HARDWARE-SOFTWARE CO-SYNTHESIS FOR DISTRIBUTED MEMORY ARCHITECTURES

by

Usman Ahmed, B.Eng., 2000,
National University of Sciences and Technology, Pakistan.

A thesis

presented to Ryerson University

in partial fulfillment of the

requirement for the degree of

Master of Applied Science

in the program of

Electrical and Computer Engineering

Toronto, Ontario, Canada, 2005

I hereby declare that I am the sole author of this thesis.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Usman Ahmed

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Usman Ahmed

Ryerson University requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Hardware-Software Co-Synthesis for Distributed Memory Architectures

Usman Ahmed,
Master of Applied Science, 2005,
Electrical and Computer Engineering,
Ryerson University

Abstract

Hardware software co-synthesis problem is related to finding an architecture, subject to certain constraints, for a given set of tasks that are related through data dependencies. The architecture consists of a set of heterogeneous processing elements and a communication structure between these processing elements. In this thesis, a new algorithm for co-synthesis is presented that targets distributed memory architectures. The algorithm consists of four distinct phases namely, processing element selection, pipelined task allocation, scheduling and best topology selection. Selected processing elements are finally mapped to a regular distributed memory architecture comprising of mesh, hypercube or quad-tree topology. The co-synthesis method is demonstrated by applying it to MPEG encoder application and various size large random graphs.

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Gul N. Khan, for his support during the course of this project. I am grateful to Ryerson University and POA Educational Foundation for providing scholarships and to National Science and Engineering Research Council of Canada (NSERC) for providing financial support through a research grant. Support of Canadian Microelectronics Corporation (CMC) for providing the prototyping systems was also very valuable. Finally, I would like to thank Rehan Hameed (Stanford University) and Fahad Ali (Gwangju Institute of Science and Technology) for various discussions on the algorithm and MPEG encoder implementation.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

INTRODUCTION

1.1 Overview

Modern multimedia, DSP and data communication applications are computationally very intensive. Computational requirements prohibit the use of a single processor to provide all the functionality at desired throughput. These performance requirements are further strained by low area (power) constraints and small window for time-to-market. Traditionally these systems were developed as a two stream process, hardware engineers delivering general purpose computer systems which were programmed by software engineers. Optimal performance is achieved when hardware and software are properly 'tuned' to each other. Typical design cycle required the system to be partitioned very early in the design cycle leaving very little room for modifications later in the design stage.

Early system partitioning, along with the separate design flows for hardware and software modules, does not fully explore design space and is prone to high cost, inefficient hardware and software. The solution is to combine hardware and software design efforts by considering the efficiency of both options and to find a design implementation that fulfills all the specification requirements with a minimal cost. Also, with increasing design spaces, selecting a feasible solution from a set of different design options becomes more and more demanding and hence a need to automate this process. The automated

process is usually referred to as hardware software co-synthesis. Main phases in this process include selection of processing elements (ASIC or general purpose CPU), allocation of tasks to these processing elements, creating a communication structure between processing elements and assigning start and finish time to each task (scheduling).

In this thesis, a co-synthesis algorithm has been presented that targets regular distributed memory architectures. Regular architectures have multiple communication paths between processing elements and therefore offer inherent support of fault tolerance. The algorithm has two distinct phases of pipelined processing element allocation and mapping the processing elements to regular distributed memory architectures. The algorithm iteratively selects processing elements and allocates tasks to these processing elements. Pipeline stages are created during the allocation process. Finally the processing elements are arranged in a regular fault tolerant topology.

1.2 Thesis Organization

This thesis is organized into five chapters. This chapter provides an introduction to co-design process and outlines the thesis organization. Chapter 2 surveys hardware software co-design and co-synthesis. This chapter also provides some details on distributed, regular architectures like hypercube, mesh and tree arrangements. Chapter 3 describes all phases of the proposed co-synthesis algorithm. These include algorithms for iterative selection of processing elements, pipelining and task allocation, mapping processing elements network to a topology and selecting the best topology for the given application.

Chapter 4 presents the experimental results. Algorithm is demonstrated by application on large random task graphs and an MPEG encoder application for a range of constraints. Output from each phase is discussed here. Chapter 5 concludes the thesis by stating some directions for future work that can improve the proposed algorithm.

CHAPTER 2

HARDWARE SOFTWARE CODESIGN

2.1 Overview

Performance requirements of most real-time embedded applications make it impossible to execute the entire application in software. To meet the performance constraints, computationally intensive portions of the application are extracted realized in the form of specialized hardware. These systems were developed as a two stream process, hardware engineers deliver the hardware components and software engineers program general purpose or application specific processors for software components. Design methodology for these systems starts with requirement analysis. Based on requirements, application is partitioned into hardware and software components and system architecture is defined. After that, hardware and software components are designed and an interface between hardware and software components is described. These components are then verified for functionality and then integrated together. Finally, the integrated environment is tested and this concludes the design. A typical design methodology for these systems is shown in Figure 2.1 [1].

Optimal performance is achieved when hardware and software are properly 'tuned' to each other. Early partitioning along with the separate design flows for hardware and software does not fully explore design space which may result in an imperfect solution; a design that is prone to high cost and inefficient hardware/software. Also from Figure 2.1,

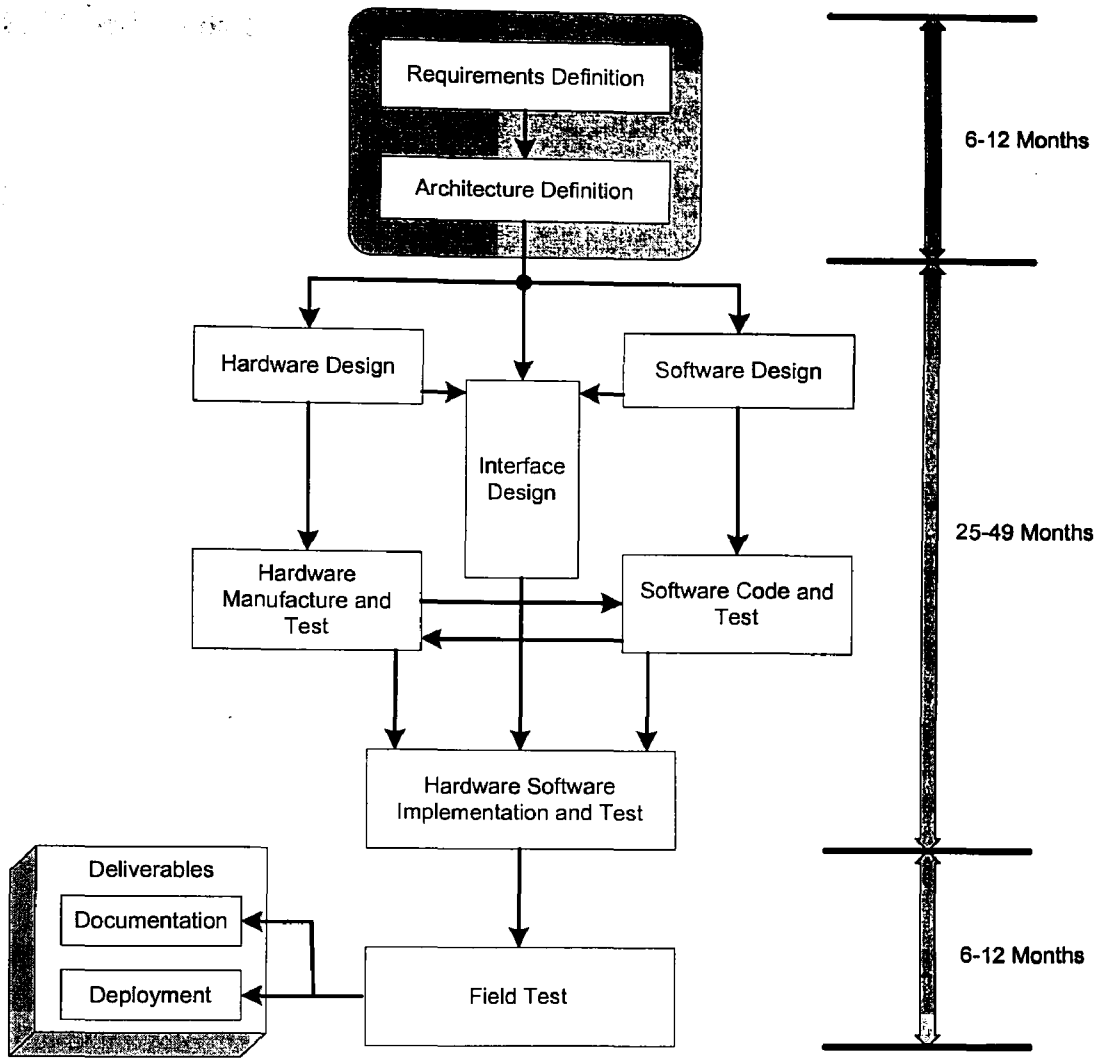

Figure 2.1 Typical System Design Practice

it can be seen that the most time consuming stage is the development of hardware and software components. Any problems (e.g. performance/area, cost issues etc.) identified in the later half of the design which may require a different hardware software partitioning would require a new hardware and software design flow costing 25-49 months worth of effort. With decreasing time-to-market windows, this may prove to be extremely costly.

To effectively address these design challenges, a unified approach that considers the efficiency of both hardware and software options is required. This approach is commonly

known as to as hardware software co-design. Hardware software co-design refers to a mechanism of jointly designing hardware and software components of a system. It is a broad term encompassing methodologies and tools that allow a designer to create hardware-software systems from a single starting point, a set of specifications. Co-design keeps hardware and software within the same 'stream' of work, with the hope of achieving better results through an integrated approach. This concept attempts to join hardware and software design efforts into a combined methodology that improves cycle time and quality while enhancing the exploration of the Hardware/Software design space. The impetus for this effort lies in the following reasons [2];

a) Computing systems deliver increasingly higher performance to end users
b) New architectures based on programmable hardware circuits can accelerate the execution of specific computations or emulate new hardware
c) Recent progress in synthesis and simulation tools has paved the way for integrating CAD environments for co-design of hardware software systems

In co-design process, at the very beginning of system development, constraints and requirements are analyzed to specify the system. The specified system is subject to an automated partitioning algorithm which partitions the system specification into hardware and software blocks. The individual hardware and software blocks are integrated and incorporated along with their interface. The integrated system is then co-simulated and evaluated for timing and resource constraints. The whole process is repeated until a satisfactory hardware software implementation is reached. Once the current

hardware/software partition satisfies all the constraints, it is integrated into a complete system and checked for functional verification.

2.2 Codesign Methodology

Co-design methodology provides techniques for analyzing the performance, area and power of the system and it provides methods for evaluating a large number of feasible design options [3]. In this section, these phases are discussed and some of work in each of these phases is reviewed.

2.2.1 Specification/Requirements Analysis

Codesign process starts with description of both hardware and software components. Describing the system level behavior is a challenging problem as it requires a high level of abstraction while requiring fine details to make it unambiguous. The output of this phase is a functional specification, which lacks any implementation detail. Different schemes have been used to specify the requirements. These include using representation of Communicating Sequential Processes (CSP) [4], VHDL [5], Codesign Finite State Machine (CFSM) [6] and C*/ Hardware C [7, 8 and 9].

Different methods have been explored to specify the system requirements. These schemes range from homogeneous modeling to heterogeneous modeling. In homogenous modeling all the requirements are specified by a common language. The examples given

above relate to homogeneous modeling. Other alternative (heterogeneous modeling) is to use hardware description languages to describe obvious hardware functions and use a software language to describe the other functions. So, when functions move across the partition only a small portion of the specification need to be translated [3]. However, software based languages, like C/C++, bias the implementation in favor of software while on the other hand the hardware description languages, like Verilog, VHDL, etc., favor the hardware implementation. Also, software based representations, e.g., C/C++, lack the mechanism to specify concurrent processes.

A recent step in this direction is the introduction of a new language to model system – SystemC [10]. System C combines the features of hardware description languages and software languages to model both software and hardware components of a system. Many companies like Synopsys, Cadence etc., are building CAD tools to simulate and synthesize a system specified in System C.

2.2.2 Partitioning

a) Graph Structure

Once a satisfactory representation of system specification is obtained, it is subject to a partitioning algorithm. The partitioning algorithm assigns parts of system description to heterogeneous implementation units e.g. ASICs (Hardware), standard or embedded microprocessors (Software), memories and so forth. The aim of partitioning task is to find a design implementation that fulfills all the specification requirements (functionality,

goals and constraints) at a minimum cost. The cost could be area cost, power cost or dollar cost of the resulting system.

Specifications for the system are read into an internal data structure. This internal representation is mostly some form of a graph. Structure of the graph is also very important for the operation of partitioning algorithm. Most commonly used graph structures are dataflow graphs and control flow graphs.

Dataflow graphs join the nodes by their data dependencies. Most digital signal processing (DSP) applications are data-flow dominated which nicely fit this graph structure. Partitioning on these graphs is performed by scheduling the nodes of the graphs to available hardware and software (general purpose processor) resources. Most commonly used scheduling algorithms are based on Hu's scheduling algorithm [11]. These algorithms are list scheduling [12] and force directed scheduling [13]. This form of graph can efficiently extract the parallelism during partitioning. However, the main disadvantage is that it cannot handle conditional branches, e.g., if-else constructs.

Control flow graph on the other hand join the nodes by their control dependencies. These graphs suit control applications which have a large number of 'if-else' constructs. Most commonly used scheme is path based scheduling algorithm [14]. This form of graph has the obvious advantage of handling the control dependencies. However, its complexity is dependent on the number of paths in the graph and it also cannot extract parallelism efficiently.

An interesting approach has been presented by Bergamaschi et al. in [15]. They combine control-flow and data-flow approaches in an adaptive scheduling algorithm. The algorithm operates on a control-flow graph and integrates the data-flow techniques into a path-based scheduling algorithm.

b) Granularity

The partitioning algorithms work by mapping the system components to heterogeneous resources (hardware or software). The size of components moved to hardware/software defines the granularity of the partitioning algorithm. Coarse grain approaches assign complete functions or processes to hardware or software resources. Nodes of the input graph in this case represent a large block of system functionality. This high level representation prevents excessive data communication across the whole application. Common examples for coarse level granularity are MPEG decode, Fast Fourier Transform (FFT), Discrete Cosine Transform (DCT) etc. Partitioning performed with at coarse granularity is also known as high level granularity.

Fine grain approaches, on the other hand deal with small operations. These operations may be either a single instruction or a bunch of instructions. Fine grain approaches usually result in a large amount of data transfers across the application. A commonly used method is to group all instructions which do not have any conditional expression among them. This group is known as basic block. Fine grain approaches are also known as low level approaches.

Granularity level directly affects the partitioning process. Optimization potential in partitioning improves from coarse grain to fine grain approaches. Data communication overhead and complexity also increases as granularity varies from high level to low level [16].

Both coarse grain and fine grain approaches have their disadvantages. For most applications, the computationally intensive parts are small loops which are hidden inside a function or process. For these cases coarse grain approaches provide costly designs by moving many redundant parts to hardware. On the other hand, fine grain approaches suffer from the fact that their space for feasible designs is so large that it is usually hard to find the global optimum. Henkel and Ernst have proposed an interesting approach which uses flexible granularity. Basic blocks are clustered together in partitioning objects which range from a single block to an entire function [17].

c) Optimization Techniques

Optimal Approaches

At the heart of partitioning process, there operates an optimization algorithm. Goal of the optimization algorithm is to select an optimum hardware software partition. Problem can be setup in different ways, e.g., to maximize the performance, to minimize the cost (area, power etc.) or to maximize the performance with minimum area. Simple hardware software partitioning is known to be an NP (Non-polynomial) complete problem [18]. Different optimization algorithms have been used to partition the system in hardware and software. A simple approach is to try out all possible solutions and then select the best

option. This approach, though gives the best result, is not feasible because design space is very large even for simple problems.

Integer linear programming (ILP) is another technique that can be used to solve partitioning problem [19]. Linear program is a formulation in which a set of linear equations define the objective function and associated constraints. If the desired solution can only have integer values, the formulation is known as ILP. Simplex technique is a well known method to solve such linear equations. Partitioning problem can be expressed as a set of linear objective function subject to linear constraints. Prakash and Parker have provided such a formulation for hardware software partitioning [20]. Although this approach provides optimal result but the computation time makes it infeasible for large problems.

Branch-and-bound algorithm is another procedure to solve optimization problems. It is a very general algorithm that can even be used to solve ILP formulations. Feasible solutions are arranged in form of a decision tree where leaves of the tree represent all possible options. Algorithm makes a decision at node (branch) based on some criteria and computes a lower bound for all solutions in the corresponding sub-tree. If the bound has higher cost than any of the previously found solutions, the sub-tree is excluded from further search [12]. Hafer and Hutchings have used branch and bound approach to solve ILP problem [21]. Vemuri and Chatha have used a branch and bound algorithm for hardware software partitioning [22]. Branch and bound algorithms can considerably

reduce the search space however the worst case complexity of this algorithm still remains exponential.

Another interesting optimization approach is Dynamic programming. Dynamic programming decomposes optimization problem into a sequence of stages such that the optimal solution to the original problem must contain optimal solutions to each of these stages. Algorithm operates at each stage, makes a decision and moves to the next stage. Decisions made at a stage do not depend on the previous decisions [13]. Shrivastava et al. used dynamic programming to solve partitioning problem [23]. Chang and Pedram applied dynamic programming to solve coarse grained partitioning problem [24], while Knudsen and Madsen used dynamic programming for performing hardware software partitioning at fine granularity [25]. Use of dynamic programming can be efficient however efficiency mainly depends on how the problem has been divided into stages and complexity to reach an optimal solution at a given sub-problem.

Heuristic Approaches

All the methods mentioned above provided an optimal solution. However, because of the exact nature, all the algorithms are computationally very expensive. These algorithms can only be applied to small problems (e.g, a task graph with a small number of nodes). For larger problems, heuristics are often employed as they provide a 'good quality' solution in a reasonable amount of time. Heuristic techniques only perform a limited search on the feasible design space and they cannot guarantee optimality of the solution.

Greedy algorithms are commonly used as heuristics. In greedy algorithm, the decision is made at each step without taking into account any previous or later decisions. Greedy algorithms are usually 'up-hill' or 'down-hill' techniques and therefore, are liable to stuck in local minima or local maxima. Optimality can be guaranteed if problem exhibits certain conditions [12], but most real life problems do not follow these conditions and thus greedy approaches are only used as a heuristic. Ernst et at., used a greedy approach by starting from all software solution and moved parts to hardware [8]. Gupta and Micheli used a complementary greedy approach by starting with an all hardware solution and moving parts to software at each step [9].

Simulated annealing is another popular heuristic optimization technique [26]. Simulated annealing based on annealing which is a process to obtain low energy states of a solid material in a heat bath. The temperature of the heat bath is increased to a maximum, which melts the solid. The temperature is then slowly decreased according to a given cooling schedule until a low energy state of the solid (a perfect crystal) is reached. The algorithm is controlled by a temperature parameter, which begins at a high value and decreases as the system "cools" and stabilizes. A cost function is required to evaluate the system for each state. Initially, during high temperature, states which increase system quality are always accepted, and states which decrease design quality are accepted randomly. As the temperature approaches zero, only the states which decrease system cost are accepted. The conditional acceptance is based on the probability P, which is given as: $P(\Delta E) = e^{-\Delta E / T}$ where ΔE represents the change in cost of the overall system. Unlike greedy heuristics, simulated annealing often accepts changes which decrease the

quality of a design, in hope of achieving a better final design. Henkel and Ernst have used simulated annealing in hardware software partitioning [17].

Tabu search is another heuristic optimization technique to allow local search methods to overcome a local optimum [27]. Tabu search uses a form of short-term memory used to keep a search from becoming trapped in local minima. A Tabu list is formed that contains the moves of the recent past but they are forbidden for a certain number of iterations. During the optimization process, solutions are checked against the Tabu list. A solution that is on the list will not be chosen for the next iteration. Tabu list forms the core of the search and keeps the process from cycling in one neighborhood of the solution space. Sometimes it may be useful to overrule the Tabu condition by what is known as 'aspiration criteria'. A commonly used aspiration criterion is to accept a system state if it results in an overall low cost. A technique to expand the search of design space is called the 'diversification strategy'. A simple diversification strategy is to restart the search from the initial system state after a specified number of iterations. Eles et al. applied Tabu search on hardware software partitioning and compares the results with a simulated annealing based approach [28].

Genetic Algorithm is also a heuristic technique that can be used to solve optimization problems. Algorithm starts from an initial population and selects a set of 'parents' based on a 'fitness function'. These parents are then used to breed a next generation by performing 'crossovers' and 'mutation'. Main idea is that as the algorithm progresses, stronger and fitter chromosomes will survive and their next generation will also have a

high probability of being fitter. Dick and Jha used genetic algorithm in performing hardware software partitioning [29]. Wiangtong et al., provide an interesting comparison between simulated annealing, tabu search and genetic algorithms, when applied to partitioning [30].

2.2.3 Estimation

During the partitioning process, it is often required to estimate the performance of a task on a specific processing element (software or hardware) and the cost associated with the processing element (area, power etc.). This is needed to determine the quality of a particular hardware software partition.

Estimating the hardware performance required to determine the maximum clock frequency of the hardware block. At the same time this has to be done in quick time so that the partitioning algorithm can analyze a large number of possibilities. Solution is to use high level synthesis techniques to determine the clock frequency for the longest path through the logic. Henkel and Ernst have developed a path scheduling based estimation technique that can estimate hardware performance very efficiently [31, 32]. Vahid and Gajski have also proposed an estimation technique [33]. Their technique used incremental hardware cost estimation by updating a previous estimate rather than re-estimating for a new partition.

Software performance estimation problem is also similar to hardware 'worst-case execution time' problem. Earlier approaches used instruction set simulators to estimate the software time [7], which is similar to the use of synthesis techniques to estimate hardware performance. Park and Shaw proposed one of the earliest software estimation algorithms using path enumeration [34]. Li et al., developed another efficient software estimation algorithm which also considered the effect of instruction caches [35]. Ye et al. also developed an algorithm for software performance estimation [36]. Finally, on system performance estimation, Yen and Wolf developed an algorithm for estimating the performance of a set of tasks on a multiprocessor system [50]. Each processor in the system used a rate monotonic scheduler to schedule the tasks.

2.2.4 Co-simulation / Co-verification

After successful partitioning, system needs to be verified against specifications for completeness and functional correctness. A common verification strategy is to simulate the final system. Co-simulation refers to an integrated simulation of hardware and software components. Challenges in hardware-software co-simulation include 'speed', 'accuracy' and 'interactive debugging'. Speed is required to simulate a reasonable number of test sequences. Accuracy refers to generating the same simulation outputs in as the implementation would result. Interactive debugging is the ability to step through system execution, examining intermediate values, and backtracking to debug the system.

Separate verification strategies for hardware and software exist but these are quite different. For hardware software co-design where the final system contains heterogeneous components (hardware and software) simulation is a problem. One approach is to simulate hardware and software separately. This technique can be efficient but it leads to problems with synchronizing the results. Another co-simulation approach could be to use register transfer level (RTL) or gate level hardware models of the processor to simulate software execution in a hardware verification environment. However, these simulations are too slow to simulate software in a reasonable amount of time. A traditional approach to co-simulate hardware and software is to use Verilog simulator. Verilog's Programming Language Interface (PLI) was used to co-simulate hardware and software components. Hardware component is described using Verilog and software interacts with the hardware using Unix's sockets. This is illustrated in the following figure.

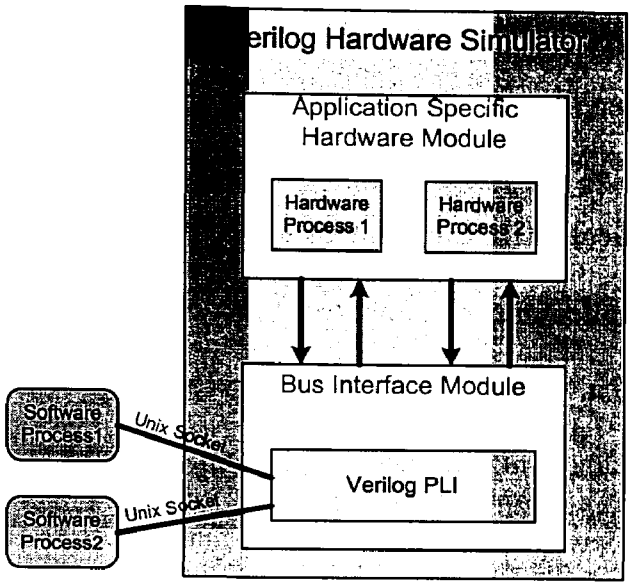

Figure 2.2 Verilog based Co-simulation

The Ptolemy environment [37], developed at Berkeley, is also a key research tool in this area. It focuses on system-level modeling and simulation and provides a high-level support for a variety of applications. Cortes et al., have proposed another co-verification methodology for embedded systems using a Petri-net based representation [38]. They used symbolic model checking to prove the correctness of the system. Ghosh et al., provided a set of techniques to speed up simulation of processors and peripherals without significant timing accuracy loss [39]. They developed a co-simulator that can be used for joint debugging of hardware and software. Finally, Hsiung proposes a formal verification approach using linear hybrid automata [40]. His approach simplifies the state-space explosion that occurs in formal verification of complex systems.

