Chapter 1

Introduction

Improvements in computer performance have traditionally been the result of higher clock speed, pipelining, cache memories, multiple execution units, hardwired control units, and RISC. With the appearance of re-configurable devices such as field programmable gate arrays (FPGAs), a new means for increasing computer performance has become possible. Systems using these re-configurable devices, such as configurable computing machines, need compilers to configure their hardware resources into architectures specialized for the tasks to be executed on them. There is a need for building compilers that can handle a wide variety of input tasks and at the same time make use of the available hardware resources in the most efficient manner possible. High-quality partitioning is necessary to achieve acceptable utilization of the re-configurable resources [20]. The goal of this thesis is to look at the partitioning problem for configurable computers (that use re-configurable devices such as FPGAs) and present algorithms to solve three variants of the partitioning problem specific to configurable computing machines.

1.1 Motivation

By using architectures that are optimized for a specific task, application specific computers can provide significant performance improvements over general-purpose computers. Custom computing machines (CCMs) utilize re-configurable devices such as field programmable gate arrays (FPGAs) to adapt to new tasks. By programming the re-configurable resources, CCMs make application specific computers more feasible. Often CCMs are implemented with static RAM based FPGAs and an inter-connection resource for a multiple FPGA network. For example, Splash-2 [35] is a CCM containing sixteen processing elements (PE), each of which contains a Xilinx XC4010 FPGA [33] and an

SRAM. Each PE is connected to its neighboring PEs and a central crossbar. The crossbar is configurable and has the capability of allowing any PE to communicate with any other PE. A CCM may also have a general purpose CPU and perhaps a specialized chip such as a Digital Signal Processor (DSP). The primary benefit of the re-configurable devices such as FPGAs is that the resulting hardware is neither rigid nor permanent, and the hardware is programmable. An alternative to CCMs is Application Specific Integrated Circuits (ASIC) which are designed for each specific application. An ASIC implementation can be somewhat faster than the corresponding CCM implementation; however, each ASIC implementation requires the fabrication of new hardware. A CCM is built with commercial off-the-shelf (COTS) parts and can be shared by many applications. The current generation requires only tens of milliseconds to re-configure for a new application specific computation [33]. Configurable computers based on FPGAs are capable of accelerating suitable applications by a factor of 10 to 1000 [14, 16, 17] when compared to traditional processor-based architectures. Further, the use of a conventional RISC processor with a re-configurable processing element could prove to be more cost-effective than existing CCMs that have five times as many FPGAs [18] [19].

Examples for contemporary CCMs are the Tower of Power [37] and the SLAAC project [38]. The structures of these CCMs in general will have heterogeneous and distributed configurable computing architectures for use in multiple applications. The general structure of such a CCM is shown in Figure 1.1.

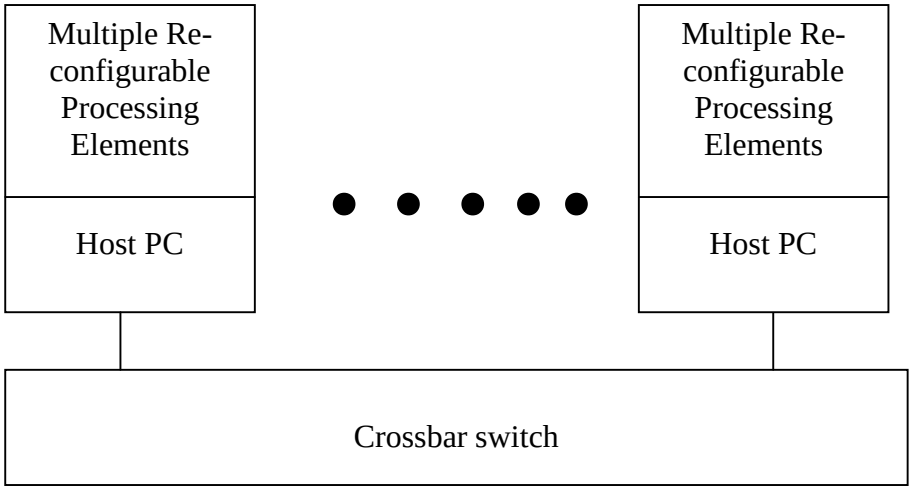

Figure 1.1. General structure of a contemporary CCM.

