested using virtual prototyping.

2.4 System Specification

There are many languages developed for the specification and verification of hardware. VHDL[Ins88] is the most well-known and probably most widely used language. VHDL can be used for the specification, verification, and synthesis of hardware at different levels. HardwareC[MK88] was developed mainly for synthesis, and is based on the C programming language. Though not developed for hardware specification and verification, there are other languages devised for the purpose of describing protocols among communicating processes. Specification and Description Language(SDL)[BS91] has been developed by the CCITT for the specification of a telecommunication system. CSP (Communicating Sequential Processes)[Hoa78] is a programming language developed to formally specify the communication and synchronization between processes.

The capability of HDLs describing desired behaviors algorithmically greatly reduced the amount of specification work for complicated hardware and the overall functionality is easily verified. However, the language description is still tedious for an early design phase and not intuitive. There are other efforts that focus on building a graphic interface with which the designer can capture design at a conceptual stage. StateChart[DH89] and SpecChart[NVG91] are tools that help the designer to describe a system as a set of a concurrent processes. The communication channel among processes is described as a global variable and the process behavior is captured as a state diagram. The final design is translated into a VHDL description, which is simulated with VHDL simulators.

While the above tools are designed to have a general applicability, there are other graphic tools with description languages that are designed for specific types of digital systems. Ptolemy[KL93] uses synchronous dataflow(SDF) to describe the behavior of a DSP application. The Alta group developed a set of tools for specifying multimedia and telecommunication applications around a components library which is a well-defined set of building blocks in designing the aforementioned systems. The design can test various architectural level configurations of building blocks in a library.

The system specification is often given in a natural language which is not formal but convenient. Translating an informal specification into a formal specification would greatly reduce the burden of specifying a system in a formal language. Granacki and Parker [GP87] developed a translator using artificial intelligence natural language understanding techniques which can construct a formal specification of a system from a specification written in natural language.

Internal representations for system specification are developed as part of specification effort. Codesign Finite State Machine (CFSM) was proposed by Chiodo *et al.* [CGJ+94]. The CFSM is a model based on a network of FSMs communicating with each other. The reactive nature of CFSM is suitable for a control-dominated system. Both hardware and software specification in a higher-level language such as Esterel, StateCharts, and a subset of VHDL can be translated into CFSM. Similarly, the concurrent process model of SpecChart is internally represented as a set

of FSMs[NVG91]. Srivastava and Brodersen used a queuing network representation for the system specification described in VHDL in their work for SIERA[SB95].

2.5 Transformation

Transforming a given specification is another area of research to improve the system quality before starting detail designs. Process transformation for system-level design was proposed by Hagerman and Thomas [HT92]. Two types of transformations, namely module expansion and behavior merging, are used to explore the trade-off between implementation size and performance. Module expansion merges two physical modules into one while behavior merging combines two processes into one. Adams and Thomas[AT95] considered another transformation scheme for hardware-software codesign. A given behavioral description is clustered into a number of tasks and metrics for each task in different implementations that can be moved are defined. Then, fractions of code are moved among processes or to new processes to make architectural trade-offs.

2.6 Simulation and Verification

To speed up the verification process of a system during development, the underlying representation for a system specification is constructed such that the simulation based on the high-level modeling of components is possible. A system-level simulation tool called Ptolemy was developed at the University of California, Berkeley[KL95]. Systems such as multi-rate signal processing systems, asynchronous signal processing systems, and communication networks can be simulated. Blocks in Ptolemy can be represented in one of the supported paradigms such as synchronous dataflow, dynamic dataflow, discrete event, and digital-hardware modeling. A system can be specified using multi-paradigm components depending on the demands of the application. Chiodo *et al.*. [CGJ+94] used CFSM as a model for co-simulating both hardware and software. The time behavior of a CFSM is constructed with an equivalent FSM network. Then a timed sequence of events is checked to determine whether it is consistent with the specification.

Thomas, Adams, and Schmit[TAS93] presented a co-simulation environment. Under the given single processor system architecture model, hardware simulation is performed with Verilog processes that communicate with software processes by means of the Unix socket utility.

2.7 Design Space Exploration

With the capability of simulating a multi-paradigm system specification, various manual trade-off analyses are possible. However, no work is known to us to automate the design space exploration in the system-level design, though the importance of design decisions are pointed out by a number of researchers[SA95, SAM95, Gaj94, GNRV96]. An example of manual trade-off analyses with Ptolemy by changing parameters during simulation is given for a multiprocessor-system design for a full-duplex telephone channel simulator[KL93]. In general, any system-level simulation tool can be used for the trade-off analyses purpose.

2.8 Hardware/software codesign and partitioning

There are a number of design steps in developing an embedded system. Hardware/software partitioning, co-specification, co-simulation, software module generation, and interface synthesis are such design steps. Since works on other design steps are described in previous sections, works on hardware/software partitioning are described. A considerable research work devoted to hardware/software partitioning because partitioning a given system specification into hardware and software parts is a very important system architectural design decision for embedded system design. Since partitioning can be performed at different granularity[EHB93], Ernst, Henkel and Benner classified partitioning schemes into coarse-grain partitioning and fine-grain partitioning.

Gupta and DeMicheli[GM92] used a heuristic to perform fine-grain hardware/software partitioning. Starting from an all-hardware implementation, operations are selected to move into software implementations depending on a number of criteria.

The communication overhead, presence of unbounded delay operations, and/or decoupling of control and execution are examples of such criteria.

Ernst, Henkel and Benner[EHB93] developed a fine-grain partitioning scheme called hardware extraction. A specification is divided into basic blocks which do not have control structures such as branching or loops. The performance of blocks in both hardware and software implementations is estimated. A partitioning is optimized using simulated annealing along with the estimated performance of blocks. In computing the cost function for a partitioning, the delay caused by moving a block from software to hardware is also considered.

Thomas, Adams, and Schmit developed a set of guidelines for coarse-grain hardware-software partitioning [TAS93]. Those guidelines are based on the type of an application, the characteristics of the task function, the static properties of task behaviors, and the amount of custom hardware.

Vahid, Gong and Gajski[VGG94] proposed a heuristic that is based on binary search for hardware-software partitioning. In the proposed heuristic, instead of using a weighted cost function that considers both performance constraints and hardware size simultaneously, the possible range on the amount of custom hardware is divided into a set of ranges and used as a size constraint. The cost function is defined such that it returns 0 if there is at least one partitioning that does not violate both types of constraints. Then, the smallest hardware size is obtained by performing binary search within a possible range of hardware size with this cost function.

Kalavade and Lee [KL94] reported a constructive heuristic which traverses a task-flow graph and maps nodes into hardware or software based on an appropriate objective function.

2.9 Multichip Design

Task-level System Partitioning

Traditionally partitioning a netlist into multiple chips is one of important design tasks in developing a digital system due to the size and pin count constraints

of packages. A better optimization scheme by partitioning a specification at the behavioral level was proposed by Lagnese and Thomas [LT89]. Chen further moved the partitioning problem to the task-level[Che94]. Chen used a Genetic Algorithm and MILP to perform task-level partitioning. Clustering of functional objects to improve the partitioning process was proposed by Vahid and Gajski[VG95]. An interactive partitioning method using a set of process transformation primitives has been proposed[IOJ94].

The Physical Design Style Selection

Although the selection of a physical design style among different ASIC design styles available today is an important step in the beginning of a digital system development, there is no intensive research effort on automating a physical design style selection for VLSI chip design because of the difficulty of quantifying the effect of physical design style selection. Instead, many selection guidelines were published based on the qualitative difference among possible physical design styles [Hol87, HR91, EBCH86]. The efficiency of silicon area usage, performance, and turn-around time are primary trade-off points in selecting a physical design style for a chip.

Datapath Architecture Selection

Chen [Che94] pioneered a datapath architecture selection for a functional task in a specification. Chen used used mathematical programming to find a datapath architecture for each task in a specification such that the overall system cost is minimized while satisfying performance constraints. Later, Kalavade and Lee described the *extended partitioning problem* [KL95] in which several possible implementations were considered that may exist for every mapping of tasks to different implementation styles. The extended partitioning problem is to find a hardware/software mapping, an implementation point (A_i, t_i), and a global schedule for each node such that the user-specified latency is achieved. The problem attempts to optimize the area cost of the implementation. The authors assume the availability of two curves for every node, namely, a hardware implementation curve

and a software implementation curve. These curves are described as sets of tuples of the form (A_i, t_i), where A_i represents the area (or memory) requirement of the i-th implementation and t_i is the execution time corresponding to the implementation. Adams and Thomas partly addressed the issue of datapath architecture selection in the global allocation problem[AT95], where an implementation point is selected from the several alternatives generated by a high-level synthesis program.

System I/O synthesis

Another important design step in a digital system development is the synthesis of the I/O subsystem, in which scheduling of I/O activities and the type of a bus used for the communication among processes are determined. Filo *et al.*. [FKCM93] classified the communication among processes as blocking and non-blocking. Blocking communication implies that two processes engaged in communication wait for one another, whereas in non-blocking communication the sender/receiver continues its operation without waiting for an acknowledgment from its communication partner. Blocking communication corresponds to handshaking and non-blocking corresponds to synchronous communication. Filo *et al.*. proposed an algorithm which optimizes the interface by reducing the amount of blocking communication. PUBSS is a system to generate a relatively scheduled I/O description called Behavioral FSM from a specification in VHDL and then solves a set of linear equations to minimize the handshaking in communication[WM93]. Narayan and Gajski[NG94] proposed heuristics which find the width of a bus that serves as the physical path for a set of interface communication channels. Filo et al. [FKCM93] proposed an algorithm which optimizes the interface by reducing the amount of blocking communication.

Die clustering and MCMs

The emergence of MCMs as a package alternatives, it has been noticed that exploiting the advantages of MCMs over other packaging methods requires a good partitioning scheme. Partitioning a system into multiple dies for maximal utilization of MCMs is attempted at different abstraction levels.

There are trade-off studies comparing the cost and performance of a single big chip to that of several smaller chips packaged into an MCM. In the report by Dehkordi *et al.*.[DRB+95], an exhaustive partitioning method was used to find an optimal partition of the SUN MicroSparc system. The partitioned system with MCMs is considerable cheaper than the existing single design. Another similar trade-off study by O'Brien *et al.*. [OHK92] showed the cost advantage of MCMs over a single chip by partitioning an IBM RS/6000 system into a set of 8 chips. Both results show that task-level partitioning and package selection are the main factors in determining the total system cost and that such decisions should be considered in early design stages. Shih *et al.*. [SKT92] proposed an algorithm for structural partitioning of a system graph (a graph of interconnected combinational blocks and registers) into chips on a MCM. Though it is not directly developed for MCMs, Chen [Che94] used mathematical programming and a genetic algorithm to partition a system specification into a number of packages to minimize the cost of a system. Khan and Madisetti[KM94, KM95] used quadratic non-linear programming to partition a system for MCMs for yield consideration and low power. In their formulation, the yield is fixed to a given value and consequently limits the size of dies which can be used in a system.

2.10 Comparison with Previous Research

Work related to hardware-software codesign by Gupta *et al.* [GM92] aims at the rapid prototyping of embedded systems. Their main focus is on hardware-software partitioning, software module generation, and interface synthesis. In hardware-software partitioning, multiple implementations of tasks were not explicitly considered. Many issues related to the portions of the system designed in hardware are not addressed, e.g. selection of implementation style, partitioning of tasks into chips, and selection of packages which influence the system cost and performance. The work presented in SIERA [SB95] is not an automation tool, but a system design methodology developed from long experience in designing real-time digital systems. The most important steps such as hardware-software partitioning are done manually. The target architecture is a hierarchical bus-based system. After

the given system model is partitioned into architectural templates, the modules for hardware, software and hardware-software interfaces are synthesized using existing CAD tools. The methodology supports task-level partitioning. PTOLEMY [KL93] aims at developing a heterogeneous system design environment which provides the designer with tools to explore the tradeoffs involved in design. Therefore, the main emphasis is to develop simulation and translation tools that integrate different models of system design. In the design methodology of PTOLEMY, a designer can verify his or her design decisions with the available tools. SOS [Pra93] and ProPart [Che94] of the USC project are similar to this work in their use of mathematical programming to solve the multiple facets of the system-level synthesis problem concurrently. The main focus of the SOS system is to synthesize a heterogeneous multiprocessor architecture, to find a global schedule for the tasks, and to allocate the tasks to processors so as to maximize the performance and minimize cost. SOS assumes either point to point interconnection or a bus-based multiprocessor architecture. In SOS, each task is realized in hardware or software on one of the processors of the target architecture. Our work addresses style selection, task-level system partitioning and packaging for full hardware designs. ProPart [Che94] performs task-level system partitioning and package selection, but the main objective is to optimize the system in terms of cost assuming a homogeneous implementation style for all tasks. In our work, we not only handle heterogeneous physical design styles for tasks, we use the time-to-market as the objective function. Interface configuration was not fully addressed by Chen[Che94], and multichip modules were not considered in package selection by Chen[Che94]. The cost model of a system used in ProPart does not include the effect of yield and therefore, a bigger chip is always preferred over a smaller one, unless the size of the chip is not limited by the cavity size of the available packages. The distinguishing features of our work from other system-level design automation work in the literature are the following: (1) We consider the non-design issues such as physical design style, yield, packaging, and prototyping time in the early phase of system development, (2) We focus on automating the optimization steps by providing a mean for trade-off analysis rather than automating synthesis step, and (3) We solve the subproblems of system-level design concurrently, without isolating them.

Chapter 3

The Multi-Chip Design Problem and Model

In Chapter 1, we described the system-level design problem and proposed a design methodology to reduce the search space rapidly and systematically. In this section, we introduce the multi-chip system design problem as a special case of the system-level design problem. We start with a real example to describe the multi-chip design problem and describe the associated design steps. Then a description of our first-order analytical model that captures the multi-chip system design problem is presented.

3.1 The Multi-Chip Architecture Design Problem with an Example

Example 4.1

In this hypothetical system development project, the project goal is to build an MPEG-1 encoder board within 6 months and the cost of each unit should be less then $500. The specification and performance requirements of the system are given in the MPEG-I standard[Gal91]. The specification is captured graphically and algorithmically in the task-flow graph shown in Figure 3.1 as a set of communicating tasks which are written in VHDL and verified with an HDL simulator. The available physical design styles for building chips are FPGA, gate array, standard cell style, and commercial off-the-shelf (COTS) devices. For the FPGA

design style, available die types provided by a vendor are 3K-gate, 5K-gate, and 10K-gate while die types provided by a gate array vendor are 5K-gate, 10K-gate, and 15K-gate. For the standard-cell style, a vendor provides a cell library. If necessary, MCMs can be considered for packaging dies and there are a number of package types available for dies.

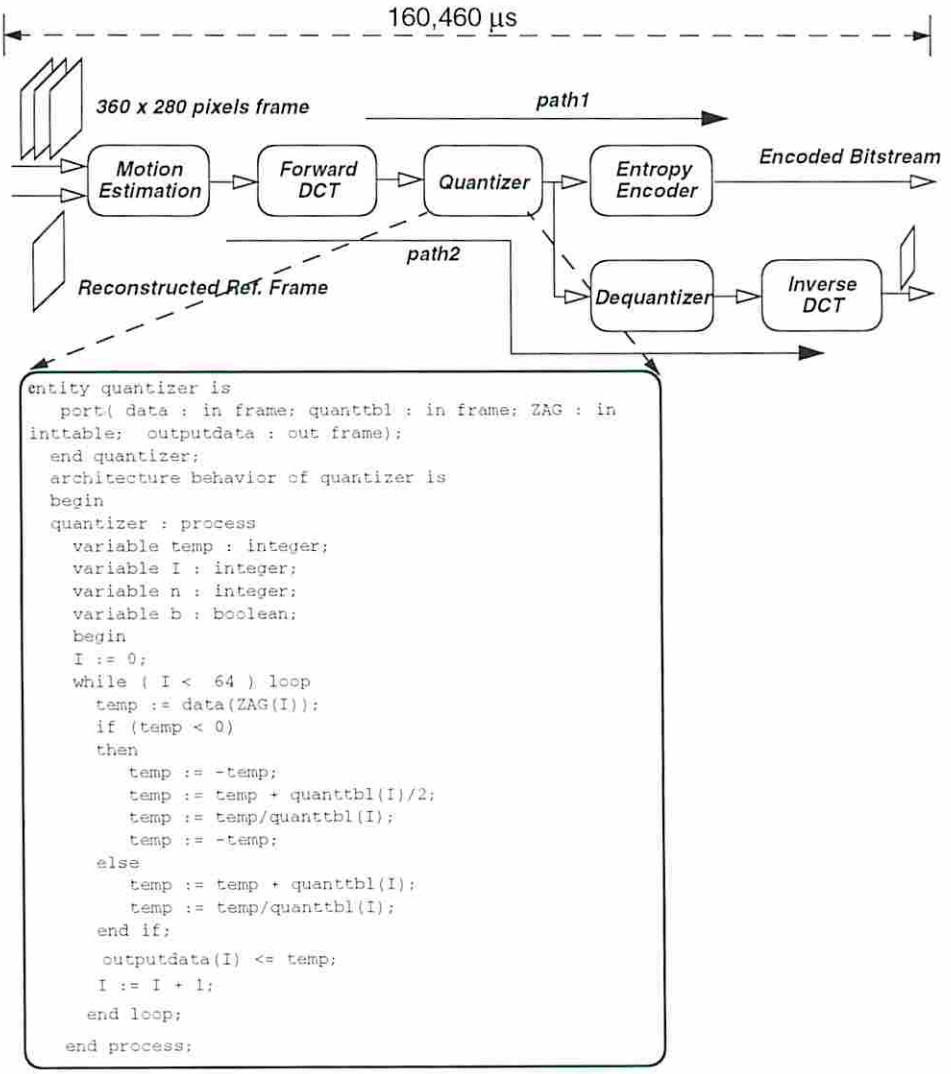


Figure 3.1: A specification for the MPEG encoder.

An example design which meets this specification is shown in Figure 4.4.

System design consists of a following set of activities which cannot be ordered to achieve optimal results because the decisions interact.

1. *physical design style selection* in which a physical design style for each task in the specification is selected from the given design styles such as FPGA, gate array, standard cell, and COTS,

2. *architecture selection* in which a datapath architecture for the behavior of each task is selected,

3. *task-level system partitioning* in which the entire set of tasks is partitioned into groups that will be implemented as separate dies,

4. *die selection* if the task partition is implemented with either FPGA or gate array style, a die type such as 10K-gate die is selected for each task partition from a given die library in die selection,

5. *die clustering* in which dies are partitioned into groups that will be packaged separately,

6. *substrate technology selection* in which a substrate technology is selected for each die cluster if the die cluster is implemented with MCMs,

7. *package selection* in which a package type that houses each die cluster is selected from a given package library,

8. *channel partitioning* in which channels are partitioned into groups such that channels in a group share a single physical bus,

9. *bus selection* in which the bus type for each channel partition is selected, and

10. *task scheduling* in which the ordering and overlapping of execution of tasks are determined.

The next sections describe each of the activities as an individual step. We describe their interaction later in Section 3.2.

3.1.1 Physical Design Style Selection

Before starting a design, the designer should decide what physical design styles should be used for the system. Understanding the impact of different physical design styles on the system cost, performance, power, and prototyping time is important in making a correct physical design style selection. The 4 physical design styles which we are considering in this thesis are FPGA, gate array, standard cell and COTS. Figure 3.2(a) shows the comparison of prototyping time of the same design in different styles while Figure 3.2(b) shows the difference of max clock frequencies and costs of systems implemented in different styles.

The development time of a multi-chip digital system can be roughly divided into seven components - logic design time, test generation time, layout time, mask generation time, fabrication time, packaging time, and fault testing time. In terms of design time, which is composed of the first three components, logic design time, test generation time, and layout design time, there is not a significant difference between physical design styles (except COTS) if we assume the use of CAD tools since the complexity of designing logic circuitry depends on specification more than on physical design style. However, prototyping time varies with the selected physical design style. The FPGA style has a strong advantage in prototyping time. Because of the programmable architecture of FPGA devices, they do not require mask generation, fabrication, packaging, or testing[1] Although the gate array style cannot avoid the above prototyping steps, it still takes less time for mask generation and fabrication than the standard cell style since considerably fewer masks and fabrication steps are required. In case the design process should be repeated for some reason, the short prototyping time of the FPGA style gives FPGA implementations a strong advantage.

The cost of a chip is composed of non-recurring engineering (NRE) cost and material cost. NRE cost such as design cost and mask generation cost should be

[1] There are two classes of FPGAs. The Xlinx class, which are reprogrammable, can be tested in advance. The Actel class, which are one-time programmable, cannot be tested.

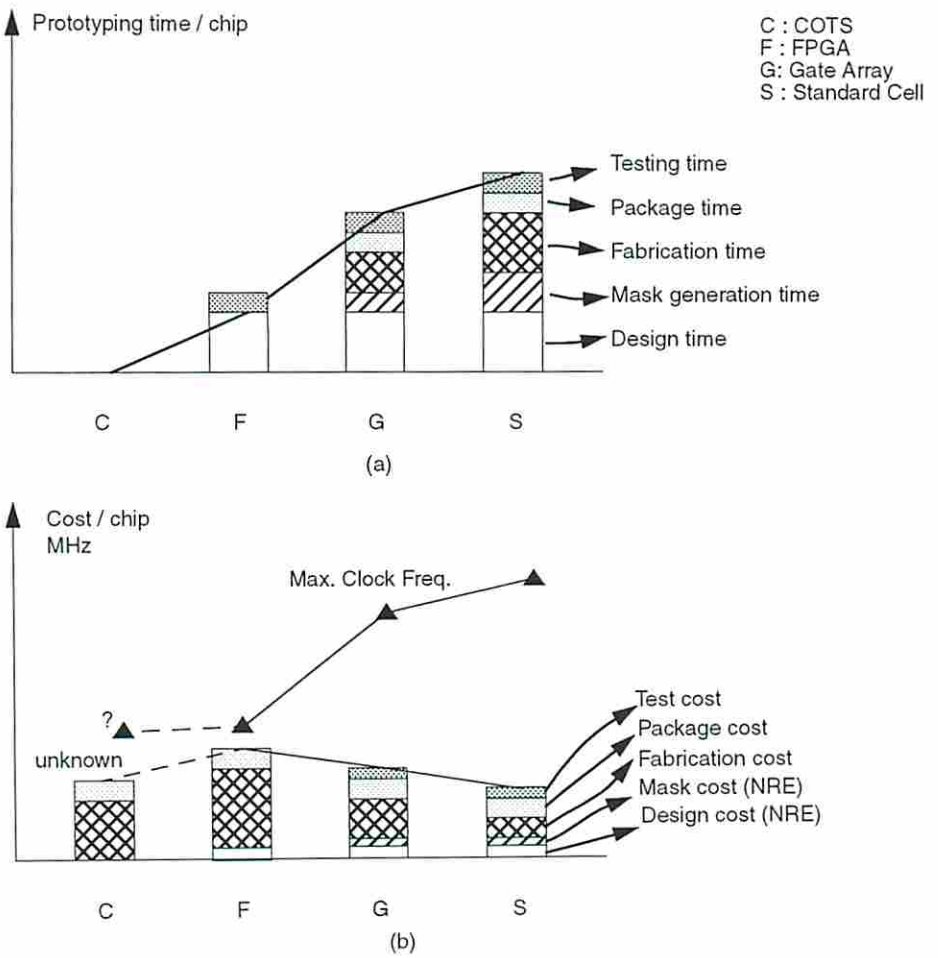


Figure 3.2: Comparison of different physical design styles. (a) Prototyping time (b) System clock frequency and cost

amortized, while fabrication cost, packaging cost, and testing cost constitute the cost of a chip as follows:

$$c = \frac{NRE}{n} + m \tag{3.1}$$

where c is the cost of a chip, and n is the number of dies produced. m is the material cost for manufacturing a chip, which is modeled by Murphy[Mur64] as follows:

$$m = \frac{C_f S_d}{Y_p Y_t Y_w} + \frac{C_t}{Y_p Y_t} + \frac{C_p}{Y_p} \tag{3.2}$$

where C_f is the cost of wafer processing for unit area, S_d is the size of a chip, C_t is the cost of testing a chip, C_p is the cost of packaging a chip, Y_w is the yield of wafers, Y_t is the bare die yield, and Y_p is the packaged chip yield.

Since FPGA chips are mass produced, the cost of NRE for each FPGA die is negligible. For gate array and standard cell design styles, the effect of NRE on die cost will vary depending on the volume of production n. The material cost of a die is mainly determined by the size of a die. The gate density of a die is low for an FPGA compared to a gate array or standard cell die because of the programmable interconnection and logic architecture of FPGA chips. Low gate density of FPGA dies makes the same design consume more silicon area than other physical design styles. Gate array dies have higher gate density than FPGA dies. Standard cell styles offer highest gate density of the styles considered. Therefore, it can be said that standard cell implementation becomes more cost effective as the number of system units produced increases.

In terms of performance, the FPGA style is the slowest physical design style, again because its programmable architecture causes longer delays in interconnection and logic. Gate array style designs in general have lower performance than standard cell designs for which optimized cell libraries are used. This is shown in Figure 3.2 (b).

The characteristics of COTS chips cannot be generalized, except that they do not require any chip prototyping time since an existing design is being reused.

Therefore, although it is desirable to use COTS chips for rapid prototyping, the effect on system performance and cost should be carefully evaluated.

So far, we have implicitly assumed that all chips in a system are implemented in a single physical design style. Frequently, a system design can be further optimized for a selected metric if we allow mixture of physical design styles in a system. Hardware/software co-design is a well-known example of mixed-style implementation of a system. Mixed-style implementation can be a good alternative when conflicting requirements cannot be satisfied with a single physical design style. For example, a system can be implemented using both FPGA and gate array design styles if an FPGA implementation alone cannot satisfy the performance constraints but the prototyping time required for gate-array implementation is not tolerable. A real example of such a mixed design with standard-cell and gate-array design is given in [FLS+92]. In such a mixed-design-style implementation, the number of possible style selections for T tasks from s styles is given by s^T. For example, if there are 4 physical design styles and 10 tasks, the number of possible style selections is $4^{10} \approx 1 \times 10^6$.

3.1.2 Datapath Architecture Selection

A given algorithm can be implemented through a number of different datapath architectures, depending on the decisions involved in synthesizing a structural representation from the algorithm. Therefore, the designer has to select a datapath architecture for each task such that the overall characteristics of a system are optimized. Knowing the characteristics of different datapath architectures in the design space is very important to make correct design decisions.

Jain, Park, and Parker[JPP92] showed that the following relationship exists between the cost and performance of different pipelined architectures for a specification.

$$(AT)_{lowerbound} = constant$$

where A is the functional area required to implement a system, T is the initiation interval for the pipelined design or is the length of the schedule for the non-pipelined design, Graphically, the relationship is shown in Figure 3.3.

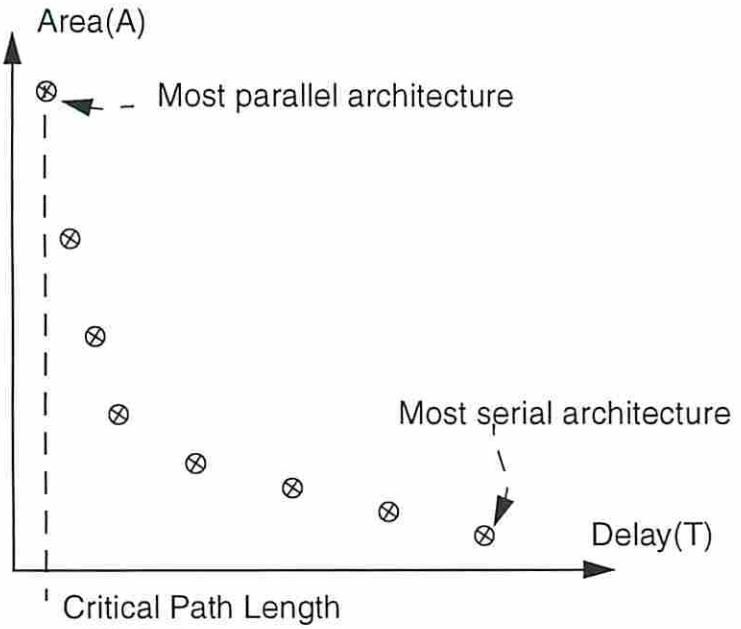


Figure 3.3: Area-delay trade-off of data path architectures

However, merely selecting an optimum solution for each step may not achieve a valid optimum solution. For example, if the designer wants to minimize the system cost, selecting minimum cost architectures for all tasks could result in the cheapest design but might violate the performance constraint. In such a case, we should select one or more tasks which are implemented with faster architectures while minimizing the system cost.

The number of possible architectures A assuming the chip design style selection has occurred is given by

$$A = \Pi_{t=1}^{T} I_{t,s} \tag{3.3}$$

where $I_{t,s}$ is the number of datapath architectures for task t in selected style s and T is the number of tasks. For a system with 10 tasks and 4 datapath architectures per task, there are $4^{10} \approx 1.0 \times 10^6$ possible architecture selections.

3.1.3 Task-Level System Partitioning

Although today's semiconductor technology can put several million gates on a single chip, the complexity of systems is increasing as fast as the integration level. As a result, there is still a frequent need to partition complex system into several chips because of the limited pin count or the physical size of package and/or die types. Although a system can be partitioned at any level of abstraction[Joh96], it is desirable to partition it at the task level because functional boundaries are preserved, which makes a design modular, better testable, and more re-usable.

Besides the evident physical size limitation, there are other reasons for task-level system partitioning. Too much integration may not be cost-effective because the yield drops faster than the area. For example, Dekhordi *et al.* showed that the overall cost of a CPU can be lowered by a factor of seven if the design is properly partitioned[DRB+95]. The cost and yield of a chip are given as follows:

$$f = \frac{C_f \cdot DieSize}{Yield} \tag{3.4}$$

$$Yield = \frac{1}{2 \cdot D_0 \cdot DieSize} \tag{3.5}$$

where f is the bare die cost, C_f is the cost per unit area associated with the fabrication facility, D_0 is the number of defects per unit area, and $DieSize$ is the size of a die. Equation 3.5 is based on the non-linear yield model along with the assumption of rectangular defect density distribution [Ber83].

The system performance also changes with task-level system partitioning because the on-chip wiring delay is usually shorter than off-chip wire delays. Therefore, more partition boundaries inserted in a path result in slower performance.

The complexity of the unconstrained task-level system partitioning problem can be computed as follows: When the number of task partitions is known, the k-way partitioning of tasks can be modeled as a combinatoric problem of partitioning

T distinguishable balls into k indistinguishable slots without allowing an empty slot. This number is known as the Stirling Cycle Number, which is given as follows[Zwi95]:

$$f(T,k) = \frac{\sum_{i=1}^{k}(-1)^{k-i}\binom{k}{i}i^T}{k!} \tag{3.6}$$

The number of ways to partition 10 tasks into 3 chips is approximately 9,330. When the number of partitions is not known, the total number of possible ways of partitioning is given as follows:

$$\sum_{k=1}^{T} f(T,k)$$

For the above example, the total number of possible task-level system partitioning is 115,975. The partitioning problem is actually somewhat different from the unconstrained partitioning problem. Tasks can only be grouped together if they are implemented with the same physical design style. We address this problem later in the next section.

3.1.4 Die Selection

When a selected physical design style for a set of tasks assigned to a partition is either FPGA or gate array, the designer has to select a die type for the partition since the range of sizes of FPGA or gate array dies is a set of discrete values. With discrete size die types, it is difficult to fully utilize the pins and gates on a die type. Such underutilization of resources also contributes to the higher cost of FPGA or gate array design than standard cell design. Figure 3.4 illustrates the utilization of pins and die areas of different physical design styles. The solid rectangles show the available die types of an FPGA or gate array style while the dotted rectangle indicates the pins and size of a standard cell implementation. A standard cell implementation can always fully utilizes resources except in the case of I/O constrained designs (The dot inside the shaded area shows an I/O constrained design which can be no smaller than the point marked with an X.).

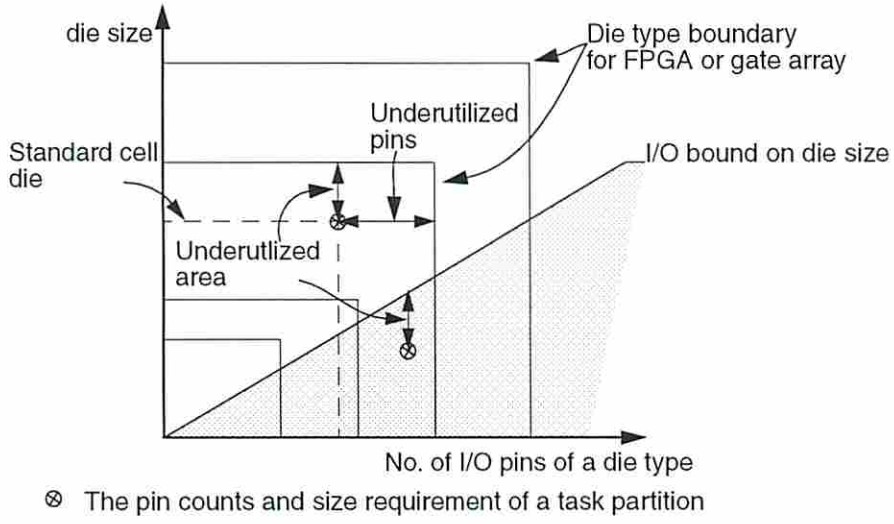


Figure 3.4: The utilization of die resources of different physical design styles

Task-level system partitioning and die selection have cyclic dependencies. If either one of them is known, the problem becomes much simpler. If task-level system partitioning is carried out before die selection, the number of pins and sizes of task partitions are known and a greedy approach like *best fit* algorithm solves the die selection optimally. But the overall utilization of dies might not be optimal.

If die selection is done first, the number of dies and their sizes are known and the problem becomes the well-known partitioning problem under size and pin constraints. The advantage of this approach is that task-level system partitioning can be done such that the given dies are maximally utilized. However a different die selection might yield a better partitioning, and so the two design steps are mutually dependent.

3.1.5 Die Clustering and Package Selection

A package type should be selected to house each die. Selecting the right packaging strategy is important since the cost of packaging comprises a considerable part of

the total system cost. System design should be conducted such that the resulting design fits into available package types. Power dissipation, cavity size, and I/O pin count are the physical constraints that determine the selection of a package type. Deciding on the packaging strategy for a system is not a trivial task, owing to the increasing number of packaging options ranging from the plain PWB with separately packaged chips to surface mounting technology(SMT) to Multi-Chip Module(MCM) technology[Tum92, Sch92, SA95]. Good package design in general can result in considerable cost, performance, and physical savings in a system[OHK92, Siu92]. O'brien *et al.*[OHK92] showed with a package estimator that different packaging options result in diverse performance and size of a system. The estimated performance and physical size with various packaging technologies for the IBM RISC System/6000 processor are shown in Figure 3.5[OHK92].