2.3 *Hardware Software Co-synthesis*

In hardware design, synthesis refers to construction of a structure of digital circuit starting from a specification in the form of some data-flow graph [12]. This structure basically represents collection of some resources (e.g., adder, multiplier, ALU etc.), interconnection between these resources and corresponding control logic. On the other hand, in software domain, synthesis refers to the conversion of system specifications into a group of basic instructions (software program) that can be executed on some general purpose processor. This software synthesis is mostly done by a compiler which converts programming language based specification into hardware specific instructions.

In the domain of hardware software co-design, where the resulting system consists of both hardware and software components, co-synthesis process is the conversion of system specification (along with a set of technology parameters) into hardware architecture and corresponding software architecture. System specification identifies both functional and non functional requirements and is specified in the form of a task graph. Hardware architecture consists of processing elements (PE's) and communication channels connecting these PE's, which can either be a general purpose processor or an application specific hardware (ASIC). Software architecture defines the allocation of tasks on PE's, scheduling of tasks and scheduling of communication channels [41].

Hardware software co-synthesis is tightly coupled with hardware software partitioning, which was described earlier. Partitioning process is a subset of co-synthesis. Co-synthesis process selects the processing elements, maps tasks onto them, arrange them in a specific

architecture and identify the schedule for each task. Partitioning, on the other hand is merely the process of mapping tasks to hardware or software components. However, this difference is subtle and partitioning and co-synthesis terms are also used interchangeably in literature.

2.3.1 Co-synthesis Phases

Co-synthesis process involves selecting processing engines, allocating tasks to processing engines and scheduling tasks on processing engines. For increased throughput requirements, co-synthesized system may also need to be pipelined. However, it must be noted that these phases are not always cleanly separated and some of these phases may be merged together in a co-synthesis environment. In the following sections, each of these phases is described.

a) Resource selection

Co-synthesis environments are mostly library based, where a number of different processing elements are available to be added in to the system. Processing elements are the implementation units which can be either an application specific hardware or a programmable processor. Performance of each task on the available processing element is also known. Resource selection refers to the selection of processing elements that when added to the system improves the performance without violating any non functional parameters (area, power etc.). Techniques for estimation earlier can be used to estimate

the performance and cost for adding a specific processing element. Resource selection involves the number as well as type of processing elements added into the system.

b) Task Allocation

Next phase in co-synthesis process is to allocate the tasks to the processing elements that have been added into the system. This phase uses the techniques described earlier in the partitioning phase. Tasks are allocated to the processing elements such that overall performance of the system is maximized. Another important parameter to be considered is the inter-task communication (also known as inter-process communication; IPC). Data communication between two tasks can be of the same magnitude as their execution times. Allocating two tasks to two different processing elements based on individual performance gain can actually reduce the system execution time if the tasks have high data communication among them. Task allocation is known to be NP-complete and is therefore a computationally hard problem [18]. Heuristics are often employed to perform task allocation.

c) Scheduling

Scheduling is the process of assigning start times to each task after they have been allocated to some processing engine. Processing elements may have more than one task allocated onto them and the order in which various tasks execute on a processing element has a direct influence on the overall system performance. Tasks cannot be executed unless the required data is available. This data is available only after the execution of

parent tasks. If tasks allocated to a processing element have a data dependency, there is a fixed order of execution. However, if there is no such data dependency, the tasks can execute in any order.

Many scheduling algorithms exist to schedule tasks in a multiprocessor environment. Scheduling is also a NP complete problem so heuristics are usually used to perform scheduling. A common heuristic is a list scheduling [12] algorithm which is a variation of Hu's optimal algorithm [11]. In list scheduling, a priority is assigned to each task. Tasks which are on the critical path (from performance point of view) have higher priority than the other tasks. A common priority measure is the sum of execution time (maximum, minimum or median) of each task along its longest path to final task. Tasks are scheduled in the order of decreasing priority such the dependent tasks are scheduled after their parents.

Multiprocessor task scheduling is a well researched topic. Many algorithms for scheduling have been proposed for multiprocessor environments [42]. Scheduling algorithms for distributed architectures have also been proposed. Sih and Lee present an interesting algorithm, known as generalized dynamic level (GDL) scheduling [43]. In this algorithm, the priority assigned to each task is dynamic and changes as the algorithm proceeds. Priority depends on a number of factors including the static priority (sum of execution time along its longest path to final task), inter-task data communication and descendant consideration.

d) Pipelining

Sequential implementations can constrain the throughput of the system. Concurrent execution of different tasks can drastically increase the performance of the system. Pipelining is a useful technique to execute different tasks in parallel when the tasks execute in a loop fashion for a long time. Tasks executing concurrently operate on delayed data; data produced by the parent task in the previous iteration. Pipelining is also known as retiming transformation.

To illustrate the effect of pipelining, consider an example task graph shown in Figure 2.3. Task graph consists of four tasks and three different processing elements are available for task allocation. Execution time of each task is also shown. Ignoring the data communication time between tasks, a possible task allocation with corresponding sequential system execution is shown in Figure 2.4. System executes the tasks sequentially in 35 time units. Next, a pipelined execution with same task allocation is shown in Figure 2.5. Dotted lines indicate pipeline stages. Tasks A and C execute in first pipeline state, task B in second and task D in third pipeline stage. With pipelined execution, tasks now execute in 15 time units.

In software (VLIW compilers) and hardware synthesis pipelining is a well researched problem however, use of pipelining in hardware software co-synthesis is relatively recent [22], [44]. Like multiprocessor scheduling, pipelining under resource constraints is also NP complete, which leads to the use of heuristic techniques [18, 22].

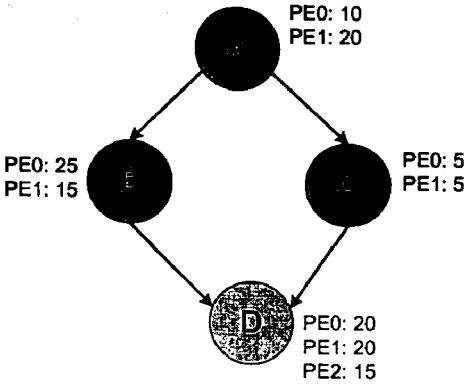

Figure 2.3 Example Task Graph with Possible Mapping

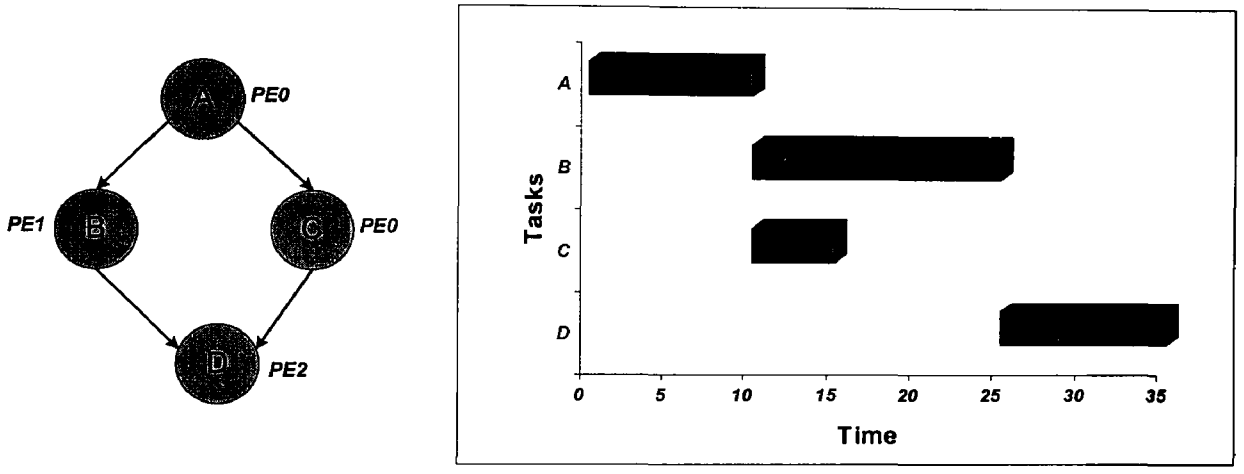

Figure 2.4 Task Allocation and Sequential Execution

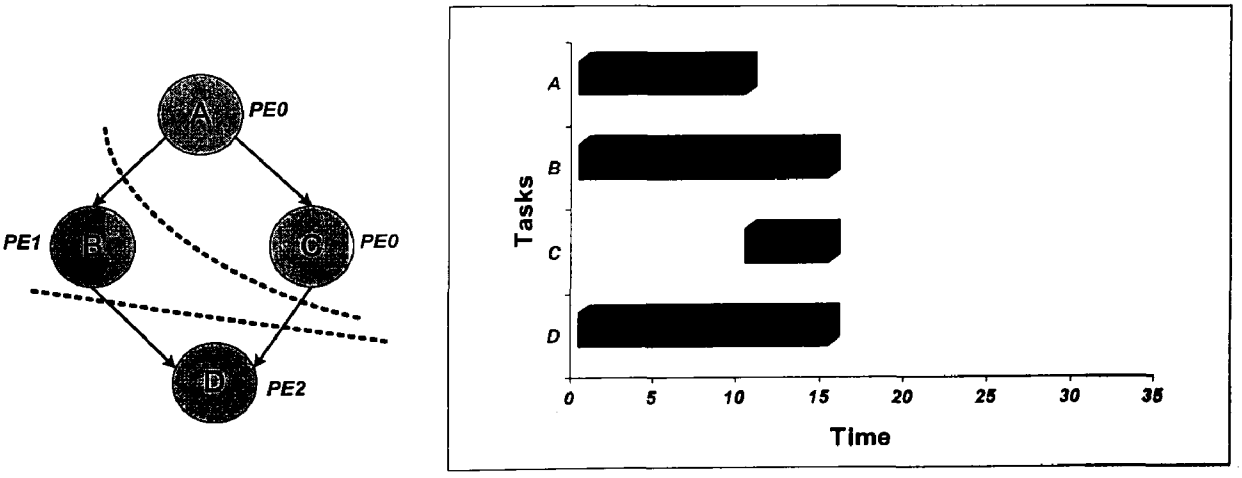

Figure 2.5 Task Allocation and Pipelined Execution

2.3.2 Target Architecture

Processing elements together with interconnection describe the architecture of the system. This architecture can range from simple shared memory based design to complex distributed memory schemes. Most simple case is a uni-processor approach, where most of the application runs on a single processor and computationally intensive parts are executed on custom hardware which communicates with the processor through a shared memory. In this case, co-synthesis reduces to a simple hardware software partitioning problem. For more complicated approaches, a distributed memory scheme is used as shared memory can become a bottleneck when a large number of processing elements communicate through single memory. Distributed memory schemes connect each processing element with a limited number of other processing elements in a regular or irregular fashion. With these schemes, data communication between different tasks becomes important as data goes through a number of stages in order to reach the destination, adding extra overhead. Finally, the target architecture of the system may not be pre-defined and is shaped during the co-synthesis process, which complicates the task allocation and scheduling processes even further.

2.3.3 Co-synthesis Approaches

Co-synthesis can be performed using optimal or heuristic based algorithms. Optimal algorithms include exhaustive search, integer linear programming approach or branch and bound algorithms. Design space for co-synthesis is much larger than simple hardware software partitioning, therefore optimal approaches are very restricted in application.

Heuristic approaches search only a limited design space can therefore not guarantee optimal results. However, most heuristic approaches provide good results in a reasonably small time. Heuristic approaches are divided into two main categories; Iterative approaches and constructive approaches. Iterative approaches start with an initial solution which is a high-cost (high area, high power) system with usually the maximum possible processing elements. This solution is subsequently improved at every iteration. As the algorithm proceeds solution is refined until is satisfy the cost constraint. Constructive algorithms on the other hand are characterized by building solution step by step. A working system is not available unless the algorithm has finished its execution.

2.3.4 Significant Co-synthesis Environments

In this section, an overview of some of the popular co-synthesis environments is presented. Pioneering work in co-synthesis was performed by Prakash and Parker [20]. They developed an algorithm for synthesis of application specific heterogeneous multiprocessor systems known as SOS. The algorithm described a formal mechanism to synthesize a heterogeneous multiprocessor system by creating a mathematical model for the constraints and objective. The problem is set up as a Mixed Integer Linear Program (MILP). Equations for constraints and objective function are developed and then linearized. Linear equations were then solved through simplex technique using Bozo program [21]. This algorithm took the input in the form of a data flow graph and synthesized an arbitrary multiprocessor topology. Algorithm produces optimal results, however, because of MILP, it is slow and is limited to only small applications.

VULCAN is an earlier co-synthesis environment developed by Gupta and DeMicheli [9]. The input for the co-synthesis process is specified in a C like language known as HardwareC. HardwareC has some modifications with simple C syntax to model hardware unambiguously. The specifications are translated into a system graph model which is control flow graph at fine granularity. A simple architecture is used which consists of a single software processor and multiple hardware blocks. Hardware software communication takes place through shared memory and to simplify computational model, hardware and software portions execute in mutual exclusion. Partitioning is performed using an iterative algorithm. Initially all the tasks are put to hardware. This mechanism is used to check if a solution exists for the given constraints. Then, tasks are subsequently moved to software to reduce hardware cost.

Another early co-synthesis environment is COSYMA, developed by Henkel and Ernst [7]. Input was specified using a superset of C language called C*. Specifications are translated to a control flow graph at a fine granularity level of base block. As VULCAN, simple target architecture is assumed which consists of a single processor and a hardware block. All the hardware modules are implemented in a single hardware block. Data communication takes place through shared memory. Earlier version of the algorithm operated at fine granularity level and used simulated annealing to partition the graph into hardware and software components. Later version introduced the idea of flexible granularity [17]. Partitioning is performed through a dynamically weighted, multidimensional objective function which takes into account area cost as well as timing

constraints. Software timing is estimated through profiling and hardware estimates are obtained through a path based estimation algorithm. This environment also assumed hardware and software block operating in mutual exclusion.

Kalavade and Lee developed a constructive algorithm for Ptolemy environment [37, 45]. Task execution time and system cost can be used as two objective functions. Their algorithm introduced the idea of Global Criticality Local Phase (GCLP). Global criticality measure identifies whether area is critical or time is critical. Local phase is used to classify tasks into extreme task, repeller task or a normal task. Algorithm works on coarse granularity. It initially concentrates on solving two-way partitioning problem and is then extended to solve multi-way partitioning problem where different an implementation bin also needs to be selected. Different implementation bins correspond to having more that two (hardware or software) possible mappings for a task. The environment concentrates on real time applications implemented by means of DSP-based architectures.

Wolf presented an architectural co-synthesis algorithm for distributed systems [41]. In this algorithm, a heterogeneous multiprocessor system is constructed iteratively. Algorithm works on a data-flow graph specified at a coarse granularity level. Initially all the tasks are out to the fastest processing element. Tasks are then re-allocated based on processing element utilization to minimize cost. After that, tasks are again re-allocated to minimize communication between various processing elements. Finally based on data communication between various processing elements, an irregular topology is created.

Tasks are scheduled by finding the longest path through the task-graph. Since task allocation is known, they are easily scheduled by forcing an order of execution. Because of heuristic nature algorithm is very fast, however it does not pipeline the execution.

Vemuri and Chatha presented a co-synthesis algorithm which supports pipelining [22]. Algorithm works on a data-flow graph, specified at a coarse granularity. Algorithm targets simple shared bus architecture with a single processor and multiple hardware blocks. Execution times of hardware and software implementation are assumed to be known in advance. Tasks are moved to hardware or software implementation using a branch and bound algorithm. Initially a hardware software partition is created which is followed by pipelined scheduling technique called RECOD (REtiming heuristic for optimal resource utilization with least shared memory utilization for hw/sw CODesigns). RECOD algorithm used heuristic techniques to create pipeline stage and utilized simulated annealing to minimize additional pipeline memory. If pipeline scheduling fails, branch and bound partitioner is invoked to get a new partition and process is repeated. Algorithm produced optimal results but is limited to task-graphs with 30 tasks.

Bakshi and Gajski also presented a co-synthesis algorithm with pipelining [44]. This algorithm also operates on a data-flow graph at coarse granularity. This algorithm tries to minimize the cost of hardware and it can perform pipelining even at task, loop and operation level. Tasks are executed in hardware only if a software implementation cannot meet timing constraints. Number of software processors is not limited and processors are selected by an exhaustive search. Pipeline stages are inserted when a task cannot be

executed in the current pipeline stage. This algorithm targets simple shared bus architecture and it does not take into account the data communication times between various tasks. Also, it does not consider the hardware cost of software processors.

A constructive co-synthesis algorithm for distributed hypercube architectures has been presented by Levman [46]. This algorithm operates on coarse grained data-flow graphs. Tasks are initially clustered into groups and assertion tasks are added for fault detection. Algorithm then iteratively adds processing elements into system taking into account performance improvement versus area cost. Task graph is pipelined a maximum number of pipelining stages by repeatedly using RECOD algorithm [22]. Tasks are allocated on processing elements to balance utilization. Processing elements are then arranged in a hypercube topology and tasks are then scheduled on each processing element. Scheduling is performed by assigning high priority to tasks that are on critical path. Tasks are re-allocated iteratively during scheduling in order to reduce excessive data communication between various processing elements.

2.4 Fault Tolerance

Fault tolerance is a process through which a system continues to work in the presence of faults. Major steps in tolerating a fault include fault detection and fault recovery. Fault detection is the process of identifying a fault. Faults can be identified by using assertions or by duplicating a task (usually by implementing it in an alternative way) and comparing its output with the original output. Assertion is the method to verify the correctness of the

output without regenerating it. Common examples of assertions are CRC check, parity check etc. Assertion requires less overhead compared to duplication where all the functionality of the task is implemented, however assertions cannot always be used. After fault has been detected, fault recovery is then performed by re-executing the faulty task again on some spare hardware.

Co-synthesis of fault tolerant systems is performed by inserting extra tasks in original task graph. These tasks are used as assertion tasks or duplicate-and-compare tasks to detect fault in the system. Following figure illustrates these assertion and duplicate-and-compare tasks. Assertion tasks have been added for tasks A and B (A_{as} and B_{as}) whereas duplicate-and-compare tasks have been added for tasks C and D (C_{du}, C_{cm} and D_{du}, D_{cm}). Dotted lines and shaded blocks indicate fault detection overhead.

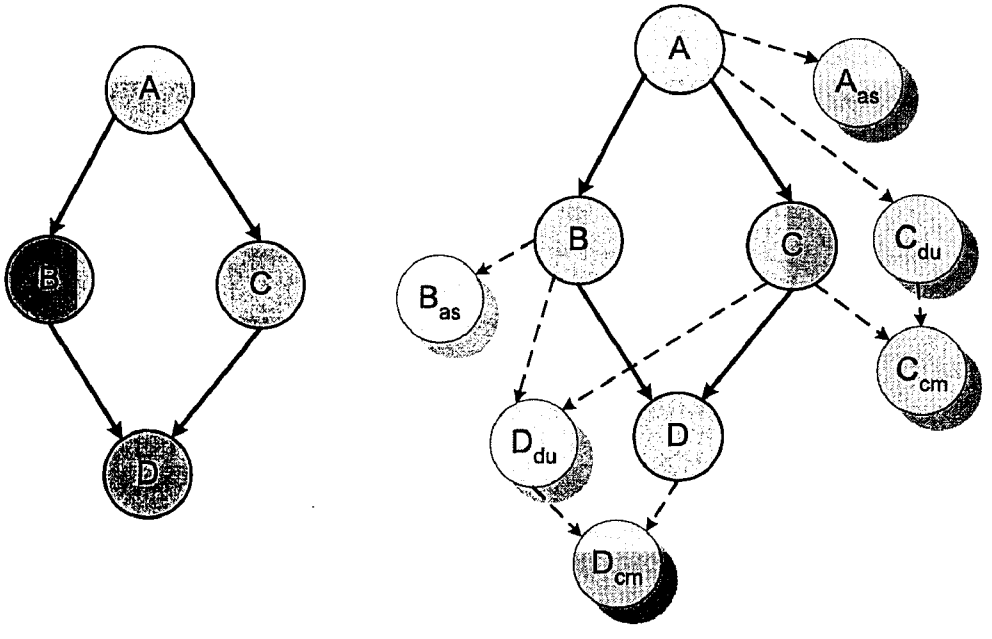

Figure 2.6 Original Task-graph and Task-graph with Fault Detection Tasks

Yajnik et al., proposed a fault tolerant co-synthesis algorithms called Task Based Fault Tolerance (TBFT) [47]. In this algorithm a fault detection task is added for each task. Duplicate-and-compare tasks are only added if an assertion task does not exist. Figure 2.6 is an example of TBFT. Dave and Jha extended TBFT and developed a Cluster Based Fault Tolerance (CBFT) algorithm [48]. CBFT uses concept of 'error transparency'. Error transparent tasks are clustered together and a single fault detection task is added for the cluster which reduces the overhead for fault detection. Group Based Fault Tolerance (GBFT) algorithm is similar to CBFT but uses a bottom-up approach and different heuristics to merge tasks in a group [46]. Its overhead is shown to be even lower than CBFT.

2.5 Distributed Memory Architectures

Shared memory architectures, where various processing elements are connected through a shared bus and communicate to each other through a shared memory, are simple and efficient if the number of processing elements is small. As number of processing elements increases, shared bus tends to become the bottleneck of the system by slowing down the communication between different processors. To overcome this problem, distributed memory architectures are employed in embedded systems. Distributed memory architecture is a scheme that connects a processing element along with its local memory to a processor-to-processor interconnection network [49]. Each processing element in this architecture has its own local memory which is not shared with any other processor. Processing elements in these architectures communicate with each other by

sending a copy of the data to other processors through the interconnection network. These architectures offer high scalability and can easily satisfy the performance requirements of modern day applications where computation on local data takes most of the time compared to data transfers across different processing elements.

Two terms associated with distributed architectures are degree and diameter. Degree of a node is the maximum number of communication links connected with a processing element. Diameter is the worst case measure of the number of communication links traversed (also known as number of hops) in transferring data between two processing elements. Regular and irregular architectures are two major classes of distributed memory architectures based on the regularity of the communication links. An architecture is regular if all nodes have same number of communication links. Most regular architectures offer inherent fault tolerance capabilities to communication link failure as multiple paths to each processing element exist.

Different distributed memory architectures have been proposed to support scalability and efficient parallel processing. These architectures differ by inter-processor communication patterns. Some common topologies of distributed memory architectures are described below.

2.5.1 Mesh Topology

Mesh topology has $n = N^2$ nodes and all nodes except the boundary nodes are connected to four immediate neighbors. Degree of this topology is 4 and its diameter is 2*(N-1). Following figure illustrates a mesh topology with N=3.

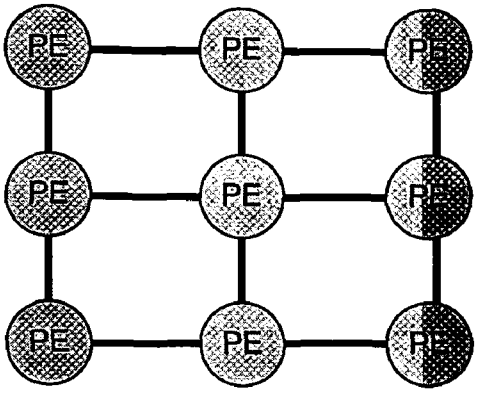

Figure 2.7 Processing Elements Arranged in Mesh Topology with N=3

2.5.2 Hypercube Topology

Hypercube topology has $n = 2^N$ nodes and all nodes are connected to N other nodes. Degree of this topology is N and its diameter is N ($\log_2 n$). Following figure illustrates a hypercube topology with N=4.

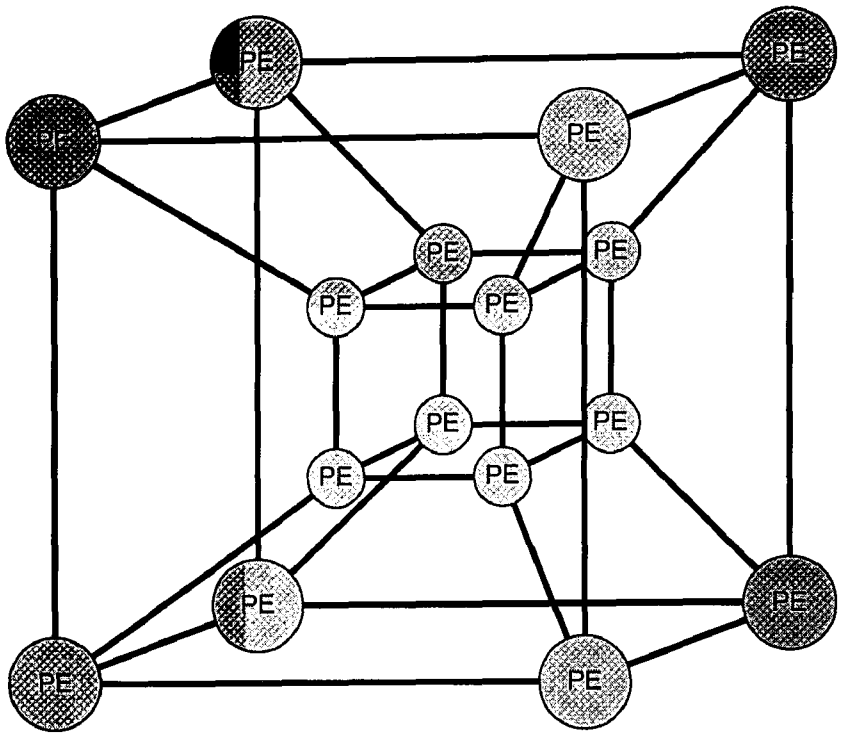

Figure 2.8 Processing Elements Arranged in Hypercube Topology with N=4

2.5.3 Tree Topology

In tree topology, processing elements are hierarchically arranged with each node at a given level is connected to 2 or more nodes and the lower lever. If 'k' denotes number of levels and 'D' is the degree of each node, then a balanced tree topology has $n = \frac{D^k - 1}{D - 1}$ nodes. Diameter of tree topology is 2*(k-1). Trees with D=2 and D=4 are known as binary and quad tree respectively. Unlike mesh and hypercube topologies, in tree topology there is only one path to each node and therefore cannot support fault tolerance. To overcome this limitation, nodes at each level are connected by extra links [49]. Trees with these extra links are also known as X-Trees [50]. Following figure illustrates a binary and quad-tree topology with k=3. Dotted lines indicate extra links.