Each unit of the CCM in Figure 1.1 contains multiple processing elements (re-configurable devices, general-purpose CPUs, or special-purpose DSPs in conjunction with a host PC. There may be several such units connected to each other via a network. The communication costs between the host PC and the embedded PEs are typically lower than the communication costs between PEs of different units.

Applications developed on CCMs include RSA cryptography, molecular biology, video compression, and image classification [17] [34]. There is a need for highly efficient and optimizing compilers to take advantage of the features of re-programmable hardware to produce high performance scalar and parallel architectures. New and complex intermediate program representations and performance models are needed to identify when an optimizing transformation is legal and profitable [32]. To capture the complexity of the target system and to make the right tradeoff decisions, compiler optimization problems have to be solved using different program models and a very efficient algorithm [32]. The resulting compiler optimization problems are most likely NP-complete [23].

The design of algorithms, tools, and software to compute efficient realizations of application programs on CCMs is the goal [17] [20]. This also includes the goal to automatically partition the modules across the CCM and then intelligently multiplex the communication links to optimize the throughput and/or total execution time. At the same time the modules to be executed have to fit onto the limited programmable hardware resources.

1.2 The problem

In this thesis we address the problem of partitioning the modules onto a configurable computing system. The problem of optimally assigning the modules of a parallel program over the processing elements (PE) of a configurable computing system is addressed. In particular the following problems are solved using a polynomial time algorithm under certain constraints:

a) The problem of partitioning chain-structured parallel and/or pipelined programs across chain-structured PEs is solved under area and pin constraints.

b) The problem of partitioning chain-structured parallel and/or pipelined programs across a limited number of processing units of a CCM whose units are connected in any pattern is solved under area, pin, and power constraints. The optimization criteria in these cases are minimizing execution times and communication times between the communicating modules. Emphasis is placed on imposing constraints, such as the area occupied by the partitioned modules, the pin limitation of the individual computing units, and the power dissipation of these modules.

c) The problem of partitioning a parallel and/or pipelined program containing a single fork to be mapped onto a CCM whose units are connected in a chain-like fashion is also solved under the constraints as mentioned for the previous case. Specifically, the module graph of the program having a single branch assumes the shape of an inverted Y.

1.3 The approach

These results extend prior research in the area of distributed computing as regards to the algorithms and solution methods employed. They also permit the efficient utilization of re-configurable devices for a wide range of problems of practical interest.

The main idea employed in solving these problems comes from Bokhari's solution for partitioning problems in parallel, pipelined, and distributed computing [1]. Unless otherwise stated, all references to Bokhari refer to [1]. A sum bottleneck path algorithm is developed that permits the efficient solution of many variants of this problem under some constraints on the structure of partitions for all three cases mentioned in Section 1.3. The algorithms used for all three cases are based on the same approach.

1.4 Thesis organization

The thesis is organized as follows. Problem formulation, prior research in partitioning in the case of FPGAs, and background for the research is described in Chapter 2. This chapter also contains a description and illustration of Bokhari's algorithm for solving the chain-structured programs partitioned over processing elements connected

in a chain-like fashion. Chapter 3 contains a description of the adaptation of Bokhari's algorithm to the case of a chain of processing units, which are part of the configurable computing machines. Chapters 4 and 5 contain descriptions of the new algorithms developed for the partitioning problems for CCMs. The algorithm presented in Chapter 4 is used to solve the second problem in Section 1.2. Chapter 5 contains a description of the algorithm for the third problem in Section 1.2. Chapter 6 contains the results for all three algorithms. This chapter also has a description of the testing done for verification of the algorithms that were implemented. The results are obtained for a specific structure of a CCM shown in Figure 1.1. Chapter 7 contains the conclusions on partitioning in the case of CCMs and on the algorithms presented in this thesis in particular. Suggestions for further improvement in the methods for partitioning in the case of CCMs are also presented.