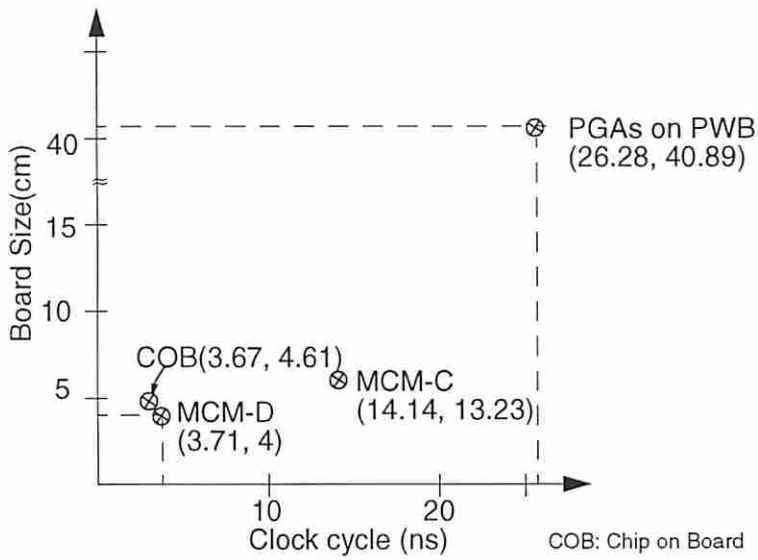


Figure 3.5: Estimated board sizes and clock cycles for 8-chip set of the RS6000 processor[OHK92]

MCMs which can package multiple chips in a single package are packaging alternatives for both the traditional single chip packages on PWBs as well as single monolithic chips. The ability to package multiple chips in a single package offers the possibility of lowering cost with proper partitioning over a large monolithic chip as

we already have discussed in Section 3.1.3. Packaging chips of mixed physical design styles is possible using MCMs[Tum92]. The tight integration of chips directly on boards without soldering pins increases the reliability of the system. The MCM technology also improves the system performance because of lesser parasitics due to the fine conducting lines. Finally, the area bonding technique can reduce the die size by removing the big output drivers and input protection, and increase the utilization of silicon area by eliminating I/O bound designs such as those shown in Figure 3.4. MCM cost strongly depends on the capability of fully testing bare dies which is a difficult problem called a *known good die* (KGD) problem[HW92]. Also the high density of gates in a smaller area requires efficient heat removal. Finally, the fine spacing between signal lines could worsen the crosstalk.

Introduction of MCMs as a possible packaging option adds another design step called *die clustering* in which dies are grouped into clusters for an MCM package. MCMs in general improve performance compared to PWBs. However, the effectiveness of using MCMs for cost improvement relies on good die clustering and even further task-level system partitioning.

A cyclic dependency exists between die clustering and package selection just as for task-level system partitioning and die selection. Similar characteristics which we observed in the previous section for task-level system partitioning and die selection exist for this pair of steps. By selecting the number of packages and their types first, we can reduce the design space significantly, but the feasibility of such package selection is not known. Clustering the dies first and selecting package types may lead to a waste of resources.

The complexity of the unconstrained die clustering step can also be expressed with the Stirling Cyclic Number given in Section 3.1.3.

3.1.6 Substrate Technology Selection

Just like selecting physical design styles for tasks plays major role in determining the characteristics of a task partition, selecting a substrate type for a die cluster determines the cost and performance of a die cluster[Hig92]. There are basically three major types of MCM technologies, namely MCM-D, MCM-C, and MCM-L.

Characteristic	MCM-L	MCM-C	MCM-D
Maximum Wiring (cm/cm^2)	300	800	250-750
Minimum Line Width (μm)	60-100	75-100	8-25
Line Space (μm)	625-2250	50-450	25-75
Maximum No. Wiring Layers	46	63	8

Table 3.1: Characteristics of different MCM technologies

Chips are placed on a laminated board which has multiple wiring layers in MCM-Ls. Wiring layers are printed inside a ceramic substrate on which chips are placed in MCM-Cs. Wiring layers are constructed by depositing thin-film materials for insulators and conductors in MCM-Ds[Lic95] In Table 3.1.6, the characteristics of different MCM technologies are shown which are summarized by Tummala[Tum92]. Selecting MCM-L technology reduces the cost but increase the system size. The performance improvement using MCM-Ls is limited. On the other hand, MCM-D technology provides very high performance and high density interconnection at a high cost.

3.1.7 Channel Allocation and Bus Selection

Recall that edges in a task flow graph are called communication channels. The communication edge in a task-flow graph must be implemented with a set of wires collectively called a *bus* in hardware. The data to be exchanged among tasks is converted into the proper format by I/O circuitry. Since the amount of data that can transferred over a bus is limited by the width of a bus, data has to be transferred using multiple transactions if the amount of data is larger than the bus width. Besides formatting data and driving the buses, the other important task of an I/O circuit is to ensure the arrival of data. This can be achieved using one of two commonly used protocol types, namely synchronous or asynchronous communication[Hay88]. In synchronous communication, the transaction is timed

and the source and destination devices are synchronized by a single clock source. In asynchronous communication, the source and destination devices coordinate the transaction using control signals. Therefore, a bus can be characterized by three parameters, namely bit-width, clock cycle, and protocols.

Bus selection affects the performance of a system, the I/O pin counts of task partitions and die clustering, and the size of a substrate or a board. On the other hand, the task-level system partitioning, die clustering, and substrate selection determine the maximum bus clock because of different parasitic capacitances of different substrates. Assuming that the parallel plate capacitance model is applicable, the capacitive load can be expressed as $C_b = W \times L \times C_u$ where C_b is the capacitance of a wire in a bus, W is the width of a wire in the bus, and L is the length of a wire, and C_u is the capacitance per unit area. A simplified equation for C_u can be $\frac{\epsilon}{d}$ where ϵ and d are the dielectric constant and thickness of insulators respectively. The length of the wire L depends on the substrate on which the wiring is done; using an MCM-D substrate will lead to shorter wires, whereas using an MCM-L or regular PWB will lead to relatively longer wires. Similarly, C_b depends on the substrate type on which the wires are laid out, whether the wiring is on-chip or off-chip.

Since buses are expensive commodities, often, the designer tries to reduce the number of buses. Yet, when buses are shared, there is a possibility of a "collision" i.e. two transactions requesting the same bus at the same time. The exact timing of a transaction depends on the scheduling of tasks and bus selection, which we discuss in the following section. If the execution time of a task is not known because of data-dependent execution or due to some asynchronous input event, a bus arbitrator is required to avoid collision among two transactions. If all bus transaction can be predetermined, as for most DSP applications, a bus can be shared without an arbitrator. If the time interval of a bus transaction over one channel does not overlap with the time interval of the transaction on another channel, the two channels can be mapped onto the same bus so that they can share a bus. A problem in which such sharing of buses among channels is done to minimize the number of buses is called *channel allocation.*

3.1.8 Task scheduling

Given the system specification shown in Figure 3.1, the exact time span for each task and the relationship of one time span to another is to be determined. Scheduling depends on a number of factors: (1) the ordering among tasks, (2) the length of a task execution, (3) whether two tasks execution can be overlapped or not, (4) whether two tasks can share a same hardware module, and (5) the mode of system operation.

The ordering among tasks is given in the specification. The length of a task execution can vary with the input data and the selected datapath architectures. The overlapping of execution of two tasks which have a data dependency among them also depends on the selected datapath architectures. If a hardware module can be shared by more than one task, this would influence the time of execution of a task which shares hardware with other tasks. A specification shown in Figure 3.1 can operate in either non-pipeline mode or pipeline mode. In non-pipelined operation, an instance of data cannot be initiated until the processing of the previous data instance is completed. In pipelined operation, a new data instance can be initiated before the previous data instance processing is completed. By designing a system in non-pipelined mode, we can obtain a cheaper design in general because more sharing of hardware is possible between different control steps. On the other hand, pipelined scheduling produce a high throughput system with additional hardware by allowing overlapped processing of data instances.

Considering a combination of the above factors, different schedules of task executions are possible. Finding a minimum execution delay or initiation latency for a given cost constraint is the problem of *task scheduling* in system-level design.

3.2 A Multi-Chip Design Problem and A Model

In the previous section, we described informally design steps at the system-level and their characteristics. In this section, we state the multi-chip system design problem formally and further explain a model that captures the relationships among design steps quantitatively.

3.2.1 Specification and User Specified Constraints

In this thesis, we model the behavior of a digital system as communicating multi-processes that run concurrently in time and represent it with a *task-flow directed hyper-graph*, $G(E, V)$, which consists of a set of task nodes V and a set of communication edges E as shown in Figure 3.1. A task node t represents a computational function in which the behavior of a task can be described algorithmically with a hardware description language and its functionality is abstracted by an associated name $TFTYPE_t$. For example, the functionality of quantizer task in Figure 3.1 is given $TFTYPE_t$=*quantizer*.

Communication among tasks is expressed as passing data from one task to another. The data can be primitive data-types such as signal, bit, and integer or composite data-types such as vector, multi-dimensional array and even aggregated data. Such communication among tasks is represented with a directed hyperedge, since a hyperedge can capture naturally data transfer from one source to destination tasks. The hyperedge is characterized by the associated volume of data VOL_e which is the amount of data that must be transferred from a source task to destination tasks through channel e after the completion of the source task.

The execution model for the task-flow graph is taken from Prakash[Pra93] i.e. a task can start execution without requiring all input data to be available and can output the results before completing execution.

The user can specify constraints on system metrics, cost, performance and the prototyping time. The cost is the price of a system unit if n copies of a prototyped system will be produced. The performance constraints are given as the processing time for an instance of data. The exact definition for the processing time will be given in Section 3.3.10.

3.2.2 Library

In our model, a library represents a set of existing choices for a design step. The structure of a library can be thought of a table in a relational database, in which an element is represented with a tuple.

A *physical design style library* is a set of different physical design styles given as follows:

$$DSL = \{q_0, q_1, \ldots, q_Q\}$$

where physical design style q is characterized by the name $SNAME$, the NRE cost, and the performance scaling factor PS, which will be explained in Section 3.3.8. In this thesis, we assume that the supported physical design styles are $SNAME_0$ =FPGA, $SNAME_1$ =gate array, $SNAME_2$ =standard cell, and $SNAME_3$ =COTS.

The *substrate technology library* for MCMs is given as follows:

$$ST = \{st_1, st_2, .., st_S\}$$

where substrate technology st is characterized by the number of signal layers $NSIG$, the via grid spacing VGS, the number of lines between vias LBV, the performance scaling factor SPS, and the manufacturing cost of a unit area $UCOST$.

An implementation i in *implementation library* I can be characterized by the size $ISIZE$, delay $IDELAY$ and style $ISTYLE \in DSL$. The implementation library is denoted as a set of implementations and expressed as

$$I = \{i_1, i_2, .., i_I\}$$

. In the same way, *die library* D, *package library* K, and *bus library* B are expressed as follows: The die library is expressed as

$$D = \{d_1, d_2, .., d_D\}$$

where d is a die type characterized by the size $DSIZE$, the I/O pin counts $DPIN$, the cost $DCOST$, the style $DSTYLE$, the fabrication time $FTIME$, the average gate size $AGATESIZE$, the average I/O pad size $PADSIZE$ and $WB = 1 + \frac{\text{routing area}}{\text{function area}}$.

The package library is expressed as

$$K = \{k_1, k_2, .., k_K\}$$

where k is a package type characterized by the cavity size $KSIZE$, the I/O pin counts $KPIN$, the cost $KCOST$, the packaging time $KTIME$.

The bus library is expressed as

$$B = \{b_1, b_2, .., b_B\}$$

where b is a bus type characterized by the bus type $BTYPE \in$ {on-chip, on-module, on-board}, the clock cycle $BCYCLE$, and the bus width $BWIDTH$. The protocol for buses in the bus library used in this thesis is synchronous.

3.2.3 Target architecture

The target architecture for the multi-chip systems we are considering is defined as follows:

Definition 1 *A* **die** *is defined as a set of functional blocks connected by a set of buses and implemented in a silicon substrate. A* **chip** *is composed of a set of dies connected by a set of buses and packaged by a package. A* **hierarchical bus-based multi-chip system** *is composed of a set of chips interconnected by a set of buses that performs the given system behavior specification in task-flow graph* $G(V, E)$.

A hypothetical example of the target architecture is shown in Figure 3.6. Circles represent abstract design entities such as task or die clusters while boxes represent physical implementation of abstract design entities such as selected datapath architecture or package types. As the system specification can be represented with a task-flow graph, the target architecture can be represented as a tree as shown in Figure 3.7. The leaf level is the task level, where an implementation (a datapath architecture in a physical design style) is chosen for each task. At the die level, tasks are grouped into sets. A die type for each task set is chosen, channels are

grouped into sets and a bus type is selected for each channel set. The task partitioning determines whether a channel will be on-chip or not. At the chip level, dies are clustered and a package type for a die cluster is selected while the die clustering determines whether a channel which is not on-chip will be on-module or on-board. Finally a system is packaged into a board and a set of buses.

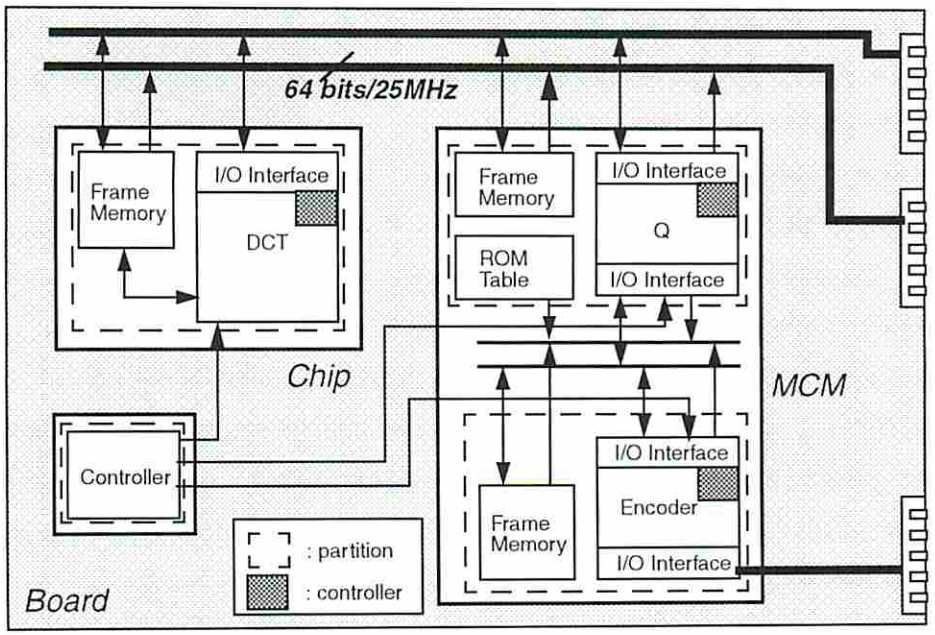


Figure 3.6: A hypothetical example of a multi-chip system architecture

The execution of a target architecture can be either non-pipelined or pipelined. Coordination among different task executions can be controlled by a centralized controller which initiates and terminates the execution of tasks and the data transfer. Another possible scheme for system control can be the asynchronous mode of operation in which a task initiate the subsequent task by indicating its termination.

In order to simplify the problem, all communications are assumed to be synchronous in our prototype system.

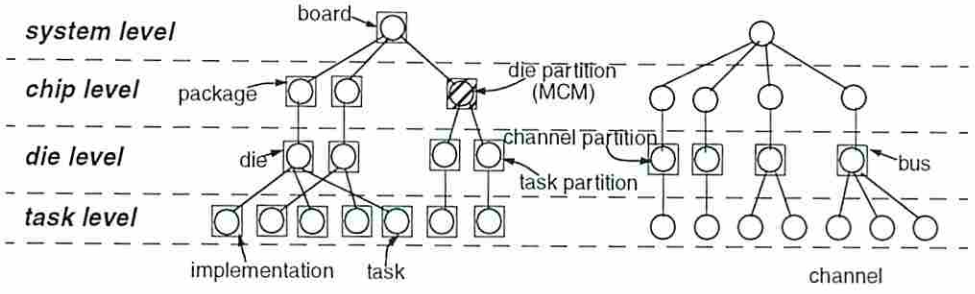


Figure 3.7: A tree representation of a multi-chip system

3.2.4 The Problem Statement

Definition 2 *A* **valid design** *at the system level is a complete set of design decisions for steps which are manufacturable. An invalid design at the system level is a set of design decisions for steps which are not manufacturable.*

Definition 3 *A* **feasible** *design at the system-level is a set of design decisions which is valid and also satisfies following conditions; Let m_0 be objective metric while $m_1, \ldots, m_n$ are constrained metrics. Let $c_1, \ldots, c_n$ are upper bound constraints on corresponding metrics. When $c_i - m_i \geq 0$, a solution is feasible for m_i. A solution is said to be feasible if it satisfy following condition:*

$$c_i - m_i \geq 0, i = 1, \ldots, n$$

Definition 4 *An* **optimized** *design at the system level is a set of design decisions which is feasible and also optimizes the objective metric associated with a design.*

Definition 5 *A* **multi-chip architecture design problem** *is to find a number of near-optimum designs for a given task-flow-graph representation of a system behavior using a set of libraries for physical design styles, substrate technology styles, datapath architectures for each tasks in a given specification, dies, buses,*

and packages by solving **physical design style selection, architecture selection, task-level system partitioning, die selection, die clustering, substrate technology selection, package selection, channel partitioning, bus selection,** *and* **task scheduling.**

The complete model for the multi-chip system design problem should include all the above design steps but the complexity of the model including all subproblems is too high. Therefore, we make the following assumptions to simplify the problem:

1. each task is implemented separately, and

2. each channel is implemented separately.

The above assumptions mean that no hardware resource such as a functional block or a bus is shared among tasks or channels. The first assumption can be justified for system-level design because few tasks perform the same function at the system level. Sharing of buses can reduce the number of pins but could increase the length of buses. For some types of MCM technologies, a few hundreds of pins can be provided easily without increasing the die size. For designs with such MCM technologies, short wiring length can be of more concern than the number of pins. The above assumptions make *channel partitioning* unnecessary and *task scheduling* trivial.

3.2.5 The Problem Approach

As outlined in Chapter 1, our system design methodology progresses in three steps. In the first part, we reduce the design space to a number of near optimum designs with optimization tools. In the second step, the designer improves the design interactively with what-if analysis tools. The final round of candidate designs are further verified with the existing detailed estimator for various system metrics[Kur91, HOK92].

Both *optimization tools* and *what-if analysis tools* require a mechanism that checks the consistency among design decisions and therefore, the validity of a design. For this purpose, we need a model that not only captures the relationship

among design steps at the system-level but also estimates various metrics associated with the design entities in a system. As we proposed before, in order to reduce the time for the first step which searches a vast design space, we need a very fast way to estimate metrics as a function of design decisions. Therefore, we developed *a set of analytical formulae* which can translate design decisions into various metrics of design entities in a system. Then the validity of a design can be checked using these estimated metrics.

For the optimization process of design decisions, we decided to use general optimization techniques. In this work, we used both **MILP** and a **Genetic Algorithm**.

3.3 A First-Order Analytical Model for Multi-Chip System Design

We start building a model by modeling each design step as a decision problem and define the relationships between steps. Then, for each design step, we define functions to compute metrics of design entities and define general validity constraints for design decisions.

3.3.1 Modeling a Design Step and Its Relationship to Other Steps

As we can see in Figure 3.7, the design steps in our model can be classified into two types, namely *partitioning steps* and *selection steps*. Both types can be thought of a decision problem in which a selection is made from a finite number of choices. For a selection step, choices are given as elements in a library while for a partitioning step, choices are the partition assignments. Both types of steps can be thought of as finding a mapping f from the domain set S to the range set T as shown in Figure 3.8(a).

$$f : S \rightarrow T$$

Such a mapping can be represented as a set of binary variables $Q = \{q_{st} \mid q_{st} = 1 \text{ if } t = f(s) \text{ and } q_{st} = 0 \text{ if } t \neq f(s),\ \forall s \in S, \forall t \in T\}$. In order to ensure f to be a function (mapping), it should satisfy the following condition:

$$\sum_{t \in T} q_{s,t} = 1$$

We defined such binary variables for each step, z_{ti} for the physical design style selection, y_{pt} for the task-level system partitioning, x_{dp} for the die selection, s_{cp} for the die clustering, m_{sv} for the substrate technology selection, w_{kp} for the package selection, and u_{be} for the bus selection. Then a design A at the system level can be represented as a set of sets of the above binary variables $\{Z, Y, X, S, M, W, U\}$. The exact definitions of the above binary variables will be given in the following sections.

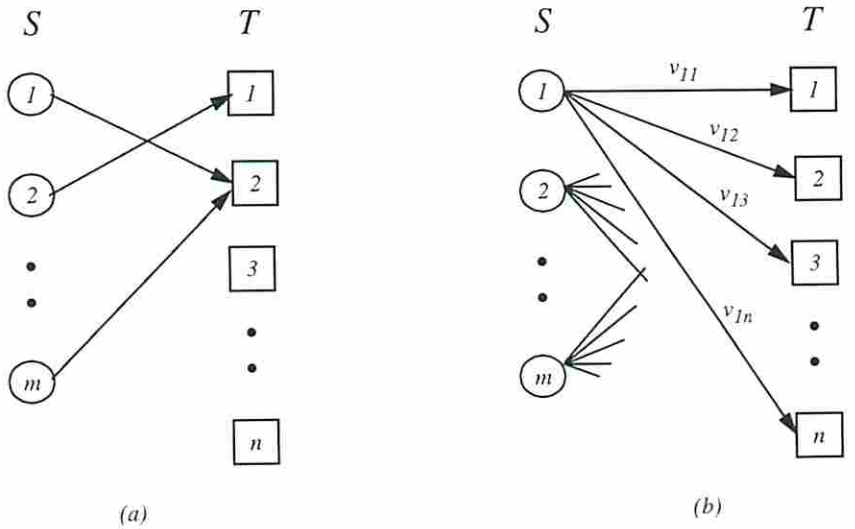


Figure 3.8: A model for a design step as a mapping with binary decision variables

With such binary variables, we can express the mapping function in linear form.

$$t = f(s) = f(s, Q) = \sum_{t \in T} q_{s,t} \cdot t$$

which means that t is chosen for s. In Section 3.2.2, we represented a library element with a parameter tuple, for example, $t = (E, G, H)$ where E, G, and H are parameters characterizing element t. Let E_t denote E of t and Let E_s be the parameter E of t chosen for s. Then we can express a function $fP(s, Q)$ that returns the parameter E of chosen element t for s as follows:

$$E_s = fP(s, Q) = \sum_{t \in T} q_{s,t} \cdot E_t$$

We call such functions *metric selection functions.*

Validity constraints take a form of relations imposed on mapping functions. For example, not all elements in the implementation library can perform the behavior of a task. Therefore, the mapping function f_{SAS} of the physical design style and architecture selection has to satisfy the following condition:

$$(i \; R \; t) \vee \overline{z_{ti}} = 1 \; , \forall t \in V$$

where $i \; R \; t$ is a relation between $i \in I$ and $t \in V$ which is defined as "the implementation i performs the function of t". z_{ti} is is a binary variable which is has a value 1 if implementation i is selected for task t or 0, otherwise. The above equation enforces the rule that z_{ti} must be 0 when $i \; R \; t$ is not true and Boolean *true* is set to 1 and *false* is set to 0.

A validity constraint can exist between two design steps. For example, there is the following validity constraint between the the physical design style selection step and the task-level system partitioning step. *Implementations of different styles cannot be grouped together in the same partition.* This can be expressed as

$$(i \; R \; j) \vee (\overline{y_{pi} \wedge y_{pj}}) = 1 \; , i, j \in V$$

where R is defined as "the physical design style of task i is same as the physical design style of task j" and y_{pi} is a binary variable which has a value 1 if task i is assigned to task partition p and 0, otherwise. The above validity constraint prevents y_{pi} and y_{pj} from taking value 1 simultaneously when implementation i and j are different styles. (i.e., the relation is not true.)

Partition Metric Function

Some validity constraints check the validity between the metrics of design entities. For example, *the size of a chosen die type for a task partition must be big enough to hold the size of a task partition.* We call constraints *metric validity constraints.* If a design entity involving a metric validity constraint is an element from a library, then such parameters can be computed with metric selection functions. But the parameters for a design entity created during the design process such as a task partition must be computed. We call the metric function that computes such metrics the *partition metric functions.* For example, in order to compute the metrics for task partition p, we need to compute the task partition size metric function $TPartSize(p, Z, Y, X, U)$ and task pin count metric function $TPartPin(p, Y, U)$ which compute the corresponding metrics from the first order effects of involved design decisions.

3.3.2 Physical Design Style and Datapath Architecture Selection Subproblem

A physical design style must be selected for each task. For each style, we indexed the physical design styles as follows; FPGA=q_0, gate array=q_1, and standard cell=q_2. Then the set of physical design styles is given as $Q = \{0, 1, 2, 3\}$. For later use, we define two disjoint subsets of Q, Q_f={FPGA, gate array} and Q_v={standard cell}. By selecting an implementation in a library, the design style and datapath architecture for a task are selected. Therefore, we treat the design style and datapath architecture selection subproblem as a single problem called *style and architecture selection* subproblem or *implementation selection* subproblem.

The mapping function for style and architecture selection is given as follows:

$$f_{SAS} : V \rightarrow I$$

where V is the set of tasks and I is a set of implementations. Let Z be a set of binary variables for style and architecture selection which is defined as follows:

$$Z = \{z_{ti} \mid z_{ti} = 1 \text{ if } i = f_{SAS}(t) \text{ and } z_{ti} = 0 \text{ if } i \neq f_{SAS}(t)\ , \forall i \in I,\ \forall t \in V\}$$

subject to the following **mapping function constraint**:

$$\sum_{i \in I} z_{ti} = 1, \forall t$$

The **metric selection functions** for a chosen implementation for task t can be expressed as follows:

The size and delay of different datapath architectures in the library are estimates obtained from the BEST estimator. BEST works differently for two different groups of physical design styles, namely the FPGA-gate array group (Q_f) and the standard cell group (Q_v). Since the BEST estimator was originally developed to estimate only standard cell implementations, we modified BEST to estimate the size and delay of datapath architectures of FPGA and gate array styles. For standard cell design, BEST produces the size estimates for datapath architectures in terms of μm^2 and the size estimate includes the area for controller and routing as well as datapath. On the other hand, for physical design styles belonging to Q_v, BEST produces the size estimate in terms of gate counts excluding routing area because the physical size in terms of μm^2 cannot be computed until the die type for a given task is selected. Therefore, two different metric functions are defined for the size estimates depending on the physical design styles.

$$\begin{aligned}
TaskSize(t, Z) &= \sum_i z_{ti} \times ISIZE_i \text{ if } ISTYLE_i \in Q_v \\
TaskGate(t, Z) &= \sum_i z_{ti} \times ISIZE_i \text{ if } ISTYLE_i \in Q_f \\
TaskDelay(t, Z) &= \sum_i z_{ti} \times IDELAY_i \\
TaskStyle(t, Z) &= \sum_i z_{ti} \times ISTYLE_i
\end{aligned}$$

where $ISIZE_i$, $IDELAY_i$, and $ISTYLE_i$ are parameters for implementation i obtained from a library as we discussed in Section 3.2.2.

The **validity constraint** for style and architecture selection is as follows:

$$TFTYPE_t = \sum_i z_{ti} \times IFTYPE_i$$

where $IFTYPE_i$ is the name of the function performed by implementation i.

3.3.3 Task-Level System Partitioning Subproblem

The mapping function for task-level system partitioning subproblem is given as follows:

$$f_{TP} : V \rightarrow P$$

where P is a set of task partitions.

Let Y be a set of binary variables for task-level system partitioning subproblem which is defined as follows:

$$Y = \{y_{pt} \mid y_{pt} = 1 \text{ if } p = f_{TP}(t) \text{ and } y_{pt} = 0 \text{ if } p \neq f_{TP}(t)\ , \forall p \in P, \forall t \in V\}$$

subject to the following **mapping function constraint**:

$$\sum_{i \in I} y_{pt} = 1, \forall t$$

Task Partition Pin Metric Function $TPartPin(p, Y, U)$

The number of pins for partition p $TPartPin$ is a function of task partitioning Y and bus selection U because task-level system partitioning determines whether a channel is cut by a task partition boundary and the number of pins required for each channel is determined by the selected bus type for the channel.

$$TPartPin(p, Y, U) = \sum_{e \in E} b(e, Y) \times t(e, p, Y) \times BusWidth(e, U)$$

where $b(e, Y)$ and $t(e, p, Y)$ are defined below:

$$b(e,Y) = \begin{cases} 1 & \text{if edge } e \text{ is cut by the given task partitioning } Y \\ 0 & \text{if edge } e \text{ is not cut by the given task partitioning } Y \end{cases} \quad (3.7)$$

$$t(e,p,Y) = \begin{cases} 1 & \text{if edge } e \text{ is incident to } t \in V_p \\ 0 & \text{if edge } e \text{ is not incident to } \forall t \in V_p \end{cases} \quad (3.8)$$

where $V_p = \{t : y_{pt} = 1\}$.

Task Partition Size Metric Function $TPartSize(p, Z, Y, X, U)$

The size $TPartSize(p, Z, Y, X, U)$ of partition p is a function not only of implementation selection Z and task-level system partitioning Y but also that of die selection X if it is implemented with either FPGA or gate array style. This is because parameters related to a chosen die type d such as the ratio of wiring area to the total area WB_d, the average gate size $AGATESIZE_d$, and the average I/O pad size $PADSIZE_d$ change the physical size of a task partition even though the datapath architecture remains same. Therefore, we need two separate metric functions for $TPartSize$ depending on the physical design style of a task partition.

The core size of a task partition implemented with the standard cell style is simply the sum of sizes of tasks allocated to a partition as follows:

$$\sum_t TaskSize(t, Z) \times y_{pt}$$

$TPartSize(p, Z, Y, U)$ for a standard cell implementation including I/O area can be expressed as follows:

$$TPartSize(p, Z, Y, U) = \sum_t TaskSize(t) \times y_{pt} + TPartPin(p, Y, U) \times PADSIZE$$

where $PADSIZE$ is the average I/O pad size of a given standard cell library.

As we discussed in the previous section, the size estimate of a datapath architecture implemented with the physical design styles in Q_f is given as the number of gates; the average gate size $AGATESIZE(\mu m^2/gate)$ should be multiplied to

obtain the physical size of the chosen implementation. Since the average gate size depends on the die type, the core size of a task partition in μm^2 is given as follows:

$$AvgGateSize(p, X) \times \sum_t TaskGate(t, Z) \times y_{pt}$$

where $AvgGateSize(p, X)$ is an average gate size of a chosen die type for partition p, which will be discussed in Section 3.3.4. In order to factor in the area consumed by the routing: The total task area, including routing is

$$WB(p, X) \times AvgGateSize(X, p) \times \sum_t TaskGate(t, Z) \times y_{pt}$$

where $WB(p, X)$ is the ratio of routing area to total die area of the chosen die type for partition p, which will be discussed in Section 3.3.4.

Finally, $TPartSize(p, Z, Y, X, U)$ including I/O area can be expressed as follows:

$$\begin{aligned} &TPartSize(p, Z, Y, X, U) = \\ &\quad WB(p, X) \times AvgGateSize(p, X) \times \sum_t TaskGate(t) \times y_{pt} + \\ &\quad TPartPin(p, Y, U) \times PadSize(p, X) \end{aligned} \tag{3.9}$$

where $PadSize(p, X)$ is the average I/O pad size of the chosen die type for partition p which will be given in Section 3.3.4.

Task Partition Style Metric Functions and a Validity Constraint

$TPartStyle(p, Z, Y)$ which computes the style of task partition p is given as follows:

$$TPartStyle(p, Z, Y) = TaskStyle(t, Z) \quad \text{if } \sum_p y_{pt} \neq 0 \text{ and } y_{pt} = 1$$

According to the above equation, the style of task partition p can be multiply defined, which is in reality not possible since the implementations of different

physical design styles cannot be put together on a single chip. Therefore, the following **validity constraint** for task-level system partitioning is required:

$$y_{pi} = 1 \text{ and } y_{pj} = 1 \longrightarrow ISTYLE_i = ISTYLE_j \tag{3.10}$$

Task Partition Cost Metric Functions

$TPartCost(p, Z, Y, X, U)$ computes the cost of a die if a task partition is implemented with the standard cell style is

$$TPartCost(p, Z, Y, X, U) = \frac{C_f \cdot TPartSize(p, Z, Y, X, U)}{Yield_p}, \quad \text{if } TPartStyle(p, Z, Y) \in Q_v$$

where C_f is the cost parameter associated with the fabrication facility and has the units dollars per unit area. The following non-linear model for yield is used to estimate $Yield_p$ [Ber83].

$$Yield_p = \frac{1}{2 \cdot D_0 \cdot TPartSize(p, Z, Y, X, U)}$$

where D_0 is the number of defects per unit area.

3.3.4 Die Selection Subproblem

The mapping function for the die selection subproblem is given as follows:

$$f_{DS} : P \rightarrow D$$

Let X be a set of binary variables for die selection which is defined as follow:

$$X = \{x_{dp} \mid x_{dp} = 1 \text{ if } d = f_{DS}(p) \text{ and } x_{dp} = 0 \text{ if } d \neq f_{DS}(p)\ , \forall p \in P, \forall d \in D\}$$

subject to the following **mapping function constraint**:

$$\sum_{d \in D} x_{dp} = max_{t \in V}(y_{pt}), \forall p \tag{3.11}$$

where $max_{t \in V}(y_{pt})$ is imposed so that a die type is selected only for non-empty partitions.

The **metric selection functions** for the selected die for partition p can be expressed as follows:

Since die types are not selected for task partitions implemented with the standard cell style, the metric selection functions for a standard cell die cannot be defined by the die parameters in a library. Instead, the metrics for a task partition are used as the metrics for a standard cell die. Therefore, again we need to maintain two separate metric functions for different groups of physical design styles. One way to avoid this complication is to have standard cell task partitions select an imaginary die type from a die library which does not have a fixed size, I/O pins, or cost. Then the other formulae defined later can be built on single metric function regardless of the physical design styles and the formulation is greatly simplified. The introduction of this imaginary die type for standard cell design can even further simplify the formulation. By associating $PADSIZE$, $AGATESIZE$ and WB with this imaginary standard cell die type, we can use Equation 3.10 for estimating the size of a task partition regardless of the physical design styles used. By making $AGATESIZE$ and WB of the imaginary standard cell die type equal to 1, the size estimate for standard cell is not affected by die selection. In Appendix A, we will show how the linearization of 3.15 and 3.17 can be avoided by using an imaginary standard cell die type of $(AGATESIZE = 1, WB = 1)$.

Let $XTPS_p = TPartStyle(p, Z, Y)$.

$$DieSize(p, Z, Y, X, U) = \begin{cases} TPartSize(p, Z, Y, X, U) & \text{if } XTPS_p \in Q_v \\ \sum_{d \in D} x_{dp} \times DSIZE_d & \text{if } XTPS_p \in Q_f \end{cases} \tag{3.12}$$

$$DiePin(p, Y, X, U) = \begin{cases} TPartPin(p, Y, U) & \text{if } XTPS_p \in Q_v \\ \sum_{d \in D} x_{dp} \times DPIN_d & \text{if } XTPS_p \in Q_f \end{cases} \tag{3.13}$$

$$DieCost(p,Z,Y,X,U) = \begin{cases} TPartCost(p,Z,Y,X,U) & \text{if } XTPS_p \in Q_v \\ \sum_{d\in D} x_{dp} \times DCOST_d & \text{if } XTPS_p \in Q_f \end{cases} \quad (3.14)$$

$$AvgGateSize(p,X) = \begin{cases} 1 & \text{if } XTPS_p \in Q_v \\ \sum_{d\in D} x_{dp} \times AGATESIZE_d & \text{if } XTPS_p \in Q_f \end{cases} \quad (3.15)$$

$$PadSize(p,X) = \sum_{d\in D} x_{dp} \times PADSIZE_d \quad (3.16)$$

$$WB(p,X) = \begin{cases} 1 & \text{if } XTPS_p \in Q_v \\ \sum_{d\in D} x_{dp} \times WB_d & \text{if } XTPS_p \in Q_f \end{cases} \quad (3.17)$$

$$FabTime(p,X) = \sum_{d} x_{dp} \times FTIME_d \quad (3.18)$$

$$DieStyle(p,X) = \sum_{d} x_{dp} \times DSTYLE_d$$

The **metric validity constraints** for die selection can now be defined with the above partition metric functions. The size, pin, and style requirements between die selection and task-level system partitioning are given as follows:

$$\begin{aligned} DieSize(p,X) &\geq TPartSize(p,Z,Y,X,U) \\ DiePin(p,X) &\geq TPartPin(p,Y,U) \\ DieStyle(p,X) &= TPartStyle(p,Z,Y) \end{aligned}$$

3.3.5 Die Cluster Subproblem

The mapping function for die clustering is given as follows:

$$f_{DC} : P \rightarrow C$$

where C is a set of die clusters.