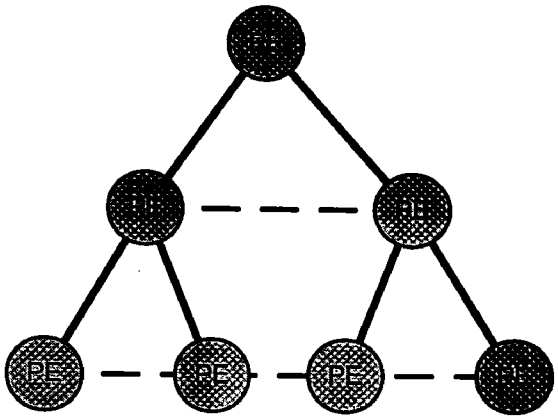

Figure 2.9 Binary Tree with 3 Levels

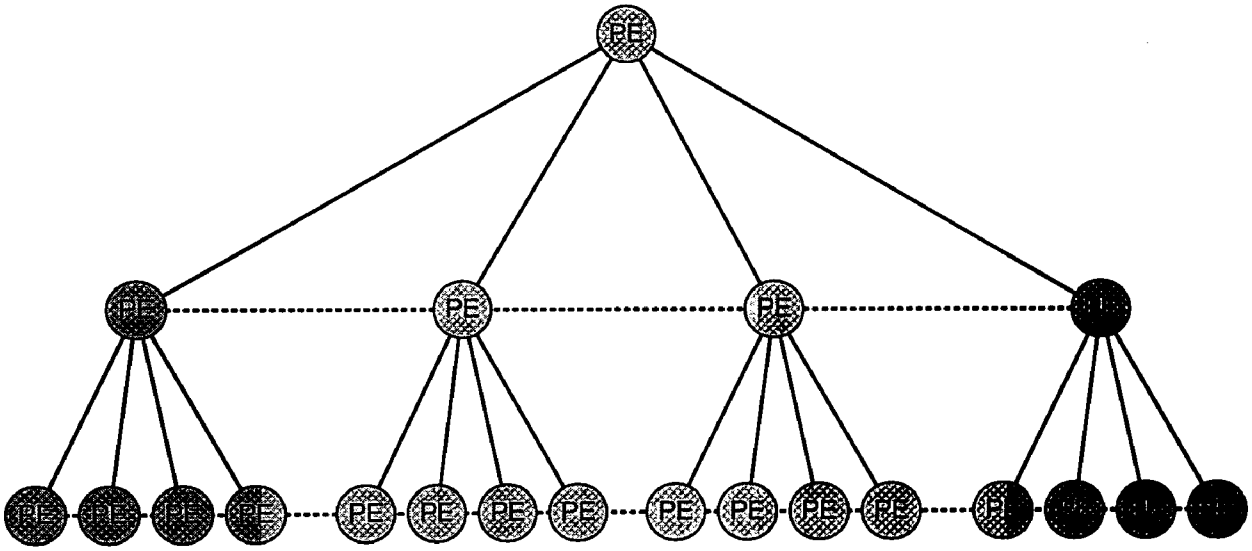

Figure 2.10 Quad Tree with 3 Levels

CHAPTER 3

HARDWARE SOFTWARE COSYNTHESIS FOR DISTRIBUTED MEMORY SYSTEMS

3.1 Introduction

In this chapter, a new hardware software co-synthesis algorithm is presented which targets regular distributed memory architectures [53]. Application is specified in the form of an acyclic task graph using data flow representation. A library of processing elements is also required. These processing elements can either be general purpose processors (software processors) or application specific hardware blocks. Algorithm also assumes that the profiling data is available which contains execution times of each task on the available processing elements. Communication links, which connect different processing elements, are all of the same width. This is required for fault tolerant capability of regular architectures so that data can be transferred through a different path in the presence of a fault. Local memory interfaces to the processing elements can have different data bus widths depending on memory bandwidth requirements of the application. Data transfers between different processing elements takes place through message passing schemes. Every processing element is assumed to have a dedicated communication interface which can communicate data while a task is executed.

Figure 3.1 provides an overview of the proposed co-synthesis algorithm. Major phases of this algorithm include selecting processing elements, allocating tasks and creating

pipeline stages, selecting a topology for the processing elements, task scheduling and reducing the pipeline period. An overview of each of these phases is provided here.

Initial phase of the algorithm is selection of processing elements iteratively followed by pipelined task allocation. Selected processing element can either be a general purpose processor or an application specific hardware block. Performance gain and additional hardware area costs are estimated for each processing element. Processing elements which gives maximum performance gain with minimum area is added to the system. After addition of each processing elements, tasks are allocated to the processing elements present in the system. Pipeline stages are also created during task allocation. This phase terminates when all the tasks are scheduled. Otherwise it continues to add more processing elements until the area constraint is violated. A dirty-list of processing elements is maintained to remove an inefficient processing element after the maximum number of processing elements gets added into the system.

After a successful task allocation, processing elements are mapped onto a regular distributed memory architecture. Topology of the processing elements is not predetermined and the algorithm selects the best topology from some well known topologies like mesh, hypercube and quad-tree topologies. Mapping processing elements to a regular topology can add delays in communicating data from one processing element to another. The topology with minimum overhead in inter-PE communication is selected.

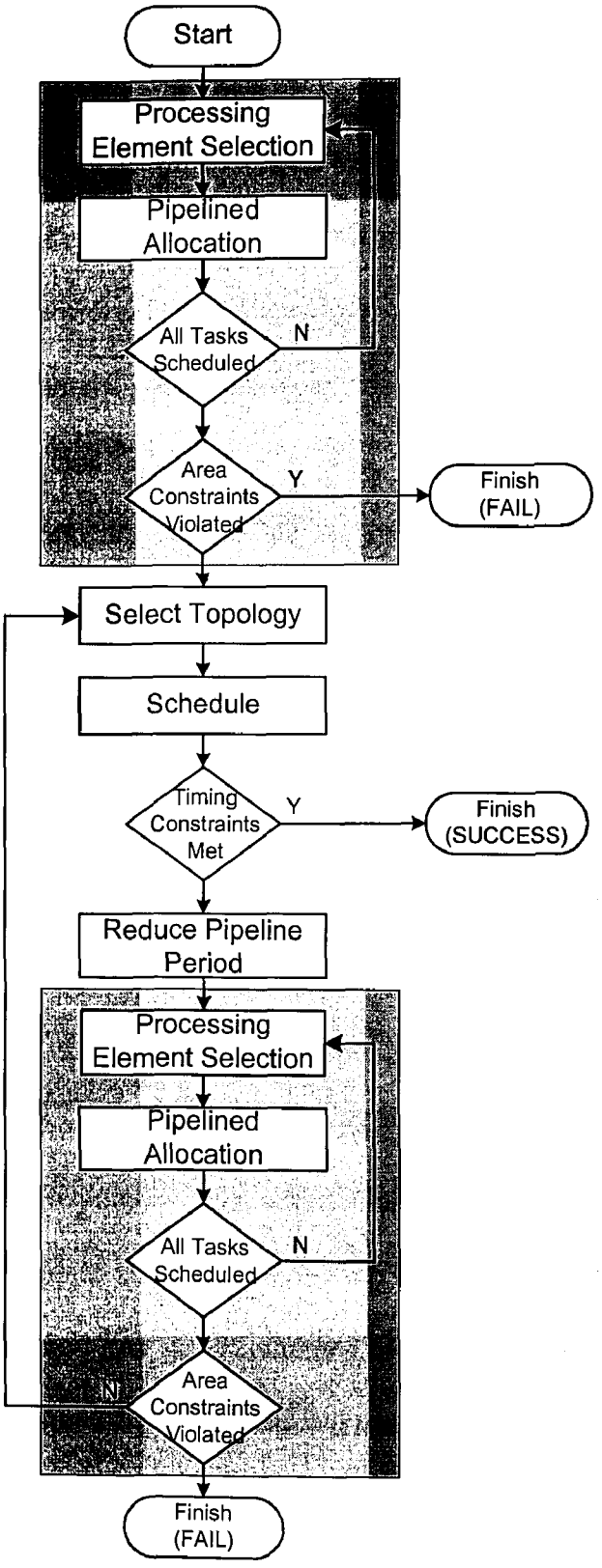

Figure 3.1 Co-synthesis Algorithm

Topology selection follows the scheduling phase. Start time for each task is identified in this stage. This phase takes into account the additional data communication delays incurred due to mapping processing elements on a regular topology. If after scheduling, all the tasks finish their execution with in required pipeline period, algorithm terminates successfully by producing regular, distributed memory architecture of hardware/software processing elements with schedule of each task. If tasks cannot be scheduled, algorithm proceeds to reduce pipeline period.

In pipeline period reduction phase, the system pipeline period is decreased. This is attempted to cope for the additional delays introduced by mapping processing elements to regular topology. Overhead introduced by the regular topology is used to reduce the pipeline period. More processing elements are then added in the system to satisfy the new period. After successful task allocation, algorithm selects a topology and tries to schedule the tasks again. This process continues till a system configuration which meets required timing constraints is established. Following sections describe each of these phases in detail.

3.2 Processing Element Selection and Pipelined Task Allocation

In this phase, the processing elements are selected, tasks are allocated and pipeline stages are created. Processing elements are added in an iterative manner. Initially all the tasks are attempted to be scheduled on a single processing element and if this is not possible,

more processing elements are added to the system. Task allocation is performed by scheduling a task on one of the available processing elements. Task allocation takes into account external data communication time, however, a fully connected topology is assumed in the beginning.

Different algorithms have been proposed for processing element selection and task allocation but all of them have certain limitations. Wolf starts with putting each task on a separate processing element and then tries to remove less utilized processing elements [41]. Hardware processing elements are almost always fully utilized and might never be removed. The algorithm does not pipeline the tasks and therefore cannot satisfy high throughput requirements. Bakshi and Gajski allocate tasks and create pipeline stages but tasks with hardware description are always mapped onto hardware processing elements and therefore they cannot trade off between hardware and software implementations for a given task [44]. Chatha and Vemuri perform task allocation through a branch and bound partitioner and present a heuristic algorithm for pipelining [22]. However, they do not consider pipeline period while creating pipeline stages and therefore may result in redundant pipeline stages. Figure 3.2 shows the processing element selection/ pipelined task allocation phase and following sub-sections describe each of these in detail.

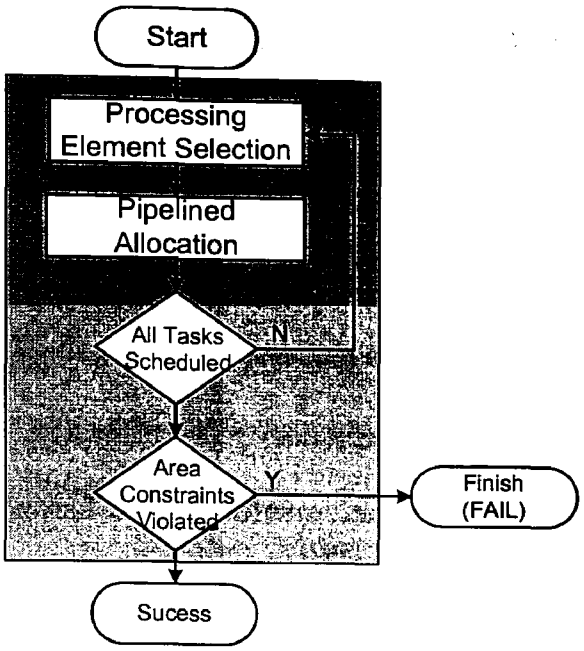

Figure 3.2 Processing Element Selection and Pipelined Allocation

3.2.1 Processing Element Selection

In this stage a hardware (HW) or software (SW) processing element is selected for addition to the system based on performance gain and its area cost. This selection involves computation of a selection coefficient for all processing elements available in the library (PE_{SELECT}). The selection coefficient consists of two factors, namely

- Performance Improvement Factor (PE_{PERF_IMPR})
- Area Cost Improvement Factor (PE_{AREA_FACTOR})

Several variables need to be defined in order to describe these factors.

- 'T_{SW}' denotes the set of tasks, which do not have a dedicated hardware resource for their execution in the current system.

- 'T_{HW}' denotes the set of tasks that have a dedicated hardware resource in the current system.
- 'PE_{SW}' and 'PE_{HW}' denote software and hardware processing elements respectively. '$SYS_{PE_{SW}}$' and '$SYS_{PE_{HW}}$' denote sets of software and hardware resources respectively in the current system.

Based on these variables, the cumulative software and hardware execution times are defined:

$$SYS_{CUM\_SW\_TIME} = \sum_{T_i} \sum_{PE_j} ExecTime(T_i, PE_j)$$

where

$T_i \in T_{SW}$, are tasks with which can execute on software processing elements

$PE_j \in SYS_{PE_{SW}}$, are software processing elements

$$SYS_{CUM\_HW\_TIME} = \sum_{T_i} \sum_{PE_j} ExecTime(T_i, PE_j)$$

where

$T_i \in T_{HW}$, are tasks which are executing on hardware processing elements

$PE_j \in SYS_{PE_{HW}}$, are hardware processing elements

These variables indicate the execution time of all the tasks, which execute on all software and hardware processing elements respectively. Next hardware improvement factor is defined, which is the improvement in system performance obtained by adding a hardware processing element to the system:

$$HW\_Imp\left(PE_{NEW}\right) = \frac{\sum_{T_i}\sum_{PE_j} ExecTime(T_i, PE_j)}{\left|SYS_{PE_{SW}}\right|} - \sum_{T_i} ExecTime(T_i, PE_{NEW})$$

where

T_i denotes the set of tasks which can execute on PE_{NEW}

$PE_j \in SYS_{PE_{SW}}$, is a software processing element which has already been added to the system

Using hardware improvement factor and cumulative hardware and software execution times, execution time of the system before (SYS_{PREV_TIME}) and after (SYS_{PREV_TIME}) the addition of a processing element are estimated.

$$SYS_{PREV\_TIME} = \frac{SYS_{CUM\_SW\_TIME}}{\left(\left|SYS_{PE_{SW}}\right|\right)^2} + SYS_{CUM\_HW\_TIME}$$

where

$\left|SYS_{PE_{SW}}\right|$ denotes the number of software processing elements currently in the system

$$SYS_{CURR\_TIME} = \begin{cases} \dfrac{SYS_{CUM\_SW\_TIME} + \sum_{T_i \in T_{sw}} ExecTime(T_i, PE_{NEW})}{\left(\left|SYS_{PE_{SW}}\right| + 1\right)^2} & if\ PE_{NEW} \in PE_{SW} \\ SYS_{PREV\_TIME} - HW\_Imp(PE_{NEW}) & if\ PE_{NEW} \in PE_{HW} \end{cases}$$

where

$|SYS_{PE\_SW}|$ denotes the number of software processing elements currently in the system

These variables are then used to calculate the performance improvement factor (PE_{PERF_IMPR}) for a given PE. This factor gives a normalized measure of the performance gain obtained by adding a particular processing element.

$$PE_{PERF\_IMPR} = 1 - \frac{SYS_{CURR\_TIME}}{SYS_{PREV\_TIME}}$$

Area cost of adding another processing element is also considered by determining area cost factor (PE_{AREA_FACTOR}). This factor identifies the area cost associated with a particular processing element. Area cost is normalized by dividing it by maximum area cost associated with any processing element available in the library (MAX_AREA).

$$PE_{AREA\_FACTOR} = 1 - \frac{Area(PE_{NEW})}{MAX\_AREA}$$

where

$Area(PE_{NEW})$ is the area cost of the newly added processing element

Using performance improvement and area cost improvement factors, finally the selection coefficient of a processing element is defined, which is the weighted sum of performance and area cost improvement factors.

$$PE_{SELECT} = k \times PE_{PERF\_IMPR} + (1-k) \times PE_{AREA\_FACTOR}$$

where

'k' is a user defined area-performance trade-off factor and $k \in [0,1]$

PE_{SELECT} is calculated for all available processing elements and the processing element with maximum selection coefficient is selected to be added into the system.

Processing elements are selected iteratively and it is possible that initially some slow processing elements get added into the system and degrade the system performance. This might result in system not meeting the required timing constraints. Algorithm uses a mechanism to remove these processing elements from the system.

Slow processing elements become the bottleneck when another processing element cannot be added into the system. This happens when;

$$\left|SYS_{PE_{SW}}\right| = \left| T_{SW} \right|$$

where

$\left| T_{SW} \right|$ is the total number of tasks that execute on a software processing element

$\left|SYS_{PE_{SW}}\right|$ is number of software processing elements in the system

Once the above condition is satisfied, a slow processing element is removed from the system. The slowest processing element is selected to be removed, which is found by using;

$$PE_{SLOW} = \max_{\forall PE \in SYS_{PE_{SW}}} \sum_{T_i \in T_{SW}} ExecTime(T_i, PE_j)$$

PE_{SLOW} is the processing element which takes maximum time to execute all software tasks. Minimum area criteria can also be used to find the slowest processing element. Once such a processing element is removed from the system, it is added to a 'DirtyPE' list. This list contains all the processing elements which have been removed from the system. Processing elements which are present in the 'DirtyPE' list are not considered in subsequent phases of processing element selection.

3.2.2 Pipelined Task Allocation

After a processing element has been added, all the tasks are then allocated to the processing elements that are currently in the system. Pipelining is also performed concurrently with the task allocation process.

First step of pipelined task allocation is to assign a priority to each task. This priority measure can be any metric but the criteria used in this algorithm assign high priority to those tasks which are on critical path (from execution point of view). Priority measure used is;

$$Priority(T_i) = Min\left(ExecTime\left(T_i\right)\right) + Max\left(Priority\left(T_j\right)\right)$$

where

$ExecTime(T_i)$ gives the execution time of Task T_i for all the available processing elements

T_j is a successor task of T_i

Priority is assigned by starting from the tail of the graph and setting the priority of each task as the sum of its minimum execution time (on any PE) and maximum priority of its successors. This priority measure is quite popular and has been used extensively in the past [12, 22]. Instead of using minimum execution time, any other statistical property can be used e.g., mean or median execution time, however, minimum metric provides slightly better results.

Pipelined allocation is the process of assigning start time to each task such that the following relationship is satisfied;

$$0 \leq StartTime(T_i) \leq ExecTime(T_i, PE) + T_{PERIOD}$$

where

StartTime(T_i) is the time at which task T_i starts its execution

ExecTime(T_i,PE) is the execution time of Task T_i on processing element where it has been allocated

T_{PERIOD} is the constraint pipeline period of the system

Pipeline allocation process initially finds starting and finishing time of each task for each processing element present in the system. Start time for each task is defined in terms of earliest start time (*EarliestStartTime*) and idle time of a processing element (***PEIdleTime***). These parameters are defined as:

$$EarliestStartTime(T_i) = MAX\left(FinishTime\left(PRED(T_i)\right)\right)$$

where

$PRED(T_i)$ = Set of all predecessors of task T_i

$$PEIdleTime(PE_j) = FinishTime\ (Last\ task\ on\ PE_j)$$

PEIdleTime is the time when a processing element becomes idle by completing the execution of all the tasks allocated to it.

Next, data communication time for task T_i is defined when T_i is allocated to PE_j. This is the time taken to transfer all the required input data of T_i to PE_j. This time is obtained by adding data transfer times of all the predecessors of T_i that are not allocated to PE_j:

$$CommTime\left(T_i, PE_j\right) = \sum_{\forall PRED\left(T_i\right) \notin PE_j} DataXfr\left(PRED\left(T_i\right)\right)$$

where

$PRED(T_i)$ = Set of all predecessors of task T_i

Now based on '*EarliestStartTime*', '*PEIdleTime*' and '*CommTime*', we define start and finish times for a task T_i when it is allocated to PE_j:

$$StartTime\left(T_i, PE_j\right) = MAX(EarliestStartTime(T_i), PEIdleTime(PE_j))$$

$$FinishTime\left(T_i, PE_j\right) = StartTime\ (T_i, PE_j) + CommTime\ (T_i, PE_j) + ExecTime\ (T_i, PE_j)$$

Using the above equations, finish time of the highest priority ready task is calculated for all processing elements that have been added to the system. Ready task is the task which

has all its predecessors allocated. Task is allocated to the processing element that has the earliest finish time. If the earliest finish time violates the pipeline period constraint, a new pipeline stage is created and task is added to the new pipeline stage. Earliest start time of the task is set to zero and finish time is calculated for all the processing elements again. The task is then allocated to the processing element, in the new pipeline stage that has earliest finish time. If earliest finish time of the task violates the pipeline period even in this new pipeline stage, pipelined allocation fails and another processing element needs to be added into the system.

3.3 Topology Selection

Task allocation process (after its completion) results in a set of processing elements which communicate with each other based on the inter-task communication. Tasks are scheduled to execute in different pipeline stages based on the timing constraints. Tasks are also assigned a start time and a finish time. System at this point meets all the constraints, however the processing elements are connected in an irregular topology. These processing elements are mapped onto a regular topology during topology selection phase.

Irregular topologies have certain disadvantages. Primarily, in irregular topologies more than one data communication paths are not guaranteed that are needed to support fault tolerance. If a communication link or a processing element fails, data can always be routed to other processing elements through a different path. Moreover, data routing is

complex in irregular topologies and such mechanisms may prove to be costly in terms of area, power etc.

In the proposed algorithm, irregular interconnection of processing elements (PE) is converted to a regular PE network. This mapping can increase data communication time, as data might have to go through a number of links. As a result additional processing elements may be added during this process. Topologies considered in this thesis are mesh, hypercube and quad-tree topologies. The algorithm selects a best topology out of these three, however approach presented here is not limited only to these schemes and any other regular topology can be added as well.

Selection of a regular topology is performed by initially generating regular topologies and assigning addresses to each location. All the processing elements are then mapped to these topologies. Mapping is performed by assigning neighbors to each processing element based on the magnitude of data traffic. Finally overhead for each topology is calculated and topology with least overhead is selected as the best topology. Following sections describe each of these steps in detail.

3.3.1 Topology Generation and Addressing

First step in topology mapping and selection is to create an empty template of nodes and assign addresses to each node. All the topologies are defined using the same template so

that the mapping does not depend on the type of topology. Topology template is defined as a graph consisting of nodes (V_T) connected by edges (E_T)

$A = \{ V_T, E_T \}$

where,

$V_T = \{v_0, v_1, ..., v_i\}$, is the set of nodes of the topology

$E_T = \{e_i = (v_x, v_y) \mid v_x, v_y \in V_T\}$, is the set of edges which connect the nodes

Each node is assigned a unique address '$Addr(v_i)$' and topology has same number of nodes as the number of processing elements in the system, that is;

$$| V_T | = |SYS_{PE_{SW}}| + |SYS_{PE_{HW}}|$$

A set of neighbors is defined for every node in the topology. Set of neighbors contain nodes which are adjacent to the current node. Set of neighbors is defined as;

$$N_{v_i} = \{ n \mid (v_i, n) \in E_T \}$$

Using these equations, topologies are generated and addresses are assigned to each node. Now the process for mesh, hypercube and quad-tree topologies is described.

a) Mesh Topology

Mesh topology is generated by arranging nodes in a grid. To determine number of rows and columns of grid, first number of processing elements in the system (*SYS_PE*) is calculated;

$$SYS\_PE = \left| SYS_{PE_{SW}} \right| + \left| SYS_{PE_{HW}} \right|$$

Using this number, number of rows and number of columns are determined using following equations;

$$MAX\_ROWS = \left\lceil \sqrt{SYS\_PE} \right\rceil$$

$$MAX\_COLS = \left\lceil (SYS\_PE \div MAX\_ROWS) \right\rceil$$

Finally, bits required to represent a row/column address is calculated as;

$$ADDR\_BITS = \left\lceil \log_2 (MAX\_ROWS) \right\rceil$$

Using these variables, nodes are added on a grid in a row-wise fashion. Addresses are assigned to each node by concatenating row and column addresses. After address assignment, the neighbors are added if they exist for given number of nodes. The existence is checked by using boundary conditions. Following pseudo code illustrates the process.

```
node = 0;
FOR row=0 to MAX_ROWS-1
      FOR col=0 to MAX_COLS-1

            //Assign address to node by concatenating row and column
            //addresses
            Addr(v_node) = (row << ADDR_BITS) | col;
            |N_v_node| = 0;   // set neighbor count to '0'

            IF (  Exist(UPPER_NEIGHBOR)  )

                  AddNeighbor(UPPER);  // add upper neighbor
            ENDIF

            IF (  Exist(LOWER_NEIGHBOR)  )

                  AddNeighbor(LOWER);  // add lower neighbor
            ENDIF

            IF (  Exist(LEFT_NEIGHBOR)  )

                  AddNeighbor(LEFT);  // add left neighbor
            ENDIF

            IF (  Exist(RIGHT_NEIGHBOR)  )

                  AddNeighbor(RIGHT);  // add right neighbor
            ENDIF

            node++;
            IF(  node == SYS_PE  )

                  RETURN;  //return when all nodes have been added
            ENDIF

      ENDFOR
ENDFOR
```

Figure 3.3 Pseudo Code for Mesh Topology Generation and Address Assignment

Following figure shows the application of above algorithm for eight nodes. At this point, only a blank template has been created and no processing element has been allocated to any node of the topology.