Chapter 2

Background

This chapter contains a description of the definitions and terms used in this thesis, followed by partitioning methods currently used for re-configurable devices. The partitioning problem in distributed computing is presented. This is followed by a review of some of the important methods and algorithms for partitioning in the case of distributed computing systems. Next, the chapter describes in detail Bokhari's method for partitioning problems in parallel, pipelined, and distributed computing.. Finally, the chapter describes why Bokhari's method has been chosen as the basis for the algorithms presented in Chapters 4 and 5.

2.1 Definitions, terms, and model used in the thesis

A formal definition of the terms used in this thesis and the graph modeling of the problem are included in this section. The implementation of an application such as image and signal processing on a distributed or configurable computer involves partitioning. These applications are represented as a set of modules, based on their functionality. Each of these modules has moderate to large granularity. The relationship between these modules is given by the module graph. The number of modules and the number of processing elements (PE) are given by m and n respectively. All problems that are dealt with in this thesis, except the problem in Chapter 5, consist of a chain of modules. Hence, the model presented will be for a chain of modules. In Chapter 5 of the thesis, the model will be modified to reflect the change in the module inter-connection pattern.

Definitions:

Module: Module is a term used to represent the individual smaller components (hardware or software) of an application with some known granularity. A module graph can be represented by a directed graph $G_m = (V, E)$, showing precedence relations. The set of vertices V, each of which represents a single module, is given by $V = \{v_1, v_2, v_3 \ldots v_i \ldots v_m\}$, where m is the total number of modules. The set of edges $E \subseteq V \times V$ represents the inter-module communication [5].

For example, typical processing steps in a communication system consist of a Fourier transform, frequency multiplication, and a band-pass filter in a pipelined fashion. Each step can be viewed as a module. The module graph is connected in series. Another example consists of modules of smaller granularity connected in a precedence relation as shown in Figure 2.1.

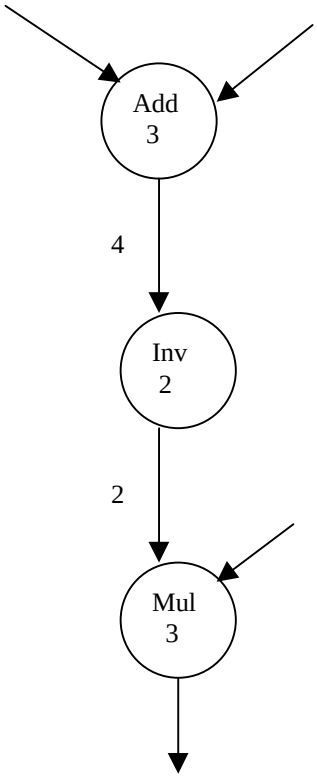

Figure 2.1. A module graph.

The weights on the nodes of G_m given by $T_e = \{t_{e1}, t_{e2}, t_{e3}... t_{ei}... t_{em}\}$ represent the individual execution times of the modules on a reference processor. The weights on the edges of G_m given by $D = \{d_{1,2}, d_{2,3}, d_{3,4}... d_{i-1,i}... d_{m-1,m}\}$ represent the amount of data transferred between the modules; for example, $d_{i-1,i}$ represents the amount of data transferred between modules v_{i-1} and v_i. For the module graph of Figure 2.1, $d_{1,2}=4$, $d_{2,3}=2$, $t_{e1}=3$, $t_{e2}=2$, and $t_{e3}=3$.

Processing Element (PE): The individual computing units of the CCM such as an FPGA, a CPU, or a DSP are referred to as processing elements. Similar to the module graph, the PEs can be represented by a PE graph $G_p=(W,F)$. The vertices of this PE graph are given by $W=\{w_1,w_2...w_i...w_n\}$. W is a set of vertices each of which represents a single PE, and n represents the total number of PEs. $F \subseteq V \times V$ is a set of edges representing the communication links between PEs. For example, Figure 2.2 represents a PE graph G_p, with four PEs connected with links. The weights on the links represent the cost of inter-PE communication. These are denoted as $C = \{c_{1,2}, c_{2,3}....c_{i-1,i}...c_{n-1,n}\}$. For the PE graph of Figure 2.2, $c_{1,2}=5$, $c_{1,3}=1$, $c_{3,4}=3$, $c_{1,4}=3$, and $c_{2,4}=6$.