Let S be a set of binary variables for die clustering which is defined as follows:

$$S = \{s_{cp} \mid s_{cp} = 1 \text{ if } c = f_{DC}(p) \text{ and } s_{cp} = 0 \text{ if } c \neq f_{DC}(p),\ \forall c \in C, \forall p \in P\}$$

subject to the following **mapping function constraint**:

$$\sum_{c} s_{cp} = max_t(y_{pt})$$

This constraint ensures that only non-empty task partitions are clustered.

Die Cluster Pin Metric Function, $DCluPin(S, U)$

The number of I/O pins for die cluster c can be computed in a similar way to that of $TPartPin$.

$$DCluPin(c, S, U) = \sum_{e \in E} b1(e, S) \times t1(e, c, S) \times BusWidth(e, U)$$

where $b1(e, S)$ and $t(e, c, S)$ are defined below:

$$b1(e, S) = \begin{cases} 1 & \text{if edge } e \text{ is cut by the given die clustering } S \\ 0 & \text{if edge } e \text{ is not cut by the given die clustering } S \end{cases}$$

$$t1(e, c, S) = \begin{cases} 1 & \text{if edge } e \text{ is incident to } t \in V_c \\ 0 & \text{if edge } e \text{ is not incident to } \forall t \in V_c \end{cases}$$

where $V_c = \{t : s_{cp} \cdot y_{pt} = 1\}$.

Die Cluster Size Metric Function $DCluSize(c, Z, Y, X, U, S, M)$

The size of a substrate $DCluSize$ on which a die cluster is mounted depends not only on the sizes of assigned dies but also on the wiring space requirement $WireSize$ for the die cluster using the selected substrate technology. Since wires can be routed under dies in MCMs, either the sum of die areas $TDieSize$ or the wiring space requirement determines the die cluster size. $WireSize$ can be computed with the estimation technique developed for PWBs. A method for estimating the size of a PWB from a given interconnected chip netlist was proposed for a single pair of wiring layers[Sch82, SO73] using Rent's rule. However, it is not suitable for MCMs because it does not consider more than two wiring layers. Hence, we used the following analytic equation expressing the wiring correlation

for multilayer PWB proposed by Hannemann[Han84], without relying on Rent's rule.

$$NSIG = K_c \frac{\lambda \times TDiePin}{L\sqrt{TND}}$$

where $L = \sqrt{WireSize}$, K_c is a constant, $TDiePin$ is the total number of die pins, TND is the total number of dies, and $\lambda(c, M) = \frac{VGS(c,M)}{1+LBV(c,M)}$. The equations computing $VGS(c, M)$ and $LBV(c, M)$ are given in Section 3.3.6.

Therefore, $WireSize$ can be computed with following equation:

$$WireSize(c, Y, U, S, M) = (K_c \frac{\lambda(c, M) \times TDiePin(c, Y, U, S)}{NSIG(c, M)})^2 \frac{1}{TND(c, S)}$$

With S, the above parameters are calculated as follows:

$$\begin{aligned} TDiePin(c, Y, U, S) &= \sum_p s_{cp} \times TPartPin(p, Y, U) \\ TND(c, S) &= \sum_p s_{cp} \end{aligned}$$

Now, $DCluSize(c, Z, Y, X, U, S, M)$ can be expressed as follows:

$$DCluSize(c, Z, Y, X, U, S, M) = \begin{cases} WireSize & \text{if } WireSize > TDieSize \\ TDieSize & \text{if } WireSize \leq TDieSize \end{cases}$$

where $TDieSize$ is given below:

$$TDieSize(c, Z, Y, X, U, S) = \sum_p DieSize(p, Z, Y, X, U) \times s_{cp}$$

The above equation represents the sum of sizes of dies on a die cluster, which is smaller than the size of a substrate because the spacing between dies and other space consumed by routing are not included.

Die Cluster Cost Metric Function, $DCluCost(c, Z, Y, X, U, S, M)$

$$SubCost(c, Z, Y, X, U, S, M) = DCluSize(c, Z, Y, X, U, S, M) \times UCost(c, M)$$

3.3.6 Substrate Technology Selection Subproblem

The mapping function for substrate technology selection subproblem is given as follows:

$$f_{SS} : C \rightarrow ST$$

where $ST = \{SMC,\ MCM\text{-}D,\ MCM\text{-}C,\ MCM\text{-}L\ \}$ and SMC is an imaginary substrate for a single chip package.

Let M be a set of binary variables for the substrate technology selection which is defined as follows:

$$M = \{m_{vc} \mid m_{vc} = 1 \text{ if } v = f_{SS}(c) \text{ and } m_{vc} = 0 \text{ if } v \neq f_{SS}(c),\ \forall c \in C, \forall v \in ST\}$$

subject to the following **mapping function constraint**:

$$\sum_{v} m_{vc} = max_p(s_{cp})$$

The **metric selection functions** for a chosen substrate technology for die cluster c can be expressed as follows:

$$\begin{aligned}
NSIG(c, M) &= \sum_{v \in ST} m_{vc} \times NSIG_v \\
VGS(c, M) &= \sum_{v \in ST} m_{vc} \times VGS_v \\
LBV(c, M) &= \sum_{v \in ST} m_{vc} \times LBV_v \\
SPS(c, M) &= \sum_{v \in ST} m_{vc} \times SPS_v \\
UCOST(c, M) &= \sum_{v \in ST} m_{vc} \times UCOST_v
\end{aligned}$$

$NSIG$, VGS, LBV, SPS, and $UCOST$ are defined in Section 3.2.2.

3.3.7 Package Selection Subproblem

The mapping function for package selection subproblem is given as follows:

$$f_{PS} : P \rightarrow K$$

Let W be a set of binary variables for package selection which is defined as follows:

$$W = \{w_{kc} \mid w_{kc} = 1 \text{ if } k = f_{PS}(c) \text{ and } w_{kc} = 0 \text{ if } k \neq f_{PS}(c),\ \forall c \in C, \forall k \in K\}$$

subject to the following **mapping function constraint**:

$$\sum_{k} w_{kc} = max_p(s_{cp}) \tag{3.19}$$

The **metric selection functions** for the chosen package for partition p can be expressed as follows:

$$\begin{aligned} PkgSize(p, W) &= \sum_{k} w_{kc} \times KSIZE_k \\ PkgCost(p, W) &= \sum_{k} w_{kc} \times KCOST_k \\ PkgPin(p, W) &= \sum_{k} w_{kc} \times KPIN_k \\ PkgTime(p, W) &= \sum_{k} w_{kc} \times KTIME_k \end{aligned}$$

Then the metric **validity constraints** can be expressed as follows:

$$PkgSize(c, W) \geq DCluSize(c) \tag{3.20}$$

$$PkgPin(c, W) \geq DCluPin(c) \tag{3.21}$$

3.3.8 Bus Selection Subproblem

The mapping function for bus selection is given as follows:

$$f_{BS} : E \rightarrow B$$

Let U be a set of binary variables for bus selection which is defined as follows:

$$U = \{u_{be} \mid u_{be} = 1 \text{ if } b = f_{BS}(e) \text{ and } u_{be} = 0 \text{ if } b \neq f_{BS}(e),\ \forall e \in E, \forall b \in B\}$$

subject to the following **mapping function constraint**:

$$\sum_{b} u_{be} = 1 \tag{3.22}$$

The **metric selection functions** for the chosen bus selected for communication channel e are

$$\begin{aligned} BusWidth(e,U) &= \sum_{b} u_{be} \times BWIDTH_b \\ BusType(e,U) &= \sum_{b} u_{be} \times BTYPE_b \end{aligned}$$

$BusClkCyc$ cannot be computed as simply as above because it depends not only on bus type but also on task implementation style or substrate technology and average length of wires. For example, even if it is an on-chip type, the maximum clock frequency of the FPGA style is different from that of the standard cell style. In the same manner, the maximum clock frequency of a multi-chip module substrate with MCM-D technology is different from that of MCM-L technology.

The bus clock cycle can change with the selection of style or technology by computing a scaling factor according to either the style or the technology as follows:

$$BusClkCyc(e,U,Z) = \sum_{b} u_{be} \times BCYCLE_b \times ps(e)$$

where $ps(e)$ is a function that compute the performance scaling factor for edge e given below:

If $BusType(e,U) = on-chip$,

$$ps(e) = \sum_{p} PerfScale(p,Z,Y) \times (1 - b(e,Y)) \times t(e,p,Y)$$

where $PerfScale(p, Z, Y)$ is defined as follows:

$$\begin{aligned} PerfScale(p, Z, Y) &= \sum_{q} QM(q, TPartStyle(p, Z, Y)) \times PS_q \\ QM(q_1, q_2) &= \begin{cases} 0 & \text{if } q_1 \neq q_2 \\ 1 & \text{if } q_1 = q_2 \end{cases} \\ q, q_1, q_2 &\in Q \end{aligned}$$

If $BusType(e, U) = on - module$,

$$ps(e) = \sum_{c} SPS(c, M) \times (1 - b1(e, S)) \times t1(e, c, S)$$

If $BusType(e, U) = on - board$,

$$ps(e) = 1$$

PS and SPS are defined in Section 3.2.2.

The **validity constraint** for bus selection is given as follows:

$$BusType(e, U) = b(e, Y) \tag{3.23}$$

From bus selection, we can compute the delay for transferring the required volume of data VOL_e over the selected bus as follows:

$$EdgeDelay(e) = \frac{VOL_e \cdot BusClkCyc(e, U, Z)}{BusWidth(e, U)}$$

3.3.9 Task Scheduling

In the beginning of this section, we made assumptions about the task implementation and channel implementation. Those assumptions simplify the task scheduling. Under those assumptions, the non-pipeline schedule is unique. Therefore,

non-pipeline scheduling is determined by decisions in other design steps. However, operating a system in non-pipelined mode under our assumption underutilizes hardware resources. For example, in the JPEG and the MPEG example, an image frame is divided into macro-blocks and each macro-block is processed separately. With non-pipelined operation, Hardware utilization is low because the rest of hardware modules are idle waiting for the completion of one hardware part. We can reduce considerably the total processing time of an image frame by processing macro blocks in a pipelined fashion and also increase the utilization of hardware.

In a pipelined scheduling of a TFG, we partition a TFG into stages as in pipelining a CDFG[PP88]. Pipeline schemes must consider resource allocation such that stages sharing resources must not be executed in parallel[PP88]. Our assumption of no hardware sharing between tasks frees our model from considering resource allocation of tasks as a complex problem. Instead, it degenerates to a one-to-one mapping onto resources. We still consider datapath resource sharing as an integral cost-performance trade-off. Consequently, task scheduling is determined solely by other design decisions and only the definition of performance becomes different under a different operation mode as will be discussed in the next section.

3.3.10 System Metric Functions

In previous sections, we defined metric functions for design entities in a system. In this section, metric functions for a system as a function of design decisions $A = \{Z, Y, X, W, U, S, M\}$ are defined. Among system metrics, we consider the cost $SCOST$, the performance $SPERF$, and the time-to-market TTM of a system in our model.

3.3.10.1 The System Cost Metric Function

The cost of a multi-chip digital system is the sum of amortized development cost, the costs for all components, and the assembly cost. The cost of each component cost is further composed of the manufacturing cost and the testing cost. We approximate the system cost function as follows:

$$SystemCost(A) = NRE/n + \sum_{chip \in system} COST_{chip}$$

where NRE is the development cost and n is the number of copies of the system that will be manufactured. NRE can be estimated by the parametric cost estimation(PCE) technique borrowed from software engineering [And95] or input by the user. Since each chip corresponds to a die cluster c, $COST_{chip}$ can be approximated with the following equation:

$$COST_c = \sum_{p \in c} DieCost(p, Z, Y, X, U) + SubCost(c, W) + PkgCost(c, W) \quad (3.24)$$

where $SubCost(c, W) = DCluCost(c, W)$.

3.3.10.2 System Performance Metric Function

The system performance metric function is defined as the execution time on *path* between two specified nodes (src, dst) in the task flow graph. Then, the system performance of a non-pipelined execution model is defined as the delay of the critical path *path* from *src* to *dst* in the TFG (see Figure 3.2.1). We have the following system performance metric function for a non-pipelined system:

$$\begin{aligned} SystemPerf(A) &= DELAY_{path} \\ &= \sum_{t \in path} TaskDelay(t, Z) + \sum_{e \in path} EdgeDelay(e) \end{aligned} \quad (3.25)$$

The performance metric function for pipelined systems is given as follows for processing n data instances:

$$SystemPerf(A, n) = DELAY_{path} + InitIntv(Z) \times n \quad (3.26)$$

$$InitInv(Z) = max_{t \in V} TaskDelay(t, Z) \quad (3.27)$$

3.3.10.3 System Time-To-Market Metric Function

Although the time-to-market is an important metric that varies with system-level design decisions, it has not been considered as a metric of a system architecture. In this section, we develop a simplified formula to estimate the prototyping time as a fraction of time to market. However, the reader is cautioned that our simplified model is built to demonstrate our research contributions. An industrial set of system architecture design tools would of necessity have a more complex model of time-to-market customized for that industry and perhaps that particular organization.

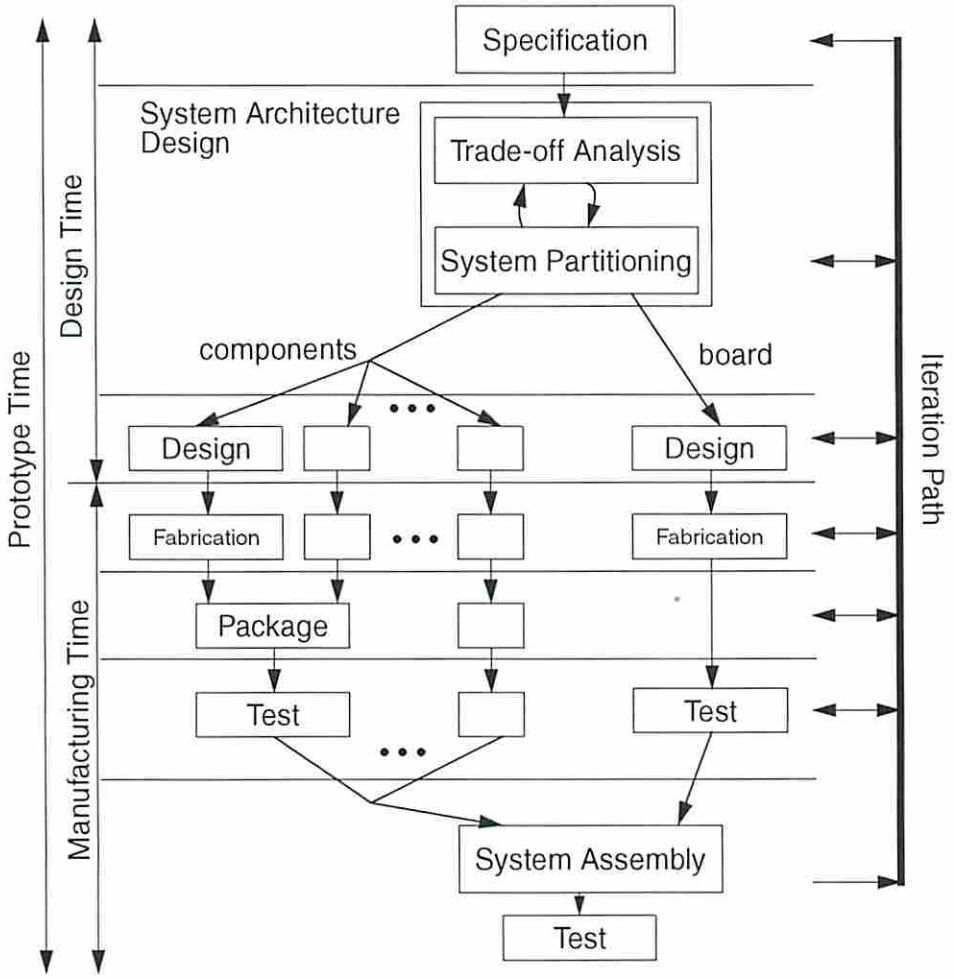


Figure 3.9: Generic design flow for multi-chip system development

In Figure 3.9, we showed the typical design flow for a multi-chip system. We defined TTM for a multi-chip digital system as a sum of two major components, namely prototyping time $TProto$ for a system and production time $TProd(n)$ for n copies of a system.

$$TTM = TProto + TProd(n)$$

We consider only $TProto$ in our prototype model. $TProto$ is further divided into two components, namely component prototyping time $CProto$ and system prototyping time $SProto$. For multi-chip systems, we assume system prototyping can be done in parallel with component prototyping as shown in Figure 3.9 and takes less time than component prototyping since the complexity of prototyping a board is much less than developing ASICs. Therefore TTM can be simplified as follows:

$$TTM \approx max(CProto, SProto) \approx CProto$$

$CProto$ is composed of the design time, the manufacturing time and the testing time of chips. In general, these terms depend on the availability of resources. For example, the design time is a function of the number of engineers and tools available while the manufacturing time and testing time vary with the availability of equipment. Design time is especially hard to estimate because tasks in the design process are not only dependent on the number of engineers and their skillfulness but also on other factors like design methodologies, tools to be used and characteristics of the system being designed. Since no known available model can estimate design time realistically, we assume that the design time is rather independent of design decisions for the subproblems considered in this dissertation. Testing time is not considered in our simplified model. Under such simplifying assumptions, the estimated time required to prototype a system is still difficult because the design time for each component is not the same and sometimes the tasks in a system have interdependencies which prohibit concurrent design. Therefore, we approximate the system manufacturing time with a total time required to develop each component in a serial manner. Minimizing the total development time eventually reduces the prototyping time and therefore, time-to-market. We divide the

manufacturing time into two components, namely the fabrication time $TFab$ and the packaging time $TPkg$. Under the above assumptions, $SystemTTM(A)$ can be expressed as follows:

$$SystemTTM(A) = \sum_{c \in system} \sum_{p \in c} FabTime(p, X) + \sum_{c} PkgTime(c, W) \quad (3.28)$$

Chapter 4

EDEN: MILP-Based Optimization Tool

In Chapter 3, we focused on how to represent an early system-level design decision, building a first-order analytical model that expresses metrics associated with design entities as a function of design decisions and also expresses validity constraints that capture interdependencies among design decisions. In this chapter, we describe one of our system architecture optimization methods called **E***arly* **D***esign* **EN***vision* (**EDEN**) that uses Mixed Integer Linear Programming (MILP) to produce a number of near-optimal system architectures.

4.1 MILP and Linearization

We first attempt to find optimized system architectures by linearizing the design model in Section 3.2 and solving it with MILP. If the model renders itself to linearization easily, MILP takes relatively less effort to develop than many other methods and the optimality of a solution is guaranteed. However, it is well-known that MILP is in general very slow and therefore, the practical size of a problem that can be solved with MILP is limited. Therefore, to see the feasibility and usefulness of using MILP in system architecture optimization, we first attempt to solve a subset of the design steps for the case that the objective metric is the system prototyping time, which corresponds to finding a feasible design that can be most quickly prototyped.

Design steps considered in this MILP-based optimization are implementation selection, task-level system partitioning, die selection, package selection, and bus

selection. The physical design styles which are considered in MILP-based optimization are FPGA, gate array, and standard cell style. The linearization step of our model is given in detail in Appendix A.

4.2 Problem Instantiation and the M-Language

In order to use an MILP solver, we must specify the objective and the constraints in a matrix file which the MILP solver can understand. Hand-coding the matrix for even a small-size MILP model is erroneous and tedious. It is practically impossible to handcode the matrix of the model given here for a design problem. An alternate is to write a program that generates a matrix file for a specified instance of the problem, e.g., the JPEG codec. This does not improve the situation significantly. First, this approach still requires the user to write a separate program for each problem. Secondly, if there is a error in final matrix format, it is difficult to debug. Third, if there is any change in the MILP model, the correction of the program to accommodate the change is very hard.

Therefore, we developed a language called M to describe the MILP formulation in a natural mathematical form. Any change made in the MILP model can be painlessly entered into the existing model. The given MILP model in M can be instantiated with a compiler called GEM which generates a matrix generator in C++. By compiling and running the matrix generator, the matrix representation is produced for the MILP solver. For example, consider the constraint shown in Equation (4.1) below.

$$\sum_{i \in E} \Theta_{ip} = PX_p \qquad \forall p \in P \tag{4.1}$$

This constraint can be coded as follows in M language.

$$Sig(i : E)(THETA\_i\_p) \;=\; PX\_p \qquad @p\{P$$

The corresponding $C++$ code generated by the GEM compiler is shown below.

```
for (int j=1; j < P_size; j++) {
     for (int i = 1; i < E_size; i++) {
```

```
            cout << "THETA" << E[i].name << P[j].name;
            count << "+";
        }
        cout << "=" << "PX" << P[j].name << '\n'
    }
```

By compiling and executing the above generator, the matrix representation of a problem is produced.

Since writing the MILP formulation and debugging it could be carried out at a high level of abstraction, we were able to save considerable amount of development and debugging time. Since the MILP formulation is popularly used to solve a number of problems is design automation, such as global routing, channel routing, floor planning, internal task scheduling and so on, we believe that the meta-language M and the GEM compiler will find general applicability.

With the M-language formulation and GEM, Figure 4.1 shows the implementation of EDEN. We used the LAMPS [AMS94] MILP solver as part of EDEN.

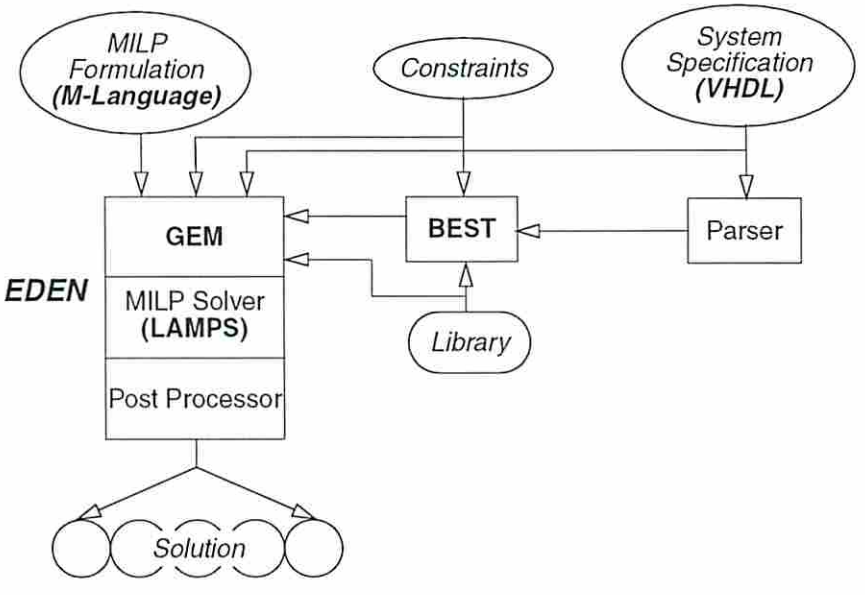


Figure 4.1: Software architecture of EDEN

4.3 Experimental Results

4.3.1 Examples

Example 1: JPEG Codec

JPEG (Joint Photographic Expert Group) standard[Wal91] is an ISO/CCITT standard for still image compression/decompression. The task flow graph for the JPEG codec is shown in Figure 4.2. The still image is divided into 8 x 8 pixel blocks. A Forward Discrete Cosine Transformation (FDCT) is performed on each block to transform the image from the spatial domain to the frequency domain. After FDCT, the macro block consists of 1 DC component and 63 AC components. Compression is attained at this step because most of the AC components have values close to 0. The frequency domain components are quantized to achieve further compression. An Entropy encoder encodes the quantized macro block using a table defined in the JPEG standard. While scanning a macro block from left to right and from top to down, the number of zeros are counted until a non-zero element is found. The number of zeros and the amplitude of non-zero elements is represented using a pair of values. The encoded bits are read from the table with this value-pair as the key.

Real time processing is not necessary in most applications of still image compression, but we shall treat the JPEG codec as a real time application. An image for JPEG or MPEG compression system has three components, one luminance component(Y) and two chrominance components(Cb and Cr). The frame size we shall consider consists of 360 $\times$ 288 pixels. The precision for a pixel is 8 bits and we consider only the Y component for our examples. A processing rate of 30 frames per second is assumed. Therefore, there are 1620 blocks of 8$\times$8 pixels in the Y component and the processing time for an 8 $\times$ 8-pixel block is 20,576 ns. The volume of data on a channel is 1 macro block i.e., 8 $\times$ 8 pixels $\times$ 8 bits= 512 bits.

Example 2 : MPEG Video Encoder

The MPEG (Motion Picture Expert Group) standard[Gal91] is an international standard for video compression. In Figure 3.1, a simplified task flow graph of

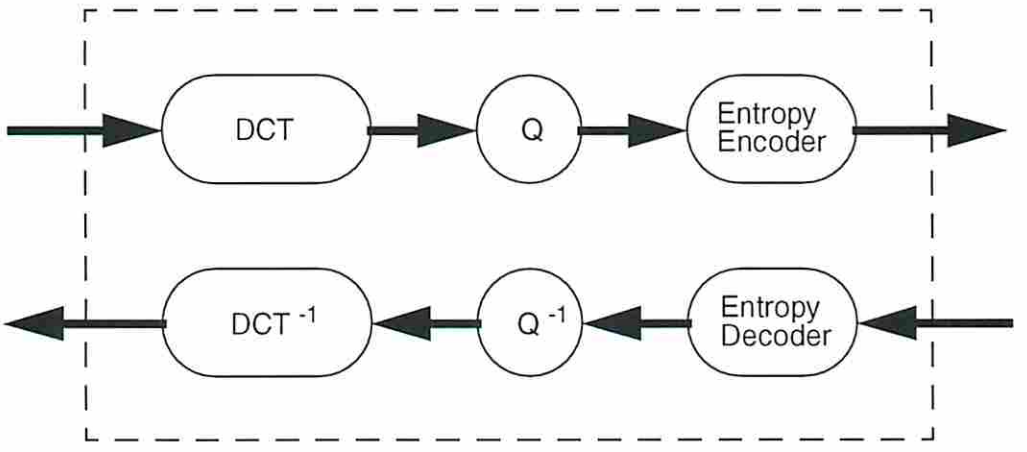


Figure 4.2: Task flow graph of the JPEG codec

an MPEG encoder is shown. A frame has three components: Y, Cb, and Cr. Each component is divided into 16 x 16 pixel blocks. Processing is carried out for each block. Motion estimation compares the current pixel block *Bcur* to blocks in the previous frame within the search range, identifies the closest matching block *Bbest* and computes a motion vector. *Bbest* is subtracted from *Bcur* and the difference is transformed through the use of the DCT into the frequency domain. As in JPEG, the transformed block information is quantized and encoded using an entropy encoder. A dequantizer and an inverse DCT are applied to the quantized block to reconstruct the current frame to be used as a reference frame for the next frame.

The frame size of the MPEG-I standard is 360 x 288 and the frame rate is up to 30 frames/sec. The macro block size for motion estimation is 16 x 16 and for the DCT is 8 x 8. The timing requirement to process a 16 x 16 macro block is 164,609 ns. with 15 frames/sec. (Y component only). The volume of data transferred on an edge of the task flow graph is assumed to be 16 x 16 x 8 = 2,048 bits.

4.3.2 Library Setup for Experiments

In order to be able to use EDEN to optimize the multi-chip system architecture designs for the JPEG codec and MPEG encoder, we first needed to build the technology library. Three physical design styles, namely, FPGA, gate array and standard cell, are represented in our current technology library. In addition to the modules available in the vendor libraries that were accessible to us, we designed and characterized several modules. We used the Xilinx XC4000 family for our FPGA design style. Modules such as an 8-bit ripple carry adder and 8-bit parallel multiplier were designed using ViewLogic and laid out using XACT, the Xilinx FPGA place and routing tool. The modules were simulated using ViewSim from ViewLogic to obtain the delay and the gate count of the modules. For gate array design, we chose VLSI Technology Inc.'s vgt350 1.2 micron gate array. The modules mentioned above were designed using VTI's portable cell library, laid out using the Gate Compiler tool from Compass Design Automation, and back annotated and simulated using the Qsim logic simulator from Compass. For standard cell design, VLSI Technology Inc.'s vsc350 1.2 micron standard cell technology was chosen. Modules designed using the gate array design style could be reused, because of the portability of the cells in VTI library. The modules were laid out using ChipCompiler and back annotated and simulated using the Qsim simulator from Compass.

With the characterized operator modules in different technologies, we obtained the estimated size and delay on possible datapath architectures with BEST[KP95]. The summarized information on the estimated architectures used for JPEG codec is given in Table 4.1. The information on the cost and size of die and package types are obtained or approximated from the available data of Xlinx databook and MOSIS service. For the die library, 4 FPGA die types and 5 gate-array die types are used. For the package library, 16 different package types are used. Finally, 12 bus types are used for the bus library.

# of estimated datapath architectures	FPGA	gate array	standard cell
DCT	7	12	9
Quantizer	1	4	3
Encoder	0	3	3
IDCT	7	12	9
Dequantizer	1	4	3
Decoder	0	3	3

Table 4.1: The number of datapath architectures for tasks in the JPEG codec predicted by BEST

4.3.3 Experimental Results for Rapidprototyping

By running GEM on our model description in M, we instantiated our model for the JPEG codec and MPEG encoder examples. The instantiated MILP formulation of the JPEG codec has 3,972 constraints and 2,266 variables and that of MPEG encoder has 4,345 constraints and 2,324 variables.

We conducted our experiments by tightening both the system performance constraints and the cost constraints to obtain design points satisfying those constraints and to study how physical design style and datapath architecture selection change with the constraints. When the performance bounds and cost bound are very loose, the formulation selects pure FPGA designs to minimize the product development time. As we tighten the performance constraints, a pure-FPGA design cannot meet the performance constraints at some point. Then the formulation forces the solution to choose the gate Array style for portions of the task-flow graph. Further tightening of timing constraints results in the selection of standard cells. When we tighten the cost constraint, the formulation forces the solution to one which exhibits cost-effective technology. For the JPEG codec, EDEN found multiple solutions which have the same prototyping time.

Our first experiment was to perform virtually unconstrained optimization i.e. both cost and performance constraints were set to large values so that the solution

space is practically unconstrained. The result of unconstrained optimization can be expected to lead to pure-FPGA designs. (Despite the fact that these designs may not satisfy realistic timing constraints, pure FPGA designs can be rapidly prototyped and are useful for functional verification through hardware emulation.) However, since the estimator BEST could not find feasible points for the entropy encoder and decoder in the FPGA design style due to the limitations of BEST, the optimum solution EDEN could find uses one FPGA partition and one gate array partition for the case of unconstrained optimization. BEST assumes the worst case number of iterations of unfixed loops making the critical path appear to be quite lengthy, when in fact the loops often terminate early. The feasible design points which correspond to EX1 (unconstrained optimization), have three partitions, two in FPGAs and the other in the gate array style. The existence of multiple solutions comes from the fact that there are many possible ways of partitioning tasks with the same number of partitions and same style selection, e.g. by changing package and die selection or with a slight change of interface configuration, a different but valid solution can be derived. We have shown one of the solutions of experiment EX2 in Figure 4.3. We conducted six sets of experiments using EDEN on the JPEG example. The constraints corresponding to these experiments (EX1 through EX6) are summarized in Table 4.2. The results obtained in these experiments are summarized in Table 4.3. As can be seen, tight constraints on cost and performance force the selection of more gate array and standard cell design styles (EX5 and EX6). A mix of the three design styles is selected for experiments EX3 and EX4. In experiment EX6, we obtained a pure standard cell design. We conducted another set of experiments for the MPEG encoder. As we already observed in the JPEG experiment, solutions are clustered in the design space. The constraints of experiments are summarized in Table 4.4. EX1 corresponds to the unconstrained case. EX2 and EX3 are performance constrained experiments. EX4 and EX5 are cost constrained experiments. EX6 was constrained in both performance and cost. The experimental results are summarized in Table 4.5. Solutions for EX3 and EX6 satisfy the performance requirement of the MPEG standard. An example of solutions found for EX3 is shown in Figure 4.4.