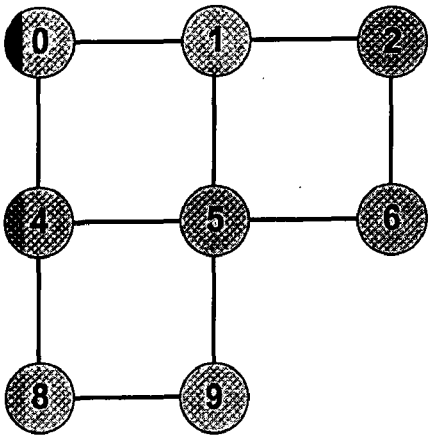

Figure 3.4 Eight Node Mesh Topology with Address Assignment

b) Hypercube Topology

Hypercube topology is generated by initially assigning addresses to each node. Addresses in this case range from '0' to 'SYS_PE', where SYS_PE is same as defined for mesh topology. After assigning the addresses, neighbors for a node are added. Degree of hypercube identifies number of neighbors for each node. Degree of a hypercube is defined as;

$$DEGREE = \lceil \log_2 (SYS\_PE) \rceil$$

Adjacent nodes in hypercube topologies have only 1-bit difference between the addresses [49]. This fact is used to assign neighbors for each node. Following figure illustrate this process.

```
//Assign address to each node
FOR node=0 to SYS_PE-1

      Addr(v_node) = node;
      |N_v_node| = 0;   // set neighbor count to '0'
ENDFOR


//Set neighbors for each node
FOR i=0 to SYS_PE-1
      FOR j=0 to SYS_PE-1

            IF ( AddrBitDiff(v_i,v_j,DEGREE) == 1 )

                  AddNeighbor(v_i,v_j);   // add v_j as v_i's neighbor
            ENDIF

      ENDFOR
ENDFOR
```

Figure 3.5 Pseudo Code for Hypercube Topology Generation and Address Assignment

Following figure shows the application of above algorithm for eight nodes.

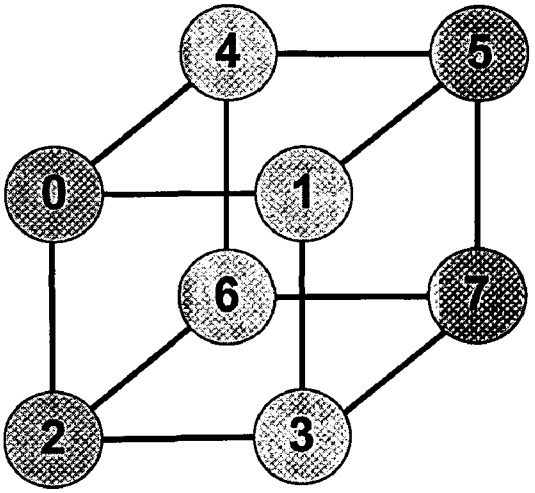

Figure 3.6 Eight Node Hypercube Topology with Address Assignment

c) Quad-Tree Topology

Tree topology is generated by assigning address and adding hierarchal neighbors as a node is created. Each created node is added to a FIFO to add further nodes and neighbors.

Finally after hierarchal links have been created, additional links are added at the same level between adjacent neighbors to support fault tolerance. Following figure describes this process.

```
node=0;
Addr(v_node)=1;
node++;
AddtoFIFO(v_node);

WHILE ( v_f = GetfromFIFO() )  //Continue till FIFO is empty

      //Add four new nodes (quad-tree) and update neighbors
      FOR i=0 to 4

            Addr(v_node) = Addr(v_f)*4 + 1; //Address for current node
            |N_v_node| = 0;   // set neighbor count to '0'
            AddNeighbor(v_f, v_node);
            node++;

            IF (node == SYS_PE)

                  BREAK; //Break when all the nodes have been created
            ENDIF

            AddtoFIFO(v_node);  //Add the new node to FIFO

      ENDFOR
ENDWHILE

//Add extra links by connecting adjacent nodes at the same level to
//support fault tolerance
CreateSameLevelNeighbors();
```

Figure 3.7 Pseudo Code for Quad-Tree Topology Generation and Address Assignment

Following figure illustrate the application of above algorithm for a quad-tree with eight nodes. Dotted lines in the figure indicate the extra links which provide multiple communication paths between different processing elements.

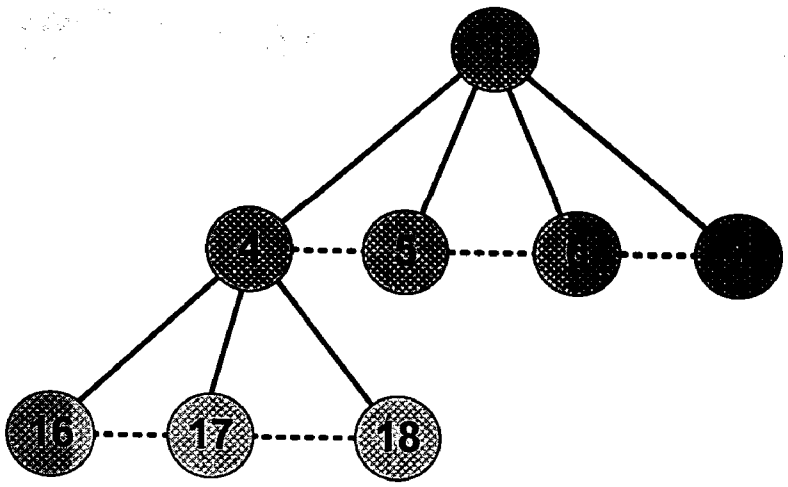

Figure 3.8 Eight Node Quad-Tree Topology with Address Assignment

3.3.2 Topology Mapping

Processing elements are mapped to a topology in a manner such that data communication delays can be minimized. To minimize the delay, processing elements are allocated to topology nodes based on their volume of communication with their neighbors. Traffic between different processing elements is defined using data communication with already allocated neighbors of a topology node. A node is allocated if a processing element has been assigned to that node (processing element has a unique address). Consequently allocated neighbors of node v_i are those nodes which are adjacent to v_i and have been assigned a processing element. These neighbors are denoted by $N_{v_i}^A$, and $N_{v_i}^A \subseteq N_{v_i}$. Neighbor traffic is then defined as;

$$NeighTraffic\left(PE_i, v_j\right) = \sum_{\forall v_a \in N_{v_j}^A} CommTime\left(PE_i, PE\left(v_a\right)\right)$$

where

$PE(v_a)$ is processing element allocated to topology node v_a

$CommTime(PE_i, PE_j)$ gives the time taken to transfer data between PE_i and PE_j

A FIFO list is used during allocation of processing elements to topology nodes. Unallocated neighbors of the current node are stored in the FIFO. This ensures that processing elements are assigned first to the immediate neighbors of allocated nodes. Following pseudo code illustrate the mapping of processing elements to topology nodes.

```
//Assign first PE to start node of the topology
PE_0 → v_0 ;
PE_0 = ALLOCATED;

//Add all the neighbors of v_0 to FIFO
AddtoFIFO( N_{v_0} );

//Continue processing while there is any un-allocated node
WHILE ( v_u = GetfromFIFO() )  //Continue till FIFO is empty

      //Calculate neighbor traffic for current node and all
      //unallocated processing elements
      FOR ALL UNALLOCATED PEs: PE_i

            FOR j=0 to | N^A_{v_u} |

                  CalculateNeighborTraffic(PE_i, v_j);
            ENDFOR
      ENDFOR

      //Get PE which has maximum neighbor traffic
      PE_A = GetMaxTrafficPE();

      //Assign PE_A to v_u
      PE_A → v_u ;
      PE_A = ALLOCATED;

      //Add unallocated neighbors of v_u to FIFO
      AddtoFIFO( N_{v_u} - N^A_{v_u} );

ENDWHILE
```

Figure 3.9 Pseudo code for Topology Mapping

3.3.3 Best Topology Selection

Best topology is selected by calculating the overhead of each topology. Overhead is the extra time spend in communicating data from one processing element to another. It is computed relative to the irregular topology, where a processing element can communicate with all the required processing elements directly. To describe overhead, neighbors of processing elements and hops required for data communication need to be

defined. Neighbors of a processing element forms a set N_{PE_i}, where each member is a processing element that needs to communicate with PE_i. Hops refer to the number of communication links traversed in a topology in order to transfer data between two given processing elements. Number of hops, denoted by N_{HOPS_T}, can be calculated from the addresses of each topology node. Following sections describe mechanism to calculate number of hops for each topology;

a) Mesh Topology

For a mesh topology, address of each node is assigned by concatenating row and column addresses. Number of hops needed to communicate data from one node to another is the sum of absolute differences of corresponding row and column addresses. For example in Figure 3.4, data transfer from node '0' (binary: 00_00) to node '9' (binary: 10_01) would require 3 hops (differences in row and column addresses is 2 and 1 respectively).

b) Hypercube Topology

For a hypercube topology, number of hops needed to communicate data from one node to the other is number of bit differences in the corresponding addresses. For example, in Figure 3.6, number of hops needed to communicate data from node '1' (binary: 001) to node '6' (binary: 110) is 3.

c) Quad-Tree Topology

Number of hops for quad tree is sum of level difference of two nodes and twice the level difference between node at lower level and first common parent of two nodes. Level of a node is defined as '$\lfloor \log_4(n_i) \rfloor$'. For quad tree of Figure 3.8, node '16' is on level 2 and node '1' is on level 0. Difference between these levels is 2. Also, first common parent of two nodes is node '1', therefore level difference between node at lower level (node '1') and first common parent is 0. Thus, number of hops required to communicate between node '16' and node '1' is 2+2(0) = 2.

Now, based on N_{PE_i} and N_{HOPS_T} overhead of a topology is defined as;

$$Overhead_T = \sum_{i=0}^{(SYS\_PE-1)} \sum_{j=0}^{|N_{PE_i}|} \left(N_{HOPS_T}(PE_i, PE_j) - 1\right) CommTime(PE_i, PE_j)$$

where,

CommTime(PE_i, PE_j) gives the time taken to transfer data between PE_i and PE_j

This overhead gives the extra time taken to transfer data between all the processing elements of the system. Topology which minimizes this overhead is selected as the best topology.

3.4 Scheduling

After processing elements are mapped to a topology, all the tasks of the application are rescheduled. Scheduling at this phase is necessary because of the extra communication

delays introduced by the regular topology. List scheduling is used to schedule tasks in this phase. The technique is similar to one used in pipelined task allocation except that every task is scheduled on the processing element which was selected for it during pipelined task allocation phase. Pipeline stages are not modified during this phase and each task is scheduled in the same pipeline stage where it was scheduled during the pipelined task allocation phase.

Scheduling is performed by selecting the highest priority task that has all its predecessors already scheduled. Priority assignment process is described in Section 3.2.2. Earliest start time and processing element idle time are then calculated for the selected task. Start time of the task is the maximum of these two quantities. Data communication time is then calculated taking into account the location of processing element for the current task and processing elements of all its predecessor tasks. Finally, finish time of the task is calculated by taking into account exact communication delays and execution time of the selected task. If task completes its execution within constraint pipeline period, ready task list is updated (as a result of scheduling of a task, more tasks may have all their predecessors scheduled). Otherwise, scheduling fails and more processing elements need to be added to the system by reducing the pipeline period. Following pseudo code illustrates the scheduling phase.

```
//Add the first task of the application to ready task list
READY_TASK = Task0;

//Continue processing while there is any un-allocated node
WHILE ( TaskR = GetReadyTask() )

      //Get the PE where task was allocated
      PE = GetTaskPE(TaskR);

      //Get the time when all parent tasks of TaskR finish their
      //execution. This time is '0' for a new pipeline stage
      EST = GetEarliestStartTime(TaskR);

      //Time when PE finishes the execution of all tasks
      //allocated before 'TaskR'
      PE_IDLE_TIME = GetPEIdleTime(PE);

      //Time to transfer all input data from parent tasks. This
      //time takes into account extra hops required as a result
      //of regular topology
      COMM_TIME = GetDataXfrTime (PE, TaskR);

      //Time taken by 'TaskR' to complete its execution on 'PE'
      EXEC_TIME = GetTaskExecTime(PE, TaskR);

      //Time when 'TaskR' finishes its execution on 'PE'
      FINISH_TIME = GetTaskFinishTime(
                                        TaskR ,
                                        MAX(EST, PE_IDLE_TIME),
                                        COMM_TIME,
                                        EXEC_TIME
                                      );

      IF(FINISH_TIME > TPERIOD)
            RETURN ERROR;
      ENDIF

      //Update Ready Task list as a result of scheduling 'TaskR'
      UpdateReadyTasks();

ENDWHILE
```

Figure 3.10 Pseudo code for Scheduling

3.5 Pipeline Period Reduction

Extra communication delays introduced as by the topology mapping may cause scheduling phase to terminate unsuccessfully. If scheduling fails, then more processing

elements are required to be added into the system so that tasks can complete their execution well within the target period and extra communication times do not violate the system pipeline period constraint. Pipeline period can be reduced by the maximum violation time factor; however other tasks may not violate the pipeline period with the same factor and this may redundantly add more processing elements in the system. Moreover, all pipeline stages do not have the same period as tasks in some pipeline stages complete earlier than the pipeline period. Therefore reducing the period by maximum violation time can result in expensive systems. Based on these factors an iterative pipeline period reduction mechanism is used in the algorithm. Pipeline period is reduced by a small amount every time scheduling phase fails. Pipeline reduction factor is defined as;

$$T_{RED\_FACTOR} = \frac{Overhead_T}{Links_M}$$

where,

$Overhead_T$ is overhead associated with the selected topology (Section 3.3.3)

$Links_M$ is the number of communication links which are missing in the regular topology and transfers across these links require multiple hops

After pipeline period is reduced, pipelined task allocation phase is repeated unless the system meets this new pipeline period. Processing elements are then mapped to regular topology and scheduling is performed with the original pipeline period constraint. When all tasks are scheduled, algorithm terminates successfully otherwise pipeline period is reduced further and the same process is repeated.

CHAPTER 4

EXPERIMENTAL RESULTS

4.1 Introduction

This chapter describes the results obtained by using the co-synthesis algorithm on different task graphs. The algorithm has been implemented in C++ programming language and Microsoft's Visual C environment was used for compilation and verification. All the experiments were conducted on a system with 512MB memory and a Pentium 4 processor running at 3.06 GHz. In the first experiment, MPEG encoder application is used for performing hardware software co-synthesis and second experiment was carried out using random graphs with up to 400 tasks.

4.2 MPEG Encoder

MPEG is a compression standard which is used to encode digital video [52]. It relies on block based motion compensation to reduce temporal redundancies and on DCT (discrete cosine transform) based compression scheme to reduce spatial redundancies. MPEG produces three types of coded frames known as I-frames (intra-coded frames), P-frames (predictively-coded frames) and B-frames (bidirectional predictively coded frames). 'P' and 'B' frames contain motion estimation information, 'I' frames on the other hand only contain discrete cosine transformed data. Most computationally intensive part of MPEG

is motion estimation where a 16x16 block of current image is searched in previous or next reference images. Search results in motion vectors which together with prediction error are stored in 'P' and 'B' frames. Prediction error is coded in the same way as 'I' frames using DCT followed by VLC (variable length coding).

MPEG encoder is specified as a data-flow based task graph for input to co-synthesis algorithm. Task graph for MPEG encoder is shown in Figure 4.1. It is a coarse grained graph consisting of 22 nodes with each node representing a large block of functionality. Numbers at the edges show the amount of data transferred between the tasks. Images in the video sequence are assumed to be in RGB format. 'Initialize' task performs the initialization and maintains a state machine to determine the type of coding ('I', 'P' or 'B') required for the current frame. 'YCbCr Conversion' task converts current image block and reference images (for 'B' and 'P' frames) from RGB to YC_bC_r format. 'Sub-Sample' task performs sub-sampling of color difference components (C_b and C_r) for the block of current image. 'Split Fwd Ref Image' and 'Split Bwd Ref Image' tasks split the forward and backward reference images into four overlapping regions for performing motion vector search over these regions in parallel. Tasks 'FS1' through 'FS4' and 'BS1' through 'BS4' perform the motion vector search over forward and backward reference images respectively. 'Fwd Motion Vector' and 'Bwd Motion Vector' tasks select the best forward and backward motion vectors respectively. 'Interpolate' task interpolates the forward and backward motion vectors for bi-directionally coded frames. 'DCT' task calculates discrete cosine transform for an image block or motion-predication error block. 'Quantize' performs quantization and 'DCAC Coding' task codes dc and ac components

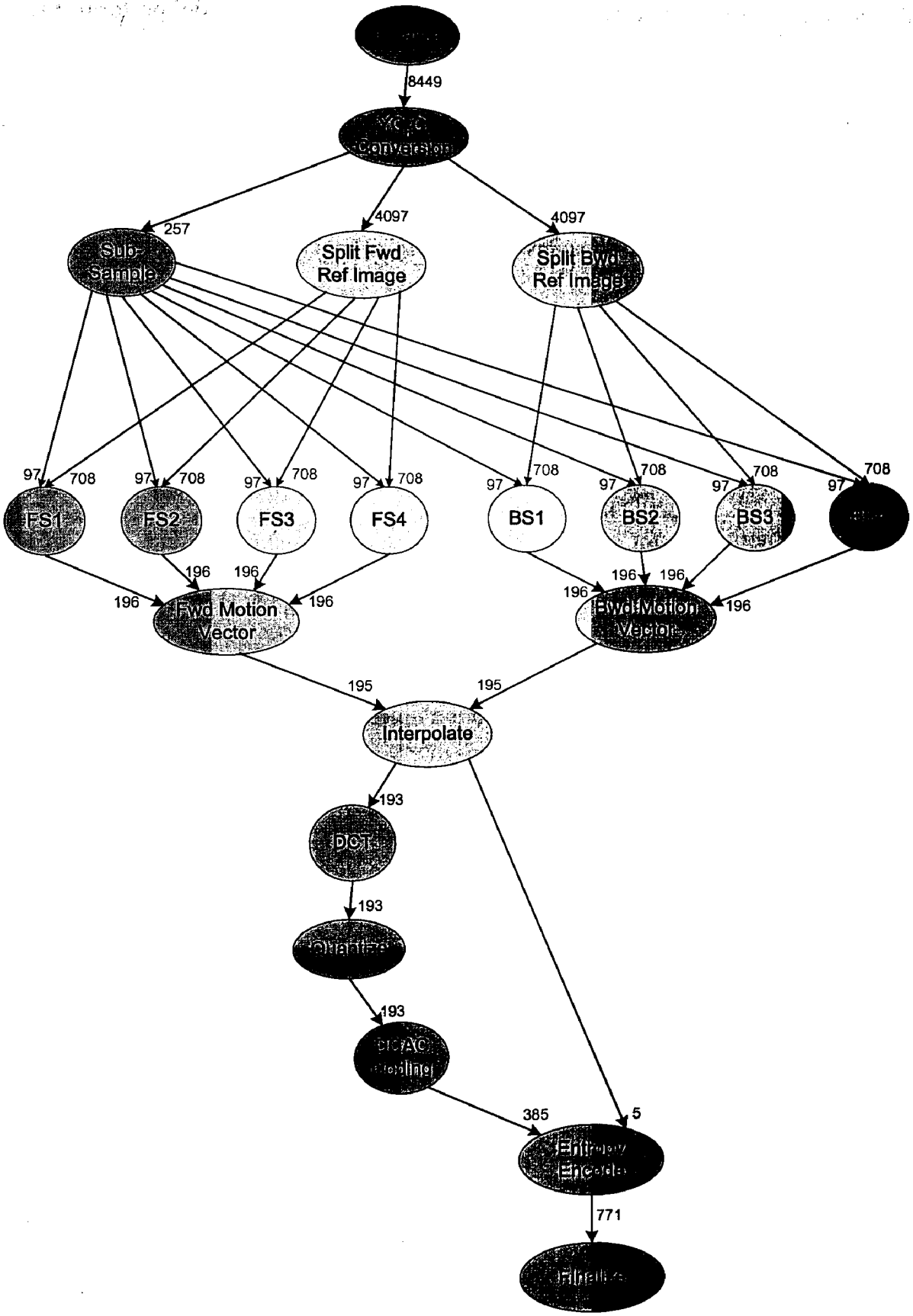

Figure 4.1 MPEG Encoder Task Graph

of quantized discrete cosine transformed block. Task 'Entropy Encode' performs variable length coding using huffman encoding scheme. Finally, 'Finalize' task packs huffman coded symbols to form the final compressed bitstream.

Software execution time for these tasks is calculated by executing each task on Altera's Nios processor. Nios CPU is available as configurable soft macro for Altera's fgpa devices. Every task is profiled on two different variants of Nios processor. One variant uses a dedicated hardware multiplier while the other uses multiple instructions to perform multiplication. Some of the tasks are also implemented as dedicated hardware. These tasks include motion vector search, DCT, quantization and dc/ac coding. Each of these tasks are implemented in Verilog hardware description language at register transfer level and then synthesized on Altera's Stratix fpga. Execution time is then the number of clock cycles required to complete the task. Execution times for some of the tasks depend on the amount of data required to be processed (e.g. run-length coding, Huffman encoding etc.). For these tasks, worst case execution time is considered. Area cost associated with each processing element is taken as the number of logic elements required to implement it on Stratix fpga. Table 4.1 lists the area cost of each processing element and
Table 4.2 gives the execution time of each task on different processing elements.

Table 4.1 Processing Element Information for MPEG Application

Name	Description	Type	Cost (Area)
NIOS_MUL	Altera's Nios embedded processor with a dedicated hardware multiplier	Software	4065
NIOS	Altera's Nios embedded processor without a dedicated hardware multiplier	Software	3662
MVS_ENG	Hardware core for motion vector search	Hardware	615
DCT_ENG	Hardware core for discrete cosine transform	Hardware	1008
QUANT_ENG	Hardware core for quantization	Hardware	712
DCAC_ENG	Hardware core for coding DC and AC components	Hardware	453

Table 4.2 Task Execution Times for MPEG Encoder Application

Task	Execution Time					
	Software NIOS_MUL	Software NIOS	Hardware MVS_ENG	Hardware DCT_ENG	Hardware QUANT_ENG	Hardware DCAC_ENG
Initialize	1194178	1245890	-	-	-	-
YCbCrConvert	6141804	11261292	-	-	-	-
SubSample	122539	199147	-	-	-	-
SplitFwdRefImage	1338842	1338842	-	-	-	-
SplitBwdRefImage	1338842	1338842	-	-	-	-
FS1	19983314	20025208	131524	-	-	-
FS2	19983314	20025208	131524	-	-	-
FS3	19983314	20025208	131524	-	-	-
FS4	19983314	20025208	131524	-	-	-
BS1	19983314	20025208	131524	-	-	-
BS2	19983314	20025208	131524	-	-	-
BS3	19983314	20025208	131524	-	-	-
BS4	19983314	20025208	131524	-	-	-
FwdMotionVector	52161	90945	-	-	-	-
BwdMotionVector	52161	90945	-	-	-	-
Interpolate	149770	304906	-	-	-	-
DCT	377600	775136	-	936	-	-
Quantize	145537	260677	-	-	468	-
DCACCoding	620181	620181	-	-	-	1214
EntropyEncoding	164918	242688	-	-	-	-
Finalize	256904	334876	-	-	-	-

MPEG encoder task graph along with task execution times and processing element hardware area information is fed to the co-synthesis algorithm. A range of constraints is provided to the algorithm to obtain a wide range of designs. The algorithm outputs a set of heterogeneous processing elements arranged in a regular distributed memory topology and a pipelined schedule for the set of tasks of the application. Major factors in the output consist of system pipeline period, area of the system, number of processing elements and number of pipeline stages created.

Algorithm was run for time constraints varying from 172000000 clock cycles to 6500000 clock cycles. Corresponding constraints on area ranged from 8000 logic elements to 38000 logic elements. Results obtained for these constraints are shown in Table 4.3. More processing elements get added into the system as the time constraints are made tighter. Also, the number of pipeline stages increase as system time period decreases. System corresponding to the tightest constraint (6500000 cycles) consists of 7 software processors and 11 dedicated hardware processing elements. The hardware processing elements correspond to 8 motion vector search engines and 3 hardware blocks for DCT, quantization and dc/ac coding. This system, giving the highest performance has the maximum area and tasks execute in 5 pipeline stages. On the other extreme, system corresponding to slowest requirements is around 26 times slower and takes around 9 times less area. System consists of only one processing elements and task execution is not pipelined. Figure 4.2 illustrates the design space exploration corresponding to various test cases.