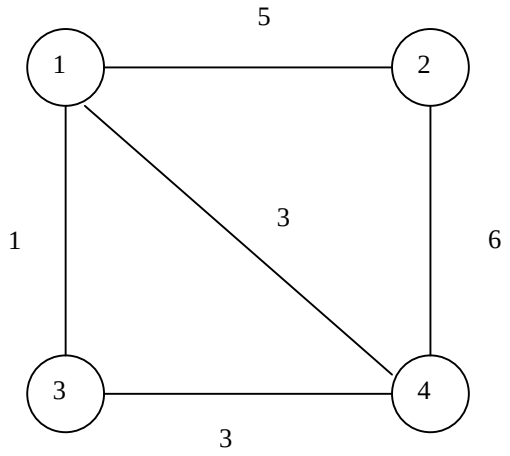

Figure 2.2. Four PEs connected by links.

The weights on the nodes represent the scaling factor to be applied to the execution times for any module being executed on that PE. The scaling factor represents

the factor by which this PE's execution rate differs from the reference PE. These are given by $SF=\{sf_1, sf_2...sf_i...sf_n\}$.

Partitioning: The process of decomposition of an application into smaller modules and grouping them in such a way so that they can be mapped onto individual PEs while satisfying the constraints and meeting objective functions is called partitioning. If n denotes the number of partitions, then partitioning divides V into n disjoint sets of vertices such that [5]

$$\bigcup_{i=1}^{n} V_i = V_n. \tag{2.1}$$

Trail: A trail is a sequence of vertices and alternating edges in the graph G such that each pair of adjacent vertices in the sequence are adjacent in the graph G and all edges in the trail are unique [36].

Modeling the problem as a graph:

The partitioning problem is to group all v_i into n sets and assign each grouping to one of the n PEs so as to minimize the computation's completion time. The model of inter-connection for the PEs will be different for the three different cases considered in the thesis and will be stated in later chapters. The problem becomes nontrivial when the modules are allowed to have individual execution times and require an explicit communication cost for partitioning communicating modules that communicate different different PEs. A PE's time during the computation is spent executing modules, communicating results, or waiting for results so that it can continue. Under any partitioning there will be at least one "bottleneck" PE that limits the computational rate. We seek the partitioning that maximizes the throughput obtained from the PEs. The other objective is to minimize the total execution time of all the modules. The weight of the nodes, which is based on the execution and communication times of the modules in the assignment graph, is calculated as follows for the case of the FPGA, CPU, or DSP. The execution of v_i needs data from the modules which immediately precede it in the module graph G_m. In the case of the graph G_m that is a chain, execution of v_i requires data only

from v_{i-1}. Let w_k denote some PE and if v_i, $v_{i+1}\ldots v_j$ are the modules residing on w_k, then the notation $<(i, j), k>$ represents the modules v_i through v_j residing on PE w_k. The time cost of $<(i, j), k>$ is $S_{i, j, k}$. If w_k is a CPU or if the time cost represents the total execution time for an FPGA,

$$S_{i, j, k} = (\sum_{l=i}^{j} t_{lk}). \tag{2.2}$$

Otherwise, if w_k is an FPGA for calculation of throughput,

$$S_{i, j, k} = (\max_{l=i}^{j} t_{lk}), \tag{2.3}$$

where, t_{lk} is the time to execute v_l on w_k and,

$$t_{lk} = t_{el} \times sf_k. \tag{2.4}$$

The communication costs for the same $<(i, j), k>$ are

$$C_{i,j,k} = d_{i-1, i} \times c_{k-1,k}, \tag{2.5}$$

where $d_{i-1, i}$ represents the amount of data exchanged between modules v_{i-1} and v_i, and, $c_{k-1,k}$ represents the cost of the communication link between PEs w_{k-1} and w_k. The term given by $C_{i,j,k}$ represents the time taken for communication between modules v_{I-1} and v_I residing on PEs w_{k-1} and w_k respectively. Total cost $T_{i,j,k}$ which includes both execution and communication cost is