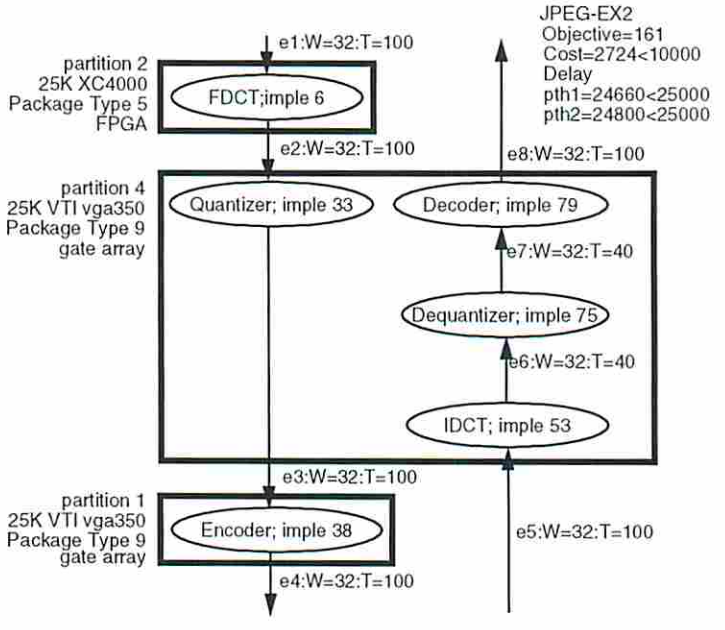


Figure 4.3: One of solutions found by EDEN for the JPEG example (EX2)

	Cost($)	Delay on Path1 (ns)	Delay on Path2 (ns)
EX1	5,000	50,000	50,000
EX2	5,000	25,000	25,000
EX3	5,000	20,000	20,000
EX4	800	50,000	50,000
EX5	500	50,000	50,000
EX6	200	20,000	20,000

Table 4.2: Constraints used for experiments on JPEG example

	Objective	Cost ($)	Delay on Path1 (ns)	Delay on Path2 (ns)	FPGA	gate array	standard cell
EX1	84	3,748	44,980	47,880	2	1	0
EX2	161	2,724	24,660	24,800	1	2	0
EX3	357	1,350	19,920	19,920	0	3	1
EX4	1,000	798	47,140	43,980	2	2	1
EX5	1,732	470	47,900	48,300	1	1	3
EX6	3,252	128	19,640	19,640	0	0	1

Table 4.3: Summary of JPEG experiment results

	Cost($)	Delay on Path1 (ns)	Delay on Path2 (ns)
EX1	10,000	500,000	500,000
EX2	10,000	190,000	190,000
EX3	10,000	164,600	164,600
EX4	2,000	500,000	500,000
EX5	500	500,000	500,000
EX6	500	164,600	164,600

Table 4.4: Constraints used for experiments on MPEG example

	Objective	Cost ($)	Delay on Path1 (ns)	Delay on Path2 (ns)	FPGA	gate array	standard cell
EX1	55	4,105	499,480	490,720	3	2	0
EX2	2,333	1,891	188,040	187,840	1	1	2
EX3	2,500	1,247	159,960	163,800	1	1	3
EX4	82	1,010	477,240	480,045	1	1	0
EX5	1,415	446	492,757	495,717	0	1	1
EX6	2,000	481	162,921	162,360	0	1	1

Table 4.5: Summary of MPEG experiment results

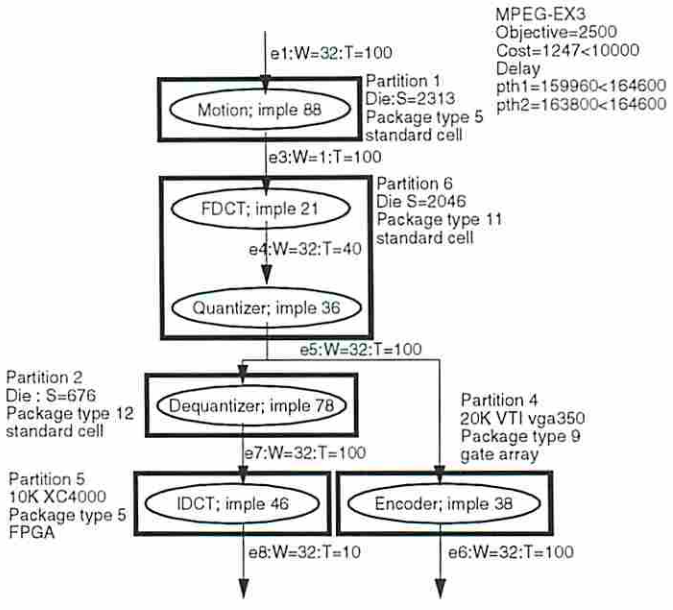


Figure 4.4: One of solutions found by EDEN for the MPEG example (EX3)

4.4 Conclusion

We have presented a software tool called EDEN for multi-chip system design optimization. EDEN was able to find valid style and implementation selections that minimized the total fabrication time. As the performance and cost constraints are tightened, the style and implementation selections move towards more manufacturing-time-intensive technologies. Based on our experiments using EDEN we now believe that using the prototyping time as an objective function is not sufficient since the current formulation merely eliminates the invalid designs while optimizing the system cost or performance are desired. As a result, many inferior designs are included in solutions such as selecting a larger package than can accommodate the die. This suggests that we need to extend EDEN for optimization of multiple objectives such as cost, performance, and prototyping time simultaneously. Through our experimentation, it became clear to us that style and datapath architecture selection have significant impacts on the behavior of a solution in the design space; following style and datapath architecture selection, the remaining

steps, i.e. task-level system partitioning, package selection, and interface configuration have impact on the characteristics of the solution. These observations can be exploited in developing efficient heuristics for the multi-chip system design optimization problem. Our current cost model was simplified on the assumption of mass production of the prototype in which the development cost of a product is negligible. As a result, high-performance styles became more cost-effective because FPGA chips are more expensive than gate-array chips under the mass-production assumption. But this is not a suitable model for rapid-prototyping and small-quantity production where the development cost is a big part of the system cost. Solving an MILP formulation for a real-size problem is computationally expensive. Often, it took more than 2 hours of CPU time to search a fraction of the design space for the first 25 solutions reported for the JPEG example. The solution time varies widely depending on the characteristics of specified system and constraints. For example, in most cases of the MPEG example, no solution was found after 15 hours CPU time. To speed up the solution process, we divided the search space by providing lower and upper bounds on the objective function. Since the LAMPS MILP solver uses branch and bound in solving integer programming problems, setting bounds on the objective function helps in reducing the run time. Without a good method of finding bounds for the objective function, we may be able to reduce run time by dividing the search range into a set of smaller ranges. We start the search range with higher prototyping time because the chance of a design satisfying the performance and cost constraints is higher when implemented in the high-prototyping styles. If a solution is found in one range, we stop the MILP solver and the next lower range is searched. This step is repeated until there is no solution in the given range. Another way of speeding up the MILP formulation is to solve the steps separately in the order of their importance as described above. For example, the style and implementation selection steps are solved first, followed by partitioning and interface configuration while variables related to the style and implementation selection are set to values found in the previous step. In summary, the MILP approach for the system architecture optimization is not practical for a complicated model such as the one we are attempt to solve for multi-chip systems. Therefore, a more efficient optimization technique is required.

Chapter 5

GARDEN: Genetic Algorithm-Based Optimization Tool

MILP is a powerful tool to express a complex model rigorously and conveniently, but even a simple model requires lengthy time to solve. The requirements of linearity on the constraints is another drawback of MILP because many factors in a design model are easier to express in non-linear form. Even though it is possible to linearize non-linear constraints, the linearization step increases the number of constraints exponentially. On the other hand, it is generally accepted that tailored heuristics are efficient in solving a problem, but it is difficult to develop a robust heuristic for a complicated constrained optimization problem. Another drawback of specialized heuristics is the difficulty of extension because many new issues and changes in a model cannot be easily incorporated in an existing heuristic because of customization.

In order to avoid the aforementioned problems of MILP and tailored heuristics, we decided to use another generic optimization technique which is general but able to be customized in some degree. We developed multi-chip system architecture optimization tool called **G***enetic* **A***lgorithm for ea***R***ly* **D***esign* **EN***vision* **GARDEN** which is based on the Genetic Algorithm approach[Mic92]. In principle, a genetic algorithm emulates the evolution process in nature. Intuitively, a genetic algorithm is suited to design problems since it is similar to a real design process. In the early stage, the designer reviews a few possible architectures and selects one that is believed to be the best (evaluation and selection), then the designer tries to improve the selected design by making a few changes (mutation). Also the

designer tries to build a better system by taking the good traits of two designs (crossover). However, the designer optimizes a design serially in the real design process while a genetic algorithm can search the design space in parallel. As we will show in our experimental results, a genetic algorithm finds solutions in much shorter time while it does not require tailoring of specialized heuristics. Incorporating non-linearity is not a problem in the GA. Easy parallelization is the another advantage of a genetic algorithm because the processes in a genetic algorithm are highly independent therefore easily parallelized to either test more complex models or handle a bigger design population[Whi93]. A genetic algorithm also has a room for customizing to exploit the characteristics of a problem as we will show in the following sections. Our systematic model and object-oriented implementation makes GARDEN easily upgradable for a more sophisticated model.

5.1 Genetic Algorithm

```
procedure Genetic Algorithm
begin
  t = 0;
  initialize P(t);
  evaluate P(t);
  while termination condition not satisfied do
  begin
        t = t + 1;
        select C(t) from P(t-1);
        recombine individuals in C(t) forming P(t);
        evaluate P(t);
  end
end
```

Figure 5.1: A generic genetic algorithm[Mic92]

A generic genetic algorithm is shown in Figure 5.1. A solution is often called an *individual* and the encoded form of an individual is called a *chromosome*. Different problems require different encoding schemes. Although a binary string representation is the most popular, it could be inefficient if the solution requires a huge

binary string. Integer or floating point numbers can also be used for encoding a solution. In the first step of a genetic algorithm, a population $\mathbf{P}$ which is a set of such encoded solutions is generated. Initial solutions are most often generated randomly but heuristics can be used. Each solution is evaluated with a fitness function which measures the quality of a solution. If the current generation of solutions satisfies one of *termination conditions*, the program stops. If any termination condition is not met, a set of promising solutions is selected from the current population and populates the intermediate population $\mathbf{C(t)}$. This process is controlled by the *selection mechanism* which is sometimes called a sampling mechanism and considered the most important step in a genetic algorithm[Mic92]. In the *elitist* approach, the solutions of high fitness are selected not only from the current population but also the parent population of the current generation. Depending upon the function which computes the number of copies which a solution can insert into the next generation, there can be variations of the sampling mechanism. *Remainder stochastic sampling* and *ranking* are representative examples of selection schemes. In each iteration, the genetic algorithm needs to introduce certain changes over the population characteristics to explore the search space. The *crossover* and *mutation* operations are mechanisms to bring in such changes in the next generation. The individuals in $\mathbf{C(t)}$ are crossed over and mutated to generate a new generation $\mathbf{P(t)}$. While crossover creates a new solution by combining parts of two chromosomes, mutation randomly changes values in a chromosome. For the new generation, the evolution process is repeated.

5.2 GARDEN Implementation

Although the overall framework of genetic algorithms is well-known, an improper implementation of a genetic algorithm for a problem can easily nullify the advantages of a genetic algorithm. A fast and high quality optimization tool based on GA demands deep understanding of both the problem and the characteristics of GA. In applying GA to the multi-chip system architecture optimization problem, the primary difficulty comes from the complex structure of the problem and its

constrained optimization nature. In this Section, we detail how such difficulties are resolved in our implementation of GARDEN.

5.2.1 Encoding and Data Structure of a Solution

Encoding of a solution greatly influence the implementation of the genetic algorithm for a specific problem. Since the role of binary decision variables in our model closely matches the role of genes in the reproduction process of living organisms, it is plausible to build the chromosome representation of a multi-chip system architecture from binary decision variables.

However merely concatenating binary decision variables into a bit string is not suitable to express a complicated structure like the tree representation of our target architecture. Further, such a long bit string complicates the definition of crossover. Since the minimum unit of design decisions at the system-level is a task, the exchange of genes at the task boundary considerably simplifies the implementation. With the bit string encoding, exchanging all decisions relevant to a task together during crossover is difficult. To simplify the exchange of genes, a binary substring representing a task is stacked together as shown in Figure 5.2. The encoding can be further simplified to take advantage of the fact that only one binary variable regarding a specific design decision for a task can have '1'. Therefore, the chromosome representation can use an integer to represent a design decision instead of bit strings. Since we also need to represent design decisions for communication channels, we use two chromosomes, namely a task chromosome and a channel chromosome, to represent a multi-chip system architecture.

The tree representation of a solution can be constructed from the encoded design decision information in a chromosome as shown in Figure 5.3. The tree representation is implemented in an object-oriented manner such that a tree is composed of hierarchically organized classes of objects at different abstraction levels. The metric functions defined for each object in Section 3.2 are associated as methods with each object class.

COTS devices can be defined as a subtree at various abstraction levels. An existing chip can be represented as a subtree rooted at package level. An existing die can be represented as a subtree rooted at the die level. An existing design in

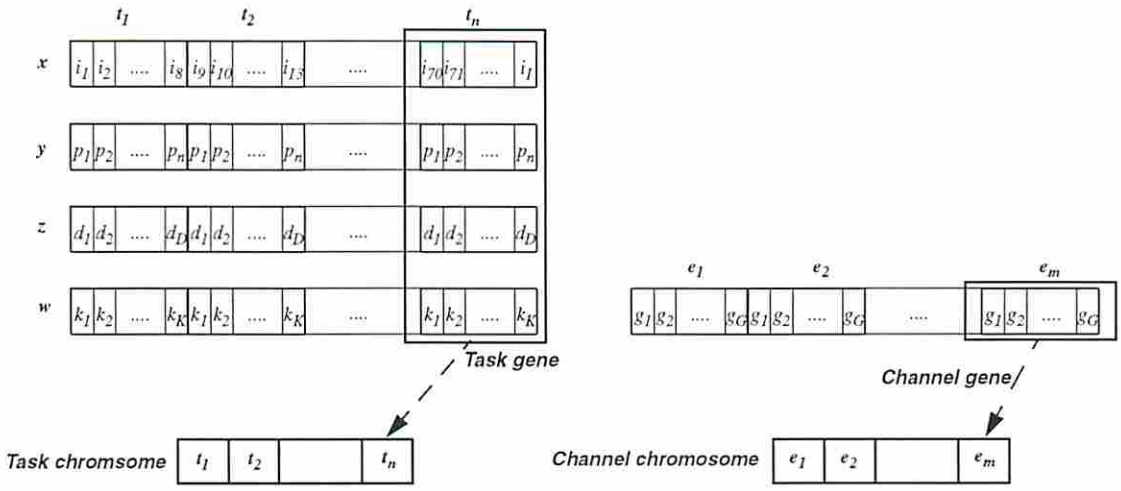


Figure 5.2: Chromosome representation of a system architecture

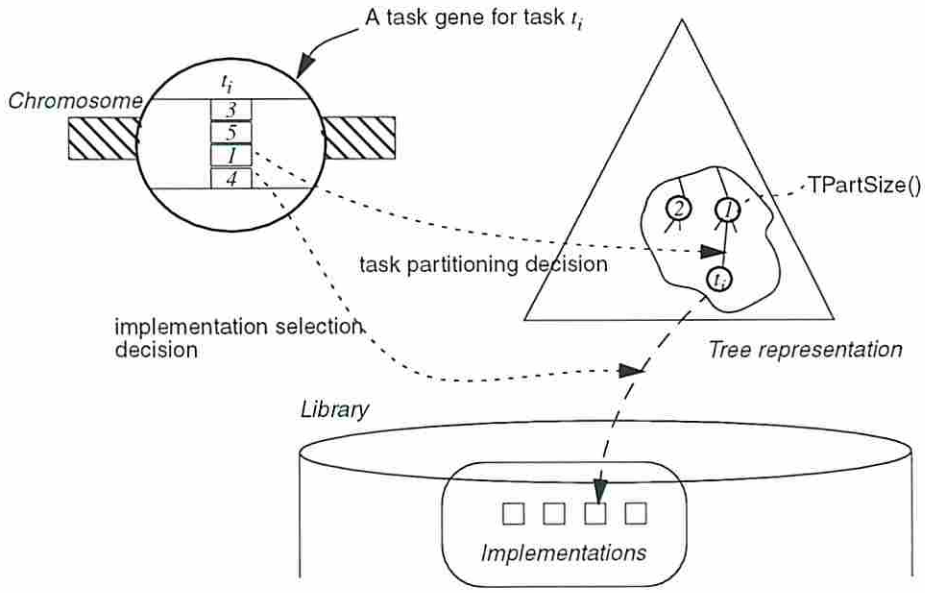


Figure 5.3: Building the tree representation from the chromosome

the form of layout can be represented as an possible datapath architecture element. The meaning of a COTS device is semantically defined by preserving the structure of a subtree during optimization process.

5.2.2 Initialization

The beginning step of the evolutionary process is to generate a number of solutions that will evolve. The initial population could be created by either random generations or heuristics. For the multi-chip system design problem, a solution can be generated fast and easily by a random number generator which fills the decision slots of the structured integer representation of a solution. There are two possible ways to make random decisions. The random decisions on the design steps can be made *concurrently* or *sequentially*. Making concurrent decisions means that a decision on each design step is made independent of other design steps while sequential decision making means that a decision on a design step respects those decisions already made on other design steps.

In concurrent random generation, decision making is free from ordering and therefore expected to generate unbiased solutions. However, because random generation could produce invalid solutions, the validity of newly created solutions must be checked. If a solution is found to violate any validity constraint, the solution is discarded.

If there are m design steps for tasks, n design steps for channels, c validity constraints for a design problem with T tasks and V channels, the total complexity of successfully creating a solution is given as follows:

$$O(m * T) + O(n * V) + O(c * g(T, V))$$

where $g(T, V)$ is the complexity of checking a validity constraint.

Let p_i be the probability that a validity constraint i is satisfied by a given set of design decisions. Then the probability of creating a valid solution by random decisions is $\prod_{i=1}^{c} p_i$. If there are 10 validity constraints and p_i=0.5, then 1 out of 10,000 tries succeed in creating a valid solution. Let t_0 be the time taken for design

decisions and let t_i be the time needed to check validity constraint i. Then time τ to generate a valid solution is given as follows:

$$\tau = t_0 + \sum_{i=1}^{c} \frac{t_i}{\prod_{j=1}^{j=c-i} p_j}$$

The above equation shows how expensive it is to generate a valid solution by *concurrent decision making*. Experimentally, the probability of generating a valid solution is 1.0×10^{-5} for even a simpler model of fewer design steps. Such low probability of generating a valid solution makes concurrent decisions impractical for our early analysis purpose, which requires fast evaluation of system architectures.

As an alternative, keeping invalid solutions in the population or *repairing* an invalid solution is considered. In the first scheme, an invalid solution is retained in the population with a given survival probability. The survival probability determines the time for initialization of the genetic algorithm. Considerably high probability is required to reduce the time for the initialization because of the high birth rate of invalid solutions, which severely corrupts the population so as to prevent the genetic algorithm from converging toward meaningful solutions. In addition, defining recombination operations such as crossover and mutation for invalid solutions is very difficult if possible at all, as we will see in the following sections. Therefore, it is not a viable option.

In the second scheme, which is called the *repair* scheme, violations on validity constraints are corrected by changing decisions in connected design steps. There can be more than one way to repair a violated validity constraint as illustrated in Figure 5.4. The dotted edge e_i represents a violated constraint. We can change either decision A or B to make e_i valid. The selection of design design steps to be repaired can be done either randomly or based on predetermined priority similar to the decision making process. Unfortunately, the random repair scheme could create an oscillation problem if a solution violates more than one constraint. Imagine a case that another constraint e_j connected to A is also violated. Though the validity constraint e_i becomes satisfied by fixing A in Figure 5.4, A can be again chosen to fix e_j in the next repair step and e_i can become invalid again as

a result of fixing e_j. In order to avoid this oscillation problem, we have to impose the ordering of fixes by predetermined priority on each design step. However, such prioritization is equivalent to sequential decision making described below. Therefore, in our GARDEN implementation, we employ *sequential decision making* for creating initial solutions.

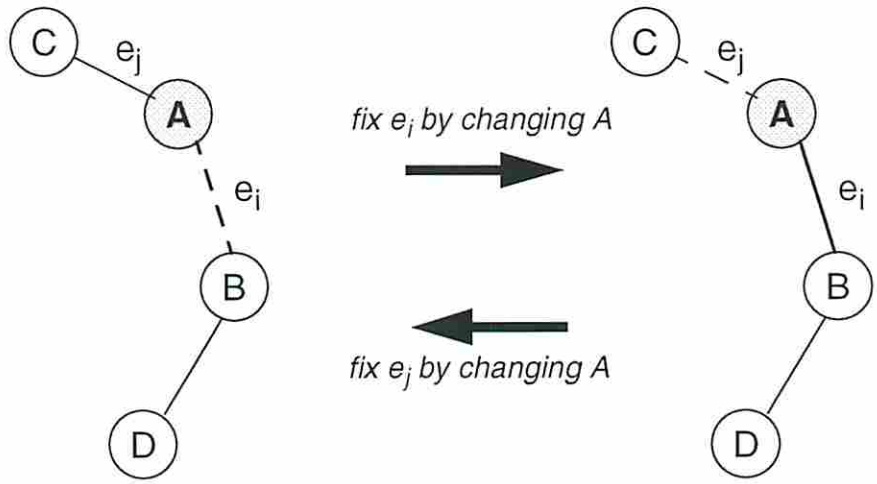


Figure 5.4: Oscillation of solution state by the random repair

In sequential decision making, design steps are solved in a predetermined order. When each design step is solved, previous design decisions are respected. For example, the ith design step must conform to design decisions on the previous $i-1$ design steps. For example, imagine that a die selection step is the first design step and the task-level system partitioning design step follows. The number of dies in a system and their types are already known when the task-level system partitioning is to be made. In sequential decision making, task-level system partitioning decision is made such that no task partition is bigger than the selected die types and needs more pins than the selected die types. Therefore, validity checking involving task-level system partitioning and die selection is unnecessary in sequential decision making. In Equation (5.1), the denominators become equal to 1, therefore, the

initialization process can be sped up considerably. The ordering of design steps used for GARDEN is shown in Figure 5.5.

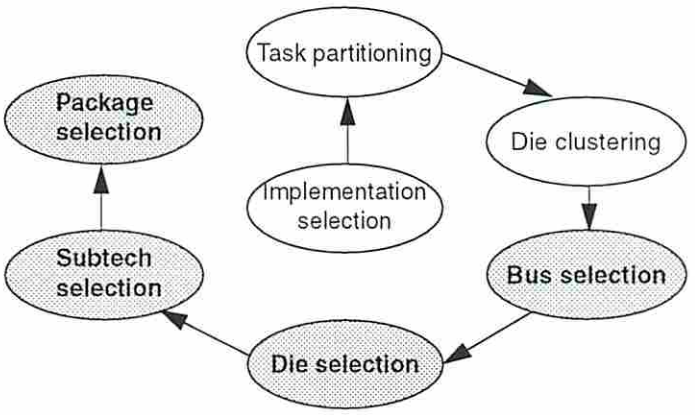


Figure 5.5: The ordering of design steps used in GARDEN

Sequential decision making exhibits an interesting property which can be exploited for further speeding up the optimization process. Since design steps are solved sequentially, some design steps can be solved optimally quite easily by ordering them properly as we discussed in Chapter 3.

Let us consider the task-level system partitioning and die selection step pair again. Solving this pair of design steps simultaneously to minimize the number of dies, the total utilization of die areas, and the number of I/O pins is a difficult optimization problem. However, if they are solved in sequence, the second problem is greatly simplified, especially when task-level system partitioning is done first, and the number of task partitions and their metrics is known. Die selection can be easily solved with a best fit algorithm. Therefore, there is no need to make a random decision in solving die selection. By ordering carefully and using a greedy algorithm for some of design steps, the design space can be reduced considerably. The shaded ovals in Figure 5.5 indicate the design steps that are solved with a greedy algorithm in GARDEN. For task-level system partitioning and die clustering, we use a max-partition heuristic which creates the maximum possible number of partitions in the beginning. Therefore, no random decision making is necessary for these design

steps. As we will show later, our crossover and mutation operators are designed to decrease the average number of partitions in the population and work fast when there are more partitions. By beginning with the maximum number of task partitions, those operators can perform search more effectively.

5.2.3 Selection

The selection mechanism used in GARDEN is *remainder stochastic selection.* In remainder stochastic selection, a solution puts a number of copies into the intermediate population $\mathbf{C(t)}$ according to the integer portion of f_i/f_{avg} where f_i is the fitness value of solution i and f_{avg} is the average fitness of solutions in the new population $\mathbf{P(t)}$. Then a solution can put an additional copy in $\mathbf{C(t)}$ with the probability corresponding to the fractional portion of f_i/f_{avg}.

5.2.4 Crossover

Crossover and mutation are two primary mechanisms to explore the design space in genetic algorithms. The generic crossover operation is done by randomly cutting a chromosome at a fixed point and swapping genes between two chromosomes at the cut point. Although this technique is simple to implement, it is not appropriate for the multi-chip system design problem for a number of reasons. Cutting a task chromosome corresponds to splitting the corresponding system architecture into two parts. The structure of a solution can be disturbed by random fixed-point splitting as illustrated in Figure 5.6. Such a random change is very likely to disturb the validity of a solution as we have seen for the initial population creation. Therefore, an expensive validity checking is required to prevent the population from being corrupted by invalid solutions. If invalid offspring are discarded, creating the next generation with crossover takes an intolerably long time as in the random initial population creation. In addition, since fixed point splitting of a solution could cut a partition into two while never merging two partitions into one, the average number of partitions of solutions increases over generations. Finally, such an exchange of fixed genes limits the possible configurations of solutions because two solutions swap only a fixed part of each solution. Therefore, a new crossover

scheme other than the fixed point crossover is desired that does not disturb the validity of a solution while not increasing number of average task partitions and allowing more exploration capability.

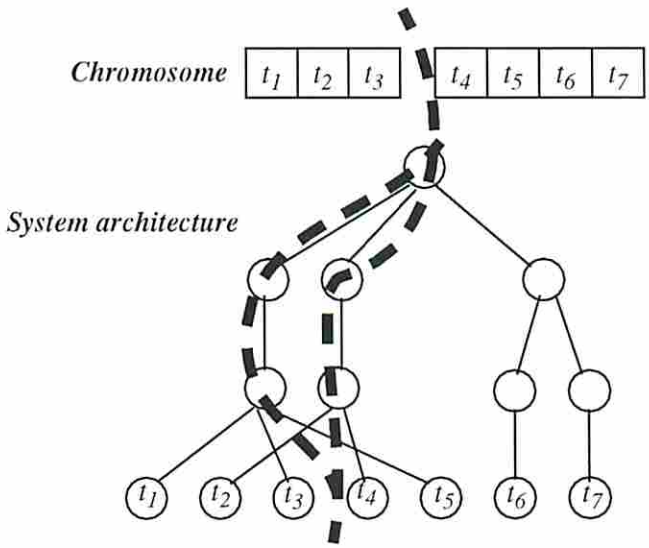


Figure 5.6: Illustration for the problem of partitioning a chromosome at a fixed point

In multi-chip design, the most fundamental unit of exchange is a package. We can create valid new designs by exchanging a set of packages between two designs if the set of tasks in exchanging sets of packages are equal. Such a condition which must be satisfied by two solutions to be crossed over is called *compatability condition* and the process of finding a compatible solution is called *mating*. The compatibility condition is defined below:

Definition 6 Compatibility condition *Let A and B be sets of task genes, $A = \{t_1, t_2, ..., t_n\}$ and $B = \{t_1, t_2, ..., t_n\}$. Let A_0 and A_1 be two disjoint and exhaustive subsets of A, i.e. $A = A_0 \cup A_1$ and $A_0 \cap A_1 = \emptyset$ such that $pkg(t_i) \neq pkg(t_j)$, $\forall i \in A_1$ and $\forall j \in A_2$ where $pkg(t_i)$ is a function that returns the package for task t_i. Let B_0 and B_1 be two disjoint and exhaustive subsets of B. B is* **compatible** *with A if $A_0 = B_0$, $A_1 = B_1$ or $A_0 = B_1$, $A_1 = B_0$.*

From the above definition, we can compute the probability of finding a compatible solution from a given population which is a sample space of the design space. Let A_1 be a subset of size m chosen out of T tasks. Let tp_1 be a set of disjoint and exhaustive subsets of A_1 such that $A_1 = \cup_{p \in tp_1} p$ and $\cap_{p \in tp_1} p = \emptyset$. Let TP_1 be a set of tp. Since $pkg(t_i) \neq pkg(t_j)$, $\forall i \in A_1$ and $\forall j \in A_2$, a compatible solution must have $tp_1 \in TP_1$ as part of its solution. The size of TP_1 is computed by Equation 3.6. For A_2 part of the solution, a compatible solution must have $tp_2 \in TP_2$ as part of its solution. Therefore, the number of compatible solutions for a given split solution in the design space is given as follows:

$$\sum_{k=1}^{m} f(m,k) \times \sum_{k=1}^{T-m} f(T-m,k))$$

Since the number of possible task-level partitionings of a system in the design space can be computed by Equation 3.6, The probability for a randomly selected solution from the population being compatible to the given split solution given as follows: From Equation

$$P(m) = (\sum_{k=1}^{m} f(m,k) \times \sum_{k=1}^{T-m} f(T-m,k))/\sum_{k=1}^{T} f(T,k)$$

The probability is plotted in Figure 5.8 from $m = 1$ to $m = 9$ for $T = 10$. Since it is reasonably high, therefore, the time to find a compatible solution is not significant if a compatibility function can be implemented in $O(n)$. One such $O(n)$ algorithm is shown in Figure 5.9.

Equipped with the above compatibility operator, we can build a crossover operator as shown in Figure 5.10 which preserves the structure of a solution while effectively searching the design space.

A solution is chosen randomly from the pool of intermediate solutions $\mathbf{C(t)}$ and is partitioned at the random package boundary into two. In the *mate* function, a compatible solution $S[j]$ is found by checking the compatibility of a candidate which is chosen randomly from $\mathbf{C(t)}$. The mate solution $S[j]$ is then split into two parts according to the boundary defined during the compatibility check. New solutions are created and stored in the next generation $\mathbf{P(t)}$ by merging parts of

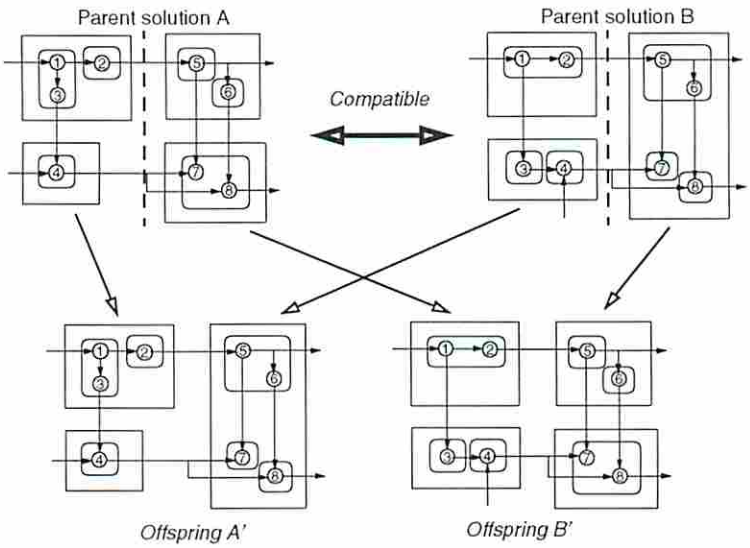


Figure 5.7: Illustration of crossover

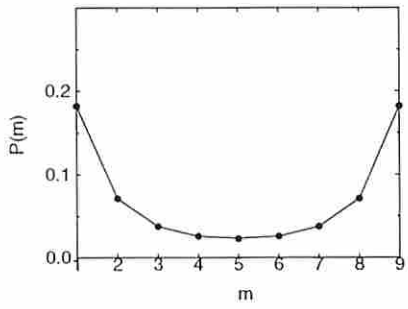


Figure 5.8: The probability of finding a compatible solution as a function of m

```
CheckCompatibility (male solution M, female solution F)
for each mark[i] of F
    mark[i] ← -1;
end
for each die partition dp in F{
    for each task gene tp in dp{
        if mark[tp] of F ≠ −1 and mark[tp] of F ≠ prev_mark
          then
                return not_compatible;
           else
                mark[tp] of F ← mark[tp] of M;
                prev_mark ← mark[tp] of M;
        fi
    end
end
return compatible;
```

Figure 5.9: Compatibility checking algorithm

```
Crossover()
randomly pick a solution S[i] from population C(t);
randomly split S[i] into two set of genes, S_0[i] and S_1[i] at the package boundary;
j ← mate(S[i]);
split S[j] into S_0[j] and S_1[j];
merge S_0[i] and S_1[j] into S[i] in P(t);
merge S_0[j] and S_1[i] into S[j] in P(t);
```

Figure 5.10: Algorithm for the crossover operation

chromosomes from both solutions. In Figure 5.11, the crossover operation for the tree representation is illustrated. A solution is split into two subset of subtrees rooted at the package level. The subtrees in one subset is represented with filled circles and the subtrees in the other subset is represented with clear circles in Figure 5.11. By merging the subsets of subtrees with the same type of circles, new solutions are created.

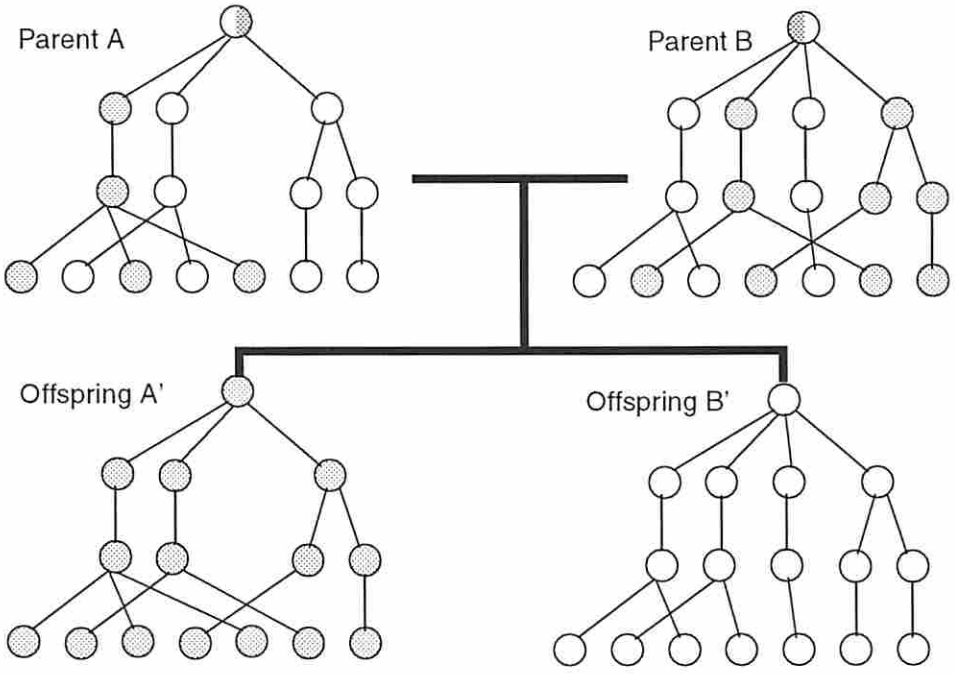


Figure 5.11: Illustration of crossover operations based on compatibility

5.2.5 Mutation

The crossover operation described in the previous section allows GARDEN to search the design space for all possible combinations of existing designs in the current population. But, as the reader can easily see, the searchable design space by crossover alone is limited to the combinations of existing design decisions in initial solutions. In addition, our crossover operator cannot work on a solution of a single package. Another reason for using the mutation operation is that certain values

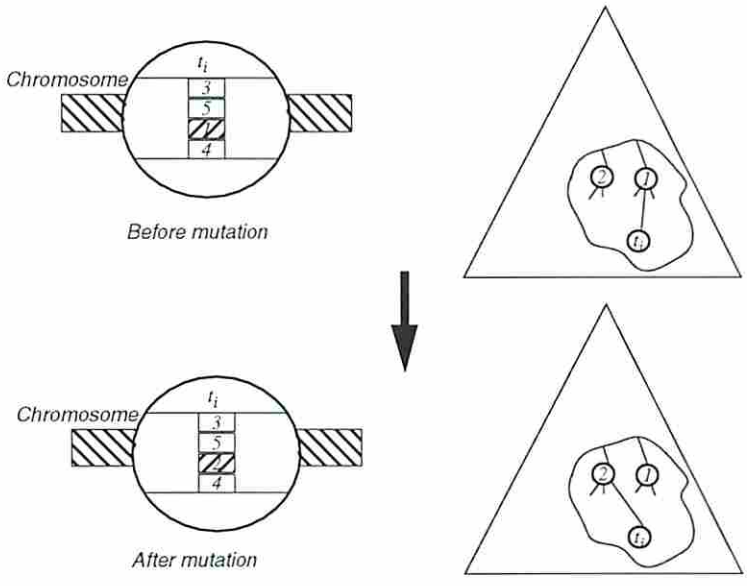


Figure 5.12: Illustration of a mutation operation for task-level system partitioning

of genes are depleted as the search progresses over generations[Whi93]. Mutation enhances the search capability of the genetic algorithm by reintroducing depleted genes into the population characteristics.

In the generic genetic algorithm, the mutation operation is implemented as a mechanism of changing part of a solution randomly. The mutation operation is illustrated in Figure 5.12 in which a task-level system partitioning decision on task t_i is changed as a result of the mutation. It is highly likely that such a random change of part of a chromosome will corrupt the validity of a design. For example, a migration of a task from task partition 1 to task partition 2 makes the destination task partition 2 bigger and the selected die type of task partition 2 might be too small to accommodate the new task partition or t_i might not be implemented with the same style as task partition 2. Such a possibility of corrupting the validity of a solution necessitates validity checking to avoid invalid solutions. Again, the situation described for the initial population generation and the generic crossover is repeated. Therefore, a scheme that can save time for the mutation operation without incurring too much computational cost is desired.