Table 4.3 Time/Area Results for MPEG Encoder

S.No.	Constraints		Results					
	Time	Area	System Time	System Area	PEs	Pipeline Stages	SW PEs	HW PEs
1.	6500000	38000	6420893	36483	18	5	7	11
2.	7500000	17000	7458521	16309	11	3	3	8
3.	8500000	17000	7592363	16309	11	3	3	8
4.	9500000	17000	8807958	16309	11	3	3	8
5.	25000000	15000	21459777	14676	9	4	3	6
6.	40100000	15000	40091097	14655	7	4	3	4
7.	45000000	13000	41484886	12831	6	4	3	3
8.	55000000	12500	53897342	12216	5	3	3	2
9.	61000000	11700	60370101	11601	4	4	3	1
10.	63000000	11500	61419853	11389	3	3	3	0
11.	95000000	8000	90544535	7727	2	2	2	0
12.	172000000	4065	171821949	4065	1	1	1	0

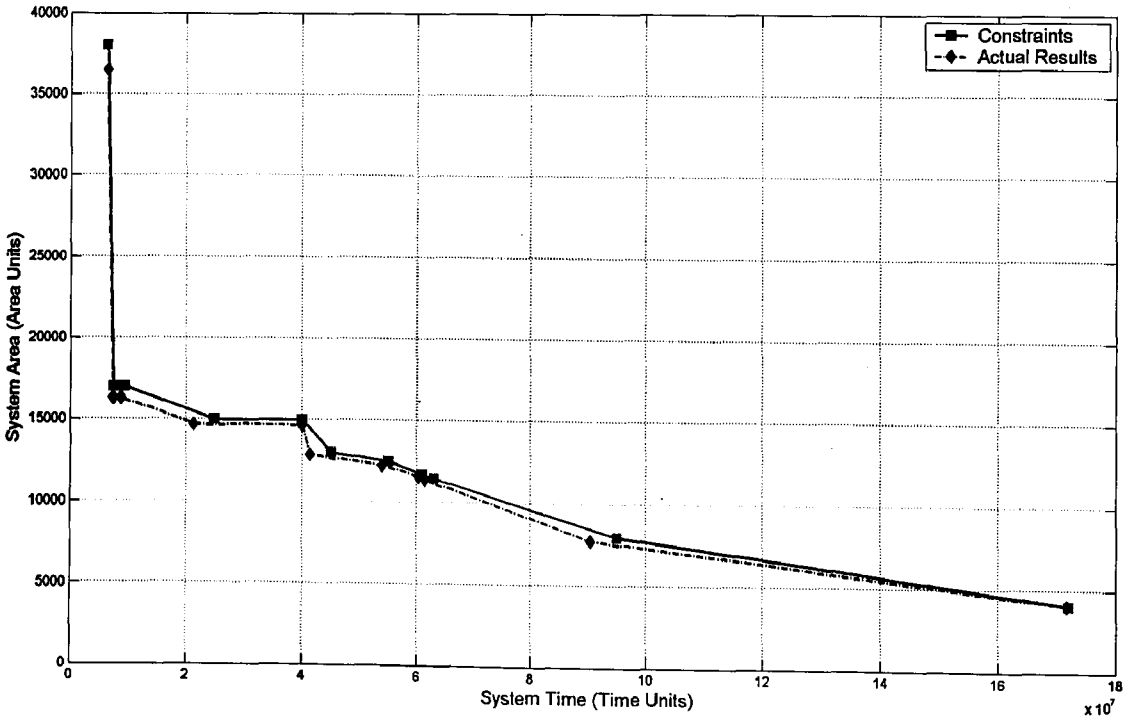

Figure 4.2 Design Space Exploration for MPEG Encoder Application

Other than timing and area results, an important characteristic of the resulting system is the arrangement of the processing elements in a regular topology. Table 4.4 lists this characteristic of the system for each test case. It shows the overhead involved in arranging processing elements to a particular topology and number of links missing in each topology when compared to a fully connected topology. It also gives the number of extra processing elements that are added to as a result of increased extra communication delays due to a regular topology.

Table 4.4 Topology Information for MPEG Encoder

S. No.	Time Constraint	Tree Topology		Mesh Topology		Hypercube Topology		Selected Topology	Extra PEs
		$Overhead_T$	$Links_M$	$Overhead_T$	$Links_M$	$Overhead_T$	$Links_M$		
1.	6500000	24051	27	13037	18	14696	24	MESH	0
2.	7500000	6201	15	7978	19	5394	15	HYPERCUBE	0
3.	8500000	6201	15	7978	19	5394	15	HYPERCUBE	0
4.	9500000	5946	16	9681	16	4017	13	HYPERCUBE	0
5.	25000000	2034	9	7458	14	2647	9	TREE	0
6.	40100000	2329	10	4746	12	2942	10	TREE	0
7.	45000000	936	5	3163	7	1549	6	TREE	0
8.	55000000	1880	6	2710	4	2295	5	TREE	0
9.	61000000	2490	4	2905	3	2905	3	TREE	0
10.	63000000	999	2	999	2	999	2	TREE	0
11.	95000000	0	0	0	0	0	0	TREE	0
12.	172000000	0	0	0	0	0	0	TREE	0

It is seen that as the time constraints become tighter and tighter, overhead and number of missing links increase. This is due to the fact that as timing constraints get smaller, more processing elements are added which increase the inter-processor communication. For mpeg encoder application, no extra processing element was required and algorithm selected all of the available topologies for different test cases. For the test case 1, Mesh topology has the lowest overhead. System in this case consists of 11 processing elements and 5 pipeline stages. Figure 4.3 shows the irregular interconnection of the processing elements before they are mapped onto a regular topology. Arrangement of processing elements in a regular topology is shown in Figure 4.4. Corresponding schedule map for each task is displayed in Figure 4.5. Figure 4.6, Figure 4.7 and Figure 4.8 show similar details for test case 4, where processing elements are arranged in a hypercube topology. Figure 4.9, Figure 4.10 and Figure 4.11 provide same details for test case 5, where tree topology has the least overhead. Finally, an interesting case is where system consists of only a single processing element. In this case there is no overhead and all the tasks execute one after the other depending on their priority. This case is illustrated in Figure 4.12.

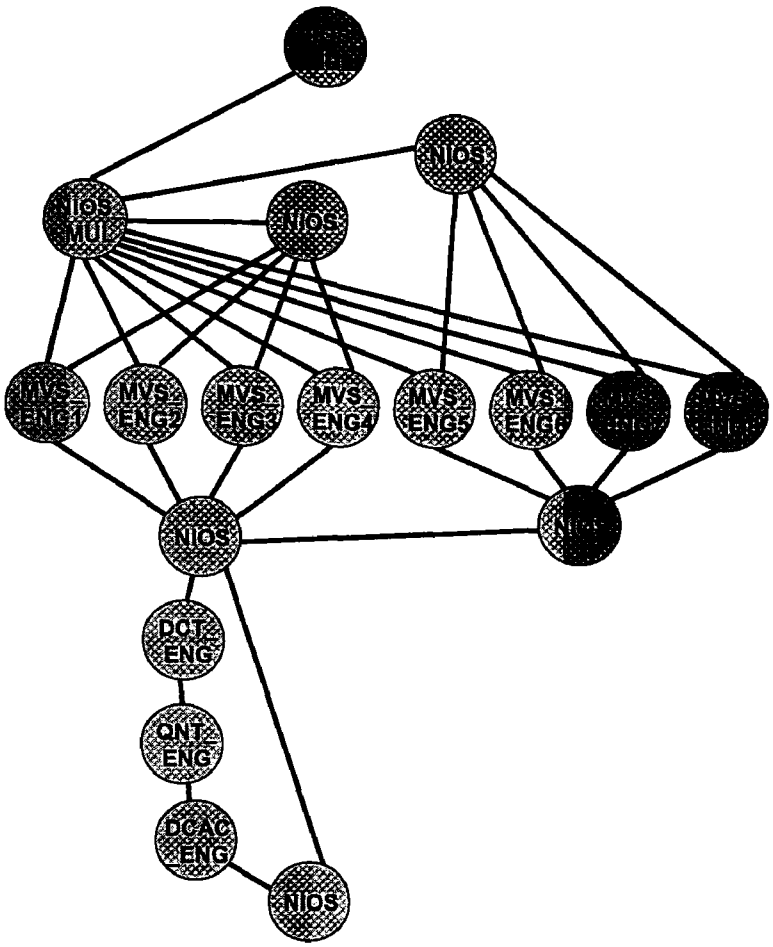

Figure 4.3 Irregular Processing Element Topology (Test case 1)

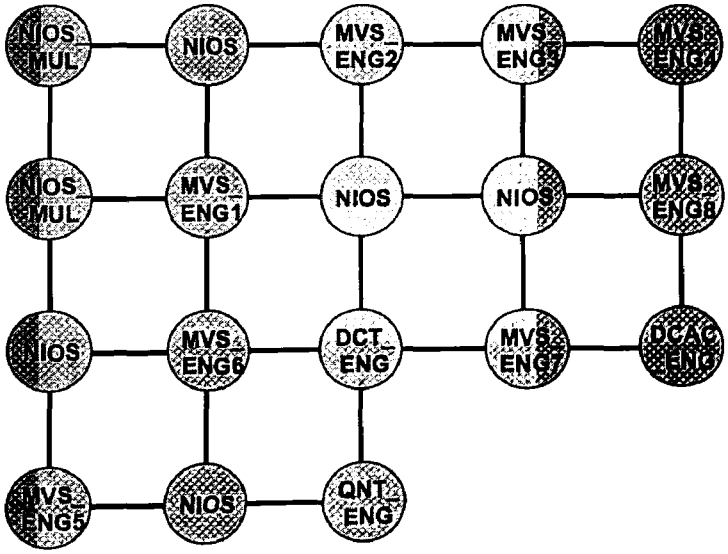

Figure 4.4 Processing Elements for MPEG Encoder Arranged in Mesh Topology

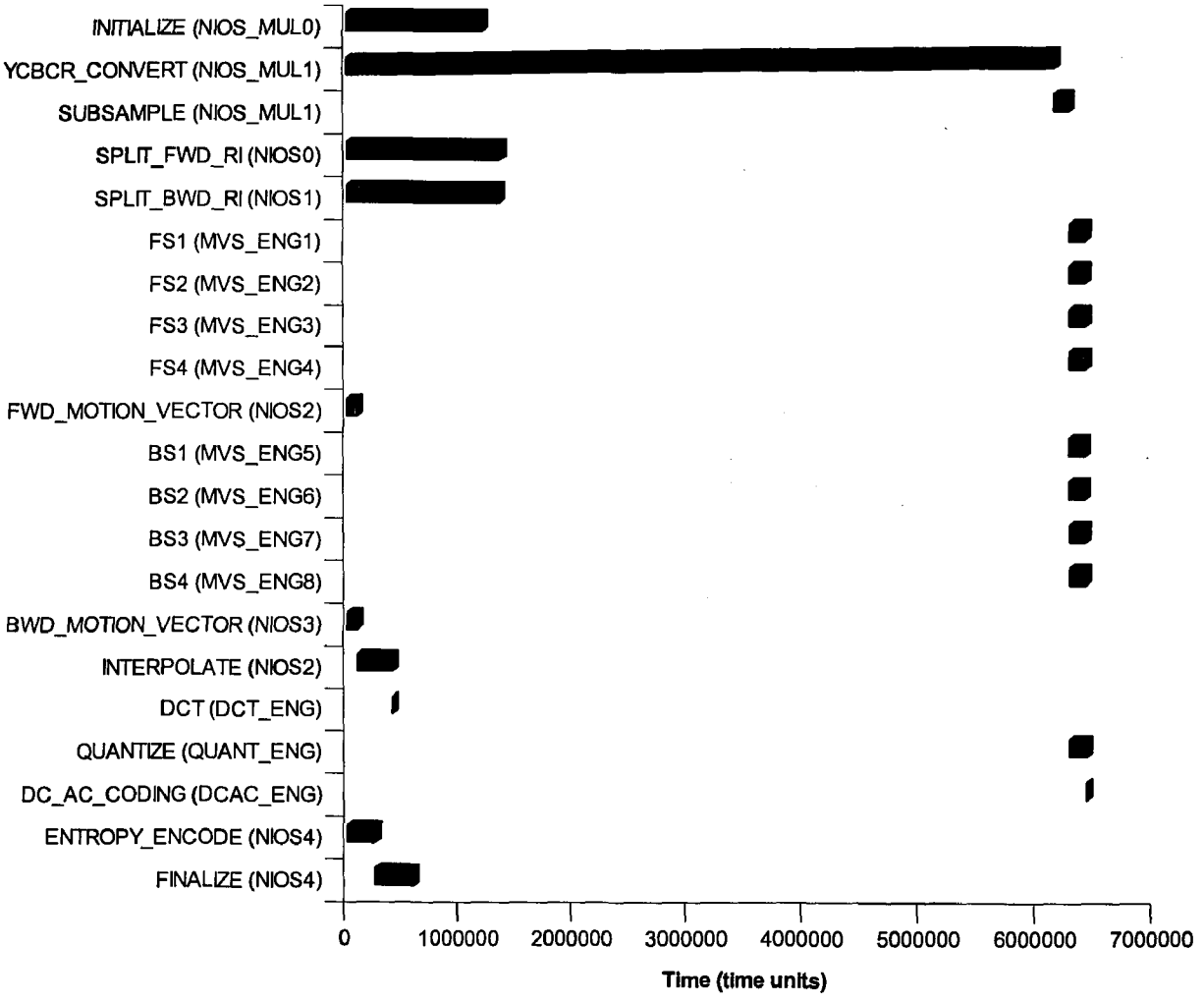

Figure 4.5 Schedule Map for Mesh Topology

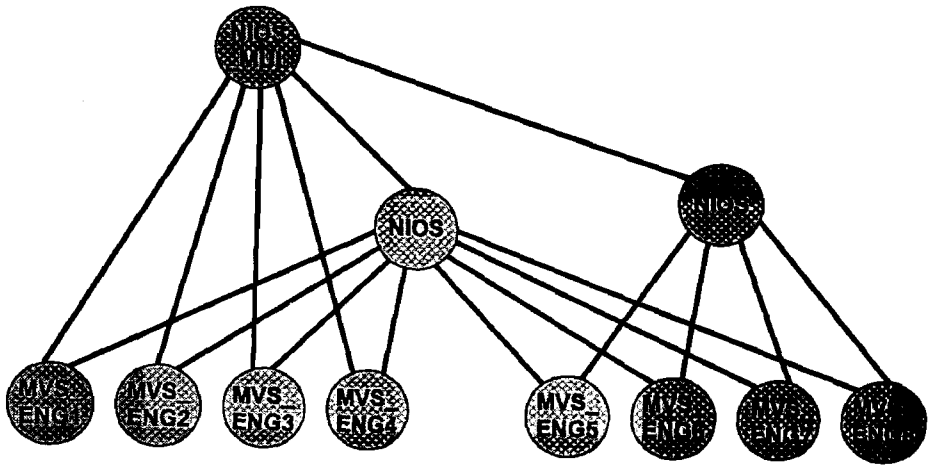

Figure 4.6 Irregular Processing Element Topology (Test case 4)

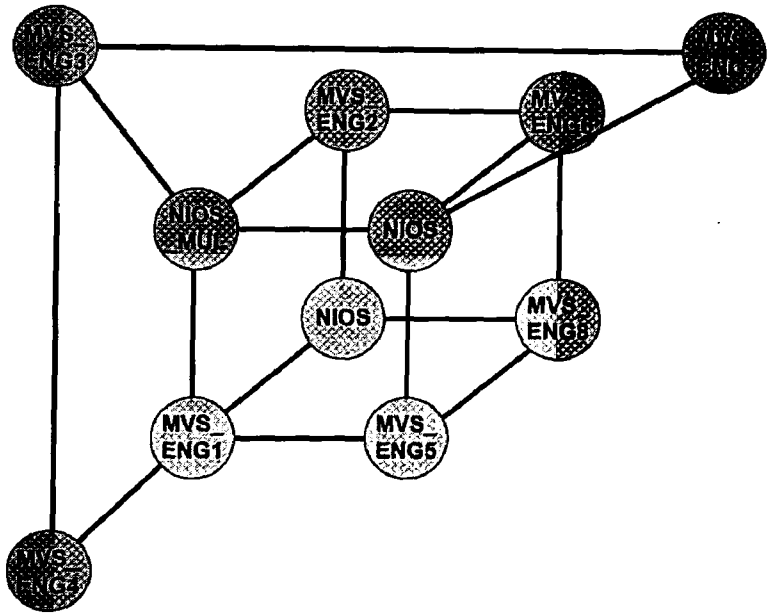

Figure 4.7 Processing Elements for MPEG Encoder Arranged in Hypercube Topology

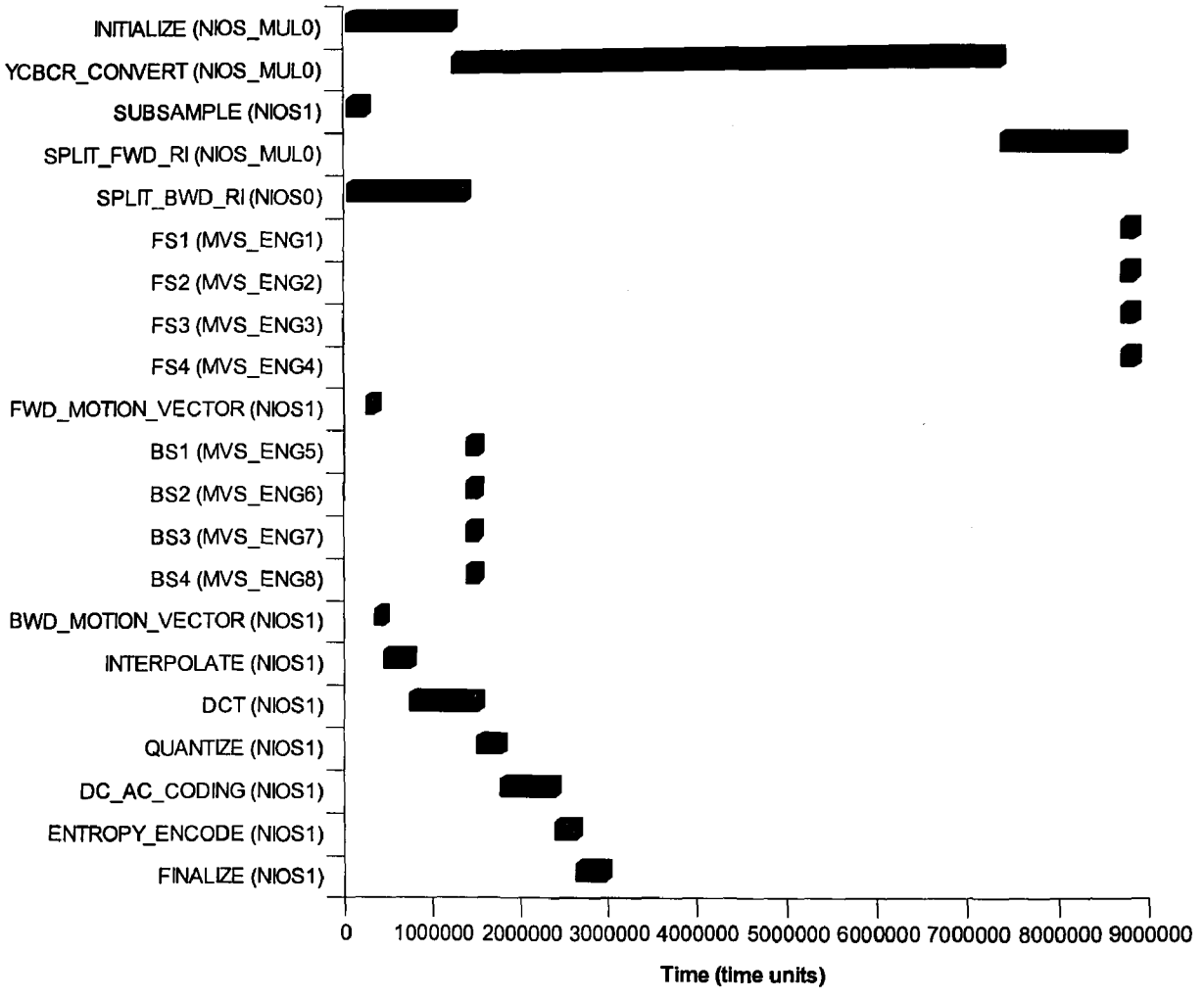

Figure 4.8 Schedule Map for Hypercube Topology

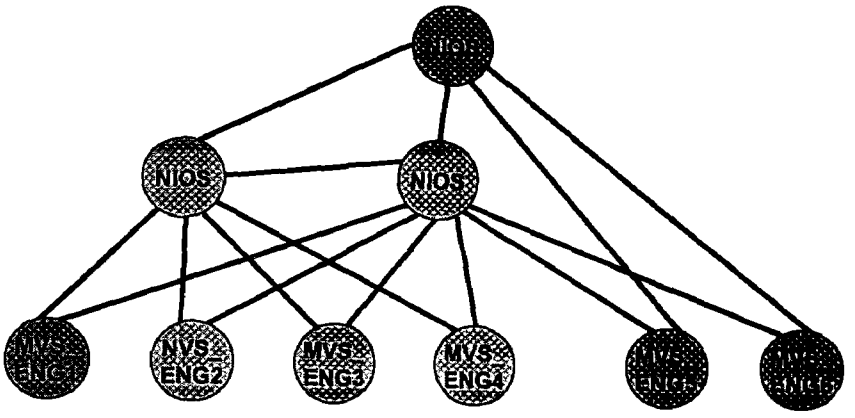

Figure 4.9 Irregular Processing Element Topology (Test case 5)