$$max(S_{i,j,k} , C_{i,j,k}), \text{ for throughput, and} \tag{2.6}$$

$$S_{i,j,k} + C_{i,j,k} \text{ for total execution time} \tag{2.7}$$

If the assignment is such that $<(1,j_1),1>$, $<(i_2,j_2),2>$ ... $<(i_n,m),n>$, then for some k, $T_{i,j,k}$ is largest in this assignment. This is defined as the bottleneck T_b, where

$$T_b = max(T_{i1,j1,1} , T_{i2,j2,2} , \ldots\ldots, T_{in,jn,n}). \tag{2.8}$$

The throughput, which is the rate of receiving the output from the PEs, is determined by T_b, where

$$Throughput = 1/ T_b. \quad (2.9)$$

The total execution time, the sum of execution times of all modules is

$$Total\ execution\ time = T_{i1,j1,1} + T_{i2,j2,2} + ,\ldots\ldots, + T_{in,jn,n}. \quad (2.10)$$

The area of module v_i is denoted by a_i. The maximum area allowable on w_k is denoted by $Size_k$. The important difference between partitioning for only CPUs and partitioning for PEs that may be CPUs, FPGAs, and DSPs is the requirement of the imposition of constraints on area, pin, and power as well as the concurrent nature of the FPGAs. The total area is the sum of the areas of the modules *i* through *j* residing on processor *k*. If this area exceeds the area supported by the processor, then that assignment cannot be made in the assignment graph. In the case of FPGAs the area constraint is imposed as

$$Area\ (i,j) \leq Size_k, \quad (2.11)$$

for some PE *k*, where

$$Area\ (i,j) = \sum_{l=i}^{j} a_l. \quad (2.12)$$

A terminal count or total number of input/output pins is

$$Count\ (i, j). \quad (2.13)$$

The maximum number of terminals that a PE can have is denoted as T_i. Pin constraint is applied by

$$Count\ (i, j) \leq T_k. \quad (2.14)$$

The power constraint is similar to area constraint in that the sum of the power consumption of the modules should not exceed a value on that processor. The power constraint is applied as

$$\sum_{l=i}^{j} p_l \leq P_k, \tag{2.15}$$

where p_l is power consumed by v_I and P_k is the power limitation of PE w_k. The constraints and objective functions for the partitioning problems considered in this thesis are listed next:

Objective functions:

1. *Throughput:* This is an objective function which is to be maximized by the partitioning algorithm. This objective function can be stated mathematically as [1] [10].

$$\textit{Obj 1:}\ \text{maximize}\ (\textit{Throughput}). \tag{2.16}$$

2. *Total execution time:* This is, again, an objective function which has to be minimized by the partitioning algorithm [1] [10] [23]. Total execution time includes the total time for the execution of all the modules and the time for all communication during the execution of those modules.

$$\textit{Obj 2:}\ \text{minimize}\ \textit{(Total execution time)} \tag{2.17}$$

Constraints:

1. *Area of each partition*: The area of each partition is constrained by the area of the PE [23]. This is expressed as

$$\textit{Area}\ (i, j) \leq \textit{Size}_k, \tag{2.18}$$

where *i* through *j* are some modules residing on PE w_k.

2. *Number of terminals or pin constraint*: Partitioning algorithms must partition the modules so that the number of pins required to connect two communicating modules residing on two different PEs should not exceed the terminal count of the PEs [23]. This constraint can be expressed as

$$Count\ (i,j) \leq T_k, \tag{2.19}$$

where i through j are some modules residing on PE w_k.

2. *Power limitation:* The sum of the power consumption for each module residing on a PE should be less than the maximum power capacity of that PE and is expressed as

$$\sum_{l=i}^{j} p_i \leq P_k, \tag{2.20}$$

where p_i is the power consumed by v_i and P_k is the power limitation of w_k.

2.2 Past and present partitioning methods used for FPGAs

Classification of partitioning algorithms:

The general partitioning problem is a well-known NP-complete problem [21] [22]. Therefore, a number of heuristic algorithms have been proposed, such as the Fiduccia-Mattheyses (FM) [2] and Kernighan-Lin [24]. In general, the algorithms can be classified broadly as a) *Optimal algorithms* that give optimal solutions under certain conditions and constraints in polynomial time such as [1], and *b) Heuristic algorithms* that give near optimal solutions under certain conditions such as [2] [24].