Our first scheme to reduce the run time of the mutation operation is to avoid generating invalid solutions, which can be accomplished by preventing any mutation that will result in a invalid solution. In the previous example, the violation of the task style constraint in mutating task-level system partitioning can be easily avoided by restricting the possible range of task partition migration to those of the same physical design style. Since restricting the mutation operation for task-level system partitioning can be done without increasing computation time significantly and the search capability remains the same, the first scheme is very effective in preventing the violation of the task style constraint. However, some mutation operations require a considerably increase in run time to be restricted and too much restriction would result in the loss of searching capability. The die size constraint in the previous example is one such constraint. With the above scheme, the mutation operation should make sure that the die type for task partition 2 has enough space for task t_i before moving which requires the recomputation of die size metric function for all other task partitions in the worst case.

We already introduced the concept of *repair* to avoid the long run time when discarding invalid solutions. Random repair does not reduce the run time because of the oscillation problem. However, just as we ordered the design steps, by ordering the design steps for repair, we can prevent the oscillation problem. We call this scheme *prioritized repair*. For example, if there is not enough space for the selected die type for task partition 2, by selecting another die type for task partition 2, a mutation operation can be successfully completed without restricting the operation. Therefore, we can retain the freedom of mutation while not increasing the run time. In the GARDEN implementation, the priority of design steps for repair is as follows;bus selection, die selection, and package selection. If a solution is irreparable, the mutation is aborted and the original solution is restored.

5.2.6 Evaluation and Fitness Functions

Each solution in the population is evaluated at the evaluation step with a function called the *fitness function*. Solutions with higher fitness will have higher probability of generating offspring. Since the behavior of a genetic algorithm is very sensitive

to the form of a fitness function, designing a good fitness function has critical importance for both run time and the quality of solutions.

There are three major metrics which we defined in Section 3.2 for the multi-chip system. With the tree representation and associated metric functions, computing metrics for a solution is easily and systematically carried out by calling metric functions in a bottom-up fashion, from the leaf elements (tasks) to the root element (system) (See Figure 5.3).

For multi-chip system design, optimization tools should be able to handle different optimization modes. For the multi-chip system design problem, the designer might desire to optimize system architectures for different objective combinations and different constraint combinations. Different optimization modes can be expressed as follows: Each system metric can be one of the metric state modes { *objective, constrained, don't-care* }. *Objective metric* is a metric that is either minimized or maximized during the optimization process and only one metric can be objective. A *constrained metric* must be satisfied by all solutions in a population to be feasible. A *don't-care metric* can have any value. As a combination, various optimization modes can be described. For example, experiments with EDEN can be expressed as (prototyping time=objective, cost=constrained, performance=constrained). If the designer wants to find a minimum cost solution under a performance constraint while not being concerned with the prototyping time, such an experiment can be expressed as (cost=objective, performance=constrained, prototyping time=don't-care).

The next important issue in designing a fitness function is the strategy of dealing with infeasible solutions. Various schemes are possible depending on the fitness function definition. For convenience, we recite our definition of feasibility of a solution as follows:

Definition 7 *A* **feasible** *design at the system-level is a set of design decisions which is valid and also satisfies following conditions: Let* m_0 *be an objective metric while* $m_1, \ldots, m_n$ *are constrained metrics. Let* $c_1, \ldots, c_n$ *be upper-bound constraints*

on corresponding metrics. When $c_i - m_i \geq 0$, a solution is feasible for m_i. A solution is said to be feasible if it satisfy the following condition:

$$c_i - m_i \geq 0, i = 1, \ldots, n$$

For the constrained-optimization problem, Defining a function with all metrics $F(m_0, c_1 - m_1, \cdots, c_n - m_n)$ is the most popular way of devising a fitness function. In order to use remainder stochastic sampling, $F \geq 0$.

The first scheme of handling infeasible solutions is to discard them and only feasible solutions are retained in the population. The following fitness function defines such a scheme:

$$F(m_0, c_1 - m_1, \cdots, c_n - m_n) = \begin{cases} m_0 & \text{if a solution is feasible} \\ 0 & \text{if a solution is not feasible} \end{cases}$$

A number of disadvantages arise using this scheme in evaluation. First, when the birth rate of infeasible solutions is higher than that of feasible solutions, initializing a population can take a very long time. Second, when the relationships among metrics are inversely proportional to each other, maintaining population size is very difficult. For example, if we want to optimize the cost, then the genetic algorithm produces more solutions that have lower cost but slower performance. Therefore, merely driving a population to the cost optimization will eventually force all solutions to be infeasible. Third, infeasible solutions can produce feasible solutions during recombination operations. So, it is desirable to keep infeasible solutions which do not violate the user-specified constraints too much, which points to the necessity of quantifying the degree of infeasibility, i.e. how much a solution violates the specified constraint as the next fitness function.

The common form of a fitness function for retaining infeasible solutions is given below:

$$F = W_0 \cdot m_0 + W_1 \cdot (c_1 - m_1) + \cdots + W_n \cdot (c_n - m_n)$$

where $W_0, W_1, \ldots, W_n$ are weighting factors. This fitness function can avoid some of the problems with the former fitness function but adds other types of problems

for the purpose of optimization. (1) Finding a set of W_i of the above conditions for a fitness function that satisfies even the simple condition $F > 0$ is difficult. (2) From fitness function value, we cannot tell whether a solution violates any constraint or not because a solution can violate one constraint and have a very high objective value and the fitness can remain positive. (3) For the same reason, we cannot tell whether one solution is better than others in terms of the objective metric.

From the above fitness function, we can identify the following desired characteristics of a good fitness function:

1. distinguishing infeasible solutions from feasible solutions,
2. distinguishing better solutions among feasible solutions, and
3. distinguishing less infeasible solutions among infeasible solutions.

Mathematically the conditions for a fitness function can be expressed as follows:

$$F(m_0, c_1 - m_1, \cdots, c_n - m_n) \geq 0 \tag{5.1}$$

$$F\,|_{c_i \geq m_i \forall i} > f(M, y)\,|_{c_i < m_i \exists i} \tag{5.2}$$

$$\frac{\partial F}{\partial m_i} < 0, i = 1 \ldots, n, \frac{\partial F}{\partial m_0} = 0 \text{ if a solution is infeasible.} \tag{5.3}$$

$$\frac{\partial F}{\partial m_i} = 0, i = 1 \ldots, n, \frac{\partial F}{\partial m_0} > 0 \text{ if a solution is feasible.} \tag{5.4}$$

The Equation 5.1 is required to use the remainder stochastic selection. The Equation 5.2 states that any feasible solution must have higher fitness than all infeasible solutions. The Equation 5.3 enforces that the objective metric should not influence the fitness of infeasible solutions. Finally, the Equation 5.4 requires that the fitness of any feasible solution should not be influenced by constrained metrics.

Devising a function that satisfies the above conditions would be difficult. Also, even if it were possible, we would need to define as many fitness functions as the number of possible optimization modes described above. The fundamental reason for difficulty with the above schemes comes from mixing different metrics in a single function.

To solve the problems associated with the above schemes, we developed a different optimization scheme for GARDEN called *Finite State Optimization*. In this scheme, each metric has its own fitness function, $f_i(m_i)$. Constraints are imposed on average population characteristics rather than on an individual solution, which allows infeasible solutions to be retained in the population. For each metric, there is a separate optimization state. In one optimization state for a metric, the corresponding fitness function is used. When the average characteristics of the population violate one of the constraining metrics, according to the predefined FSM-like control mechanism, the optimization mode switches to the other state. For GARDEN, three separate fitness functions, $f(SPERF)$, $g(SCOST)$, and $h(TTM)$ are defined. The state diagram for an optimization mode (prototyping time=objective, cost=constrained, performance=constrained) is shown in Figure 5.13.

In the beginning, solutions are optimized for either the performance or cost, which are constraint metrics, then optimized for the prototyping time after both the average fitness regarding performance and cost reach the critical values which are defined as follows:

$$\begin{aligned} CriticalPerfFitness &= f(CPerf) \\ CriticalCostFitness &= g(CCost) \end{aligned}$$

where $CPerf$ and $CCost$ are user-specified performance and cost constraints. When solutions are optimized for the prototyping time, the population characteristics can deteriorate in terms of cost and/or performance. Then, the population is again optimized for the violated metric according to the corresponding *optimizing condition*. In Figure 5.13, such conditions for optimization state transitions are denoted by P and C which are defined as follows:

$$\begin{aligned} AvgPerfFitness \geq CriticalPerfFitness &\longrightarrow D = 1 \\ AvgCostFitness \geq CriticalCostFitness &\longrightarrow M = 1 \end{aligned}$$

In Figure 5.13, $\overline{D}$ = not D and $\overline{C}$ = not C, an optimizing state transition occurs whenever a condition becomes true.

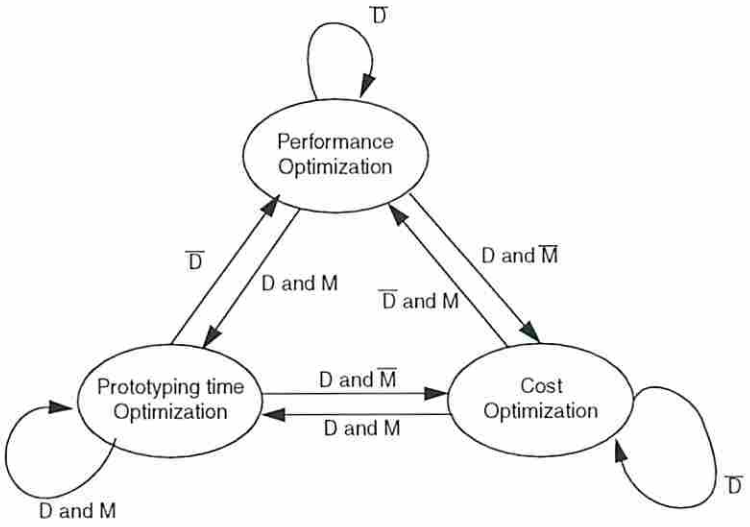


Figure 5.13: Optimization sequence in GARDEN

In Figure 5.14, the normalized average fitness values over generations is plotted. In Figure 5.14, B is the $h(TTM)$ of a best valid solution, $\bar{f}$ is the average $f(SPERF)$, $\bar{g}$ is the average $g(SCOST)$, and $\bar{h}$ is the average $h(TTM)$ of solutions in the population.

5.3 Termination Conditions

Two termination conditions are used in GARDEN, namely, *maximum generation limit* and *minimum improvement requirement*. The first condition sets the upper bound on the number of generations. The second condition, the minimum improvement requirement checks the saturation of improvement over generations, whether the fitness of the best solution improves more than a given improvement requirement whenever a new best solution is found. If the fitness improvement of

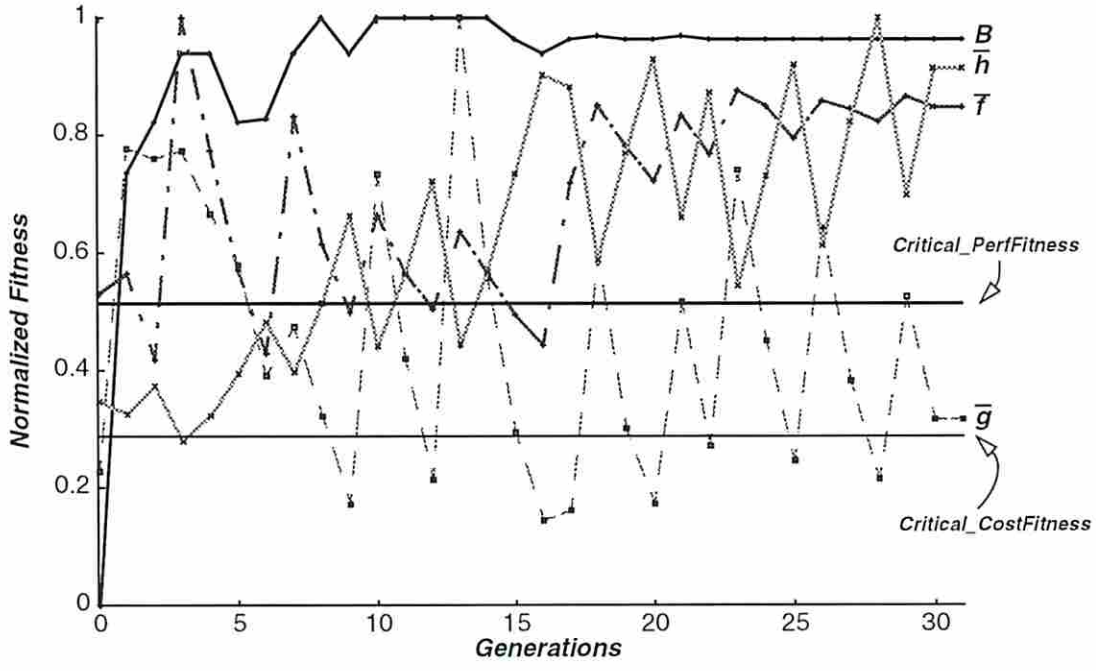


Figure 5.14: Normalized fitness functions

the new best solution is less than the specified requirement for a given number of times consecutively, GARDEN terminates.

5.4 Experimental Results

In the experiments with EDEN, we focused on finding a system that can be rapidly prototyped under different cost-performance constraints. In the experiments with GARDEN, different architectural trade-off analyses are emphasized. Since GARDEN provides controls over many system architecture alternatives, the designer can investigate the influence of design alternatives painlessly. The effects of the physical design style selection, the number of task partitions in a system, and the use of MCMs are examples of such architectural decisions that are experimented with and described in this Section.

5.4.1 Examples

The MPEG example which we used in previous section is not big enough to show the full capability of GARDEN. To demonstrate the complexity of the multi-chip design problem and the capability of GARDEN, we used the following two hypothetical examples constructed with the MPEG encoder as a building block as shown in Figure 5.15 in addition to the MPEG encoder example. The first example is constructed by instantiating the MPEG encoder 4 times and therefore has more tasks and more external connections while the second example has 4 copies of the MPEG encoder connected serially.

5.4.2 The Effect of Physical Style Selections

The first trade-off analysis is to find the effect of physical design style selections on prototyping time, system cost, system performance and the number of task-level system partitions. For each FPGA, gate array, and standard cell design style, we had GARDEN generate cost-optimized architectures under different performance constraints when 10^6 units are assumed to be produced.

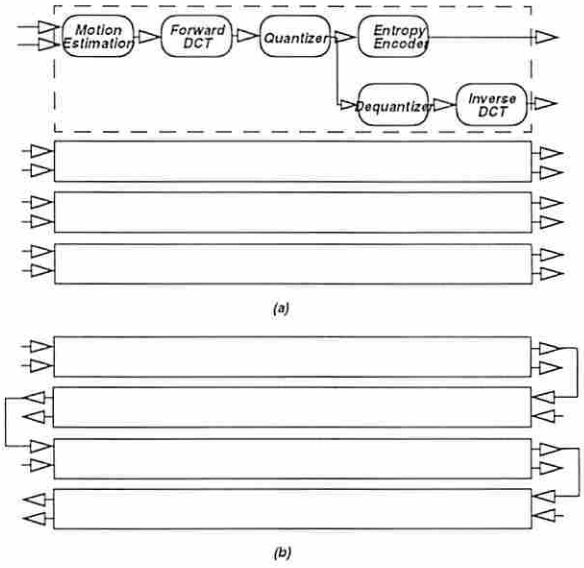
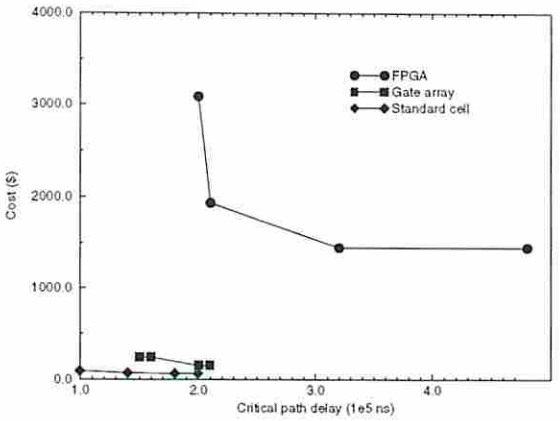


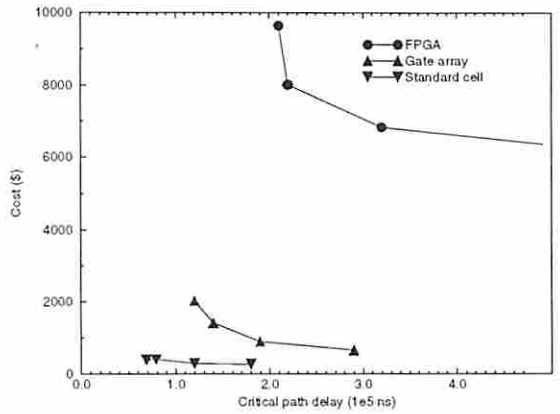
Figure 5.15: Examples used in GARDEN experiments. (a) parallel example (b) serial example

The results are plotted in Figure 5.17. It shows the design space reachable with different physical design styles. Under the mass production assumption, a system designed with the standard-cell style is superior to other designs with either FPGA or gate-array style in both performance and cost. However, the advantage of standard-cell designs comes at the cost of prototyping time. In Table 5.1, the prototyping time for the optimized designs of the MPEG example is shown together with the number of task partitions. In order to reduce the system prototyping time while maintaining the cost at a low level, a mixture of different physical design styles which fall in the area between homogeneous physical-style designs can be considered by EDEN.

The cost advantage of standard cell designs also requires a certain volume of production. In Figure 5.18, the variation of system cost per unit with the volume of production is plotted with the given NRE for standard cells and gate arrays. The standard-cell design maintains a cost advantage over other physical design styles for the MPEG encoder when the NRE cost of the standard cell is equal to \$150,000. The gate-array design becomes more cost-effective when the NRE



(a) MPEG



(b) Example 1

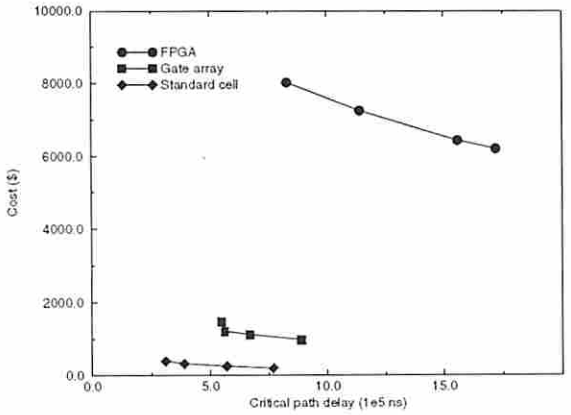


Figure 5.16: (c) Example 2

Figure 5.17: Design space reachable by different physical design styles

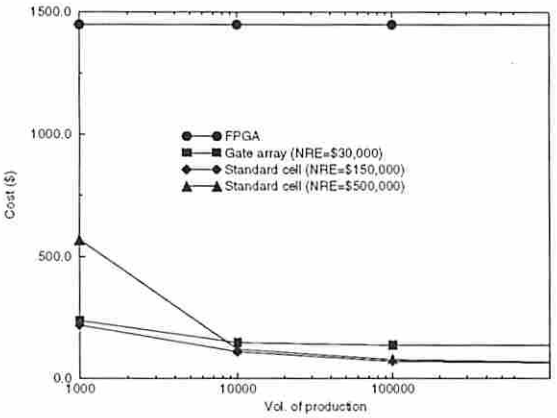
	FPGA		Gate Array		Standard cell	
	No. of TP (dies)	PTime (hour)	No. of TP (dies)	PTime (hour)	No. of TP (dies)	PTime (hour)
Ex1	3	3	4	2688	3	6048
Ex2	3	3	4	2688	3	6048
Ex3	4	3	4	2688	3	6048
Ex4	4	3	4	2688	3	6048

Table 5.1: The number of task partitions (No. of TP) and the prototyping time (PTime) for the optimized architectures for the MPEG encoder.

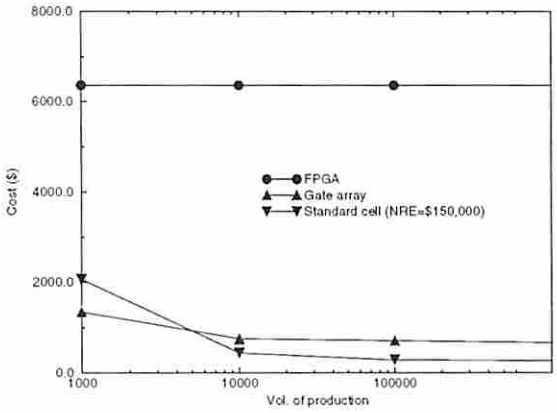
cost of the standard-cell design is $500,000. For Example 1, gate-array designs becomes cheaper than standard-cell designs for a small volume of production (less than 5,000 units).

5.4.3 The Effect of the Number of Task Partitions

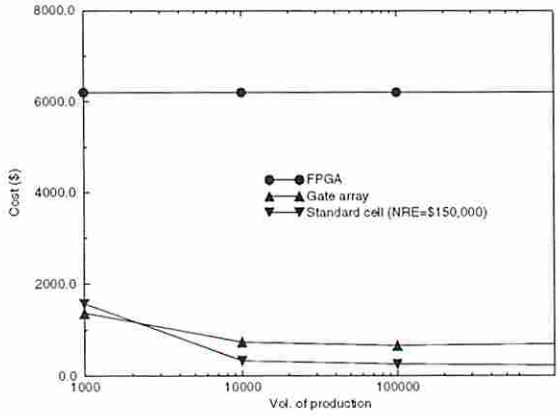
Setting the number of task partitions in the early stages of design primarily determines the project schedule and board size. Moreover, the system cost and performance are influenced by the number of task partitions. In this experiment, we investigate the effect of the number of task partitions on system characteristics. The results for the MPEG encoder are shown in Figure 5.19 (a). The cheapest designs are 3-chip systems for all constraints. The high cost of 2-chip systems are due to the high pin count requirements of one of the packages. Figure 5.19 (a) shows that neither a single chip design nor a max-chip design is cost effective and that a system can be far from the optimum because of unexpected factors like the I/O pin count requirement. The effect on the system cost of example 1 for different numbers of task partitions under different performance constraints is plotted in Figure 5.19 (b), which shows other interesting factors about design problems and GARDEN. First, except for the case of high performance systems, the low-cost designs are found between 11 and 16 partitions. Second, the cost of a system with 12 task partitions and a 2.6×10^5 ns. performance constraint is higher than designs of 12 task partitions and a higher performance requirement. From the qualitative point of view, this point is not well optimized by GARDEN. Since tools



(a) MPEG



(b) Example 1

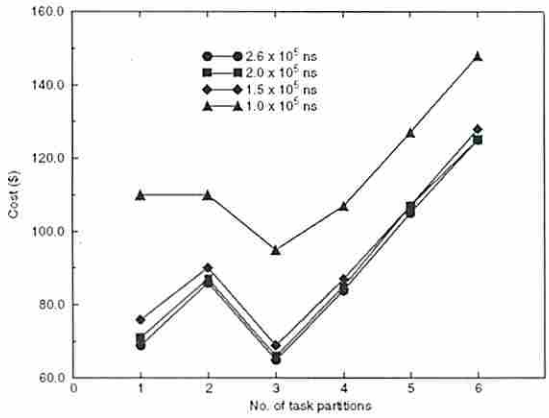


(c) Example 2

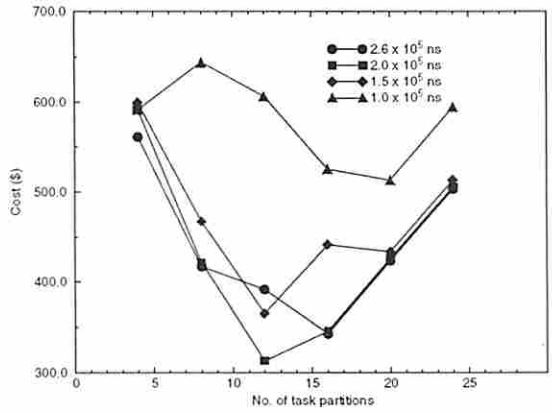
Figure 5.18: The variation of system cost per unit with the volume of production

like GARDEN cannot guarantee to find an optimal solution, there is possibility of getting a result far from an optimum. However, the designer can avoid drawing a false conclusion from non-optimal design points by identifying the trends of other optimized points. Once a point is known to deviate far from the optimum, by relaxing the termination conditions of GARDEN, the designer can try to optimize such points further. Third, at 4-chip designs, the cost of systems under different constraints are close each other while one would expect that a high-performance design has much higher cost because of bigger datapath architectures. The primary reason for this phenomenon is the high pin count requirements of Example 1. By comparing the results of experiments of the same type for Example 2 as shown in Figure 5.19 (c), it is evident that the cost for a high-performance system is higher across different numbers of task partitions. The comparison between Figure 5.19 (b) and Figure 5.19 (c) revealed that Example 2 has in general lower system cost and is more easily optimized by GARDEN.

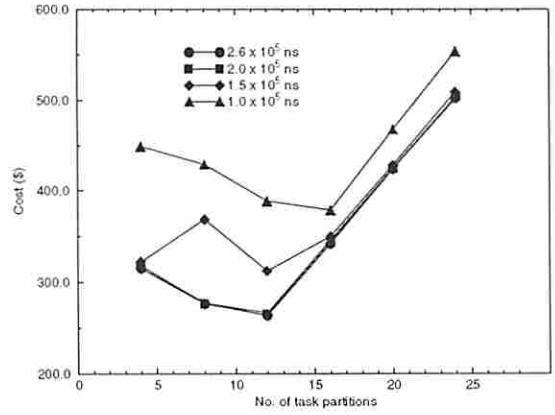
The system performance variation with the number of task partitions for the MPEG encoder and Example 2 is plotted in Figure 5.20. GARDEN generated a number of performance-optimized architectures under different cost constraints. When the cost constraint is $10,000, the critical path delay increases with the number of task partitions because of the introduction of off-chip buses while the fastest datapath architectures can be used for tasks. When the cost constraint is very tight, GARDEN must budget carefully the cost between silicon and package such that the performance is maximized within the given cost constraint. For a single chip or 2-chip design, the high I/O pin counts require an expensive package. Therefore, datapath architectures that are affordable are small and slow. On the other hand, the increasing number of task partitions increases the cost of packages and forces GARDEN to select smaller and slower datapath architectures and therefore the performance drops rapidly. GARDEN was not able to find a 6-chip design since the package cost alone is higher than the cost constraint. The experiments for Example 2 shows similar trends. the optimum performance design with the tight cost constraint is somewhere between 12 and 16 partitions. As one might expect, a more tightly constrained problem demands a better partitioning scheme to reach a better system characteristics.



(a) MPEG

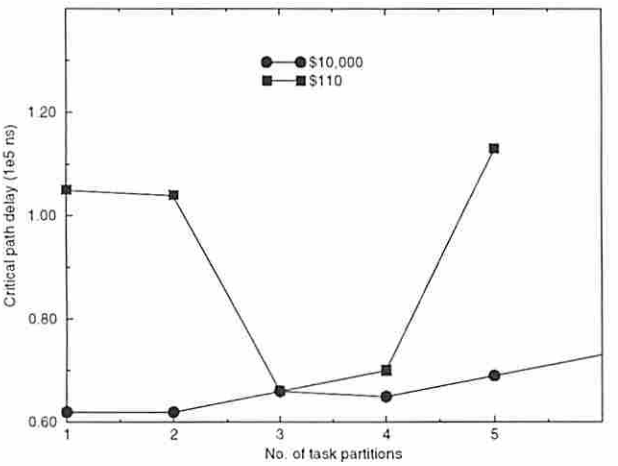


(b) Example 1

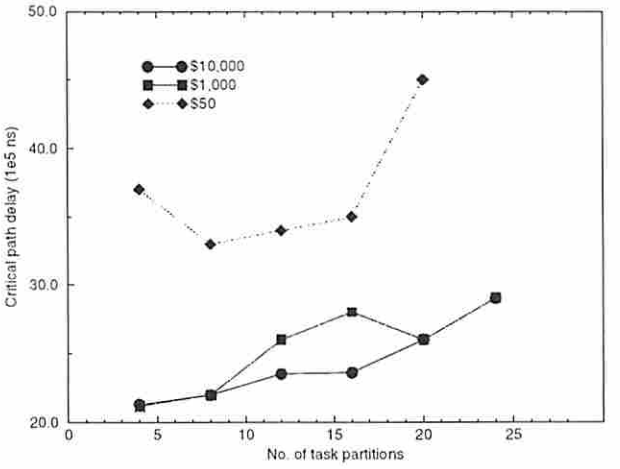


(c) Example 2

Figure 5.19: The variation of system cost by the change in the number of task partitions



(a) MPEG



(b) Example 2

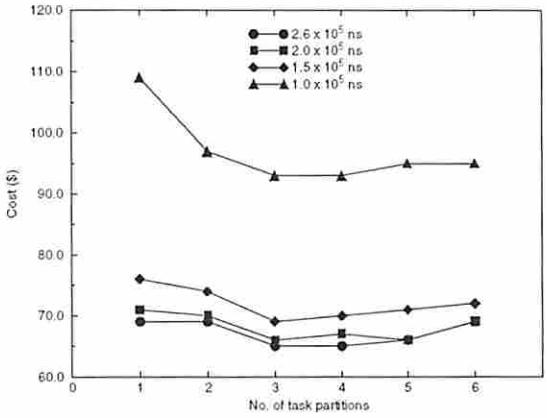
Figure 5.20: The variation of system performance with the number of task partitions.

5.4.4 The Influence of MCMs

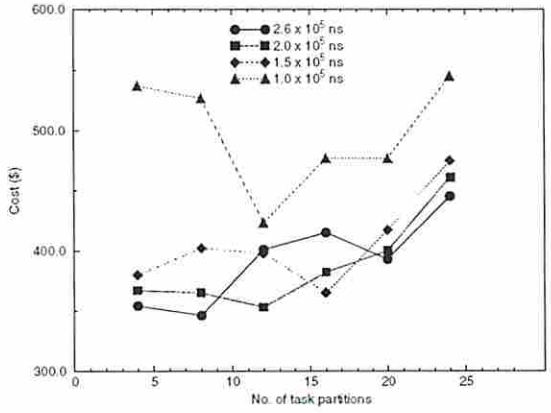
In this experiment, we investigated the effect of employing MCMs over system characteristics. For both examples, we had GARDEN find a cost-optimized system for 4,8,12,16,20, and 24 partitions. The four different performance constraints from 2.6×10^5 ns to 1.0×10^5 ns were used. The results of experiments are shown in Figure 5.21. By comparing these results to the results in Figure 5.19, we can see the effect of using MCMs as a possible packaging option. The 2-die design of the MPEG example with MCMs is not so expensive comparing to the 1-die design because the MCM packaging absorbs the pin count requirements of the 2-die design. The system cost with a higher number of dies stays rather flat compared to non-MCM designs. For low-performance designs, the cheapest designs with MCMs occurred at a bigger number of task partitions than the cases of non-MCM designs for Example 1 because MCMs can accommodate more dies in a single package. Except the high performance systems at $1.x10^5$ ns, the system cost is more expensive with MCMs than with a single chip packaging.

5.4.5 The Example Designs Optimized by GARDEN

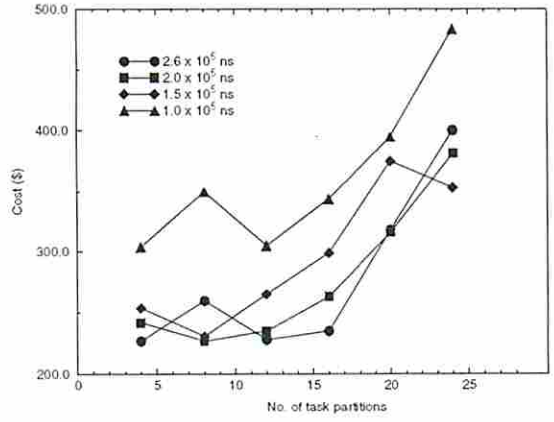
In Figure 5.22, three optimized designs, (a), (b), and (c) for the MPEG example are shown. Designs (a) and (b) are implemented with the standard-cell style and were optimized for cost under the same performance constraints. Design (a) is packaged separately while Design (b) uses MCMs for part of its design. The cost of Design (a) and (b) is roughly the same and the performance of Design (b) is better than Design (a). When MCMs are not allowed, Design (a) is partitioned into 3 dies. The package cost is a major part of the system cost. The rapid increase in the die cost prevents further integration. Another factor suppressing integration is the package cost. In our library, the package type that can accommodate 96 pins is \$60 which is the same as for the cost of 3 packages used in Design (a). By integrating the Encoder task with the Motion estimation die, the system will have two chips and the cost of the system increases \$80. Moving the Encoder task to the Inverse DCT die will not increase the total die cost significantly. However, it still requires a 100-pin die which increases the system cost by \$60. Separating the Quantizer



(a) MPEG



(b) Example 1



(c) Example 2

Figure 5.21: The variation of system cost per unit with the number of task partitions when MCMs are used.

task from the Motion estimation die and integrating it with the Encoder task can lower the total die cost but increase the system cost by $60 again which cannot be compensated for by the die cost decrease. More partitioning of the system can lower the total die costs but increase the package cost.

In Design (b), MCMs are allowed as a possible packaging option. Design (b) also has three packages. However, there are 5 dies in the system. The use of MCMs allows GARDEN to partition further the system into more dies to lower the total die cost and to use a cheap package type with 80-pins to house the first three dies. Therefore, the system cost can be lowered by $10. Unfortunately, the cost gain is compensated by the cost of the substrate. However, the performance can be improved slightly because a bigger datapath architecture can be used without incurring too much additional cost because of the low integration level. Integrating other dies into the MCM can remove more packages but requires a more expensive package type because of increased I/O pin counts.

In Design (c), the system is implemented with FPGA style only without MCMs. The die type used in the design has 139 pins and costs $333. The next smaller die type has 105 pins and cost $111. The next smaller package type costs $60. Therefore, by splitting a task partition into two dies in Design (c), we can save $111+$30=$141 for each task partition. However, the next smaller die type cannot hold the Motion estimation task. Therefore, instead of wasting the unused space of the Motion estimation die, the FDCT task can be integrated into the motion estimation die. If a die type cannot hold both tasks, the problem can be resolved by either selecting the next bigger die type or selecting a smaller datapath architectures for both tasks unless the performance constraints are not satisfied. In this case, the use of the next bigger die increases the system cost more than $1,000 so it cannot be used. Therefore, using smaller architectures is a better choice. If the performance constraints cannot be met with smaller architectures, faster architectures for other tasks can compensate for the loss of performance. The Quantizer die has the same size problem as above because of the Encoder task. The IDCT die does not have the same size problem. The selected architecture for the IDCT task can fully utilize the next smaller die and the selected architecture for the Dequantizer task can fit onto even the smallest die type, which can further lower

the system cost. However, splitting the IDCT die could introduce more communication delay of about 4,000 ns, jeopardize the performance of the system and make the design infeasible.

5.4.6 The Run Time Distribution of GARDEN

The run time of GARDEN depends on the population size, the termination conditions, and the complexity of a given problem. The random nature of the genetic algorithm makes the run time of GARDEN vary for the same number of generations. Figure 5.23 shows the plot of the run time vs. the number of generations for 64 runs on Example 1. The run times shown in Figure 5.23 are for 100 solutions in the population, 200 limits on generations without improvement, and 20 limits on less than 2% improvement. The machine used for runs is a Ultra Enterprise system which has 8 sparc CPUs and 2.0 GB memory.

5.5 Conclusion

We developed GARDEN, a genetic algorithm-based software that can optimize multi-chip system architectures based on the model described in Section 3.2. A number of problem-specific features were developed to apply genetic algorithms to the multi-chip design problem. GARDEN was able to find good solutions quickly compared to MILP-based optimization. The experiments showed the versatility of GARDEN in optimizing a multi-chip system architecture for different optimization modes.