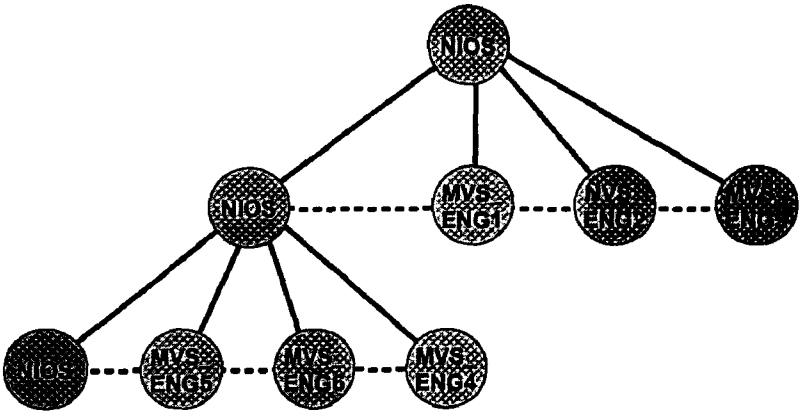

Figure 4.10 Processing Elements for MPEG Encoder Arranged in Tree Topology

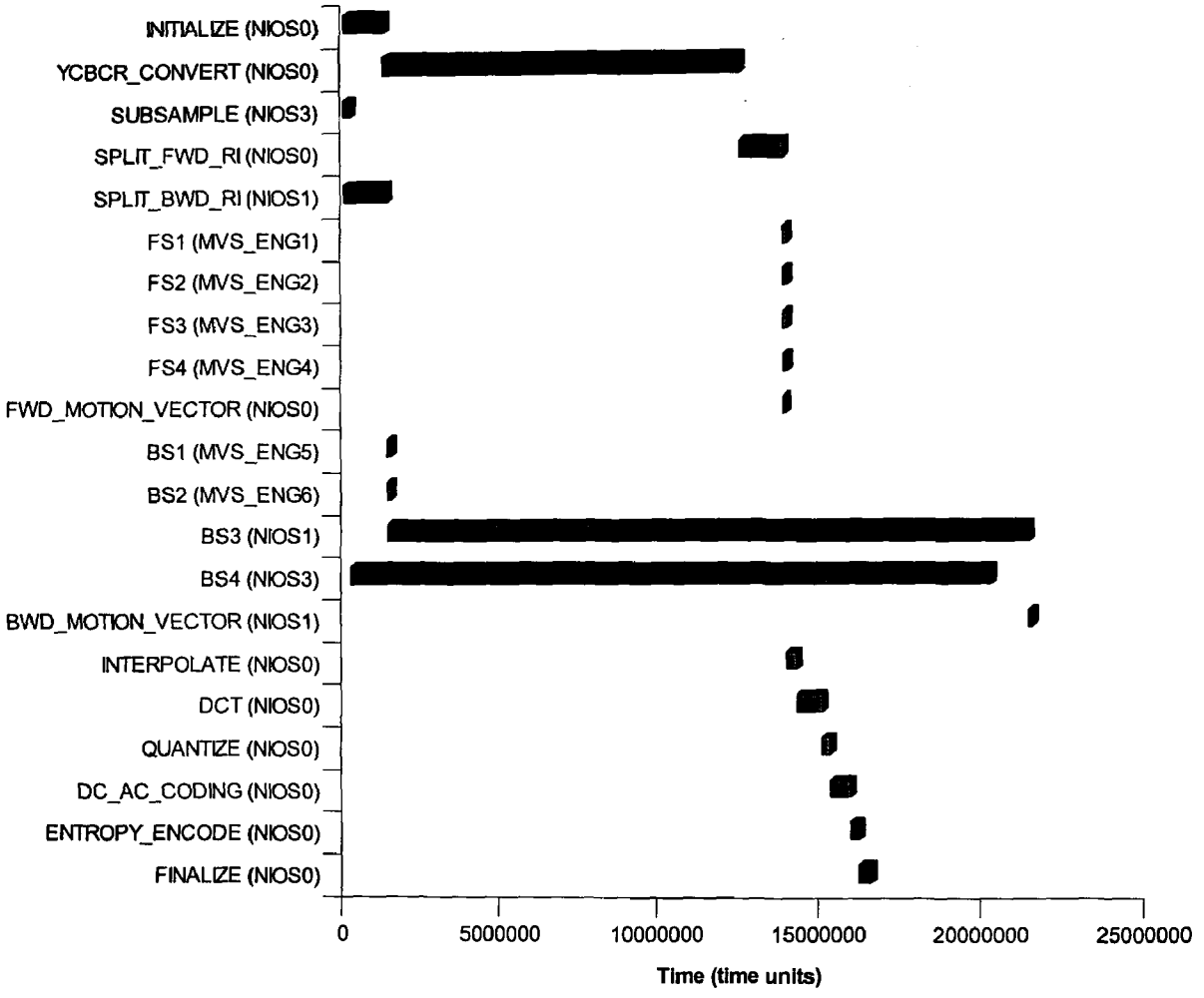

Figure 4.11 Schedule Map for Tree Topology

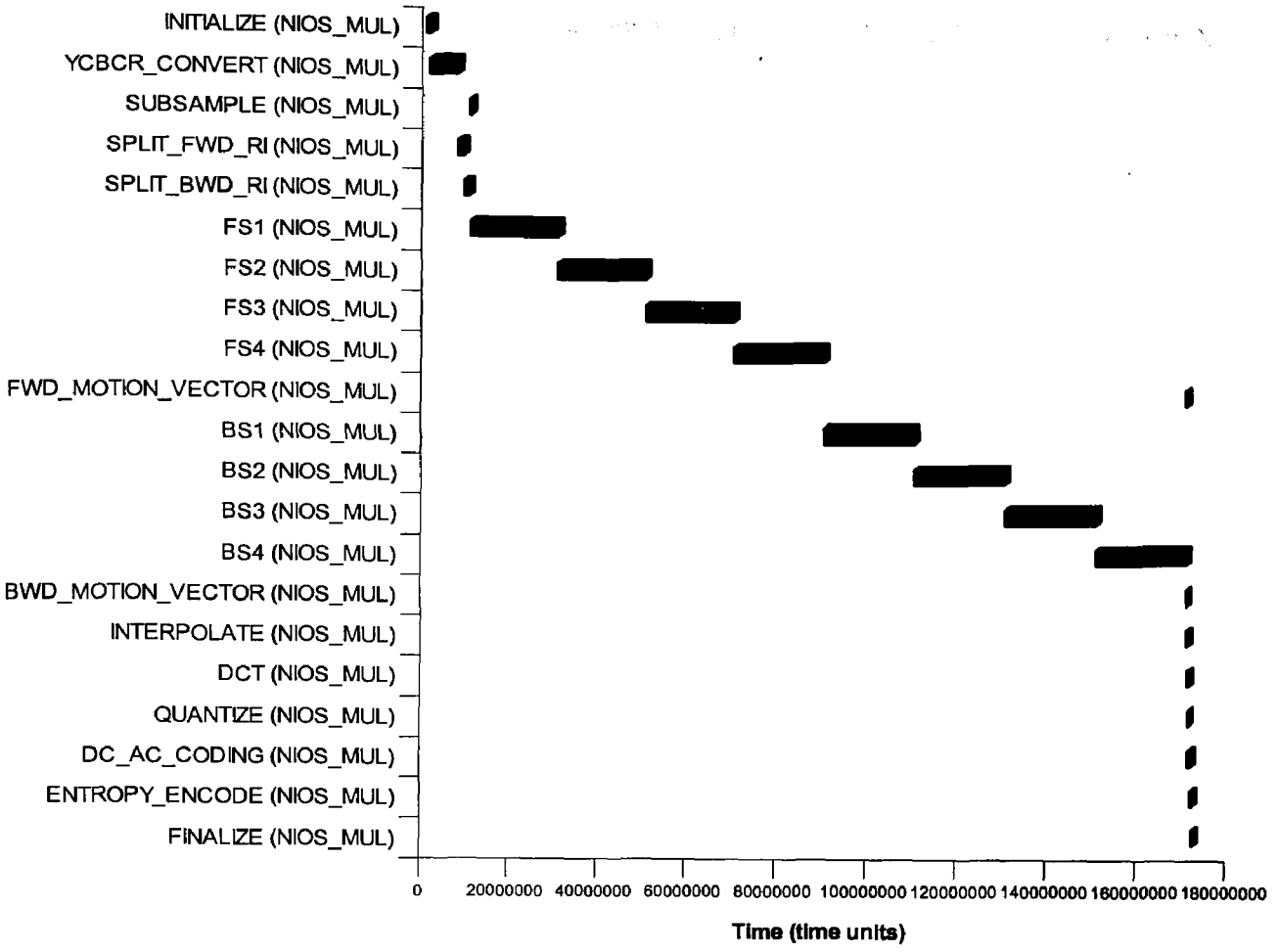

Figure 4.12 Schedule Map with only a Single Processing Element in the System

4.3 Parallel MPEG Decoding

In the second experiment, proposed algorithm is applied to a test case from Hypercube Co-synthesis algorithm [46]. Hypercube co-synthesis algorithm involves processing element selection, task scheduling and pipelining however the processing elements are always mapped to a hypercube topology. Also the algorithm uses RECOD pipelining method ([22]) that may also result in redundant pipeline stages. The task graph for this test case consists of 22 nodes with 16 MPEG decoding tasks. Pentium II processor (450

MHz) was used for software implementation and Altera FLEX10KE FPGA was employed for hardware synthesis. Following figure shows the task graph for parallel MPEG decoding.

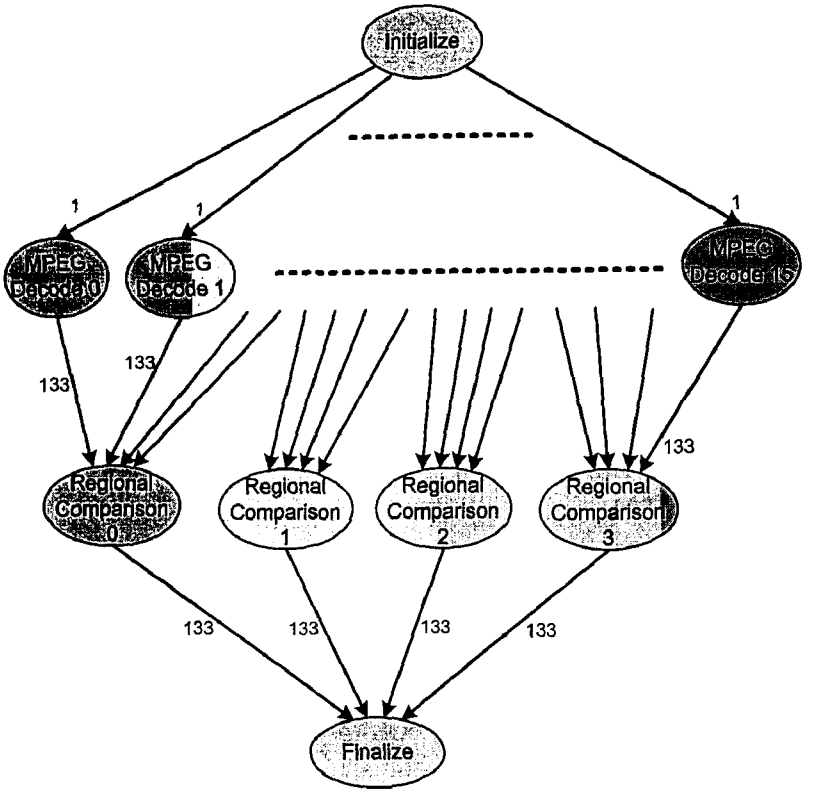

Figure 4.13 Parallel MPEG Decoding Task Graph

Proposed method is tested with the same constraints as used in the Hypercube co-synthesis technique. Area constraints were varied from 11.5M to 7.5M area units and corresponding time constraints were 30000 to 770000. Algorithm was able to meet all the constraints. Processing elements were mapped to tree and hypercube topologies. In the first case where the resulting system consists of 7 processing elements, tree topology results in 10 communication links as opposed to hypercube topology which consists of 12

links. This shows the savings in cost of the resulting system. Detailed results are shown in Table 4.5and Table 4.6. Optimal results are also provided in [46]. Results obtained by the proposed co-synthesis algorithm suggest that its timing performance is, on an average, only 0.97% more than optimal and it utilized only 0.20% more area than the optimal results. As a comparison, timing and area differences for hypercube co-synthesis algorithm were 3.75% and 0.62% respectively. This illustrates the effectiveness of the proposed algorithm. Comparison results are shown in Figure 4.14.

Table 4.5 Time/Area Results for Parallel MPEG Decoding

S.No.	Constraints		Results					
	Time	Area	System Time	System Area	PEs	Pipeline Stages	SW PEs	HW PEs
1.	40000	11500000	36320	11118452	7	4	1	6
2.	50000	11000000	45405	10258292	6	4	1	5
3.	60000	10500000	45405	10258292	6	4	1	5
4.	90000	9500000	68069	9398132	5	4	1	4
5.	120000	9000000	118040	8537972	4	4	1	3
6.	130000	8600000	118040	8537972	4	4	1	3
7.	600000	7600000	588480	7588611	2	2	1	1
8.	770000	7500000	759296	7500000	1	1	1	0

Table 4.6 Topology Information for Parallel MPEG Decoding

S. No.	Time Constraint	Tree Topology		Mesh Topology		Hypercube Topology		Selected Topology	Extra PEs
		$Overhead_T$	$Links_M$	$Overhead_T$	$Links_M$	$Overhead_T$	$Links_M$		
1.	40000	1599	4	2009	6	1869	5	TREE	0
2.	50000	1995	3	2675	5	1335	3	HYPERCUBE	0
3.	60000	1995	3	2675	5	1335	3	HYPERCUBE	0
4.	90000	2394	3	2800	3	1862	2	HYPERCUBE	0
5.	120000	2261	2	1729	1	1729	1	HYPERCUBE	0
6.	130000	2261	2	1729	1	1729	1	HYPERCUBE	0
7.	600000	0	0	0	0	0	0	TREE	0
8.	770000	0	0	0	0	0	0	TREE	0

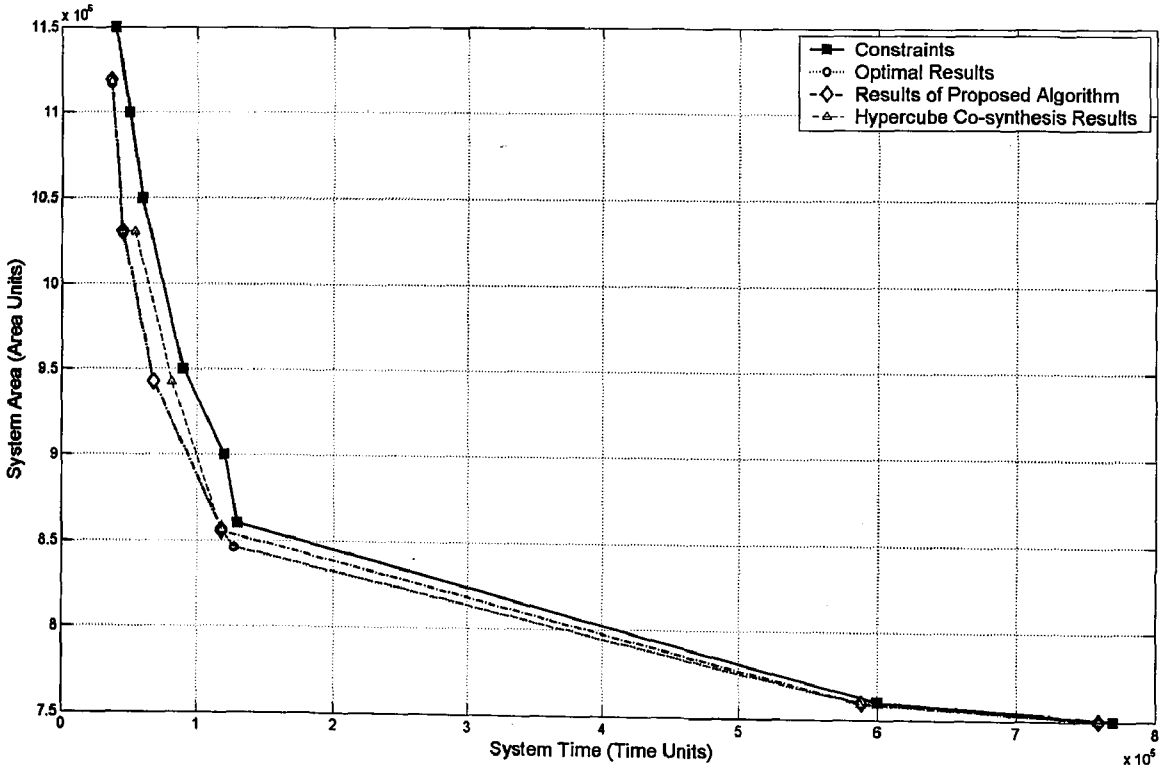

Figure 4.14 Comparison Results for Parallel MPEG Decoding

4.4 *Random Graphs*

Third experiment is conducted by performing co-synthesis on random task graphs. These graphs are generated by randomly varying number of predecessors and successors for each task, depth or number of levels and amount of data transferred between different tasks. Successors and predecessors for a task are varied from 2 to 20 and tasks without a successor are connected to the final task. Number of processing elements available for the system is also random and both hardware and software processing elements are included in the library. Execution time of each task depends on the area and type of the processing element. Hardware processing elements execute respective tasks in less time compared to software counterparts. Large random graphs are generated to conduct the experiments. Five different types of graphs are created with 50, 100, 200, 300 and 400 nodes. Multiple graphs for each type are used and each of the graphs is then tested for a range of area and timing constraints. Figure 4.15 shows a random graph of 50 nodes.

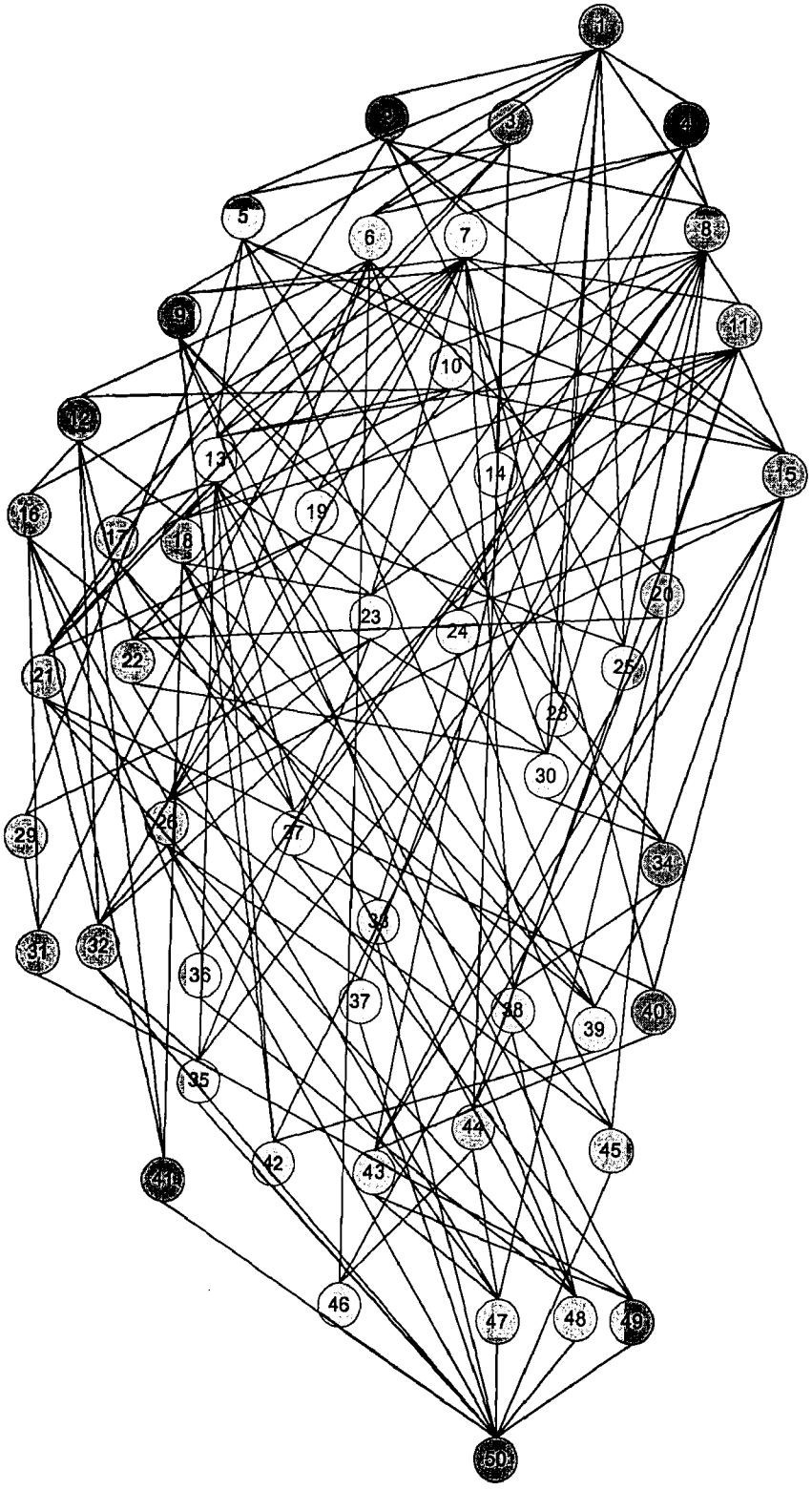

Figure 4.15 Randomly Generated 50-node Graph

Timing constraints for are varied from 25000 to 70000 time units for five different task graphs (graph 'a' to graph 'e') of 50 nodes. Area constraints range from 1800 to 16500 area units. Number of processing elements ranged from 1 to a maximum of 10 processing elements and pipeline stages varied from 1 to 9. All the topologies were selected for different cases and overhead of the topologies increased as the pipeline period became smaller. Also, for cases with tighter constraints extra processing elements were required to cater for delays introduced due to regular topology mapping. Number of missing links increase as timing constraints become smaller which is due to the fact that more processing elements get added into the system and communication between processing elements increases. Table 4.7 shows the timing/area results with number of processing elements and pipeline stages. Table 4.8 provides the topology information and Figure 4.16 illustrates the design space exploration for each graph with 50 tasks. Similar results are obtained for other graphs, however, number of missing links at smallest constraint increases with number of tasks in the graph. Subsequent tables and figures illustrate the results for 100, 200, 300 and 400 node task graphs. Finally, Figure 4.21, Figure 4.22 and Figure 4.23 show the arrangement of processing elements for 400-node task graph (graph 'e' with time constraint of 100000 time units), 200-node task graph (graph 'e' with time constraint of 125000 time units), and 300-node task graph (graph 'c' with time constraint of 200000 time units) respectively.

Table 4.7 Time/Area Results for 50 Node Graphs

Graph	Constraints		Results					
	Time	Area	System Time	System Area	PEs	Pipeline Stages	SW PEs	HW PEs
a.	25000	8500	22574	8492	8	9	6	2
a.	30000	5500	29934	5268	4	5	4	0
a.	40000	5300	30884	5268	4	4	4	0
a.	50000	2650	46840	2634	2	2	2	0
a.	67000	1400	66795	1317	1	1	1	0
b.	25000	14200	24851	14131	8	8	6	2
b.	30000	9600	27336	9529	6	7	4	2
b.	40000	7000	35579	6903	3	3	3	0
b.	50000	5000	40417	4602	2	2	2	0
b.	60000	2400	59125	2301	1	1	1	0
c.	25000	16500	23594	16456	8	7	6	2
c.	30000	11000	29092	10742	5	5	4	1
c.	35000	8100	33442	8031	3	3	3	0
c.	40000	5500	39606	5354	2	2	2	0
c.	65000	2700	60661	2677	1	1	1	0
d.	25000	11500	24144	11390	10	6	6	4
d.	30000	9600	28298	9512	8	5	5	3
d.	40000	5300	38595	5232	3	3	3	0
d.	45000	3500	43813	3488	2	2	2	0
d.	65000	1800	61778	1744	1	1	1	0
e.	30000	11000	26350	10075	5	6	5	0
e.	40000	8100	34844	8060	4	4	4	0
e.	45000	6100	42934	6045	3	3	3	0
e.	50000	4100	49468	4030	2	2	2	0
e.	70000	2100	68840	2015	1	1	1	0

Table 4.8 Topology Information for 50 Node Graphs

Graph	Time Constraint	Tree Topology		Mesh Topology		Hypercube Topology		Selected Topology	Extra PEs
		$Overhead_T$	$Links_M$	$Overhead_T$	$Links_M$	$Overhead_T$	$Links_M$		
a.	25000	30573	30	29057	26	17419	24	HYPERCUBE	2
a.	30000	15117	6	7475	4	7475	4	HYPERCUBE	0
a.	40000	15117	6	7475	4	7475	4	HYPERCUBE	1
a.	50000	0	0	0	0	0	0	TREE	0
a.	67000	0	0	0	0	0	0	TREE	0
b.	25000	21260	27	23021	22	17601	23	HYPERCUBE	1
b.	30000	9834	14	10866	13	9452	11	HYPERCUBE	0
b.	40000	6755	2	6755	2	6755	2	TREE	0
b.	50000	0	0	0	0	0	0	TREE	0
b.	60000	0	0	0	0	0	0	TREE	0
c.	25000	28498	29	25701	26	17140	24	HYPERCUBE	3
c.	30000	13322	9	13210	7	13227	8	MESH	2
c.	35000	7341	2	7341	2	7341	2	TREE	0
c.	40000	0	0	0	0	0	0	TREE	0
c.	65000	0	0	0	0	0	0	TREE	0
d.	25000	27041	37	31235	33	22872	30	HYPERCUBE	2
d.	30000	21888	22	24441	18	20920	18	HYPERCUBE	1
d.	40000	8756	2	8756	2	8756	2	TREE	0
d.	45000	0	0	0	0	0	0	TREE	0
d.	65000	0	0	0	0	0	0	TREE	0
e.	30000	12597	12	17438	10	13475	10	TREE	0
e.	40000	9450	6	7626	4	7626	4	HYPERCUBE	1
e.	45000	5958	2	5958	2	5958	2	TREE	0
e.	50000	0	0	0	0	0	0	TREE	0
e.	70000	0	0	0	0	0	0	TREE	0