Partitioning algorithms, as classified by Sherwani [23], are presented as follows. The availability of an initial partition gives rise to the classification of these algorithms as *constructive algorithms.* The inputs to a constructive algorithm are the circuit components and the netlist. The output is a set of partitions and the new netlist. Constructive algorithms are typically used to form some initial partitions, which can be improved by using other algorithms. In that sense, constructive algorithms are used as preprocessing algorithms for partitioning. They are usually fast, but the partitions generated by these algorithms may be far from optimal. *Iterative algorithms* [2] [24], on the other hand, accept a set of partitions and a net-list as input and generate an improved set of partitions with the modified net-list. These algorithms iterate continuously until the partitions cannot be improved further.

Another type of classification is based on the nature of the algorithms. *Deterministic algorithms* produce the same solutions for the same input every time [1][2][3][5][24]. *Probabilistic algorithms* are capable of producing different solutions for the same problem each time they are used. The reason for this is the use of random functions as in [25].

Partitioning can also be classified based on the process used for partitioning. *Group migration algorithms* [2][24] start with some partitions, usually generated randomly, and then move components between partitions to improve the partitioning.

Simulated annealing/evolution algorithms [25] [26] carry out the partitioning process by using a cost function that classifies any feasible solution. The partitioning process also uses a set of moves that allows movement from solution to solution.

Kernighan and Lin [5] proposed a graph bi-sectioning algorithm for a graph, which starts with a random initial partition and then uses pair-wise swapping of vertices between partitions until no improvement is possible. Schweikert and Kernighan [27] proposed the use of a net model so that the algorithm can handle hyper-graphs. Fiduccia and Mattheyses [2] reduced the time complexity of the K-L algorithm to *O(t),* where *t* is the number of terminals (number of links for a node in the graph). Kring and Newton [28] presented an algorithm using a vertex-replication technique to reduce the number of nets that cross the partitions. Goldberg and Burstein [29] suggested an algorithm, which improves upon the original K-L algorithm by using graph matching. One of the problems with the K-L algorithm is the requirement of pre-specified sizes of partition [23]. Wei and Cheng [30] proposed a ratio-cut model in which the sizes of the partitions do not need to be specified. The group migration algorithms are used extensively in partitioning VLSI circuits.

The most common method used is the Fiduccia-Mattheyses (FM) [2] partitioning. This method consists of multiple passes. In each pass, random partitions are improved iteratively by cell (node) movements from one partition to the other until no further improvement in cut-set size is observed. Many passes (each begins with a unique random starting point) are usually needed to produce good results, particularly when circuits are large. Out of the many passes, only the best pass results are saved. Information captured by all the other passes is lost.

Many modifications/improvements of the FM method are available. Krishnamurthy [3] suggested the use of gain vectors instead of single values in order to reduce ambiguity when choosing cells to move. Hagen et. al [31] introduced locality in cell selection by using the LIFO queue as the data structure, instead of a simple linked list. Although each has its own benefit, neither of them improves partitioning significantly, largely because they do not introduce any means for the algorithm to climb over steep hills and valleys where local minima reside [32].

Another partitioning algorithm suggested by Kernighan and Lin [5] is more successful in improving the results of FM [2]. The KLFM [23] algorithm divides the partitions *{A, B}* generated by FM into 4 sub-partitions $\{A_0, A_1, B_0, B_1\}$ by applying FM partitioning to *A* and *B*. These 4 sub-partitions are swapped and merged to form new initial partitions *{C, D}* for another pass of FM, i.e.,

$$C=A_0 \cup B_0,\ D=A_1 \cup B_1 \text{ or } C=A_0 \cup B_1,\ D=A_1 \cup B_0 \qquad (2.21)$$

This procedure of sub-dividing, swapping and re-partitioning is continued until no further improvement of the cut-set is observed. This method produces much better results per iteration because it explores different combinations of sub-partitions to form initial partitions, instead of just a single trial of random partition assignment. By doing so, this method is effectively exploring different ways of constructing the partitioning tree at the top.