The speed up of the optimization process comes from the reduced design space by incorporating knowledge about the problem characteristics and customized operations such that obvious infeasible solutions are eliminated in the initial population creation stage. By combining the understanding of the problem with the optimization mechanism of genetic algorithms, it was possible to develop a robust and extensible optimization tool. However, as demonstrated from the encoding of a solution to the development of recombination operators, careful design of a genetic algorithm suitable for the multi-chip system design problem was a critical part of the development of GARDEN.

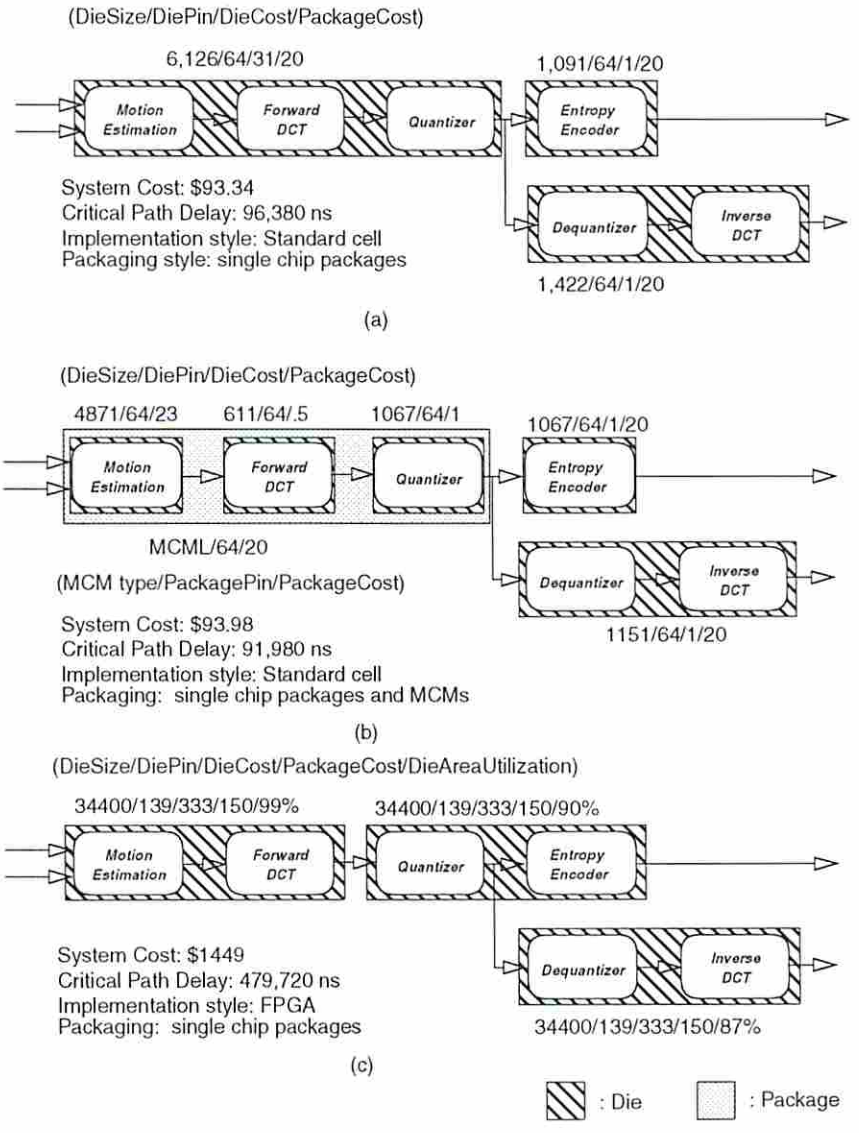


Figure 5.22: Example designs optimized by GARDEN for the MPEG encoder

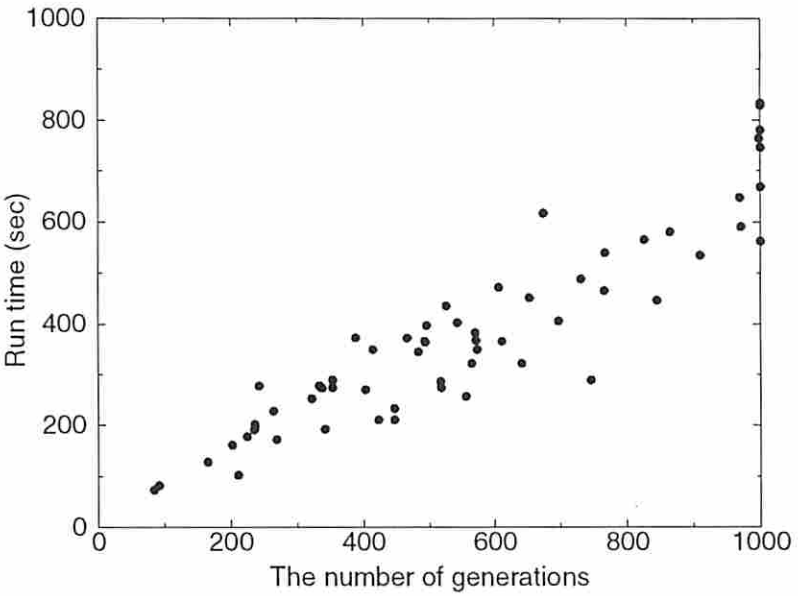


Figure 5.23: The run time vs. the number of generations of GARDEN

Versatility is the most important feature which GARDEN has for multi-chip system design. Experimental results showed how the designer can investigate various architectural trade-offs with GARDEN in the early system development stage. With such trade-off analyses of various system parameters and design alternatives, the designer can make fast and informed decisions that lead to a better system in a shorter development time. However, the design decisions for optimized systems are very sensitive to yield of dies, available die and package types and their cost distribution, and datapath architectures. Therefore, the observed trends in optimized system architectures under various optimization modes cannot be generalized.

The current implementation of GARDEN does not limit the other possibilities of improving and extending our model and optimization mechanism. The complete model and systematic organization of the GARDEN software architecture renders itself useful for other extensions easily.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

Design decisions made for system-level design steps in the early system development stage shape the overall system characteristics. Albeit its importance in the system development process, automating and assisting decision making at this abstraction level has only been researched rather recently. In this research, we attempt to define the system-level design problem and automate early design decision making. For that purpose, we developed a model and optimization tools that can help designers in making critical system-level design decisions.

We defined the multi-chip system design problem as a special case of the system-level design problem, in which an optimized hierarchical bus-based multi-chip architecture is constructed for a given system specification and user constraints. The multi-chip system design problem is a complex problem composed of interrelated design steps which addresses many development issues of the multi-chip system design problem simultaneously. In this research, we consider design steps, namely *physical design style selection, datapath architecture selection, task-level system partitioning, die selection, die clustering, substrate technology selection, package selection, and bus selection.* These tasks are modeled as *binary decision problems* and formulas computing metrics of design entities are developed based on the first order effect of design decisions. The interrelationships among design steps called validity constraints are defined with the computed metrics.

On this underlying model of the multi-chip system design steps, two optimization tools were developed. In EDEN, our MILP-based optimization tool, our first-order analytical model was linearized. To speed up the instantiation of a design problem in MILP, we defined a language called *M* which can express the formulation in the mathematical form effortlessly. (See Appendix C for the *M* language description of the linearized formulation). The *M* language description can be processed by a compiler called *GEM* to generate a generator which will be compiled and run to produce the matrix representation of a problem. EDEN was specifically formulated for rapid-prototyping. Experimental results to show the change of system architectures that can be rapidly prototyped under different user constraints were described.

A customized implementation of a genetic algorithm for multi-chip system design optimization was developed. GARDEN was designed such that various architectural alternatives can be evaluated under different optimization modes. Therefore, GARDEN is capable of versatile trade-off analyses. The experimental results of such architectural trade-off analyses for different physical design styles, the use of MCMs, and the number of task partitions are given.

There are other possibilities of using EDEN or GARDEN. Partitioning of a system into FPGA chips and COTS device selection are some examples of such applications. As indicated by its capability of producing a new solution by mixing different solutions together with different objectives, GARDEN can be used for improving existing designs with a mix-and-match of promising designs. Such capability can be exploited in finding a better solution from solutions optimized for different system metrics.

We believe that tools like EDEN and GARDEN are in demand to manage the increasing complexity of the modern system development process in the early phase of system development. Such tools can help to reduce time for trade-off analyses for informed decision making, increase the correctness of design decisions and therefore, reduce the time-to-market.

Our formulation shows how a systematic model for system-level design decision making can be developed. Our model can be extended to include other architectural alternatives. The possibility of applying GA for optimizing a complex problem like multi-chip system architecture is another important contribution. Our work shows the design issues of applying GA to the complex constrained optimization problem. New approaches for encoding, recombination operator design, and the optimization mode control mechanism were developed.

Our work has a number of limitations imposed from the simplifying assumptions. However, we believe that our work is the ground work for more practical tools which might be developed in future.

6.2 Future Work

Our current work can be continued in three different directions in the future. The first direction is to make our model more sophisticated and realistic. The second direction is to further extend our tool to cope with new optimization challenges for other system-level design issues such as mixed supply voltage power optimization or intellectual property-based design. The third possible direction is toward a more accurate and fast trade-off analysis and optimization.

Model Enhancements

The following design issues should be incorporated in the future model.

1. architecture trade-offs for I/O and memory subsystems,
2. consideration for design-for-testability,
3. power consideration in package selection, and
4. power budgeting among tasks.

Coping with New Optimization Challenges

Our model and GARDEN can be extended further to cope with new optimization challenges in the future. A mixture of different feature size in a system to achieve

a better the cost-performance trade-off[Mal94] can be performed with GARDEN. Power optimization for a system with a mixture of designs with different supply voltages can be done also by extending GARDEN. Recently incorporating intellectual property (IP) which is existing design data becomes an important issue in digital system design because the rapidly growing popularity of the internet makes the marketing of design information possible. By employing IPs in a system, the design time can be greatly reduced. However, deciding which IP is good for system development requires a systematic evaluation method. Since GARDEN has the facility to handle a COTS device which can be considered to be an IP, GARDEN can be used in making decisions on the use of IPs. Finally, in our hardware model, we assumed that no hardware is shared among tasks. In general, it is a realistic assumption at this time. However, a system can be designed such that hardware can be shared among tasks in the future. Sharing can be easily implemented for FPGAs because of the programmability of FPGAs. While an FPGA chip performs a specified task at one moment of the system execution, the same chip can be programmed to perform a different task in the next moment. The sharing of hardware necessitates solving a non-trivial scheduling problem in a multi-chip system design which must be handled by future system architecture tools.

Increasing the Accuracy of the Estimates While Reducing the Runtime

Our model considers only the first-order effect of design decisions. For example, in computing the size of dies, our model does not take into account placement and routing, which also has big influence on the size. Considering such factors in the early design phase is an almost impossible task. However, the first effect consideration alone cannot guarantee the correctness of a design decision though it can lower the risk of incorrect decisions considerably. The goal of our three-step system architecture optimization is to balance accuracy and speed. However, more detailed and accurate estimation can be made to further eliminate the possibility of a wrong design decision. Since there exist a number of good estimators that can compute various metrics more accurately, such sophisticated estimators can

be used. To achieve speeding up the architecture optimization while increasing the accuracy, a distributed system over LAN or parallel processors can be used. The metric functions in the analytical model in GARDEN can be simply replaced with calls for detailed estimators running on different machines over a network.

Reference List

[And95] J. Anderson. Projecting RASSP benefits. In *Proceedings 2nd Annual RASSP Conference*, 1995.

[AT95] J. Adams and D. E. Thomas. Multiple-process behavioral synthesis for mixed hardware/software systems. In *Proc. of 8th International Symposium on System Synthesis*, 1995.

[Ber83] W. Bertram. Yield and reliability. In S. M. Sze, editor, *VLSI Technology*, page 599. McGraw-Hill, 1983.

[Boe88] B. Boehm. A spiral model of software development and enhancement. *Computer*, pages 61–72, May 1988.

[BS91] D. Belina and A. Sarma. *SDL with Applications from Protocol Specifications*. Prentice Hall, 1991.

[CGJ+94] M. Chiodo, P. Giusto, A. Jurecska, M. Marelli, H. Hsieh, A. Sangiovanni-Vincentelli, and L. Lavagno. Hardware-software codesign of embedded systems. *IEEE Micro*, pages 26–36, 1994.

[Che94] C. Chen. *System-level Design Techniques and Tools for Synthesis of Application-Specific Digital Systems*. PhD thesis, University of Southern California, 1994.

[DB96] T. P. Darr and W. P. Birmingham. An attribute-space representation and algorithm for concurrent engineering. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, pages 21–35, 1996.

[DH89] D. Drunsinsky and D. Harel. Using StateCharts for hardware description and synthesis. *IEEE Trans. on CAD*, 1989.

[DP94] J. C. DeSouza-Batista and A. Parker. Optimal synthesis of application specific heterogeneous pipelined multiprocessors. In *The International Conf. on Application-Specific Array Processors*, 1994.

[DRB+95] P. Dehkordi, K. Ramamurthi, D. Bouldin, H. Davidson, and P. Sandborn. Impact of packaging technology on system partitioning: A case study. In *Proc. of IEEE Multi-chip Module Conference*, 1995.

[EBCH86] C. K. Erdelyi, R.A. Bechade, M.P. Concannon, and W.K. Hoffman. Custom and semi-custom design. In S. Goto, editor, *Design Methodologies*, chapter 1. North-Holland, 1986.

[EHB93] R. Ernst, J. Henkel, and T. Benner. Hardware-software co-synthesis for microcontrollers. *IEEE Design & Test of Computer*, pages 64–75, December 1993.

[FKCM93] D. Filo, D. Ku, C Coelho, and G. De Micheli. Interface optimization for concurrent systems under timing constraints. *IEEE Trans. on Very Large Scale Integration Systems*, 1(3):268, September 1993.

[FLS+92] H. Fujiwara, M. Liou, M. Sun, K. Yang, M. Maruyama, K. Shomura, and K. Ohyama. An all-ASIC implementation of a low bit-rate video codec. *IEEE Transactions on Circuits and Systems for Video Technology*, 2(2):123, 1992.

[Gaj94] D. Gajski. A system-design methodology: Executable-specification refinement. In *Proc. of the European Conference on Design Automation*, 1994.

[Gal91] Didier Le Gall. MPEG:a video compression standard for multimedia application. *Communications of the ACM*, 34(4), April 1991.

[GKP85] J. Granacki, D. Knapp, and A. C. Parker. The ADAM Design Automation System: Overview, Planner and Natural Language Interface. In *Proc. of the 22nd Design Automation Conf.*, pages 727–730, June 1985.

[GM92] R. Gupta and G. Micheli. System synthesis via hardware-software codesign. Technical Report CSL-TR-92-548, Stanford University, 1992.

[GNRV96] D. Gajski, S. Narayan, L. Ramachandran, and F. Vahid. System design methodologies:aiming at the 100 h design cycle. *IEEE Trans. on Very Large Scale Integration Systems*, 4:70–82, 1996.

[GP87] J. Granacki and A. Parker. PHRAN-SPAN: A Natural Language Interface for System Specifications. In *Proceedings of the 1987 Design Automation Conf.*, pages 416–422. ACM/IEEE, June 1987.

[GR94] K. Gong and L. Rowe. Parallel MPEG-1 video encoding. In *Picture Coding Symposium*, 1994.

[Han84] R. Hannemann. Physical technology for VLSI systems. In *Proc. of the Int'l Conf. on Computer Design*, pages 48–53, 1984.

[Hay88] J. P. Hayes. *Computer Architecture and Organization*, chapter 6. McGraw-Hill, 1988.

[Hig92] L. Higgins III. Perspectives on multi-chip modules: Substrate alternatives. In *Proc. of IEEE Multi-chip Module Conference*, pages 12–15, 1992.

[Hoa78] C. Hoare. Communicating sequential processes. *Comm. of the ACM*, August 1978.

[HOK92] B. Hahne, D.E. O'Brien, and J.P. Krusius. Package architecture design and optimization tool AUDiT:Version 4.2 for inhomogeneous systems. In *Proceedings of 42nd Electronic Components Conference*, pages 326–331, 1992.

[Hol87] E. Hollis. *Design of VLSI Gate Array ICs.* Prentice-Hall, Inc., 1987.

[HR91] J. Huber and M. Rosneck. *Successful ASIC Design the First Time Through.* Van Nostrand Reinhold, New York, 1991.

[HRP96] D. H. Heo, C. P. Ravikumar, and A. Parker. A synthesis tool targeted for rapid delivery of electronic systems. In *Electronic Design Process Workshop 1996*, 1996.

[HT92] J. W. Hagerman and D. E. Thomas. Process transformation for system level synthesis. In *Highlevel Synthesis Workshop*, 1992.

[Hun92] Y. Hung. *High-Level Synthesis with Pin Constraints for Multiple-Chip Design.* PhD thesis, University of Southern California, 1992.

[HW92] J. K. Haggle and R. J. Wagner. High-yield assembly of multichip modules through known-good IC's and effective test strategies. *Proceedings of the IEEE*, pages 1965–1994, 1992.

[Ins88] Institute of Electrical and Electronics Engineers. *IEEE Standard VHDL Language Reference Manual*, 1988.

[IOJ94] T. Ismail, K. O'Brien, and A. Jerraya. Interactive system-level partitioning with PARTIF. In *EDAC for testing*, 1994.

[Joh96] F. M. Johannes. Partitioning of VLSI circuits and systems. In *Proc. of the 33rd Design Automation Conf.*, 1996.

[JPP92] R. Jain, A. Parker, and N. Park. Predicting system-level area and delay for pipelined and nonpipelined designs. *IEEE Trans. on CAD*, 11(8):955, August 1992.

[KL93] A. Kalavade and E. Lee. A hardware-software codesign methodology for DSP applications. *IEEE Design & Test of Computers*, September 1993.

[KL94] A. Kalavade and E. Lee. A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem. In *Proc. of Codes/CASHE '94, Third Intl. Workshop on Hardware/Software Codesign*, 1994.

[KL95] A. Kalavade and E. Lee. The extended partitioning problem:hardware/software mapping and implementation-bin selection. In *Proc. of 6th Intl. Workshop on Rapid System Prototyping*, 1995.

[KM94] S. Khan and V. Madisetti. Yield-based system partitioning strategies for MCM and ASEM design. In *Proceedings of IEEE Multi-Chip Module Conference*, pages 144–149, 1994.

[KM95] S. Khan and V. Madisetti. System partitioning of MCMs for low power. *IEEE Design & Test of Computer*, pages 41–52, 1995.

[KP85] D. Knapp and A. C. Parker. A Unified Representation for Design Information. In *Proceedings of the IFIP Conf. on Hardware Description Languages*, August 1985.

[KP95] K. Kucukcakar and A. Parker. A methodology and design tools to support system-level VLSI design. *IEEE Trans. on Very Large Scale Integration Systems*, 3(3), September 1995.

[Kuc91] K. Kucukcakar. *System-Level Synthesis Techniques with Emphasis on Partitioning and Design Planning.* PhD thesis, University of Southern California, September 1991.

[Kur91] Kurdahi. Last: Layout area and shape function estimation. In *Proc. of the European Conference on Design Automation*, 1991.

[Lic95] J. Licari. Introduction. In J. Licari, editor, *Multichip Module Design, Fabrication, and Testing*, pages 1–19. McGraw-Hill, Inc, 1995.

[LT89] E. Lagnese and D. E. Thomas. Architectural Partitioning for System Level Design. In *Proc. of the 26th Design Automation Conf.*, June 1989.

[Mal94] W. Maly. Cost of silicon viewed from VLSI design perspective. In *Proc. of the 31st Design Automation Conf.*, pages 135–142, 1994.

[MC80] C. Mead and L. Conway. *Introduction to VLSI Systems*, chapter 7. System Timing. Addison Wesley, 1980.

[McC86] E. J. McCluskey. *Logic Design Principles with Emphasis on Testable Semicustom Circuits.* Prentice Hall, 1986.

[Mic92] Z. Michalewicz. *Genetic Algorithms + Datastruture = Evolution Programs.* Springer-Verlag, 1992.

[MK88] G. De Micheli and D. Ku. HERCULES - a system for high-level synthesis. In *Proc. of the 25th Design Automation Conf.*, 1988.

[MMC88] A. C. Parker M. McFarland and R. Camposano. Tutorial on High-Level Synthesis. *Proc. of the 25th Design Automation Conf.*, Jul 1988.

[Mur64] B. T. Murphy. Cost-size optima of monolithic integrated circuits. *Proceedings of IEEE*, December 1964.

[NG94] S. Narayan and D. Gajski. Synthesis of system-level bus interfaces. In *Proc. of the European Conference on Design Automation*, 1994.

[NVG91] S. Narayan, F. Vahid, and D. Gajski. System specification and synthesis with the SpecCharts language. In *Proc. of the Int'l Conf. on Computer Aided Design*, 1991.

[OHK92] D. O'Brien, B. Hahne, and J. Krusius. Multichip modules vs high-density printed wiring boards: A trade-off study. In *Proceedings of 42th Electronic Components Conference*, 1992. 31-35.

[PCG93] A. Parker, C. Chen, and P. Gupta. Unified system construction. In *Proc. 4th SASIMI Workshop, Nara, Japan*, pages 109–118, 1993.

[PP88] N. Park and A. C. Parker. Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications. *IEEE Trans. on Computer-Aided Design*, March 1988.

[Pra93] S. Prakash. *Synthesis of Application-Specific Multiprocessor Systems.* PhD thesis, University of Southern California, 1993.

[Pre87] R. Pressman. *Software Engineering.* McGraw-Hill, 1987.

[RHU+93] S. Rao, M. Hatamian, M. Uyttendaele, S. Narayan, J. O'Neill, and G. Uvieghara. A real-time P*64/MPEG video encoder chip. In *Proceedings of IEEE International Solid-State Circuits Conference*, 1993.

[Ric94] M. A. Richard. The rapid prototyping of application specific signal processors(RASSP) program:overview and status. In *Proceedings 1st Annual RASSP Conference*, 1994.

[SA95] P. Sandborn and M. Abadir. Multichip module design. In J. Licari, editor, *Multichip Module Design, Fabrication, and Testing*, pages 21–78. McGraw-Hill, Inc, 1995.

[SAM95] G. A. Shaw, J. C. Anderson, and V. K. Madisetti. Accessing and improving current practice in the design of application-specific signal processors. In *Proc. of the Int'l. Conf. on Acoustics, Speech and Signal Processing*, 1995.

[SB95] M. Srivastava and R. Brodersen. SIERA: A unified framework for rapid-prototyping of system-level hardware and software. *IEEE Trans. on CAD*, 14(6):676–693, June 1995.

[Sch82] D. C. Schmidt. Circuit pack parameter estimation using Rent's rule. *IEEE Trans. on CAD*, CAD-1(4):186–192, October 1982.

[Sch92] L. W. Schaper. Design of multichip modules. *Proceedings of the IEEE*, 80(12):1955–1964, December 1992.

[Siu92] W. M. Siu. MCM and monolithic VLSI, perspectives on dependencies, integration, performance and economics. In *Proc. of IEEE Multi-chip Module Conference*, 1992.

[SKT92] M. Shih, E. Kuh, and R. Tsay. System partitioning for multi-chip modules under timing and capacity constraints. In *Proc. of IEEE Multi-chip Module Conference*, pages 123–126, 1992.

[SO73] I. E. Sutherland and D. Oestricher. How big should a printed circuit board be? *IEEE Trans. on Computers*, pages 537–542, May 1973.

[TAS93] D. Thomas, J. Adams, and H. Schmit. A model and methodology for hardware-software codesign. *IEEE Design & Test of Computers*, pages 6–15, 1993.

[Tum92] R. R. Tummala. Multichip packaging - a tutorial. *Proc. of IEEE*, 8(12):1924–1941, 1992.

[VG95] F. Vahid and D. Gajski. Clustering for improved system-level functional partitioning. In *International Symposium on System Synthesis*, 1995.

[VGG94] F. Vahid, J. Gong, and D. Gajski. A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning. In *Proc. of the EuroDAC*, 1994.

[Wal91] G. K. Wallace. The JPEG still picture compression standard. *IEEE Tran. on Consumer Electronics*, December 1991.

[Whi93] D. Whitley. A genetic algorithm tutorial. Technical Report CS-93-103, Colorado State University, 1993.

[WM93] W. Wolf and R. Manno. High-level modeling and synthesis of communicating processes using VHDL. *IEICE Trans. on Information and Systems*, E76D(9):1039–1046, September 1993.

[WP91] J. Weng and A. C. Parker. 3D Scheduling: High-Level Synthesis with Floor planning. In *Proc. of the 28th Design Automation Conf.*, pages 668–673, July 1991.

[Zwi95] D. Zwinllinger, editor. *CRC Standard Mathematical Tables and Formulae.* CRC, 1995.

Appendix A

Linearization for MILP

Notations and Conventions

Lower case alphabets are used to denote binary variables, e.g. y_{pt}. Upper case letters are used for linear/integer variables and constants. Names of linear/integer variables begin with an X to distinguish them from constants. Greek letters are used to denote intermediate variables introduced for the purpose of linearization of constraints. Intermediate binary variables are in lower case Greek letters and intermediate linear variables and constants are in upper case Greek letters.

A.1 Lemmas for Linearization

We introduce a few useful lemmas that simplify linearization procedure.

Lemma A.1.1 *Let S be a set of real numbers and $y \in S$. A non-linear equation of the form $x \geq max_{y \in S}(y)$ in the formulation can be linearized as follows:*

$$x \geq y, \ \forall y \in S$$

A non-linear equation of the form $x \leq min_{y \in S}(y)$ in the formulation can be linearized as follows:

$$x \leq y, \ \forall y \in S$$

The linearization step with the above lemma increases the number of constraints by $S - 1$.

Lemma A.1.2 *Let S be a set of binary numbers and $y \in S$. A non-linear equation of the form $x = max_{y \in S}(y)$ in the formulation can be linearized as follows:*

$$\begin{aligned} x &\geq y, \forall y \in S \\ x &\leq \sum_{y} y, \forall y \in S \end{aligned}$$

A non-linear equation of the form $x = min_{y \in S}(y)$ in the formulation can be linearized as follows:

$$\begin{aligned} x &\leq y, \forall y \in S \\ x &\geq \sum_{y} (y-1) + 1, \forall y \in S \end{aligned}$$

The linearization step with the above lemma increases the number of constraints by $2S - 1$.

Lemma A.1.3 *A product $x \cdot y$ of two binary variables appearing in a constraint can be replaced with a binary variable z as follows:*

$$\begin{aligned} z &\leq x \\ z &\leq y \\ z &\geq x + y - 1 \end{aligned}$$

Proof. When both x and y are equal to 0, we have $z \leq 0, z \geq -1$. When either of x or y takes the value 0, the above inequalities becomes $z \leq 0, z \leq 1, z \geq 0$, and therefore $z = x \cdot y = 0$. When both x and y are equal to 1, we have $z \leq 1, z \leq 1, z \geq 1$, therefore, $z = x \cdot y = 1$.

□

The linearization step with the above lemma increases the number of constraints by 3.

Lemma A.1.4 *A product $L \cdot y$ of a finite positive real variable L and a binary variable y in the MILP formulation can be replaced with a positive real variable M as follows:*

$$\begin{aligned} M &\leq y \cdot MAXL \\ M &\leq L \\ M &\geq L + (y-1) \cdot MAXL \end{aligned}$$

where $MAXL$ is the upper bound on the variable L.

Proof. When $y = 0$, the constraints can be rewritten as $M \leq 0, M \leq L$ and $M \geq L - MAXL$. The last term implies $M \geq 0$ because $L - MAXL \leq 0$ and thus $M = L \cdot y = 0$. When $y = 1$, the constraints can be rewritten as $M \leq MAXL, M \leq L$ and $M \leq L$, implying $M = L$.

□

The linearization step with the above lemma increases the number of constraints by 3.

A.2 Linearization of Non-linear Formulations for Task-Level System Partitioning

Function $b(e, Y)$, Equation (3.7)

$b(e, Y)$ can be expressed as follows:

$$b(e, Y) = 1 - \sum_{p} min_{t \in V_e}(y_{pt})$$

Let $b_e = b(e, Y)$. the above equation can be linearized with Lemma A.1.2 by replacing $min_{t \in V_e}(y_{pt})$ with λ_{ep} as follows:

$$b_e = 1 - \sum_{p} \lambda_{ep}$$

$$\begin{aligned} \lambda_{ep} &\leq y_{pt}, \ \forall t \in V_e \\ \lambda_{ep} &\geq \sum_t (y_{pt} - 1) + 1 \end{aligned}$$

Function $t(e, p, Y)$, Equation (3.8)

$t(e, p, Y)$ can be expressed as follows:

$$t(e, p, Y) = max_{t \in V_e}(y_{pt})$$

$t(e, p, Y)$ can be linearized with Lemma A.1.2 by replacing $\zeta_{ep} = max_{t \in V_e}(y_{pt})$.

$$\begin{aligned} \zeta_{ep} &= t(e, p, Y) \\ \zeta_{ep} &\geq y_{pt}, \ \ t \in V_e \\ \zeta_{ep} &\leq \sum_{t \in V_e} y_{pt} \end{aligned}$$

Task Partition Pin Metric Function, $TPartPin(p, Y, U)$

Let $XTPPIN_p = TPartPin(p, Y, U)$ and $XBW_e = BusWidth(e, U)$. By substituting $b_e = b(e, Y)$ and $\zeta_{ep} = t(e, p, Y)$, $TPartPin(p, Y, U)$ can be written as follows:

$$XTPPIN_p = \sum_{e \in E} b_e \times \zeta_{ep} \times XBW_e$$

According to Lemma A.1.3, we can replace $b_e \times \zeta_{ep}$ with δ_{ep}. Then,

$$XTPPIN_p = \sum_{e \in E} \delta_{ep} \times XBW_e$$

where δ_{ep} is subject to following additional constraints according to Lemma A.1.3:

$$\begin{aligned} \delta_{ep} &\leq b_e \\ \delta_{ep} &\leq \zeta_{ep} \\ \delta_{ep} &\geq \zeta_{ep} + b_e - 1 \end{aligned}$$

We can further linearize the above equation by replacing $\delta_{ep} \times XBW_e$ with Θ_{ep} and the following additional constraints;

$$\begin{aligned}
\Theta_{ep} &\geq \delta_{ep} \times MAXBW \\
\Theta_{ep} &\leq XBW_e \\
\Theta_{ep} &\geq XBW_e + (\delta_{ep} - 1) \times MAXBW
\end{aligned}$$

Task Partition Size Metric Function, $TPartSize(p, Z, Y, X, U)$

Let $XTPSIZE_p = TPartSize(p, Z, Y, X, U)$ and $XTSIZE_t = TaskSize(t, Z)$. Then, the equation for $TaskSize(t, Z)$ can be written as follows:

$$\begin{aligned}
XTPSIZE_p = &\sum_d \sum_t y_{pt} \times x_{dp} \times XTSIZE_t \times WB_d \times AVGGATESIZE_d + \\
&\sum_d XTPPIN_p \times x_{dp} \times PADSIZE_d
\end{aligned}$$

Let $\Lambda_{dpt} = y_{pt} \times x_{dp} \times XTSIZE_t$ and $\Delta_{dp} = XTPPIN_p \times x_{dp}$. Then,

$$XTPSIZE_p = \sum_d \sum_t \Lambda_{dpt} \times WB_d \times AVGGATESIZE_d + \sum_d \Delta_{dp} \times PADSIZE_d$$

We need to linearize Λ_{dpt}. Let $\gamma_{dpt} = y_{pt} \times x_{dp}$. Then, $\Lambda_{bpt} = \gamma_{dpt} \times XTSIZE_t$. It can be linearized by Lemma A.1.4 as follows:

$$\begin{aligned}
\Lambda_{dpt} &\geq XTSIZE_t + (\gamma_{dpt} - 1) \times MAXTSIZE \\
\Lambda_{dpt} &\leq XTSIZE_t \\
\Lambda_{dpt} &\leq \gamma_{dpt} \times MAXTSIZE
\end{aligned}$$

According to Lemma A.1.3, the new binary variable γ_{dpt} is subject to the following additional constraints:

$$\begin{aligned}
\gamma_{dpt} &\leq y_{pt} \\
\gamma_{dpt} &\leq x_{dp} \\
\gamma_{dpt} &\geq x_{dp} + y_{dp} - 1
\end{aligned}$$

Finally, Δ_{dp} is subject to the following additional constraints:

$$\begin{aligned} \Delta_{dp} &\geq XTPPIN_p + (x_{dp} - 1) \times MAXTPPIN \\ \Delta_{dp} &\leq x_{dp} \times MAXTPPIN \end{aligned}$$

Task Cost Metric Function, $TPartCost(p, Z, Y, X, U)$

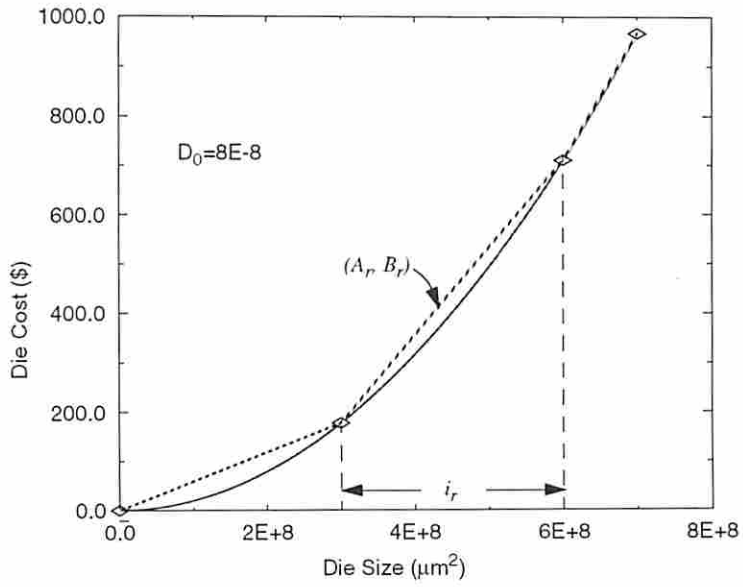


Figure A.1: Piece-wise approximation of die cost vs. die size for the standard cell implementation

$TPartCost(p, Z, Y, X, U)$ can be linearized by a piece-wise linear approximation as shown Figure A.1. Let $XTPCOST_p = TPartCost(p, Z, Y, X, U)$ and let $R = \{i_r : i_r = [S_r, S_{r+1}), \bigcup_r i_r = \{s : 0 \leq s \leq MAXTPSIZE\}, r = 1, 2, .., n\}$. Let A_r and B_r be the slope and offset of the line approximating $XTPCOST_p$ within range r as follows (see Figure A.1):

$$\begin{aligned} A_r &= \frac{Yied(S_{r+1}) - Yield(S_r)}{S_{r+1} - S_r}, \; r = 1, 2, \ldots, n \\ B_r &= \frac{Yied(S_r)S_{r+1} - Yield(S_{r+1})S_r}{S_{r+1} - S_r}, \; r = 1, 2, \ldots, n \end{aligned}$$

We also introduced a new binary variable μ_{rp} defined below:

$$\mu_{rp} = \begin{cases} 1 & \text{if } S_r \leq XTPSIZE_p < S_{r+1} \\ 0 & \text{if } S_r > XTPSIZE_p \text{ or } S_{r+1} \leq XTPSIZE_p \end{cases}$$

The above definition can be linearized as follows:

$$\begin{aligned} XTPSIZE_p &< \sum_r \mu_{rp} \times S_{r+1} \\ XTPSIZE_p &\geq \sum_r \mu_{rp} \times S_r \end{aligned}$$

The following equation ensures that only one μ_{rp} can take on the value 1 for a non-empty partition p.

$$\sum_r \mu_{rp} = min_t(y_{pt})$$

With Lemma A.1.2, we can linearize the above equation by substituting $min_t(y_{pt})$ with τ_p as follows:

$$\begin{aligned} \sum_r \mu_{rp} &= \tau_p \\ \tau_p &\geq y_{pt} \\ \tau_p &\leq \sum_t y_{pt} \end{aligned}$$

Then we can express the piece-wise approximation in linear form as follows:

$$XTPCOST_p \approx \sum_r A_r \times \mu_{rp} \times XTPSIZE_p + \sum_r B_r \times \mu_{rp}$$

In order to linearize the above equation, we substitute $\Phi_{rp} = \mu_{rp} \times XTPSIZE_p$,

$$XTPCOST_p = \sum_r A_r \times \Phi_{rp} + \sum_r B_r \times \mu_{rp}$$

where Φ_{rp} is subject to following constraints:

$$\Phi_{rp} \leq \mu_{rp} \times MAXTPSIZE$$

$$\begin{aligned}\Phi_{rp} &\leq XTPSIZE_p \\ \Phi_{rp} &\geq XTPSIZE_p + (\mu_r p - 1) \times MAXTPSIZE\end{aligned}$$

Style Constraint for TP, Equation (3.10)

Let $XTSTYLE_t = TaskStyle(t, Z)$:

$$\begin{aligned}\alpha_p &\geq max_{t \in V_p}(XTSTYLE_t),\ \forall t \in V_p \\ &= y_{pt} \times XTPSTYLE_t,\ \forall t \in V \qquad \text{(A.1)} \\ \beta_p &\leq min_{t \in V_p}(XTSTYLE_t),\ \forall t \in V_p \\ &= y_{pt} \times ISTYLE_t,\ \forall t \in \qquad \text{(A.2)}\end{aligned}$$

Then, for a given partition p, in order to satisfy the constraint given in Equation (3.10),

$$\alpha_p = \beta_p \qquad \text{(A.3)}$$

Therefore, the validity constraint given in (3.10) can be replaced with Equation (A.1)-(A.3).