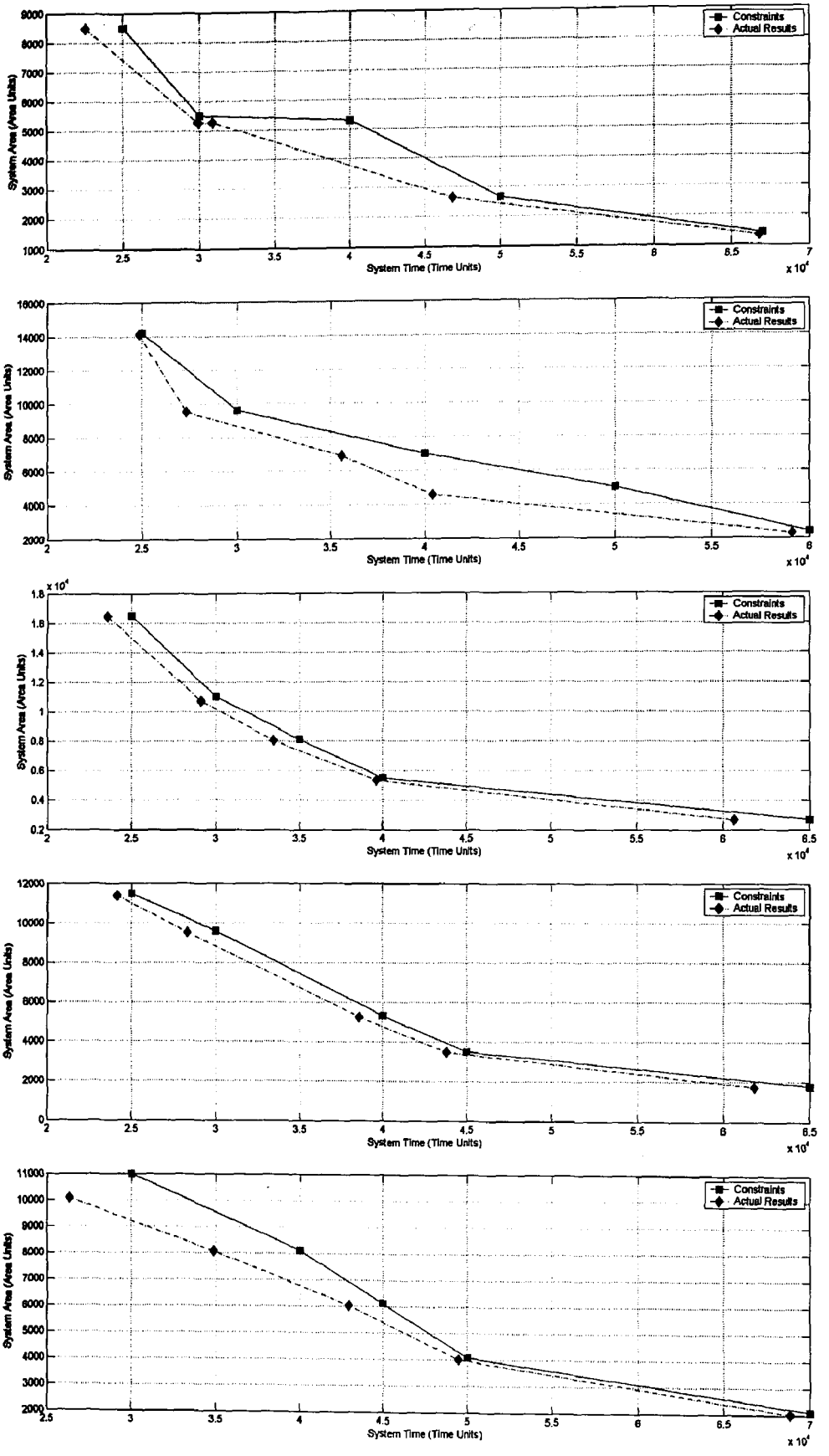

Figure 4.16 Design Space Exploration for 50 Node Graphs

Table 4.9 Time/Area Results for 100 Node Graphs

Graph	Constraints		Results					
	Time	Area	System Time	System Area	PEs	Pipeline Stages	SW PEs	HW PEs
a.	50000	14000	40091	13045	11	11	8	3
a.	70000	6100	62675	6016	4	4	4	0
a.	90000	4600	85038	4512	3	3	3	0
a.	100000	3100	93355	3008	2	2	2	0
a.	135000	1600	134534	1504	1	1	1	0
b.	35000	70000	33824	68035	21	14	19	11
b.	50000	33000	44775	32435	11	10	9	7
b.	70000	15000	62958	14635	6	7	4	3
b.	90000	7500	89201	7120	2	2	2	0
b.	135000	3600	125803	3560	1	1	1	0
c.	40000	26000	36381	25377	11	10	9	2
c.	50000	17500	49736	17028	8	8	6	2
c.	60000	11500	56830	11412	5	5	4	1
c.	80000	5600	79706	5566	2	2	2	0
c.	120000	2800	115622	2783	1	1	1	0
d.	50000	17000	40964	16936	11	11	9	2
d.	60000	12000	53047	11075	7	7	6	1
d.	80000	5500	79409	5433	3	3	3	0
d.	100000	3800	88138	3622	2	2	2	0
d.	130000	1900	125303	1811	1	1	1	0
e.	55000	17000	52583	16504	10	10	7	3
e.	57000	15000	56430	14280	9	9	6	3
e.	65000	12000	60545	11424	6	6	5	1
e.	90000	4500	84821	4448	2	2	2	0
e.	130000	2224	117628	2224	1	1	1	0

Table 4.10 Topology Information for 100 Node Graphs

Graph	Time Constraint	Tree Topology		Mesh Topology		Hypercube Topology		Selected Topology	Extra PEs
		$Overhead_T$	$Links_M$	$Overhead_T$	$Links_M$	$Overhead_T$	$Links_M$		
a.	50000	75759	57	75032	52	57230	48	HYPERCUBE	6
a.	70000	30198	6	19159	4	19159	4	HYPERCUBE	0
a.	90000	24300	2	24300	2	24300	2	TREE	0
a.	100000	0	0	0	0	0	0	TREE	0
a.	135000	0	0	0	0	0	0	TREE	0
b.	35000	169524	151	185060	144	116600	138	HYPERCUBE	11
b.	50000	108692	64	94309	56	80141	53	HYPERCUBE	4
b.	70000	35078	11	26265	11	35065	11	MESH	3
b.	90000	0	0	0	0	0	0	TREE	0
b.	135000	0	0	0	0	0	0	TREE	0
c.	40000	84391	62	85423	55	70341	52	HYPERCUBE	3
c.	50000	62372	29	65579	23	48566	20	HYPERCUBE	3
c.	60000	29133	9	20795	7	29670	8	MESH	0
c.	80000	0	0	0	0	0	0	TREE	0
c.	120000	0	0	0	0	0	0	TREE	0
d.	50000	86426	64	82731	57	65858	54	HYPERCUBE	5
d.	60000	57457	24	60766	22	45394	21	HYPERCUBE	3
d.	80000	17562	2	17562	2	17562	2	TREE	0
d.	100000	0	0	0	0	0	0	TREE	0
d.	130000	0	0	0	0	0	0	TREE	0
e.	55000	83788	38	75404	36	64168	31	HYPERCUBE	5
e.	57000	64198	32	58827	28	53242	27	HYPERCUBE	5
e.	65000	43530	13	47018	13	39613	11	HYPERCUBE	2
e.	90000	0	0	0	0	0	0	TREE	0
e.	130000	0	0	0	0	0	0	TREE	0

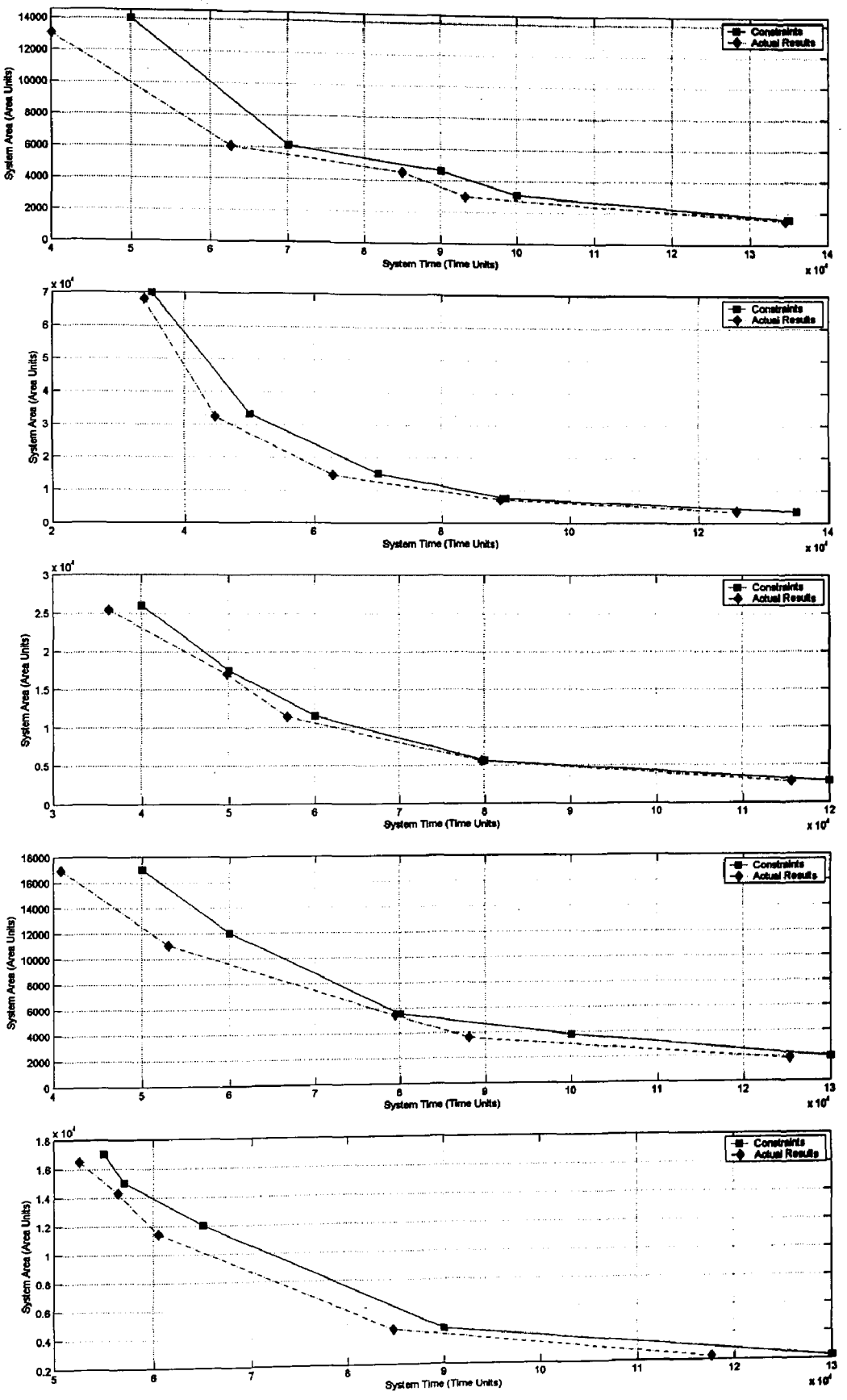

Figure 4.17 Design Space Exploration for 100 Node Graphs

Table 4.11 Time/Area Results for 200 Node Graphs

Graph	Constraints		Results					
	Time	Area	System Time	System Area	PEs	Pipeline Stages	SW PEs	HW PEs
a.	95000	25000	94430	22718	12	12	10	2
a.	125000	12000	121990	11228	6	6	5	1
a.	150000	9000	132613	8908	4	4	4	0
a.	200000	4500	184755	4454	2	2	2	0
a.	260000	2300	256035	2227	1	1	1	0
b.	95000	28000	88896	27673	12	12	10	2
b.	125000	14000	124634	13958	7	7	5	2
b.	170000	8500	155297	8229	3	3	3	0
b.	200000	5500	177806	5486	2	2	2	0
b.	260000	2800	242599	2743	1	1	1	0
c.	100000	23000	98365	22437	9	8	7	2
c.	125000	12500	123889	12336	4	4	4	0
c.	170000	9300	151555	9252	3	3	3	0
c.	200000	6200	173985	6168	2	2	2	0
c.	260000	3084	250705	3084	1	1	1	0
d.	100000	16500	96527	16035	10	10	6	4
d.	125000	11000	116951	10633	7	7	4	3
d.	175000	7500	153308	7470	3	3	3	0
d.	200000	5000	178012	4980	2	2	2	0
d.	260000	2500	250029	2490	1	1	1	0
e.	100000	20000	95929	19850	9	9	8	1
e.	120000	13000	117500	12533	6	6	5	1
e.	125000	10000	124652	9756	4	4	4	0
e.	200000	5000	171165	4878	2	2	2	0
e.	260000	2500	247290	2439	1	1	1	0

Table 4.12 Topology Information for 200 Node Graphs

Graph	Time Constraint	Tree Topology		Mesh Topology		Hypercube Topology		Selected Topology	Extra PEs
		$Overhead_T$	$Links_M$	$Overhead_T$	$Links_M$	$Overhead_T$	$Links_M$		
a.	95000	228059	80	218399	71	171768	66	HYPERCUBE	4
a.	125000	86409	16	96510	12	90295	13	TREE	2
a.	150000	70012	6	45378	4	45378	4	HYPERCUBE	1
a.	200000	0	0	0	0	0	0	TREE	0
a.	260000	0	0	0	0	0	0	TREE	0
b.	95000	204237	78	222386	70	152689	69	HYPERCUBE	5
b.	125000	98766	18	92556	17	96302	15	MESH	3
b.	170000	39507	2	39507	2	39507	2	TREE	0
b.	200000	0	0	0	0	0	0	TREE	0
b.	260000	0	0	0	0	0	0	TREE	0
c.	100000	158977	38	155680	35	120176	34	HYPERCUBE	2
c.	125000	76090	6	47717	4	47717	4	HYPERCUBE	0
c.	170000	35657	2	35657	2	35657	2	TREE	0
c.	200000	0	0	0	0	0	0	TREE	0
c.	260000	0	0	0	0	0	0	TREE	0
d.	100000	146262	44	137651	41	118430	38	HYPERCUBE	3
d.	125000	71213	18	96623	17	71770	14	TREE	1
d.	175000	34123	2	34123	2	34123	2	TREE	0
d.	200000	0	0	0	0	0	0	TREE	0
d.	260000	0	0	0	0	0	0	TREE	0
e.	100000	161684	47	150442	38	134081	39	HYPERCUBE	3
e.	120000	86789	14	87654	13	88005	12	TREE	2
e.	125000	68810	6	43170	4	43170	4	HYPERCUBE	0
e.	200000	0	0	0	0	0	0	TREE	0
e.	260000	0	0	0	0	0	0	TREE	0

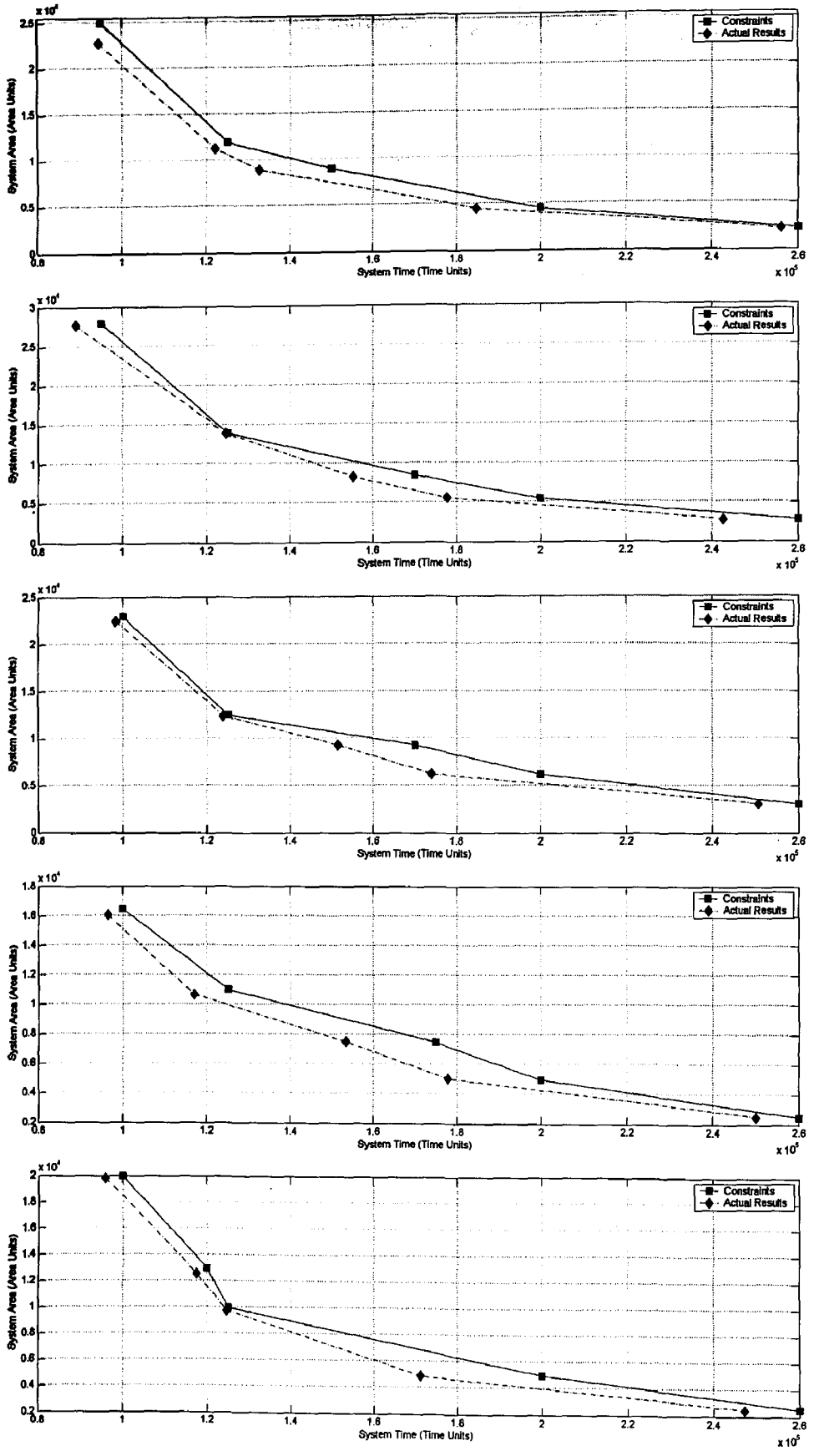

Figure 4.18 Design Space Exploration for 200 Node Graphs

Table 4.13 Time/Area Results for 300 Node Graphs

Graph	Constraints		Results					
	Time	Area	System Time	System Area	PEs	Pipeline Stages	SW PEs	HW PEs
a.	150000	18000	131113	17066	14	14	12	2
a.	200000	8500	183401	8232	6	6	6	0
a.	250000	4200	249678	4116	3	3	3	0
a.	300000	3000	271854	2744	2	2	2	0
a.	400000	1500	377918	1372	1	1	1	0
b.	150000	16000	130661	15930	10	11	10	0
b.	200000	9600	177877	9558	6	6	6	0
b.	260000	6500	203441	6372	4	4	4	0
b.	300000	3200	281363	3186	2	2	2	0
b.	400000	1600	393717	1593	1	1	1	0
c.	150000	15000	147645	14249	9	10	7	2
c.	200000	11000	197525	10164	6	6	5	1
c.	250000	6100	247438	6057	3	3	3	0
c.	300000	4100	259514	4038	2	2	2	0
c.	400000	2019	369897	2019	1	1	1	0
d.	150000	12000	141493	11168	8	8	8	0
d.	200000	5600	194796	5584	4	4	4	0
d.	260000	4200	241119	4188	3	3	3	0
d.	300000	2800	263491	2792	2	2	2	0
d.	400000	1396	372880	1396	1	1	1	0
e.	150000	13500	148745	13134	8	8	7	1
e.	200000	7500	179266	7408	4	4	4	0
e.	250000	5600	223493	5556	3	3	3	0
e.	300000	3800	259033	3704	2	2	2	0
e.	400000	1852	366398	1852	1	1	1	0

Table 4.14 Topology Information for 300 Node Graphs

Graph	Time Constraint	Tree Topology		Mesh Topology		Hypercube Topology		Selected Topology	Extra PEs
		$Overhead_T$	$Links_M$	$Overhead_T$	$Links_M$	$Overhead_T$	$Links_M$		
a.	150000	403973	120	419309	108	301949	102	HYPERCUBE	9
a.	200000	221603	20	178106	16	177289	16	HYPERCUBE	2
a.	250000	67615	2	67615	2	67615	2	TREE	0
a.	300000	0	0	0	0	0	0	TREE	0
a.	400000	0	0	0	0	0	0	TREE	0
b.	150000	358779	72	349114	64	261110	60	HYPERCUBE	5
b.	200000	222590	20	166553	16	178741	16	MESH	2
b.	260000	117737	6	82765	4	82765	4	HYPERCUBE	1
b.	300000	0	0	0	0	0	0	TREE	0
b.	400000	0	0	0	0	0	0	TREE	0
c.	150000	243640	40	245260	33	206090	35	HYPERCUBE	3
c.	200000	155663	14	140544	11	150622	12	MESH	2
c.	250000	67490	2	67490	2	67490	2	TREE	0
c.	300000	0	0	0	0	0	0	TREE	0
c.	400000	0	0	0	0	0	0	TREE	0
d.	150000	250263	42	234790	36	176151	32	HYPERCUBE	3
d.	200000	112324	6	90272	4	90272	4	HYPERCUBE	0
d.	260000	60410	2	60410	2	60410	2	TREE	0
d.	300000	0	0	0	0	0	0	TREE	0
d.	400000	0	0	0	0	0	0	TREE	0
e.	150000	230686	33	216663	30	171106	28	HYPERCUBE	3
e.	200000	90726	6	63284	4	63284	4	HYPERCUBE	1
e.	250000	55373	2	55373	2	55373	2	TREE	0
e.	300000	0	0	0	0	0	0	TREE	0
e.	400000	0	0	0	0	0	0	TREE	0

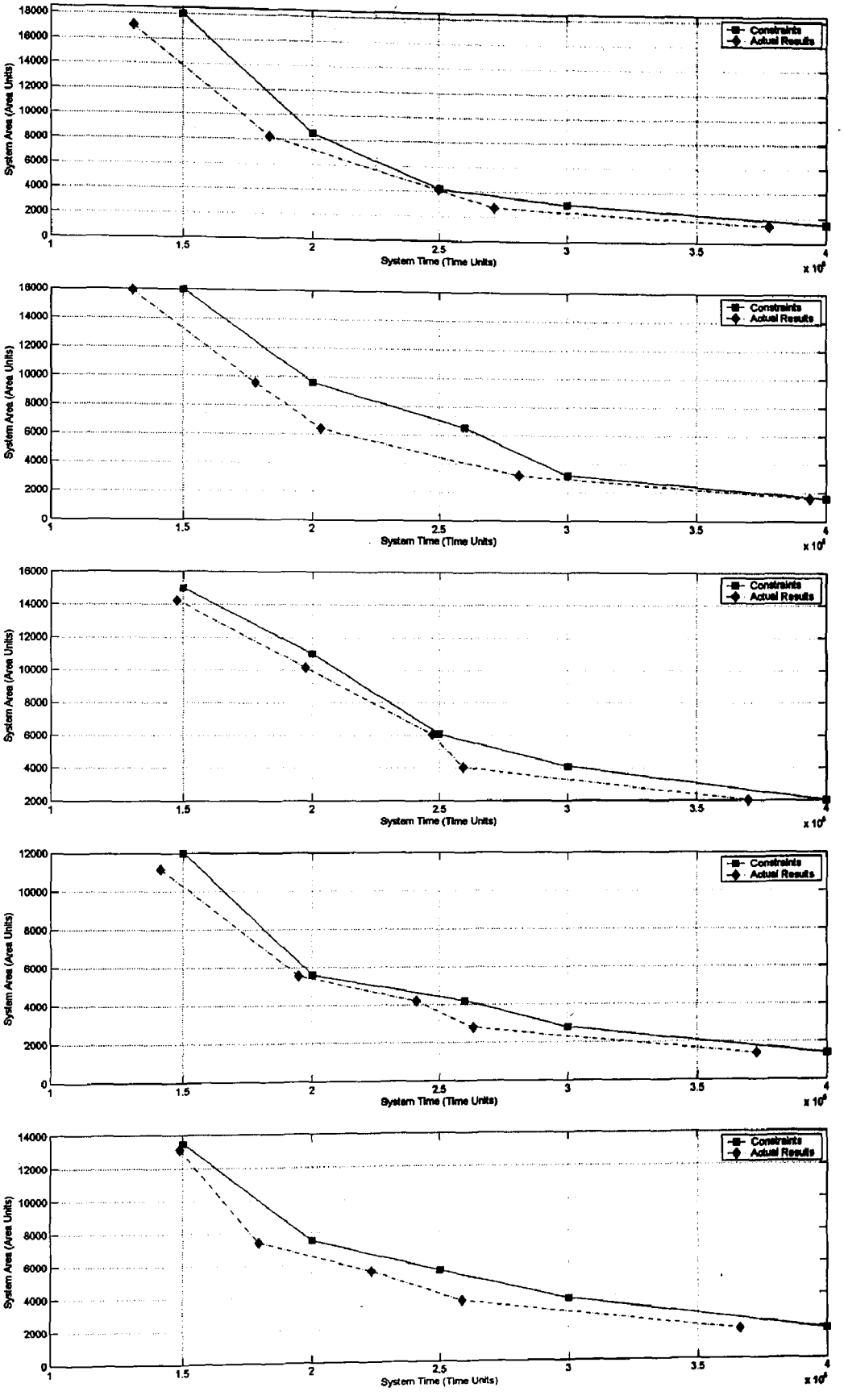

Figure 4.19 Design Space Exploration for 300 Node Graphs

Table 4.15 Time/Area Results for 400 Node Graphs

Graph	Constraints		Results					
	Time	Area	System Time	System Area	PEs	Pipeline Stages	SW PEs	HW PEs
a.	200000	16000	193942	15736	11	10	8	3
a.	250000	11500	223619	11250	6	6	6	0
a.	300000	7500	256960	7500	4	4	4	0
a.	400000	3800	354636	3750	2	2	2	0
a.	500000	2000	496379	1875	1	1	1	0
b.	200000	20000	185507	19199	13	12	8	5
b.	250000	15000	247651	14565	11	10	6	5
b.	300000	9300	262363	9268	4	4	4	0
b.	400000	4700	363667	4634	2	2	2	0
b.	500000	2500	496114	2317	1	1	1	0
c.	170000	23000	166779	22800	10	10	9	1
c.	200000	15300	198317	15267	7	7	6	1
c.	250000	10500	237502	10044	4	4	4	0
c.	350000	5100	334429	5022	2	2	2	0
c.	500000	2600	493521	2511	1	1	1	0
d.	200000	25000	194678	24094	7	7	7	0
d.	230000	21000	207343	20652	6	6	6	0
d.	250000	14000	241836	13768	4	4	4	0
d.	350000	6900	347966	6884	2	2	2	0
d.	500000	3500	497429	3442	1	1	1	0
e.	250000	10050	224331	10017	8	8	6	2
e.	300000	6500	260789	6464	5	5	4	1
e.	350000	4800	335686	4722	3	3	3	0
e.	400000	3200	368259	3148	2	2	2	0
e.	550000	1600	503096	1574	1	1	1	0