2.3 Comparison of distributed and configurable computing systems

A distributed computing system has processors, which need not be identical. Some processors have unique capabilities in the network [4]. For example, a specific processor may have a hardware floating-point unit and thus be able to carry out floating-point operations with higher speed than a processor without such hardware. Similarly, some processors may be able to do byte manipulation more efficiently than others. Although all processors may be dissimilar, each module is able, in general to execute on any processor. This is possible if all modules are written in a high-level language and separate object versions of these modules are available for each of these processors. The modules of the program are to be assigned to the processing units such that the sum of the execution and communication costs is minimized [4]. A configurable computing system, on the other hand, is made to have architectures that are optimized for specific modules. CCMs usually provide significant performance improvements over general-purpose computers [15]. CCMs are implemented with re-configurable devices, and an inter-connection resource for the multiple re-configurable devices. A CCM may also have a CPU and, perhaps, a specialized chip such as a Digital Signal Processor (DSP).

The execution time of all the modules on a processor in a distributed system is the sum of the execution times of each of the modules because they are executed on a time-sharing basis on the processor. However, the modules executing on a re-configurable device such as an FPGA will execute concurrently because the FPGA can be configured to do so. Hence, the total execution time of all the modules on an FPGA is the highest of the individual execution times of the modules residing on that FPGA. In other words, the total time of execution is the time of the bottleneck module on that unit and is given by Equation 2.3.

When dealing with processors, there is no worry about fitting certain modules on a processor because any number of modules can be executed on a processor on a time-sharing basis. However, in the case of FPGAs, the modules have to fit inside them so that they can be configured to be executed concurrently. Hence, the area constraint has to be applied for CCMs given by Equation 2.11. Note that the method employed by Athanas [37] can be used to alleviate this constraint.

Another important point to be considered in the case of CCMs is the pin limitation. Consider the case of a group of modules concurrently executing on an FPGA. If a majority of these modules have communication outside the FPGA, then there is a severe pin limitation. Note that the method employed for the virtual wires project at M. I. T. can alleviate this constraint [14].

2.4 Related work in distributed computing systems

Configurable computing systems are similar to distributed computing systems except for the constraint limitations such as the size of the program and the pin limitation. Hence, we look at some of the mapping and partitioning methods developed for the distributed processing systems. We look at how and why these apply to CCMs in the next chapter.

A distributed computing system is made up of various processing units comprised of several CPUs. The interacting modules comprising a distributed program must be assigned to the processing units to make use of the resources of the system efficiently. A distributed computing system has some configuration of processors each with its own memory, control, and arithmetic capability. The following module assignment problem is addressed. Given a distributed system of some configuration made up of *n* processing units and a program made up of *m* interacting modules, assign the modules of the program to the processing units such that the sum of the execution and communication costs is minimized. In other words, the amount of inter-processor communication is minimized, while taking advantage of specific powers of some processing units. The module assignment problem has been shown to be NP-complete for the general *n* processing unit system [21] [22].

Stone [6] has suggested an optimal algorithm for the problem of assigning modules to two processors by making use of the well-known network flow algorithm in related two-terminal network graphs. He has shown how the network flow model can be extended to systems made up of three or more processors. For the three-processor case, an algorithm that finds the optimal assignment has been developed [7]. If the structure of a distributed program is constrained in certain ways, it is possible to find the optimal assignment over a system made up of any number or processors in polynomial time. When the structure is constrained to be a tree, a shortest tree algorithm developed by Bokhari [8] yields the optimal assignment. Towsley [5] generalized the results of Bokhari to the case of series-parallel graphs. All these works are well summarized in [9].

In Stone's [6] and Bokhari's [8] model, the distributed computing system is assumed to be fully interconnected, i.e., there exists a communication link between any two processors. If a distributed program is a chain-structured parallel or pipelined program, it can be optimally partitioned over a chain or ring of processors under the constraint that each processor is assigned a contiguous sub-chain of modules [1]. In this case, the objective of the assignment is to minimize the load of a bottleneck processor, rather than to minimize the total load of the processors. Next, we look in detail at Bokhari's solution onto a chain of modules partitioned to a chain of processors.

2.5 Bokhari's method for solving the linear module and processor case

This algorithm produces the optimal partitioning of a pipelined program onto a linear processor network. The problem of optimally assigning the modules of a parallel program over the processors of a computer system is addressed. A sum-bottleneck path algorithm is developed that permits the efficient solution of many variants of this problem. The problem is solved under constraints on the structure of the partitions.