With Lemma A.1.4, Equation (A.1) can be linearized by replacing $y_{pt} \times XTPSTYLE_t$ with $\mho_{pt}$ as follows:

$$\alpha_p \geq \mho_{pt}$$

where $\mho_{pt}$ is subject to the following constraints:

$$\begin{aligned}\mho_{pt} &\leq y_{pt} \times MAXISTYLE \\ \mho_{pt} &\leq XTPSTYLE_t \\ \mho_{pt} &\geq XTPSTYLE_t + (y - 1) \times MAXISTYLE\end{aligned}$$

Similarly, Equation A.2 can be linearized as follows:

$$\beta_p \leq \mho_{pt}$$

where $MAXISTYLE$ is the maximum value of $ISTYLE$ and in this case, it is $MAXISTYLE = 2$. For each p, the above linearization step increases the number of constraints by $2V - 1$.

A.3 Linearization of Non-linear Die Selection Formulation

Function Mapping Constraint for DS

Since $min_t(y_{pt})$ was linearized with τ_p, the function mapping constraint can be written as follows:

$$\sum_b x_{bp} = \tau_p$$

Die Size Metric Function, $DieSize(p, Z, Y, X, U)$

In order to linearize partition metric functions for DS, we need introduce the following binary variable:

$$n_{pq} = \begin{cases} 1 & \text{if } XTPSTYLE_p = q \\ 0 & \text{if } XTPSTYLE_p \neq q \end{cases}$$

where $q \in Q$.
Since each non-empty task partition must have one and only one design style, the following two constraints are necessary.

$$XTPSTYLE_p = \sum_q n_{pq} \cdot q$$

$$\sum_q n_{pq} = \tau_p$$

Let $XDSSIZE_p = DieSize(p, Z, Y, X, U)$. Then, Equation (3.12) can be expressed:

$$\begin{aligned} XDSSIZE_p &= \sum_{q \in Q_v} n_{pq} \times XTPSIZE_p + \sum_{q \in Q_f} n_{pq} \times \sum_d x_{dp} \times DSIZE_d \\ &= \sum_{q \in Q_v} n_{pq} \times XTPSIZE_p + \sum_{q \in Q_f} \sum_d n_{pq} \times x_{dp} \times DSIZE_d \end{aligned}$$

By substituting $\Xi_{pq} = n_{pq} \times XTPSIZE_p$ and $\eta_{dpq} = n_{pq} \times x_{dp}$, the above equation becomes,

$$XDSSIZE_p = \sum_{q \in Q_v} \Xi_{pq} + \sum_{q \in Q_f} \sum_b \eta_{dpq} \times DSIZE_d$$

where Ξ_{pq} and η_{dpq} are subject to the following constraints:

$$\begin{aligned} \eta_{dpq} &\leq n_{pq} \\ \eta_{dpq} &\leq x_{dp} \\ \eta_{dpq} &\geq n_{pq} + x_{dp} - 1 \\ \Xi_{pq} &\leq n_{pq} \times MAXTPSIZE \\ \Xi_{pq} &\leq XTPSIZE_p \\ \Xi_{pq} &\geq XTPSIZE_p + (n_{pq} - 1) \times MAXTPSIZE \end{aligned}$$

Die Pin Metric Function, $DiePin(p, Y, X, U)$

Let $XDSPIN_p = DiePin(p, Y, X, U)$. Then, Equation (3.13) becomes,

$$XDSPIN_p = \sum_{q \in Q_v} n_{pq} \times XTPPIN_p + \sum_{q \in Q_f} \sum_d n_{pq} \times x_{dp} \times DPIN_d$$

Because term $n_{pq} \times x_{dp}$ is already linearized, we replace it with η_{dpq}. Let $\xi_{pq} = n_{pq} \times XTPPIN_p$.

$$XDSPIN_p = \sum_{q \in Q_v} \xi_{pq} + \sum_{q \in Q_f} \sum_d \eta_{dpq} \times DPIN_d$$

where ξ_{pq} introduces the following constraints:

$$\begin{aligned}
\xi_{pq} &\geq XTPPIN_p + (n_{pq} - 1) \times MAXTPPIN \\
\xi_{pq} &\leq XTPPIN_p \\
\xi_{pq} &\leq n_{pq} \times MAXTPPIN
\end{aligned}$$

Die Cost Metric Function, $DieCost(p, Z, Y, X, U)$

Let $XDSCOST_p = DieCost(p, Z, Y, X, U)$ and
$XTPCOST_p = TPartCost(p, Z, Y, X, U)$. Then Equation (3.14) can be rewritten as follows:

$$XDSCOST_p = \sum_{q \in Q_v} n_{pq} \times XTPCOST_p + \sum_{q \in Q_f} \sum_{b} n_{pq} \times x_{dp} \times DCOST_p$$

Let $\Omega_{pq} = n_{pq} \times DCOST_p$. Then,

$$XDSCOST_p = \sum_{q \in Q_v} \Omega_{pq} + \sum_{q \in Q_f} \sum_{b} \eta_{dpq} \times DCOST_p$$

where Ω_{pq} is subject to following constraints:

$$\begin{aligned}
\Omega_{pq} &\leq n_{pq} \times MAXDCOST \\
\Omega_{pq} &\leq DCOST_p \\
\Omega_{pq} &\geq DCOST_p + (n_{pq} - 1) \times MAXDCOST
\end{aligned}$$

Average Gate Size, Pad Size, and Routing-Area-Ratio Metric Functions

Equation (3.15),(3.17), the metric functions for the average gate size, and the routing-area-ratio of a chosen die can be linearized easily by setting the value in

the corresponding library element, i.e., by letting $AVGGATESIZE_d = 1$ and $WB_d = 1$ for $q \in Q_v$, Equation (3.15),(3.17) are simplified as follows:

$$\begin{aligned} AvgGateSize(p, X) &= \sum_{d \in D} x_{dp} \times AVGGATESIZE_d \\ WB(p, X) &= \sum_{d \in D} x_{dp} \times WB_d \end{aligned}$$

A.4 Linearization of System Metric Functions

Modified System Cost Metric Function

We do not consider MCMs in the MILP model and further simplify the system cost metric function by assuming $n \to \infty$. Then, Equation 3.25 must be modified as follows:

$$COST_c = \sum_{p \in c} DieCost(p, Z, Y, X, U) + PkgCost(p, W)$$

Linearization of System Performance Metric Function

In order to make the equation computing $DELAY_{path}$ linear, we need to linearize the following term in Equation (3.25):

$$CommDelay(e) = \frac{VOL_e \cdot BusCycle(e, U)}{BusWidth(e, U)}$$

Let $XDPBIT_e = \frac{BusCycle(e,U)}{BusWidth(e,U)}$:

$$CommDelay(e) = VOL_e \times XDPBIT_e$$

Then $XDBPIT_e$ can be simplified as follows:

$$\begin{aligned} XDPBIT_e &= \frac{BusCycle(e, U)}{BusWidth(e, U)} \\ &= \frac{\sum_b u_{be} \times BCYCLE_b}{\sum_b u_{be} \times BWIDTH_b} \end{aligned}$$

$$= \sum_b u_{be} \times \frac{BCYCLE_b}{BWIDTH_b}$$

Linearization of System TTM Metric Function

$SystemTTM(A)$ is used as an objective function in our MILP model but Equation (3.28) is not appropriate to be used for the objective function as it is, since there can be a number of different architectures with the same TTM but the characteristics of the architectures in other system metric are quite different. Therefore, we modified Equation (3.28) as follows such that an architecture with smaller implementation size is preferred among solutions having the same TTM.

$$Obj = \sum_p XTPSIZE_p \times (FabTime(p, X) + PkgTime(p, W)) \quad \text{(A.4)}$$

In order to linearize the above equation, we take advantage of the following facts:

1. the time for fabrication and packaging for FPGA implementation is equal to 0, and
2. the times for packaging for both the gate array and the standard cell implementations are equal.

Since the fabrication and packaging time of a chip differ only by the physical design styles, $TFab$ and $TPkg$ can have only finite number of values depending on the design styles. Let $TFab = \{TF_q : q \in Q\}$ and $TPkg = \{TK_q : q \in Q\}$. Let GA and SC denote the gate array and standard cell design style respectively. Let $P_q = \{p : XTPSTYLE_p = q, p \in P\}$. Then, Equation (A.4) can be expressed as follows:

$$\begin{aligned} Obj = & \sum_{p_{FPGA}} XTPSIZE_p \times (FabTime(p, X) + PkgTime(p, W)) + \\ & \sum_{p_{GA}} XTPSIZE_p \times (FabTime(p, X) + PkgTime(p, W)) + \\ & \sum_{p_{SC}} XTPSIZE_p \times (FabTime(p, X) + PkgTime(p, W)) \end{aligned}$$

Since $FabTime(p, X) \in TFab$ and $PkgTime(p, W) \in TPkg$,

$$\begin{aligned} Obj &= \sum_{p_{FPGA}} XTPSIZE_p \times (TF_{FPGA} + TK_{FPGA}) \\ &\quad + \sum_{p_{GA}} XTPSIZE_p \times (TF_{GA} + TK_{GA}) \\ &\quad + \sum_{p_{SC}} XTPSIZE_p \times (TF_{SC} + TK_{SC}) \end{aligned}$$

Since $TF_{FPGA} = TK_{FPGA} = 0$ and $TK_{GA} = TK_{SC} = TKC$,

$$\begin{aligned} Obj &= \sum_{p_{GA}} XTPSIZE_p \times (TF_{GA} + TKC) + \sum_{p_{SC}} XTPSIZE_p \times (TF_{SC} + TKC) \\ &= (TF_{GA} + TKC) \sum_{p_{GA}} XTPSIZE_p + (TF_{SC} + TKC) \sum_{p_{SC}} XTPSIZE_p \\ &= (TF_{GA} + TKC) \sum_{p} n_{p,GA} \times XTPSIZE_p \\ &\quad + (TF_{SC} + TKC) \sum_{p} n_{p,SC} \times XTPSIZE_p \end{aligned}$$

Let $MANU_{GA} = TF_{GA} + TKC$, $MANU_{SC} = TF_{SC} + TKC$, and $\Gamma_{pq} = n_{pq} \times XTPSIZE_p$. Then

$$Obj = MANU_{GA} \sum_{p} \Gamma_{p,GA} + MANU_{SC} \sum_{p} \Gamma_{p,SC}$$

where $\Gamma_{p,GA}$ and $\Gamma_{p,SC}$ are subject to the following linearization constraints:

$$\begin{aligned} \Gamma_{p,GA} &\leq n_{p,GA} \times MAXTPSIZE \\ \Gamma_{p,GA} &\leq XTPSIZE_p \\ \Gamma_{p,GA} &\geq XTPSIZE_p + (n_{p,GA} - 1) MAXTPSIZE \\ \Gamma_{p,SC} &\leq n_{p,SC} \times MAXTPSIZE \\ \Gamma_{p,SC} &\leq XTPSIZE_p \\ \Gamma_{p,SC} &\geq XTPSIZE_p + (n_{p,SC} - 1) MAXTPSIZE \end{aligned}$$

Appendix B

Complete List of Linearized Formulae

B.1 Definition of Sets

$$
\begin{aligned}
Q &= \{0,1,2\} \\
Q_f &= \{\text{ FPGA, gate array }\} \\
Q_v &= \{\text{standard cell}\} \\
R &= \{i_r : i_r = [S_r, S_{r+1}), \bigcup_r i_r = \{s : 0 \leq s \leq MAXTPSIZE\}, r = 1,2,..,n\}\} \\
V_e &= \{t : \ t \text{ is connected to } e \ , t \in V\} \\
V_p &= \{t : y_{pt} = 1\}
\end{aligned}
$$

B.2 Physical Design Style and Architecture Selection Subproblem

$$\sum_{i \in I} z_{ti} = 1$$

Metric Selection Functions

$$TaskSize(t, Z) = \sum_i z_{ti} \times ISIZE_i$$

$$
\begin{aligned}
TaskDelay(t, Z) &= \sum_i z_{ti} \times IDELAY_i \\
TaskStyle(t, Z) &= \sum_i z_{ti} \times ISTYLE_i
\end{aligned}
$$

Validity Constraint

$$
TFTYPE_t = \sum_i z_{ti} \times IFTYPE_i
$$

B.3 Task-Level System Partitioning Subproblem

$$
\sum_{i \in I} y_{pt} = 1
$$

Function $b(e, Y)$

$$
\begin{aligned}
b_e &= 1 - \sum_p \lambda_{ep} \\
\lambda_{ep} &\leq y_{pt}, \ \forall t \in V_e \\
\lambda_{ep} &\geq \sum_t (y_{pt} - 1) + 1
\end{aligned}
$$

Function $t(e, p, Y)$

$$
\begin{aligned}
\zeta_{ep} &\geq y_{pt}, \quad t \in V_e \\
\zeta_{ep} &\leq \sum_{t \in V_e} y_{pt}
\end{aligned}
$$

Task Partition Pin Metric Function, $TPartPin(p, Y, U)$

$$
\begin{aligned}
XTPPIN_p &= \sum_{e \in E} \Theta_{ep} \\
\Theta_{ep} &\geq \delta_{ep} \times MAXBW
\end{aligned}
$$

$$
\begin{aligned}
\Theta_{ep} &\leq XBW_e \\
\Theta_{ep} &\geq XBW_e + (\delta_{ep} - 1) \times MAXBW \\
\delta_{ep} &\leq b_e \\
\delta_{ep} &\leq \zeta_{ep} \\
\delta_{ep} &\geq \zeta_{ep} + b_e - 1
\end{aligned}
$$

Task Partition Size Metric Function, $TPartSize(p, Z, Y, X, U)$

$$
\begin{aligned}
XTPSIZE_p &= \sum_d \sum_t \Lambda_{dpt} \times WB_d \times G_d + \sum_d \Delta_{dp} \times PADSIZE_d \\
\Lambda_{dpt} &\geq XTSIZE_t + (\gamma_{dpt} - 1) \times MAXTSIZE \\
\Lambda_{dpt} &\leq XTSIZE_t \\
\Lambda_{dpt} &\leq \gamma_{dpt} \times MAXTSIZE \\
\Delta_{dp} &\geq XTPPIN_p + (x_{dp} - 1) \times MAXTPPIN \\
\Delta_{dp} &\leq x_{dp} \times MAXTPPIN \\
\gamma_{dpt} &\leq y_{pt} \\
\gamma_{dpt} &\leq x_{dp} \\
\gamma_{dpt} &\geq x_{dp} + y_{pt} - 1
\end{aligned}
$$

Style Constraint for TP, Equation (3.10)

$$
\begin{aligned}
\alpha_p &\geq \mho_{pt} \\
\beta_p &\leq \mho_{pt} \\
\alpha_p &= \beta_p \\
\mho_{pt} &\leq y_{pt} \times MAXISTYLE \\
\mho_{pt} &\leq XTPSTYLE_t \\
\mho_{pt} &\geq XTPSTYLE_t + (y - 1) \times MAXISTYLE
\end{aligned}
$$

Task Partition Style

$$XTPSTYLE_p = \alpha_p$$

Task Partition Cost Metric

$$\begin{aligned}
XTPCOST_p &= \sum_r A_r \times \Phi_{rp} + \sum_r B_r \times \mu_{rp} \\
\Phi_{rp} &\leq \mu_{rp} \times MAXTPSIZE \\
\Phi_{rp} &\leq XTPSIZE_p \\
\Phi_{rp} &\geq XTPSIZE_p + (\mu_r p - 1) \times MAXTPSIZE \\
\sum_r \mu_{rp} \times S_{r+1} &\geq XTPSIZE_p \\
\sum_r \mu_{rp} \times S_r &< XTPSIZE_p \\
\sum_r \mu_{rp} &= \tau_p \\
\tau_p &\geq y_{pt} \\
\tau_p &\leq \sum_t y_{pt}
\end{aligned}$$

B.4 Die Selection Subproblem

$$\sum_b x_{bp} = \tau_p$$

Die Size Metric Function, $DieSize(p, Z, Y, X, U)$

$$\begin{aligned}
XDSSIZE_p &= \sum_{q \in Q_v} \Xi_{pq} + \sum_{q \in Q_f} \sum_d \eta_{dpq} \times DSIZE_d \\
\Xi_{pq} &\leq n_{pq} \times MAXTPSIZE \\
\Xi_{pq} &\leq XTPSIZE_p \\
\Xi_{pq} &\geq XTPSIZE_p + (n_{pq} - 1) \times MAXTPSIZE \\
\eta_{dpq} &\leq n_{pq}
\end{aligned}$$

$$
\begin{aligned}
\eta_{dpq} &\leq x_{dp} \\
\eta_{dpq} &\geq n_{pq} + x_{dp} - 1 \\
\sum_{q} n_{pq} \cdot q &= XTPSTYLE_p \\
\sum_{q} n_{pq} &= \tau_p
\end{aligned}
$$

Die Pin Metric Function, $DiePin(p, Y, X, U)$

$$
\begin{aligned}
XDSPIN_p &= \sum_{q \in Q_v} \xi_{pq} + \sum_{q \in Q_f} \sum_{d} \eta_{dpq} \times DPIN_d \\
\xi_{pq} &\leq n_{pq} \times MAXTPPIN \\
\xi_{pq} &\leq XTPPIN_p \\
\xi_{pq} &\geq XTPPIN_p + (n_{pq} - 1) \times MAXTPPIN
\end{aligned}
$$

Die Cost Metric Function, $DieCost(p, Z, Y, X, U)$

$$
\begin{aligned}
XDSCOST_p &= \sum_{q \in Q_v} \Omega_{pq} + \sum_{q \in Q_f} \sum_{b} \eta_{dpq} \times DCOST_p \\
\Omega_{pq} &\leq n_{pq} \times MAXDCOST \\
\Omega_{pq} &\leq DCOST_p \\
\Omega_{pq} &\geq DCOST_p + (n_{pq} - 1) \times MAXDCOST
\end{aligned}
$$

Metric Selection Function for the Physical Design Style

$$
XDSSTYLE_p = \sum_{d} x_{dp} \times DSTYLE_d
$$

Validity Constraints

$$
\begin{aligned}
XDSSIZE_p &\geq XTPSIZE_p \\
XDSPIN_p &\geq XTPPIN_p \\
XDSSTYLE_p &= XTPSTYLE_p
\end{aligned}
$$

B.5 Package Selection Subproblem

$$\sum_{b} w_{kp} = \tau_p$$

Metric Selection Functions

$$\begin{aligned} XPKGSIZE_p &= \sum_{k} w_{kp} \times KSIZE_k \\ XPKGCOST_p &= \sum_{k} w_{kp} \times KCOST_k \\ XPKGPIN_p &= \sum_{k} w_{kp} \times KPIN_k \end{aligned}$$

Validity Constraints

$$XPKGSIZE_p \geq XDSSIZE_p \quad \text{(B.1)}$$

$$XPKGPIN_p \geq XTPPIN_p \quad \text{(B.2)}$$

B.6 Bus Selection Subproblem

$$\sum_{d} u_{be} = 1$$

Metric Selection Functions

$$\begin{aligned} XBSWIDTH_e &= \sum_{d} u_{be} \times W_d \\ XBSCYCLE_e &= \sum_{d} u_{be} \times D_d \end{aligned}$$

Validity Constraint

$$b_e = \sum_d u_{be} \times b_d$$

B.7 User Constraints on the System Metrics and the Objective Function

B.7.1 The System Cost Constraint

$$\sum_p XDSCOST_p + XPKGCOST_p \leq GivenCost$$

B.7.2 System Performance Metric Function

$$\sum_{t \in path} XTDELAY_t + \sum_{e \in path} Vol_e \times XDPBIT_e \leq GivenDelay$$

$$XDPBIT_e = \sum_b u_{be} \times \frac{BCYCLE_b}{BWIDTH_b}$$

B.7.3 Objective Function:Modified System Time-To-Market Metric Function

$$\begin{aligned}
Obj &= MANU_{GA} \sum_p \Gamma_{p,GA} + MANU_{SC} \sum_p \Gamma_{p,SC} \\
\Gamma_{p,GA} &\leq n_{p,GA} \times MAXTPSIZE \\
\Gamma_{p,GA} &\leq XTPSIZE_p \\
\Gamma_{p,GA} &\geq XTPSIZE_p + (n_{p,GA} - 1)MAXTPSIZE \\
\Gamma_{p,SC} &\leq n_{p,SC} \times MAXTPSIZE \\
\Gamma_{p,SC} &\leq XTPSIZE_p \\
\Gamma_{p,SC} &\geq XTPSIZE_p + (n_{p,SC} - 1)MAXTPSIZE
\end{aligned}$$

Appendix C

MILP Formulation in the M Language

```
TITLE mpeg;
#
# Physical Design Style and Architecture Selection Subproblem
#
sisAA:Sig(i:I)(Bz_t_i) = 1 @t{V;
sisB:Sig(i:I)(Bz_t_i * S.i) - LSX_t = 0 @t{V;
sisC:Sig(i:I)(Bz_t_i * D.i) - LDX_t = 0 @t{V;
sisE:Sig(i:I)(Bz_t_i * Q.i) - IQX_t = 0 @t{V;
sisF:Sig(i:I)(Bz_t_i * F.i) - F.t = 0 @t{V;

#
# Task-level System Partitioning Subproblem
#
tlpA:Sig(p:P)(By_p_t) = 1 @t{V;
tlpAB:Sig(p:P)(By_p_ex) = 0;
#
# b(e,Y)
#
tlpAB:B_e - 1 + Sig(p:P)(Bla_e_p) = 0 @e{E;
tlpAC:Bla_e_p - By_p_t<= 0 @t{V~e @e{E @p{P;
tlpAD:Bla_e_p - Sig(t:V~e)( By_p_t - 1 ) - 1 >= 0 @e{E @p{P;
#
```

```
# t(e,p,Y)
#
tlpS:Bze_e_p - By_p_t >= 0 @t{V~e @e{E @p{P;
tlpT:Bze_e_p - Sig(t:V~e)(By_p_t) <= 0 @e{E @p{ P;
#
# Task Partition Pin Metric Function
#
tlpO:Sig(e:E)(ITH_e_p) - IPX_p = 0 @p{P;
tlpR:ITH_e_p - Bde_e_p * Wmax <= 0 @e{E @p{P;
tlpQ:ITH_e_p - IWX_e <= 0 @e{E @p{P;
tlpP:ITH_e_p - IWX_e - Bde_e_p * Wmax + Wmax >= 0 @e{E @p{P;
tlpU:Bde_e_p - B_e <= 0  @e{E @p{P;
tlpV:Bde_e_p - Bze_e_p <= 0  @e{E @p{P;
tlpW:Bde_e_p + 1 - Bze_e_p - B_e >= 0  @e{E @p{P;
#
# Task Partition Size Metric Function
#
tlpE:Sig(b:B)(Sig(t:T)(LG_b_p_t * WB.b * G.b)) + Sig(b:B)(IDE_b_p *
     PAD.b)  - LSX_p = 0 @p{P;
tlpF:LG_b_p_t - LSX_t - Bg_b_p_t * Smax + Smax >= 0 @b{B @p{P @t{V;
tlpG:LG_b_p_t - LSX_t <= 0 @b{B @p{P @t{V;
tlpH:LG_b_p_t - Bg_b_p_t * Smax <= 0 @b{B @p{P @t{V;
tlpL:IDE_b_p - IPX_p - Bx_b_p * Pmax + Pmax >= 0  @b{B @p{P;
tlpM:IDE_b_p - IPX_p <= 0 @b{B @p{P;
tlpN:IDE_b_p - Bx_b_p * Pmax <= 0 @b{B @p{P;
tlpI:Bg_b_p_t - By_p_t <= 0 @b{B @p{P @t{V;
tlpJ:Bg_b_p_t - Bx_b_p <= 0 @b{B @p{P @t{V;
tlpK:Bg_b_p_t - Bx_b_p - By_p_t + 1 >= 0 @b{B @p{P @t{V;
#
# Implementations of different design style cannot  be assigned to
# the same partition
#
```

```
tlpB:Ial_p - Imh_p_t >= 0 @p{P @t{V;
tlpC:Ibeta_p - Imh_p_t <= 0 @p{P @t{V;
tlpD:Ial_p - Ibeta_p = 0 @p{P;
tlpBA:Imh_p_t - By_p_t * Qmax <= 0 @p{P @t{V;
tlpBB:Imh_p_t - IQX_t <= 0 @p{P @t{V;
tlpBC:Imh_p_t - IQX_t - By_p_t * Qmax + Qmax >= 0 @p{P @t{V;
#
# Task Partition Cost Metric Function
#
bsW:LC_p - Sig(r:R)(A.r*LPI_r_p)-Sig(r:R)(B.r*Bmu_r_p) = 0 @p{P;
bsZ:LPI_r_p - Bmu_r_p * Smax <= 0 @r{R @p{P;
bsY:LPI_r_p - LSX_p <= 0 @r{R @p{P;
bsX:LPI_r_p - LSX_p - Bmu_r_p * Smax + Smax >= 0 @r{R @p{P;
bsT:LSX_p-Sig(r:R)(Bmu_r_p * S.{r+1}) < 0 @p{P;
bsU:LSX_p - Sig(r:R)(Bmu_r_p * S.r) >= 0 @p{P;
bsS:Sig(r:R)(Bmu_r_p)-Bta_p =0 @p{P;
#
# Die Selection Subproblem
#
bsA:Sig(b:B)(Bx_b_p) - \Bta_p = 0  @p{P;
bsAB:Bta_p - Sig(t:T)(By_p_t) <= 0 @p{P;
bsAA:Bta_p - By_p_t >= 0 @t{T @p{P;
#
# Die Size Metric Function
#
bsF:LBSX_p - Sig(q:Qf)(Sig(b:B)(Be_b_p_q * S.b)) - Sig(q:Qv)(LXI_p_q)
    = 0 @p{P;
bsL:LXI_p_q - Bn_p_q * Smax <= 0 @p{P @q{Qv;
bsK:LXI_p_q - LSX_p <= 0 @p{P @q{Qv;
bsJ:LXI_p_q - LSX_p - Bn_p_q * Smax + Smax >= 0 @p{P @q{Qv;
bsG:Be_b_p_q - Bn_p_q <= 0 @b{B @p{P @q{Qf;
bsH:Be_b_p_q - Bx_b_p <= 0 @b{B @p{P @q{Qf;
```

```
bsI:Be_b_p_q - Bn_p_q - Bx_b_p + 1 >= 0 @b{B @p{P @q{Qf;
bsE:IQX_p - Sig(q:Q)(Bn_p_q * Q.q) = 0  @p{P;
bsD:Sig(q:Q)(Bn_p_q) - Bta_p = 0 @p{P;
#
# Die Pin Metric Function
#
bsN:IBPX_p - Sig(q:Qf)(Sig(b:B)(Be_b_p_q * P.b)) - Sig(q:Qv)(Ixi_p_q)
    = 0 @p{P;
bsQ:Ixi_p_q - Bn_p_q * Pmax <= 0 @p{P @q{Qv;
bsP:Ixi_p_q - IPX_p <= 0 @p{P @q{Qv;
bsO:Ixi_p_q - IPX_p - Bn_p_q * Pmax + Pmax >= 0 @p{P @q{Qv;
#
# Die Cost Metric Function
#
bsO:LBCX_p - Sig(q:Qf)(Sig(b:B)(Be_b_p_q * C.b)) - Sig(q:Qv)(LOM_p_q )
    = 0 @p{P;
bs1:LOM_p_q - LC_p <= 0 @q{Qv @p{P;
bs2:LOM_p_q - LC_p - Bn_p_q * Cmax + Cmax >= 0 @q{Qv @p{P;
bs3:LOM_p_q - Bn_p_q * Cmax <= 0 @q{Qv @p{P;
#
# Metric Selection Functions for the Fabtime and the Design Style
#
bsB:IBQX_p - Sig(b:B)(Bx_b_p * Q.b) = 0 @p{P;
#
# Validity Constraints
#
bsM:LBSX_p - LSX_p >= 0 @p{P;
bsR:IBPX_p - IPX_p >= 0 @p{P;
bsC:IQX_p - IBQX_p = 0 @p{P;
#
# Package Selection
#
```

```
psA:Sig(k:K)(Bw_k_p) - Bta_p = 0 @p{P;
#
# Metric Selection Functions
#
psD:LKSX_p - Sig(k:K)(Bw_k_p * S.k) = 0 @p{P;
psG:LKCX_p - Sig(k:K)(Bw_k_p * C.k) = 0 @p{P;
ps:IKPX_p - Sig(k:K)(Bw_k_p * P.k) = 0 @p{P;
#
# Validity Constraints
#
psE:LKSX_p - LBSX_p >= 0 @p{P;
psF:IKPX_p - IPX_p >= 0 @p{P;
#
# Bus Selection Subproblem
#
icE:Sig(b:B)(Bd_e_c)=1 @e{E;
#
# Metric Selection Functions
#
icA:Sig(b:B)(Bd_e_c * W.c) - IWX_e = 0 @e{E;
icB:Sig(b:B)(Bd_e_c * D.c) - LDX_e = 0 @e{E;
#
# Validity Constraint
#
icC:Sig(b:B)(Bd_e_c * b.c) - B_e = 0 @e{E;
#
# User Constraints on the System Metrics and the Objective Function
#
# System Cost
#
CC:Sig(p:P)(LBCX_p) + Sig(p:P)(LKCX_p) <= Cs;
#
```

```
# System Performance
#
TC:Sig(t:T~path)(LDX_t)+Sig(e:E~path)(V.e*LBTX_e) - D.path <= 0 @path{PATH;
icD:Sig(b:B)(Bd_e_c * B.c) - LBTX_e = 0 @e{E;
#
# Objective Function
#
OBJ:Sig(p:P)(LGa_p_ga * MANUga)+Sig(p:P)(LG_p_sc*MANUsc);
obja:LG_p_ga - n_p_ga * Smax <= 0 @p{P;
objb:LG_p_ga - LSX_p <= 0 @p{P;
objc:LG_p_ga - LSX_p - n_p_ga * Smax + Smax >= 0 @p{P;
obja:LG_p_sc - n_p_sc * Smax <= 0 @p{P;
objb:LG_p_sc - LSX_p <= 0 @p{P;
objc:LG_p_sc - LSX_p - n_p_sc * Smax + Smax >= 0 @p{P;
end
```

Appendix D

The VHDL Descriptions of Tasks in the JPEG and MPEG examples

In this appendix, the VHDL specifications for tasks in the JPEG and MPEG examples are listed. The specifications are developed based on a public C implmentation. The first JPEG specification was simulated with ViewLogic VHDL simulator and part of the first JPEG specification is given here. The first JPEG specification was customized considerably for VHDL2DDS parser and BEST. Some of such modifications are as follows:

1. Data interfaces between tasks through files were removed,
2. "inout" I/O type was removed,
3. Some constants declaration in the JPEG package file were moved to each task, and
4. Unfixed loops were fixed.

For the MPEG encoder design, a motion estimation description was written.