Table 4.16 Topology Information for 400 Node Graphs

Graph	Time Constraint	Tree Topology		Mesh Topology		Hypercube Topology		Selected Topology	Extra PEs
		$Overhead_T$	$Links_M$	$Overhead_T$	$Links_M$	$Overhead_T$	$Links_M$		
a.	200000	367748	61	327633	55	300090	49	HYPERCUBE	6
a.	250000	269909	20	218650	16	196887	16	HYPERCUBE	2
a.	300000	148085	6	84897	4	84897	4	HYPERCUBE	1
a.	400000	0	0	0	0	0	0	TREE	0
a.	500000	0	0	0	0	0	0	TREE	0
b.	200000	407296	67	451284	63	331531	57	HYPERCUBE	8
b.	250000	308627	39	317030	41	263956	35	HYPERCUBE	7
b.	300000	165444	6	107113	4	107113	4	HYPERCUBE	1
b.	400000	0	0	0	0	0	0	TREE	0
b.	500000	0	0	0	0	0	0	TREE	0
c.	170000	373574	63	376688	55	300727	53	HYPERCUBE	3
c.	200000	250075	24	260269	22	190527	19	HYPERCUBE	1
c.	250000	139359	6	95387	4	95387	4	HYPERCUBE	0
c.	350000	0	0	0	0	0	0	TREE	0
c.	500000	0	0	0	0	0	0	TREE	0
d.	200000	318971	30	298067	26	235154	24	HYPERCUBE	2
d.	230000	276251	20	197729	16	203177	16	MESH	2
d.	250000	132607	6	86574	4	86574	4	HYPERCUBE	0
d.	350000	0	0	0	0	0	0	TREE	0
d.	500000	0	0	0	0	0	0	TREE	0
e.	250000	290773	28	302624	24	219367	22	HYPERCUBE	3
e.	300000	146109	9	107334	7	146173	8	MESH	2
e.	350000	98506	2	98506	2	98506	2	TREE	0
e.	400000	0	0	0	0	0	0	TREE	0
e.	550000	0	0	0	0	0	0	TREE	0

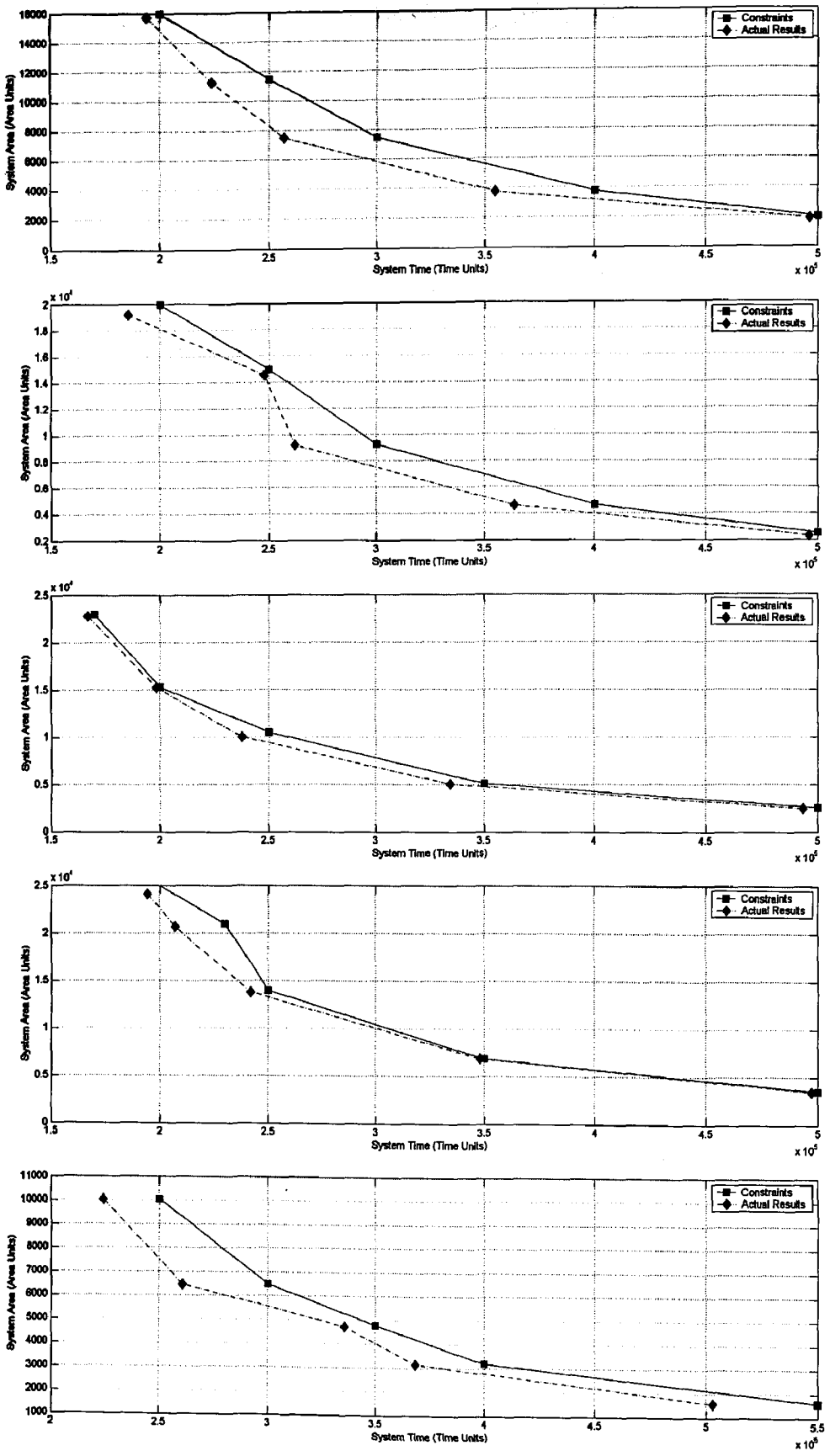

Figure 4.20 Design Space Exploration for 400 Node Graphs

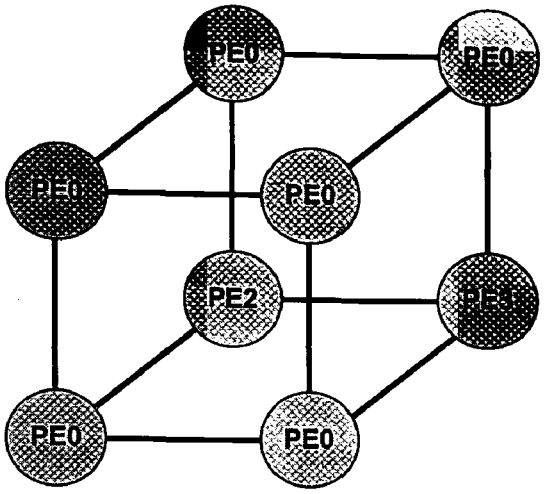

Figure 4.21 Topology Mapping for 400-node Graph (Graph 'e', T_{PERIOD}=100000)

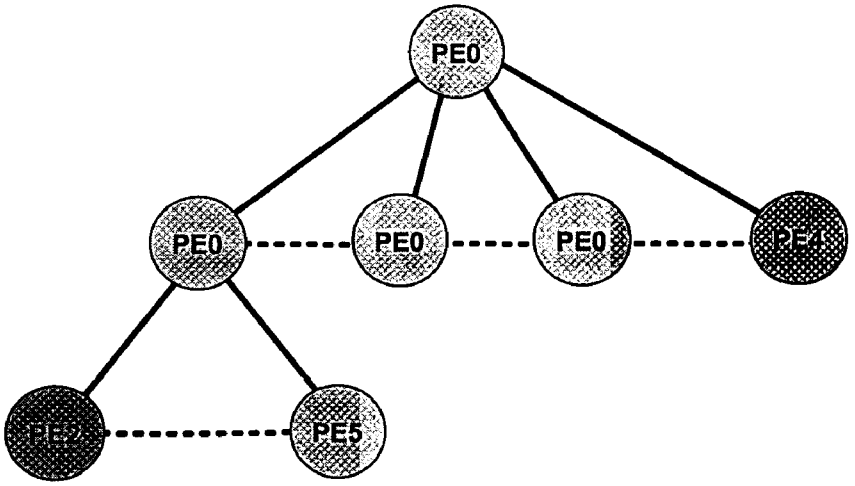

Figure 4.22 Topology Mapping for 200-node Graph (Graph 'd', T_{PERIOD}=125000)

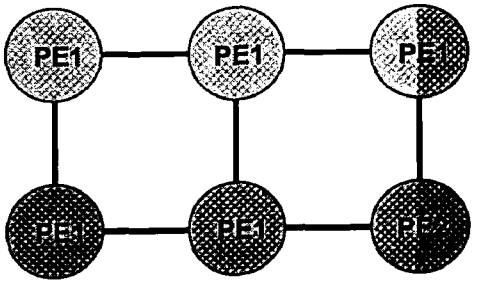

Figure 4.23 Topology Mapping for 300-node Graph (Graph 'c', T_{PERIOD}=200000)

4.5 *Algorithm Execution Time*

Time taken by algorithm to generate a co-synthesized hardware software system is an important criterion for its effectiveness. To study this effect, execution time of the algorithm for each test case was recorded. Algorithm works iteratively thus its execution time is not dependent solely on the number of nodes. Tight constraints require many iterations and therefore take more time. Figure 4.24 shows minimum, average and maximum execution times for graphs with a wide range of tasks. Algorithm was able to provide final output with in 2.5 seconds for largest graph having 400 tasks. Worst case execution time of the algorithm for graphs up to 100 nodes is even less than 500 milliseconds. This clearly shows its usefulness as, for example Vemuri and Chatha's algorithm takes 30 minutes for a 30 node graph [22].

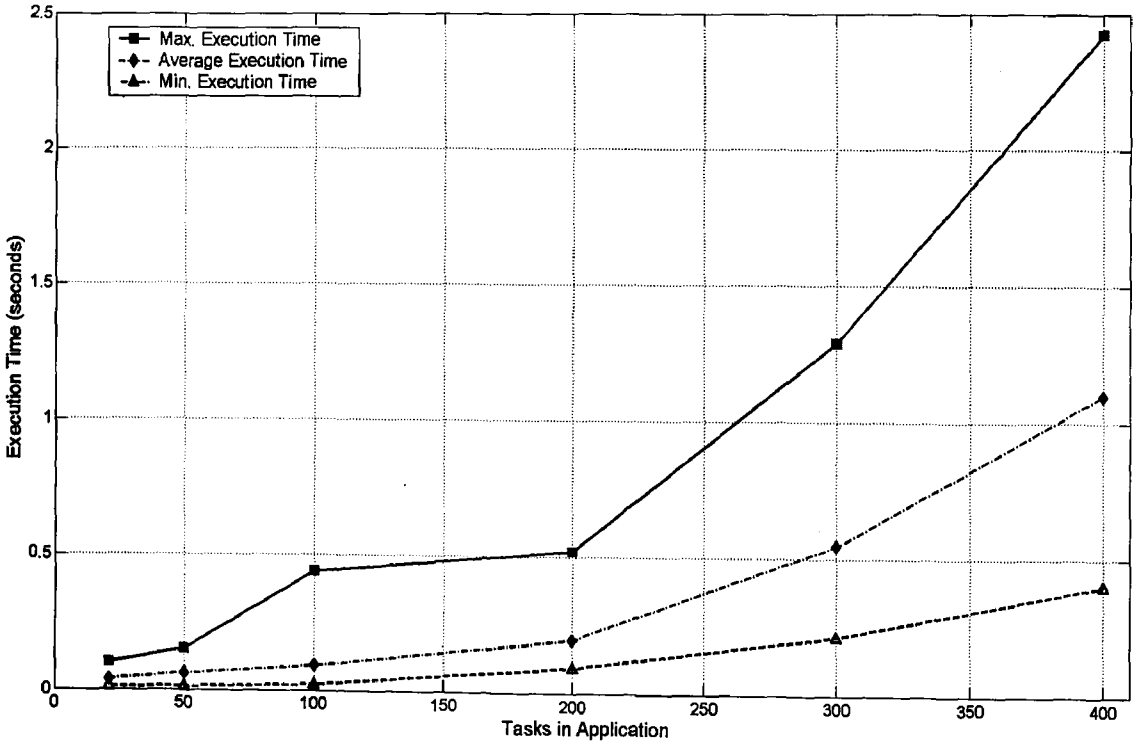

Figure 4.24 Algorithm Execution Time

CHAPTER 5

CONCLUSION AND FUTURE WORK

In this thesis a new co-synthesis algorithm for fault tolerant applications is presented. The algorithm targets regular distributed memory architectures by arranging processing elements in mesh, hypercube and quad-tree topologies. A data flow graph specifying the application, library of heterogeneous processing elements and profile information for each task is provided to the algorithm along with constraints for pipeline period and area cost of the resulting system. The co-synthesis algorithm then operates on these inputs and selects necessary processing elements and creates pipeline stages for task execution. It performs co-synthesis by adding processing elements in the system in an iterative manner. Main phases of the algorithm include processing element selection, pipelined task allocation, topology mapping and scheduling. Processing elements which give maximum performance are added into the system. Tasks are then scheduled on the available processing elements based on their priority and pipeline stages are created when a task cannot be scheduled in the current pipeline stage. When timing constraints are met, the processing elements are mapped to a regular topology. Topology that has the least communication delay is selected. Finally, scheduling is performed to see if all the tasks still meet the timing constraints. More processing elements are added in the system if timing constraints are violated.

Different experiments were conducted to demonstrate the efficacy of the proposed algorithm. In the first experiment, MPEG encoder application has been used for co-

synthesis. Application was tested with a wide range of timing and area constraints and algorithm was able to generate pipelined schedules for each of the test case in a short span of time. Other experiments were conducted on large size random graphs consisting of up to 400 tasks. The algorithm assumes coarse grained task graphs, therefore graph with 400 tasks represent a very large application. Algorithm was able to find good results for each of the test case in a very short time. Algorithm generated results that range from a single processing to tens of processing elements based on performance requirements and it explored the design space well. Different topologies were selected depending on the nature of inter-task communication.

Although proposed algorithm gives good results, it can be improved in a number of ways. Processing elements are currently selected depending on the performance improvement and corresponding area cost. Tradeoff between these conflicting requirements is done using a constant area-performance tradeoff factor. One of the enhancements in this approach could be to dynamically change this factor by deriving it through some system characteristics or performance/area requirements.

Co-synthesis technique presently requires acyclic data flow graphs. Many applications require previous iteration data in tasks that lead to the same data for the current iteration. Modeling such applications with data-flow graphs require cyclic graphs which have feedback edges. The method could be enhanced to handle these applications by using a special edge such that the successor task should not wait for predecessor task to complete

its execution. Data required through these edges is produced in previous iterations and is therefore always available for the current iteration.

Presently, non-functional requirements are specified only as area and pipeline period constraints. More non-functional requirements can be added. Reliability is an important metric for fault-tolerant systems. System reliability can be modeled and the cost function of the algorithm can be modified to include reliability as another constraint. Similarly, system power can also be used as another non-functional requirement.

Finally, another direction for future work could be to substitute reconfigurable logic devices in place of hardware blocks for co-synthesis. If configuration time of such devices happen to be very small compared to the task execution time, then same device can be reconfigured to perform a different task instead of using other processing elements. This can lead to substantial savings in the cost of the target system.

REFERENCES

[1] V. Madisetti, "Rapid digital system prototyping: current practice, future challenges," IEEE Design and Test of Computers, vol. 13, no. 3, pp.12-22, August 1996.

[2] G.D. Micheli, "Computer-aided hardware software codesign," in IEEE Micro, vol. 14, no. 4, pp. 11-16, August 1994.

[3] W. Wolf, "A decade of hardware/software codesign," IEEE Computer vol. 36, no. 4, pp. 38-43, April 2003.

[4] D.E. Thomas, J.K. Adams, H. Schmit, "A model and methodology for hardware-software codesign," IEEE Design and Test of Computers, vol. 10, no. 3, pp. 6-15, September 1993.

[5] S. Kumar, J. Aylor, B. Johnson, W. Wulf, "A framework for hardware/software codesign," IEEE Computer, vol. 26, no. 12, pp. 39-45, December 1993.

[6] M. Chiodo, P. Guisto, A. Jurecska, H. C. Hsieh, A.S. Vincentelli, L. Lavagno, "Hardware software codesgin of embedded systems," IEEE Micro, vol. 14, no. 4, pp. 26-36, August 1994.

[7] J. Henkel, Th. Benner, R. Ernst,W. Ye, N. Serafimov and G.Glawe, "COSYMA: A software–oriented approach to hardware/software codesign," The Journal of Computer and Software Engineering, vol. 2, no. 3, pp. 293–314, 1994.

[8] R. Ernst, J. Henkel, T. Benner, "Hardware/software cosynthesis for microcontrollers," IEEE Design and Test of Computers, vol. 10, no. 4, pp. 64-75, December 1993.

[9] R.K. Gupta and G.D. Micheli, "Hardware-software cosynthesis for digital systems," IEEE Design and Test of Computers, vol. 10, no. 3, pp. 29-41, September 1993.

[10] http://www.systemc.org

[11] T. C. Hu, "Parallel sequencing and assembly line problems," Operations Research, vol. 9, no. 6, pp. 841-848, 1961.

[12] G. DeMicheli, "Synthesis and Optimization of Digital Circuits," McGraw-Hill, 1994.

[13] P. Paulin, J. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 8, no. 6, pp. 661-679, June 1989.

[14] R. Camposano, "Path based scheduling for synthesis," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 10, no. 1, pp. 85-93, January 1991.

[15] R. A. Bergamaschi, S. Raje, I. Nair, L. Trevillyan, "Control-flow versus data-flow-based scheduling: combining both approaches in an adaptive scheduling system," IEEE Transactions on VLSI Systems, vol. 5, no. 1, pp. 82-100, March 1997.

[16] R. Ernst, "Codesign of embedded systems: status and trends," IEEE Design and Test of Computers, vol. 15, no. 2, pp. 45-54, April 1998.

[17] J. Henkel, R. Ernst, "An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques," IEEE Transactions on VLSI Systems, vol. 9, no. 2, pp. 273-289 , April 2001.

[18] K. Melhorn, "Graph Algorithms and NP-Completeness," New York: Springer-Verlag, 1977.

[19] G. Nemhauser, L.Wolsey, "Integer and Combinatorial Optimization," Wiley, New York, 1988.

[20] S. Prakash and A.C. Parker, "SOS: Synthesis of application specific heterogeneous multiprocessor systems," Journal of Parallel and Distributed Computing, vol. 16, pp. 338-351, December 1992.

[21] L.J. Hafer and E. Hutchings, "Bringing up Bozo," Technical Report (CMTR TR-02), School of Computing Science, Simon Fraser University, BC, Canada. March 1990.

[22] K. S. Chatha and R. Vemuri, "Hardware-software partitioning and pipelined scheduling of transformative applications," IEEE Transactions on VLSI Systems, vol. 10, no. 3, pp. 193-208, June 2002.

[23] Aviral Shrivastava, Mohit Kumar, Sanjiv Kapoor, Shashi Kumar, M. Balakrishnan, "Optimal hardware/software partitioning for concurrent specification using dynamic programming," Proceedings of International Conference on VLSI Design, Calcutta, India, pp. 110-113, January 2000.

[24] Jui-Ming Chang, Massoud Pedram, "Codex-dp: Co-Design of communicating systems using dynamic programming," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 19, no. 7, pp. 732-744 , July 2000.

[25] P. V. Knudsen, J. Madsen, "PACE: A dynamic programming algorithm for hardware/software partitioning," Proceedings of 4th International Workshop on Hardware/Software Codesign, Pittsburgh, PA, USA, pp. 85-92, 1996.

[26] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, "Optimization by simulated annealing," Science, vol. 220, no. 4598, pp 671-680, May 1983.

[27] F. Glover, E. Taillard, D. de Werra, "A user's guide to tabu search," Annals of Operations Research, vol. 41, no. 0, pp. 3-28, 1993.

[28] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli, "System level hardware/software partitioning based on simulated annealing and tabu search," Design Automation for Embedded Systems, vol. 2, no. 1, pp. 5-32, January 1997.

[29] R. Dick, N. Jha, "Mogac: a multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 17, no. 10, pp. 920–935, October 1998.

[30] T. Wiangtong, P. Cheung, and W. Luk, "Comparing three heuristic search methods for functional partitioning in hardware-software codesign," Journal of Design Automation for Embedded Systems, vol. 6, pp. 425-449, 2002.

[31] J. Henkel and R. Ernst, "A path-based technique for estimating hardware runtime in HW/SW-cosynthesis," Proceedings of 8th International Symposium on System Synthesis, Cannes, France, pp. 116-121, 1995.

[32] J. Henkel and R. Ernst, "High level estimation techniques for usage in hardware/software co-design," Proceedings of Asia South Pacific Design Automation Conference (ASPDAC '98), Yokohama, Japan, pp. 353-360, 1998.

[33] F. Vahid and D.D. Gajski, "Incremental hardware estimation during hardware/software functional partitioning," IEEE Transactions on VLSI Systems, vol. 3, no. 3, pp. 459-464, September 1995.

[34] C-Y. Park and A.C. Shaw, "Experiments with a program timing tool based on a source-level timing scheme," Computer, vol. 24, no. 5, pp. 48-57, May 1991.

[35] Y-T. Li, S. Malik, and A. Wolfe, "Performance estimation of embedded software with instruction cache modeling," ACM Transactions on Design Automation of Electronic Systems, vol. 4, no. 3, pp. 380-387, July 1999.

[36] W. Ye, R. Ernst, T. Benner, and J. Henkel, "Fast timing analysis for hardware-software co-synthesis," Proceedings IEEE International Conference on Computer Design (ICCD '93), Cambridge, MA, USA, pp. 452–457, 1993.

[37] J. Buck, S. Ha, E. A. Lee, D.G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," International Journal of Computer Simulation, vol. 4, pp. 155-182, April 1994.

[38] L.A. Cortes, P. Eles, Z. Peng, "Verification of embedded systems using a Petri net based representation," Proceedings of 13th International Symposium on System Synthesis, Madrid, Spain, pp. 149-155, September 2000.

[39] A. Ghosh, M. Bershteyn, R. Casley, C. Chien, A. Jain, M. Lipsie, D. Tarrodaychik, O. Yamamo, "A hardware-software co-simulator for embedded system design and debugging," Proceedings of Asia South Pacific Design Automation Conference (ASPDAC '95), Makuhari, Japan, pp. 155-164, September 1995.

[40] P. A. Hsiung, "Hardware-software timing co-verification of concurrent embedded real-time systems," IEE Proceedings Computers and Digital Techniques, vol. 147, no. 2, pp. 83-92, March 2000.

[41] W. Wolf, "An architectural co-synthesis algorithm for distributed, embedded computing systems," IEEE Transactions on VLSI Systems, vol. 5, no.2, pp. 218-229, June 1997.

[42] K. Konstantinides, R. Kaneshiro, and J. Tani, "Task allocation and scheduling models for multi-processor digital signal processing," IEEE Transactions on Acoustics, Speech, Signal Processing, vol. 38, no. 12 pp. 2151–2161, December 1990.

[43] G. C. Sih, E.A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," IEEE Transactions on Parallel Distributed Systems, vol. 4, no. 2, pp. 175–187, February 1993.

[44] S. Bakshi, D.D. Gajski, "Paritioning and pipelining for performance-constrained hardware/software systems," IEEE Transactions on VLSI Systems, vol. 7, no. 4, pp. 419-432, Dec. 1997.

[45] A. Kalavade, E.A. Lee, "The Extended Partitioning Problem: Hardware/Software Mapping, Scheduling and Implementation-bin Selection," Journal of Design Automation of Embedded Systems, vol. 2, no. 2, pp. 125-163, March 1997.

[46] J. Levman, "Hardware software co-synthesis of heterogeneous hypercube architectures for fault tolerant embedded systems," MASc. Thesis, Dept. of Electrical and Computer Engineering, Ryerson University, 2004.

[47] S. Yajnik, S. Srinivasan, N. K. Jha, "TBFT: A Task Based Fault Tolerance Scheme for Distributed Systems," Proceedings of International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV, USA, pp. 483-489, October 1994.

[48] B. P. Dave, N. K. Jha, "COFTA: Hardware-Software Co-Synthesis of Heterogeneous Distributed Embedded Systems for Low Overhead Fault Tolerance," IEEE Transactions on Computers, vol. 48, no. 4, pp. 417-441, April 1999.

[49] Ralph Duncan, "A Survey of Parallel Computer Architectures," Computer, vol. 23, no.2, pp. 5-16, Feb. 1990.

[50] A.M. Despain, D. A. Patterson, "X-Tree: A tree structured multi-processor computer architecture," Proceedings of the 5th annual symposium on Computer architecture, Palo Alto, CA, USA, pp. 144-151, April 1978.

[51] T. Yen, W. Wolf, "Performance Estimation for Real-Time Distributed Embedded Systems," IEEE Transactions on Parallel and Distributed Systems, vol. 9 no. 11, pp. 1125-1136, November 1998.

[52] Didier Le Gall, "MPEG: A video compression standard for multimedia applications," Communications of the ACM, vol. 34, no. 4, pp. 46-58, April 1991.

[53] Usman Ahmed, Gul N. Khan, "A new processor allocation and pipelining approach for hardware software co-synthesis," Proceedings of the 18th annual Canadian Conference on Electrical and Computer Engineering (CCECE'05), Saskatoon, SK, Canada, May 2005.