D.1 JPEG Package

```
-- jpeg.vhd
-- Types and Functions for JPEG decorder
-- by DongHyun Heo
```

```
  -- This is for v2dss
  -- Remove inout variables and temporily remove emit_bits
  -- Remove assert statement.
  -- Remove the body of the function and procedures to
  --   reduce the run-time
  -- constants are moved to corresponding files

  package jpeg is

------------------------------------------------
-- TYPE DEFNITION
------------------------------------------------
    --type long is array(0 to 31) of vlbit;
    type long is array(0 to 31) of bit;
    type frame is array( 0 to 63 ) of integer;
    type inttable is array( 0 to 63 ) of integer;
    type huftable is array( 0 to 255 ) of integer;
    type outbuf   is array( 0 to 63 ) of integer;

------------------------------------------------
-- CONSTANT DEFNITION
------------------------------------------------

    constant ZAG : inttable := (
        0,  1,  8, 16,  9,  2,  3, 10,
        17, 24, 32, 25, 18, 11,  4,  5,
        12, 19, 26, 33, 40, 48, 41, 34,
        27, 20, 13,  6,  7, 14, 21, 28,
        35, 42, 49, 56, 57, 50, 43, 36,
        29, 22, 15, 23, 30, 37, 44, 51,
        58, 59, 52, 45, 38, 31, 39, 46,
```

```
    53, 60, 61, 54, 47, 55, 62, 63
);

constant dchufco : huftable := (
    0, 2, 3, 4, 5, 6, 14, 30,
    62, 126, 254, 510, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
```

```
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0
);
constant dchufsi : huftable  := (
    2, 3, 3, 3, 3, 3, 4, 5,
    6, 7, 8, 9, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
```

```
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0
);
constant achufco : huftable := (
    10, 0, 1, 4, 11, 26, 120, 248,
    1014, 65410, 65411, 0, 0, 0, 0, 0,
    0, 12, 27, 121, 502, 2038, 65412, 65413,
    65414, 65415, 65416, 0, 0, 0, 0, 0,
    0, 28, 249, 1015, 4084, 65417, 65418, 65419,
    65420, 65421, 65422, 0, 0, 0, 0, 0,
    0, 58, 503, 4085, 65423, 65424, 65425, 65426,
    65427, 65428, 65429, 0, 0, 0, 0, 0,
    0, 59, 1016, 65430, 65431, 65432, 65433, 65434,
    65435, 65436, 65437, 0, 0, 0, 0, 0,
    0, 122, 2039, 65438, 65439, 65440, 65441, 65442,
    65443, 65444, 65445, 0, 0, 0, 0, 0,
    0, 123, 4086, 65446, 65447, 65448, 65449, 65450,
    65451, 65452, 65453, 0, 0, 0, 0, 0,
    0, 250, 4087, 65454, 65455, 65456, 65457, 65458,
    65459, 65460, 65461, 0, 0, 0, 0, 0,
    0, 504, 32704, 65462, 65463, 65464, 65465, 65466,
    65467, 65468, 65469, 0, 0, 0, 0, 0,
    0, 505, 65470, 65471, 65472, 65473, 65474, 65475,
    65476, 65477, 65478, 0, 0, 0, 0, 0,
    0, 506, 65479, 65480, 65481, 65482, 65483, 65484,
    65485, 65486, 65487, 0, 0, 0, 0, 0,
    0, 1017, 65488, 65489, 65490, 65491, 65492, 65493,
    65494, 65495, 65496, 0, 0, 0, 0, 0,
```

```
    0, 1018, 65497, 65498, 65499, 65500, 65501, 65502,
    65503, 65504, 65505, 0, 0, 0, 0, 0,
    0, 2040, 65506, 65507, 65508, 65509, 65510, 65511,
    65512, 65513, 65514, 0, 0, 0, 0, 0,
    0, 65515, 65516, 65517, 65518, 65519, 65520, 65521,
    65522, 65523, 65524, 0, 0, 0, 0, 0,
    2041, 65525, 65526, 65527, 65528, 65529, 65530, 65531,
    65532, 65533, 65534, 0, 0, 0, 0, 0
);
constant achufsi : huftable := (
    4, 2, 2, 3, 4, 5, 7, 8,
    10, 16, 16, 0, 0, 0, 0, 0,
    0, 4, 5, 7, 9, 11, 16, 16,
    16, 16, 16, 0, 0, 0, 0, 0,
    0, 5, 8, 10, 12, 16, 16, 16,
    16, 16, 16, 0, 0, 0, 0, 0,
    0, 6, 9, 12, 16, 16, 16, 16,
    16, 16, 16, 0, 0, 0, 0, 0,
    0, 6, 10, 16, 16, 16, 16, 16,
    16, 16, 16, 0, 0, 0, 0, 0,
    0, 7, 11, 16, 16, 16, 16, 16,
    16, 16, 16, 0, 0, 0, 0, 0,
    0, 7, 12, 16, 16, 16, 16, 16,
    16, 16, 16, 0, 0, 0, 0, 0,
    0, 8, 12, 16, 16, 16, 16, 16,
    16, 16, 16, 0, 0, 0, 0, 0,
    0, 9, 15, 16, 16, 16, 16, 16,
    16, 16, 16, 0, 0, 0, 0, 0,
    0, 9, 16, 16, 16, 16, 16, 16,
    16, 16, 16, 0, 0, 0, 0, 0,
    0, 9, 16, 16, 16, 16, 16, 16,
    16, 16, 16, 0, 0, 0, 0, 0,
```

```
        0, 10, 16, 16, 16, 16, 16, 16,
        16, 16, 16, 0, 0, 0, 0, 0,
        0, 10, 16, 16, 16, 16, 16, 16,
        16, 16, 16, 0, 0, 0, 0, 0,
        0, 11, 16, 16, 16, 16, 16, 16,
        16, 16, 16, 0, 0, 0, 0, 0,
        0, 16, 16, 16, 16, 16, 16, 16,
        16, 16, 16, 0, 0, 0, 0, 0,
        11, 16, 16, 16, 16, 16, 16, 16,
        16, 16, 16, 0, 0, 0, 0, 0
    );

------------------------------------------------
-- SUBPROGRAM DECLARATION
------------------------------------------------
    function int2long( n : integer ) return long;
    function long2int ( s : long ) return integer;
    function iOR ( x : integer; y : integer ) return integer;
    function iAND ( x : integer; y : integer ) return integer;
    function rshlong( x: integer; n : integer ) return integer;
    function lshlong( x: integer; n : integer ) return integer;
    function arshlong( x: integer; n : integer ) return integer;

    function descale( x: integer; n : integer ) return integer;

    procedure emit_bits ( code : integer;
                          size : integer;
                          huff_put_buffer_in :  integer;
                          huff_put_buffer_out : OUT integer;
                          huff_put_bits_in : integer;
                          huff_put_bits_out : OUT integer;
                          output_buffer : OUT outbuf;
```

```
                              bytes_in_buffer_in :  integer;
                              bytes_in_buffer_out: out integer );
  end jpeg;

-------------------------------------------------
-- PACKAGE BODY
-------------------------------------------------
  package body jpeg is

-------------------------------------------------
-- int2long
-------------------------------------------------
  function int2long ( n : integer ) return long is
    variable FourByte : long;
    variable i, m : integer;

    begin
           return FourByte;

     end int2long;

-------------------------------------------------
-- long2int
-------------------------------------------------
  function long2int ( s : long ) return integer is

    variable i, it : integer;
    variable t : long;

    begin
        return it;
    end long2int;
```

```
-----------------------------------------------
-- iOR
-----------------------------------------------
    function iOR ( x : integer; y : integer ) return integer is

    variable bz : long;

    begin
       return long2int(bz);
    end;

-----------------------------------------------
-- iAND
-----------------------------------------------
    function iAND ( x : integer; y : integer ) return integer is

    variable bz : long;

    begin
       return long2int(bz);
    end;

-----------------------------------------------
-- lshlong
-----------------------------------------------
    function lshlong( x: integer; n : integer ) return  integer is

    variable iz, m : integer;

    begin
```

```
        iz := x;
        m  := n;
        while ( m > 0 ) loop
                iz := iz * 2;
                m  := m - 1;
       end loop;
       return iz;

    end lshlong;

-------------------------------------------------
-- rshlong
-------------------------------------------------
    function rshlong( x: integer; n : integer ) return  integer is

    variable z, m : integer;

    begin
       return z;
    end rshlong;

-------------------------------------------------
-- arshlong
-------------------------------------------------
    function arshlong( x: integer; n : integer ) return  integer is

    variable z : integer;

    begin
       return rshlong( z, n );
    end arshlong;
```

```
-----------------------------------------------
-- descale
-----------------------------------------------
   function descale( x: integer; n : integer ) return integer is

   begin
       return arshlong(  x + lshlong( 1, n-1 ), n);
   end descale;

-----------------------------------------------
-- emit_bits
-----------------------------------------------
    procedure emit_bits ( code : integer;
                          size : integer;
                          huff_put_buffer_in :  integer;
                          huff_put_buffer_out : out integer;
                          huff_put_bits_in : integer;
                          huff_put_bits_out : out integer;
                              output_buffer : out outbuf;
                          bytes_in_buffer_in : integer;
                          bytes_in_buffer_out : out integer ) is

    variable put_buffer : integer;
    variable put_bits : integer;
    variable c : integer;
    variable huff_put_buffer : integer;
    variable huff_put_bits : integer;
    variable bytes_in_buffer : integer;

    begin

    end emit_bits;
```

```
end jpeg;
```

D.2 Forward DCT Specification

```
--
-- dct.vhdl
--
-- source : from dct.vhdl
-- constant in the loop condition is replaced with the real value

use work.jpeg.all;

entity dct is
   port( indata : in frame; outdata : out frame );
end dct;

architecture behavior of dct is
  constant DCTSIZE : integer := 8;
  constant DCTSIZE2 : integer := 64;
  constant CONST_BITS : integer := 13;
  constant PASS1_BITS : integer := 2;

  constant XFF : integer := 255;
  constant XF0 : integer := 240;

  constant FIX_0_298631336 : integer := 2446;
  constant FIX_0_390180644 : integer := 3196;
  constant FIX_0_541196100 : integer := 4433;
  constant FIX_0_765366865 : integer := 6270;
  constant FIX_0_899976223 : integer := 7373;
  constant FIX_1_175875602 : integer := 9633;
```

```
  constant FIX_1_501321110 : integer := 12299;
  constant FIX_1_847759065 : integer := 15137;
  constant FIX_1_961570560 : integer := 16069;
  constant FIX_2_053119869 : integer := 16819;
  constant FIX_2_562915447 : integer := 20995;
  constant FIX_3_072711026 : integer := 25172;

begin

dct:process

   variable  tmp0, tmp1, tmp2, tmp3, tmp4,
             tmp5, tmp6, tmp7 : integer;
   variable  tmp10, tmp11, tmp12, tmp13 : integer;
   variable  z1, z2, z3, z4, z5 : integer;
   variable  I, J: integer;
   variable  data : frame;
   variable  L, M : integer;

begin
   I := 0;
   while (I < 8) loop
     L:= I*DCTSIZE; M:=I*DCTSIZE+7;
     tmp0 := indata(L) + indata(M);
     tmp7 := indata(L) - indata(M);
     L:= I*DCTSIZE+1; M:=I*DCTSIZE+6;
     tmp1 := indata(L) + indata(M);
     tmp6 := indata(L) - indata(M);
     L:= I*DCTSIZE+2; M:=I*DCTSIZE+5;
     tmp2 := indata(L) + indata(M);
     tmp5 := indata(L) - indata(M);
     L:= I*DCTSIZE+3; M:=I*DCTSIZE+4;
```

```
tmp3 := indata(L) + indata(M);
tmp4 := indata(L) - indata(M);

 -- Even part per LL&M figure 1
 -- note that published figure is faulty;
 -- rotator "sqrt(2)*c1" should be "sqrt(2)*c6".

tmp10 := tmp0 + tmp3;
tmp13 := tmp0 - tmp3;
tmp11 := tmp1 + tmp2;
tmp12 := tmp1 - tmp2;

 L := I*DCTSIZE; M:= I*DCTSIZE+4;
 data(L) := lshlong( tmp10+tmp11, PASS1_BITS);
 data(M) := lshlong( tmp10-tmp11, PASS1_BITS);

 z1 := (tmp12+ tmp13)* FIX_0_541196100;

 L := I*DCTSIZE+2; M:= I*DCTSIZE+6;
 data(L) := descale( z1+tmp13*FIX_0_765366865,
                          CONST_BITS-PASS1_BITS);
 data(M) := descale( z1-tmp12*FIX_1_847759065,
                          CONST_BITS-PASS1_BITS);

 -- Odd part per figure 8
 -- note paper omits factor of sqrt(2).
 -- cK represents cos(K*pi/16).
 -- i0..i3 in the paper are tmp4..tmp7 here.

 z1 := tmp4 + tmp7;
 z2 := tmp5 + tmp6;
 z3 := tmp4 + tmp6;
```

```
   z4 := tmp5 + tmp7;
   z5 := (z3+ z4)*FIX_1_175875602;  -- sqrt(2) * c3

   tmp4 :=tmp4 * FIX_0_298631336; -- sqrt(2) * (-c1+c3+c5-c7)
   tmp5 :=tmp5 * FIX_2_053119869; -- sqrt(2) * ( c1+c3-c5+c7)
   tmp6 :=tmp6 * FIX_3_072711026; -- sqrt(2) * ( c1+c3+c5-c7)
   tmp7 :=tmp7 * FIX_1_501321110; -- sqrt(2) * ( c1+c3-c5-c7)

   z1 := - z1 * FIX_0_899976223; -- sqrt(2) * (c7-c3)
   z2 := - z2 * FIX_2_562915447; -- sqrt(2) * (-c1-c3)
   z3 := - z3 * FIX_1_961570560; -- sqrt(2) * (-c3-c5)
   z4 := - z4 * FIX_0_390180644; -- sqrt(2) * (c5-c3)

   z3 := z3 + z5;
   z4 := z4 + z5;
   L := I*DCTSIZE+7;
   data(L) :=descale(tmp4 + z1 + z3, CONST_BITS-PASS1_BITS);
   L := I*DCTSIZE+5;
   data(L) :=descale(tmp5 + z2 + z4, CONST_BITS-PASS1_BITS);
   L := I*DCTSIZE+3;
   data(L) :=descale(tmp6 + z2 + z3, CONST_BITS-PASS1_BITS);
   L := I*DCTSIZE+1;
   data(L) :=descale(tmp7 + z1 + z4, CONST_BITS-PASS1_BITS);

   I := I + 1; -- advance pointer to next row
end loop;

-- 2nd phase of DCT

--wait until clk = '1';

I := 0;
```

```
 while ( I < 8 ) loop
   L := I; M:= DCTSIZE*7+I;
   tmp0 := data(L) + data(M);
   tmp7 := data(L) - data(M);
   L := DCTSIZE*1+I; M:= DCTSIZE*6+I;
   tmp1 := data(L) + data(M);
   tmp6 := data(L) - data(M);
   L := DCTSIZE*2+I; M:= DCTSIZE*5+I;
   tmp2 := data(L) + data(M);
   tmp5 := data(L) - data(M);
   L := DCTSIZE*3+I; M:= DCTSIZE*4+I;
   tmp3 := data(L) + data(M);
   tmp4 := data(L) - data(M);

-- Even part per LL&M figure 1
-- note that published figure is faulty;
-- rotator "sqrt(2)*c1" should be "sqrt(2)*c6".

   tmp10 := tmp0 + tmp3;
   tmp13 := tmp0 - tmp3;
   tmp11 := tmp1 + tmp2;
   tmp12 := tmp1 - tmp2;

   L := DCTSIZE*0+I; M:= DCTSIZE*4+I;
   outdata(L)  <= descale(tmp10 + tmp11, PASS1_BITS+3);
   outdata(M)  <= descale(tmp10 - tmp11, PASS1_BITS+3);

   z1 := (tmp12 + tmp13) * FIX_0_541196100;
   L := DCTSIZE*2+I; M:= DCTSIZE*6+I;
   outdata(L)  <= descale(z1 + tmp13 * FIX_0_765366865,
                          CONST_BITS+PASS1_BITS+3);
   outdata(M)  <= descale(z1 - tmp12 * FIX_1_847759065,
```

```
                    CONST_BITS+PASS1_BITS+3);

        -- Odd part per figure 8
        -- note paper omits factor of sqrt(2+I).
        -- cK represents cos(K*pi/16+I).
        -- i0..i3 in the paper are tmp4..tmp7 here.

        z1 := tmp4 + tmp7;
        z2 := tmp5 + tmp6;
        z3 := tmp4 + tmp6;
        z4 := tmp5 + tmp7;
        z5 := (z3 + z4)* FIX_1_175875602; -- sqrt(2) * c3

        tmp4 := tmp4*FIX_0_298631336; -- sqrt(2) * (-c1+c3+c5-c7)
        tmp5 := tmp5*FIX_2_053119869; -- sqrt(2) * ( c1+c3-c5+c7)
        tmp6 := tmp6*FIX_3_072711026; -- sqrt(2) * ( c1+c3+c5-c7)
        tmp7 := tmp7*FIX_1_501321110; -- sqrt(2) * ( c1+c3-c5-c7)
        z1 := -z1 * FIX_0_899976223; -- sqrt(2) * (c7-c3)
        z2 := -z2 * FIX_2_562915447; -- sqrt(2) * (-c1-c3)
        z3 := -z3 * FIX_1_961570560; -- sqrt(2) * (-c3-c5)
        z4 := -z4 * FIX_0_390180644; -- sqrt(2) * (c5-c3)

        z3 := z3 + z5;
        z4 := z4 + z5;

        L := DCTSIZE*7+I;
        outdata(L)  <= descale(tmp4 + z1 + z3,
CONST_BITS+PASS1_BITS+3);
        L := DCTSIZE*5+I;
        outdata(L)  <= descale(tmp5 + z2 + z4,
CONST_BITS+PASS1_BITS+3);
        L := DCTSIZE*3+I;
```

```
        outdata(L)  <= descale(tmp6 + z2 + z3,
CONST_BITS+PASS1_BITS+3);
        L := DCTSIZE*1+I;
        outdata(L)  <= descale(tmp7 + z1 + z4,
CONST_BITS+PASS1_BITS+3);
        I := I + 1;     -- advance pointer to next column

   end loop;  -- end 2nd phase

   end process;

end behavior;
```

D.3 Quantizer Specification

```
--
-- quantizer.vhd
--
-- source : converted from the public domain jpeg software
--
-- remove vlbit;

use work.ijpeg.all;

entity quantizer is
   port( data : in frame; quanttbl : in frame;
         ZAG : in inttable; outputdata : out frame);
end quantizer;

architecture behavior of quantizer is

begin
```

```
quantizer : process

  variable temp : integer;

  variable I, J : integer;

  variable n : integer;
  variable b : boolean;

  begin

  I := 0;
  J := 0;
  while ( I <  64 ) loop

    temp := data(ZAG(I));

    -- divide by quanttbl[J], ensuring proper rounding

    if (temp < 0)
    then
      temp := -temp;
      temp := temp + rshlong( quanttbl(J), 1 );
      temp := temp/quanttbl(J);
      temp := -temp;
    else
      temp := temp + rshlong(quanttbl(J), 1);
      temp := temp/quanttbl(J);
    end if;

    outputdata(I) <= temp;
```

```
      I := I + 1;
      J := J + 1;
    end loop;


  end process;

end behavior;
```

D.4 Huffman Encoding Specification

```
  --
  -- huff.vhd
  --
  -- source : converted from the public domain jpeg software
  -- declare the input at port
  -- the iteration of a loop is fixed.
  -- inout type is seperated into in and out type

  use work.jpeg.all;

  entity huffman is
     port( data : in frame; output_buffer : out outbuf );
  end huffman;

  -- Encode the DC coefficient difference per section F.1.2.1

  architecture behavior of huffman is

  begin

    huffman : process
```

```
    variable temp, temp2, nbits, r, i, j, k : integer;
    variable  data : frame;
    variable huff_put_buffer : integer;
    variable huff_put_bits : integer;
    variable output_buffer : outbuf;
    variable bytes_in_buffer : integer;
    variable i2 : integer;
    begin

      -- Initialize variables
      huff_put_buffer := 0;
      huff_put_bits   := 0;
      bytes_in_buffer := 0;

      temp2 := data(0);
      temp  := temp2;

      if (temp < 0)
      then
        temp := -temp;    -- temp is abs value of input
        -- For a negative input,
        -- want temp2 = bitwise complement of abs(input)
        -- This code assumes we are on a two's complement
-- machine
        temp2 := temp2 - 1;
      end if;

      -- Find the number of bits needed for the magnitude of the
      -- coefficient
      nbits := 0;
      i:=0;
```

```
while(i < 16 ) loop
--while (temp > 0) loop
--nbits := nbits + 1;
  nbits := nbits + 1;
  temp := rshlong( temp, 1);
  i := i+1;
end loop;

-- Emit the Huffman-coded symbol for the number of bits
emit_bits(dchufco(nbits), dchufsi(nbits),
  huff_put_buffer, huff_put_buffer,
  huff_put_bits, huff_put_bits, output_buffer,
  bytes_in_buffer, bytes_in_buffer );

-- Emit that number of bits of the value, if positive,
-- or the complement of its magnitude, if negative.
if (nbits /= 0)  -- emit_bits rejects calls with size 0
then
  emit_bits( temp2, nbits,
    huff_put_buffer, huff_put_buffer,
    huff_put_bits, huff_put_bits, output_buffer,
    bytes_in_buffer, bytes_in_buffer );
end if;

-- Encode the AC coefficients per section F.1.2.2

r := 0;                    -- r = run length of zeros
k := 1;
while ( k < 64 ) loop

  temp := data(k);
```

```
        if ( temp = 0 )
        then
          r := r + 1;
        else

          -- if run length > 15,
          -- must emit special run-length-16 codes (0xF0)
          --while ( r > 15 ) loop
          if ( r > 15) then
            emit_bits( achufco(240), achufsi(240),
              huff_put_buffer, huff_put_buffer,
              huff_put_bits, huff_put_bits, output_buffer,
              bytes_in_buffer, bytes_in_buffer );
            r := r - 16;
          end if;
          --end loop;

          temp2 := temp;

          if (temp < 0)
          then
            temp := -temp;  -- temp is abs value of input
            -- This code assumes we are on a two's complement
            -- machine
            temp2 := temp2 - 1;
          end if;

          -- Find the number of bits needed for the magnitude
-- of the coefficient

          nbits := 1;     -- there must be at least one 1 bit
                temp := rshlong( temp, 1);
```

```
    i2 := 0;
    while ( i2 < 16 ) loop
    --while (temp > 0 ) loop
      nbits := nbits + 1;
            temp := rshlong( temp, 1);
      i2 := i2 + 1;
    end loop;

    -- Emit Huffman symbol for run length / number of bits

    i := lshlong(r, 4) +  nbits;
    emit_bits(achufco(i), achufsi(i),
        huff_put_buffer, huff_put_buffer,
        huff_put_bits, huff_put_bits, output_buffer,
        bytes_in_buffer, bytes_in_buffer );

    -- Emit that number of bits of the value, if positive,
    -- or the complement of its magnitude, if negative.

    emit_bits( temp2, nbits,
        huff_put_buffer, huff_put_buffer,
        huff_put_bits, huff_put_bits, output_buffer,
        bytes_in_buffer, bytes_in_buffer );

    r := 0;
  end if;

  k := k + 1;

end loop;

-- If the last coef(s) were zero,
```

```
        -- emit an end-of-bdata code

        if (r > 0)
        then
          emit_bits(achufco(0), achufsi(0),
                huff_put_buffer, huff_put_buffer,
                huff_put_bits, huff_put_bits, output_buffer,
                bytes_in_buffer, bytes_in_buffer );
        end if;

  end process;

end behavior;
```

D.5 Inverse DCT Specification

```
use work.ijpeg.all;

entity idct is
        port( indata : in frame; outdata : out frame );
end idct;

architecture behavior of idct is

begin

idct : process

  variable tmp0, tmp1, tmp2, tmp3 : integer;
  variable tmp10, tmp11, tmp12, tmp13 : integer;
  variable z1, z2, z3, z4, z5 : integer;
  variable I : integer;
```

```
variable data : frame;

begin

I := 8-1;
while (I >= 0) loop

  -- Even part: reverse the even part of the forward DCT.
  -- The rotator is sqrt(2)*c(-6).

  z2 :=  indata(8*I+2);
  z3 :=  indata(8*I+6);

  z1 := (z2+z3)* FIX_0_541196100;
  tmp2 := z1 - z3 * FIX_1_847759065;
  tmp3 := z1 + z2 * FIX_0_765366865;

  tmp0 := lshlong((indata(8*I+0) +  indata(8*I+4)), CONST_BITS);
  tmp1 := lshlong((indata(8*I+0) -  indata(8*I+4)), CONST_BITS);

  tmp10 := tmp0 + tmp3;
  tmp13 := tmp0 - tmp3;
  tmp11 := tmp1 + tmp2;
  tmp12 := tmp1 - tmp2;

  -- Odd part per figure 8; the matrix is unitary and hence its
  -- transpose is its inverse.
  -- i0..i3 are y7,y5,y3,y1 respectively.

  tmp0 :=  indata(8*I+7);
  tmp1 :=  indata(8*I+5);
  tmp2 :=  indata(8*I+3);
```

```
tmp3 :=  indata(8*I+1);

z1 := tmp0 + tmp3;
z2 := tmp1 + tmp2;
z3 := tmp0 + tmp2;
z4 := tmp1 + tmp3;
z5 := (z3 + z4)* FIX_1_175875602; -- sqrt(2) * c3

tmp0 := tmp0 * FIX_0_298631336; -- sqrt(2) * (-c1+c3+c5-c7)
tmp1 := tmp1 * FIX_2_053119869; -- sqrt(2) * ( c1+c3-c5+c7)
tmp2 := tmp2 * FIX_3_072711026; -- sqrt(2) * ( c1+c3+c5-c7)
tmp3 := tmp3 * FIX_1_501321110; -- sqrt(2) * ( c1+c3-c5-c7)
z1 := - z1 * FIX_0_899976223; -- sqrt(2) * (c7-c3)
z2 := - z2 * FIX_2_562915447; -- sqrt(2) * (-c1-c3)
z3 := - z3 * FIX_1_961570560; -- sqrt(2) * (-c3-c5)
z4 := - z4 * FIX_0_390180644; -- sqrt(2) * (c5-c3)

z3 := z3 +  z5;
z4 := z4 +  z5;

tmp0 := tmp0 +  z1 + z3;
tmp1 := tmp1 +  z2 + z4;
tmp2 := tmp2 +  z2 + z3;
tmp3 := tmp3 +  z1 + z4;

-- Final output stage: inputs are tmp10..tmp13, tmp0..tmp3

data(8*I+0) :=  descale(tmp10 + tmp3, CONST_BITS-PASS1_BITS);
data(8*I+7) :=  descale(tmp10 - tmp3, CONST_BITS-PASS1_BITS);
data(8*I+1) :=  descale(tmp11 + tmp2, CONST_BITS-PASS1_BITS);
data(8*I+6) :=  descale(tmp11 - tmp2, CONST_BITS-PASS1_BITS);
data(8*I+2) :=  descale(tmp12 + tmp1, CONST_BITS-PASS1_BITS);
```

```
  data(8*I+5) :=  descale(tmp12 - tmp1, CONST_BITS-PASS1_BITS);
  data(8*I+3) :=  descale(tmp13 + tmp0, CONST_BITS-PASS1_BITS);
  data(8*I+4) :=  descale(tmp13 - tmp0, CONST_BITS-PASS1_BITS);

  I := I - 1;
end loop; -- while

-- Pass 2: process columns.
-- Note that we must descale the results by a factor of
-- 8 == 2**3, and also undo the PASS1_BITS scaling.

I := 8-1;
while( I>=0 ) loop
  -- Even part: reverse the even part of the forward DCT.
  -- The rotator is sqrt(2)*c(-6).

  z2 :=  data(I+8*2);
  z3 :=  data(I+8*6);

  z1 := (z2 + z3)* FIX_0_541196100;
  tmp2 := z1 - z3 * FIX_1_847759065;
  tmp3 := z1 + z2 * FIX_0_765366865;

  tmp0 := lshlong(( data(8*0) +  data(8*4)), CONST_BITS);
  tmp1 := lshlong(( data(8*0) -  data(8*4)), CONST_BITS);

  tmp10 := tmp0 + tmp3;
  tmp13 := tmp0 - tmp3;
  tmp11 := tmp1 + tmp2;
  tmp12 := tmp1 - tmp2;

  -- Odd part per figure 8; the matrix is unitary and hence its
```

```
-- transpose is its inverse.
-- i0..i3 are y7,y5,y3,y1 respectively.


tmp0 :=  data(I+8*7);
tmp1 :=  data(I+8*5);
tmp2 :=  data(I+8*3);
tmp3 :=  data(I+8*1);

z1 := tmp0 + tmp3;
z2 := tmp1 + tmp2;
z3 := tmp0 + tmp2;
z4 := tmp1 + tmp3;
z5 := (z3 + z4)* FIX_1_175875602; -- sqrt(2) * c3

tmp0 := tmp0 * FIX_0_298631336; -- sqrt(2) * (-c1+c3+c5-c7)
tmp1 := tmp1 * FIX_2_053119869; -- sqrt(2) * ( c1+c3-c5+c7)
tmp2 := tmp2 * FIX_3_072711026; -- sqrt(2) * ( c1+c3+c5-c7)
tmp3 := tmp3 * FIX_1_501321110; -- sqrt(2) * ( c1+c3-c5-c7)
z1 := - z1 * FIX_0_899976223; -- sqrt(2) * (c7-c3)
z2 := - z2 * FIX_2_562915447; -- sqrt(2) * (-c1-c3)
z3 := - z3 * FIX_1_961570560; -- sqrt(2) * (-c3-c5)
z4 := - z4 * FIX_0_390180644; -- sqrt(2) * (c5-c3)

z3 := z3 +  z5;
z4 := z4 +  z5;

tmp0 := tmp0 +  z1 + z3;
tmp1 := tmp1 +  z2 + z4;
tmp2 := tmp2 +  z2 + z3;
tmp3 := tmp3 +  z1 + z4;
```

```
    -- Final output stage: inputs are tmp10..tmp13, tmp0..tmp3

    outdata(I+8*0) <=  descale(tmp10 + tmp3,
                                        CONST_BITS+PASS1_BITS+3);
    outdata(I+8*7) <=  descale(tmp10 - tmp3,
                                        CONST_BITS+PASS1_BITS+3);
    outdata(I+8*1) <=  descale(tmp11 + tmp2,
                                        CONST_BITS+PASS1_BITS+3);
    outdata(I+8*6) <=  descale(tmp11 - tmp2,
                                        CONST_BITS+PASS1_BITS+3);
    outdata(I+8*2) <=  descale(tmp12 + tmp1,
                                        CONST_BITS+PASS1_BITS+3);
    outdata(I+8*5) <=  descale(tmp12 - tmp1,
                                        CONST_BITS+PASS1_BITS+3);
    outdata(I+8*3) <=  descale(tmp13 + tmp0,
                                        CONST_BITS+PASS1_BITS+3);
    outdata(I+8*4) <=  descale(tmp13 - tmp0,
                                        CONST_BITS+PASS1_BITS+3);


  end loop; -- while
  end process;
end behavior;
```

D.6 Dequantizer Specification

```
use work.ijpeg.all;

entity dequantize is
        port(datain : in frame; quanttbl : in frame;
     ZAG : in inttable; data : out frame);
end dequantize;
```

```
architecture behavior of dequantize is

begin

dequantize : process

  variable s, k, r : integer;
  variable blkn, ci : integer;

  begin

    --DC coefficient dequantization

    --Descale and output the DC coefficient (assumes ZAG[0] = 0)
    s := datain(0);
    data(0) <= s * quanttbl(0);

    --Section F.2.2.2: decode the AC coefficients
    --Since zero values are skipped, output area must be
    --zeroed beforehand
    k := 1;
    while ( k < 64 ) loop
        r := datain(k); -- ac coefficients
        s := iAND(r, 15);
        r := rshlong(r, 4);
        data(ZAG(k)) <= s * quanttbl(k);
        k := k + 1;
    end loop;
  end process;
end behavior;
```

D.7 Huffman Decoder Specification

```
use work.ijpeg.all;

entity dehuff is
        port(data : out frame, get_buffer : in integer);
end dehuff;

architecture behavior of dehuff is

begin

  dehuff : process

  variable l : integer;
  variable code : int32;
  variable bits_left : integer;

  if ( bits_left = 0 ) then
       bits_left=15;
  else
       bits_left := bits_left - 1;
  end if;
  code := rshlong(get_buffer, bits_left) and 1;

  --Assume that the macro block detection is done
  --Recover DC coefficent
    l := 1;
    while (code > dcmaxcode(l)) loop

      if ( bits_left = 0 ) then
        bits_left=15;
```

```
  else
    bits_left := bits_left - 1;

  code := rshlong(get_buffer, bits_left) and 1;

  code = lshlong(code,1) or code;
  l := l + 1;
end loop;
s := dchuffval(valptr(l) + code - dcmincode(l));
if ( s /= 0 ) then
  if ( bits_left >= s )
  then
    bits_left := bits_left - s;
    r := get_buffer >> bits_left
    r := r and bmask(s);
  else
    r := fill_buffer_..
  end if;

  if ( r < extend_test(s) )
  then
    s := r + extend_offset(s);
  else
    s := r;

endif
data(0) <= s;

--AC coefficient
k := 1;
while ( k < DCTSIZE2 ) loop
  l := 1;
```

```
while (code > acmaxcode(l)) loop
  if ( bits_left = 0 )
  then
    bits_left=15;
  else
    bits_left := bits_left - 1;
  end if;
  code := rshlong(get_buffer, bits_left) and 1;
  code = lshlong(code ,1) or code;
  l := l + 1;
end loop;
r := achuffval(valptr(l) + code - acmincode(l));
s := r and 15;
r := rshlong(r, 4);

if (s /= 0)
then
  k := k+r;
  r := get_bits(s);
  if ( r < extend_test(s) )
  then
    s := r + extend_offset(s);
  else
    s := r;
  end if;
else
  if (r /= 15)
  then
    break;
  k := k + 15;
endif
```

```
    data(k) <= s;
    k := k + 1;
end loop;
}
```

D.8 Motion Estimation Specification

```
-- Motion Estimation Block

entity motion is
  port( current : in macro_block;
        prev : in macro_block;
        xmin : out integer;
        ymin : out integer
       );
end motion;

architecture behavior of motion is

   begin

   motion:process

   variable x, i : integer;
   variable dmin, dp : integer;
   variable dp0,dp1,dp2,dp3,dp4,dp5,dp6,dp7,
            dp8,dp9,dp10,dp11,dp12,dp13,dp14,
            dp15 : integer;

   begin

     dmin := 1000;
```

```
dp:=0;
x:=0;
while( x  < 256 ) loop
    i := 0;
    while( i < 16 ) loop
        dp0:=current(0)-prev(0);
      dp1:=current(1)-prev(1);
      dp2:=current(2)-prev(2);
      dp3:=current(3)-prev(3);
      dp4:=current(4)-prev(4);
      dp5:=current(5)-prev(5);
      dp6:=current(6)-prev(6);
      dp7:=current(7)-prev(7);
      dp8:=current(8)-prev(8);
      dp9:=current(9)-prev(9);
      dp10:=current(10)-prev(10);
      dp11:=current(11)-prev(11);
      dp12:=current(12)-prev(12);
      dp13:=current(13)-prev(13);
      dp14:=current(14)-prev(14);
      dp15:=current(15)-prev(15);
      dp:=dp0+dp1+dp2+dp3+dp4+dp5+dp6+dp7;
         +dp8+dp9+dp10+dp11+dp12+dp13+dp14+dp15+dp;

      i:= i + 1;

    end loop; -- I
    if dp < dmin then
      dmin := dp;
      xmin <= x;
      ymin <= y;
    end if;
```

```
        x := x + 1;
      end loop; -- x
    end process;
end behavior;
```

Appendix E

The Libraries Used in the Experiments

The characteristics of package types, die types, bus types, and substrate technology types used for GARDEN examples are given here.

Name	Style	Size ($1 \times 10^4 \mu m^2$)	Cost ($)	Pins	Pkgtime (hours)
k1	SCM	5,800	5	40	336
k2	SCM	13,000	20	80	336
k3	SCM	20,000	60	100	336
k4	SCM	30,000	100	156	336
k5	SCM	50,000	150	200	336
k6	SCM	70,000	220	299	336
k7	MCM	15,000	5	40	336
k8	MCM	40,000	20	80	336
k9	MCM	60,000	60	100	336
k10	MCM	90,000	100	156	336
k11	MCM	150,000	150	200	336
k12	MCM	210,000	220	299	336

Table E.1: Package library

Name	Style	FabTime (hours)	Cost ($)	Size (μm^2)
d1	FPGA	1	37.50	10,000
d2	FPGA	1	110.70	17,000
d3	FPGA	1	333.00	34,000
d4	FPGA	1	1422.00	62,000
d5	gate array	336	0.63	1,400
d6	gate array	336	3.70	3,400
d7	gate array	336	17.50	7,400
d8	gate array	336	82.10	16,900
d9	gate array	336	128.00	21,000
ds	standard cell	1680	-	-

Name	AGATESIZE ($\mu m^2/gate$)	Pins	PADSIZE($\mu m^2/gate$)	WB
d1	3.42	80	0.00	1.7
d2	2.89	105	0.00	1.7
d3	2.65	139	0.00	1.7
d4	2.49	199	0.00	1.7
d5	1.42	80	0.00	1.5
d6	0.96	125	0.00	1.5
d7	0.96	184	0.00	1.5
d8	0.85	270	0.00	1.5
d9	0.84	302	0.00	1.5
ds	1	-	6.73	1.2

Table E.2: Die library

Name	Bit width	Clock Cycle (ns)	Type
b0	1	40	on-chip
b1	2	40	on-chip
b2	4	40	on-chip
b3	8	40	on-chip
b4	16	40	on-chip
b5	32	40	on-chip
b6	1	100	on-board
b7	2	100	on-board
b8	4	100	on-board
b9	8	100	on-board
b10	16	100	on-board
b11	32	100	on-board
b12	32	100	on-module

Table E.3: Bus Library

	MCM-D	MCM-C	MCM-L
No. of Layers	5	10	6
Via Grid Space (μm)	75	450	1250
Line btn Vias	1	1	1
Perf. scale	1.00	0.26	0.10
Unit area Cost ($\$/\mu m^2$)	0.009	0.006	0.0009
NRE ($)	5,000	3,000	1,000
Manu. Time (hours)	336	336	336

Table E.4: Substrate Technology Library