The approach to this problem is to first construct a layered graph that contains all information about the execution and communication times of the modules. A path in this graph corresponds to the assignment of subsequences of modules to processors. The weight of the heaviest edge in a path corresponds to the time required to execute the assignment in parallel/pipelined fashion. Thus, having constructed the graph, all that remains is to find the minimum bottleneck path in the graph (of all the paths, the one on which the heaviest edge has minimum weight).

Each layer corresponds to a processor and the label on each node corresponds to a sub-chain of modules. Any path connecting nodes *s* to *t* corresponds to an assignment of modules to processors. The rule for generating this layered graph for a problem with *m* modules and *n* processors is as follows.

Each layer contains all sub-chains of nodes, i.e., all pairs $<i, j>$ such that $1 \leq i \leq j \leq m$. A node labeled $<i, j>$ is connected to all nodes $<j+1, k>$ in the layer below it. All nodes $<1, i>$ $(<i,m>)$ in the first (last) layer are connected to node *s (t)*. As stated previously, each path from *s* to *t* represents an assignment of sub-chains to processors

under the contiguity constraint. If this path contains the node <*i, j*> of layer *k,* then this represents the assignment of modules *i* through *j* to processor *k*. There is a path in this graph corresponding to every possible contiguous sub-chain assignment that utilizes all processors.

Weights can be now added to the edges of this layered graph as follows. In layer *k* each edge emanating downwards from node <*i, j*> is first weighted with the time required for processor *k* to process nodes *i* through *j*. This accounts for the computation time. The communication time is now included in the graph. To the weight on the edge joining node <*a, b*> in layer *k* to node <*b + 1, d*> in layer *k + 1* is added the time to communicate between modules *b* and *b + 1* over the link connecting processors *k* and *k+1*. The influence of both the amount of data transmitted between modules *b* and *b+1* as well as the speed of the link between processors *k* and *k+1* can be included in the graph.

To consider memory constraints on individual processors, it suffices to add up the memory requirements of all modules in every sub-chain. If the sum of memory requirements for nodes *i* through *j* exceeds the capacity of processor *k,* node <*i, j*> in layer *k* is deleted, along with all edges incident on it [1].

2.5.1 Illustration of Bokhari's algorithm

Illustration of Bokhari's algorithm: Step 1

Draw a layered graph in which every layer corresponds to a processor and each node <*i, j*> in a layer corresponds to a sub-chain of modules *i* through *j*.

1. Layer 1 contains nodes < *1,j*> *j=1 to m- (n-1)*

2. Layers *k=2,…n- 1* contain nodes <*i,j*> where, *i =k* to *m -(n-k),* and *j = i* to *m -(n-k)*

3. Layer *n* contains nodes <*i, m*> *i=n* to *m.*

As an example, *m=5*, *n=3* gives the layers as shown in Figure 2.4. The modules are shown to have execution times as labels in Figure 2.3. Nodes of the assignment graph are constructed for five modules and three processors in Figure 2.4.

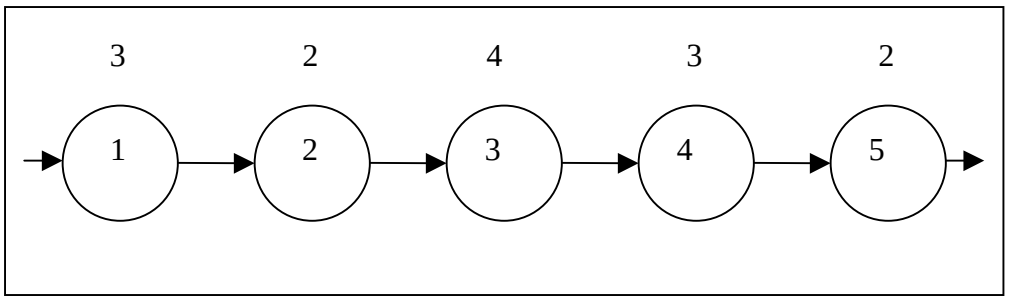

Figure 2.3. A module graph with five modules.

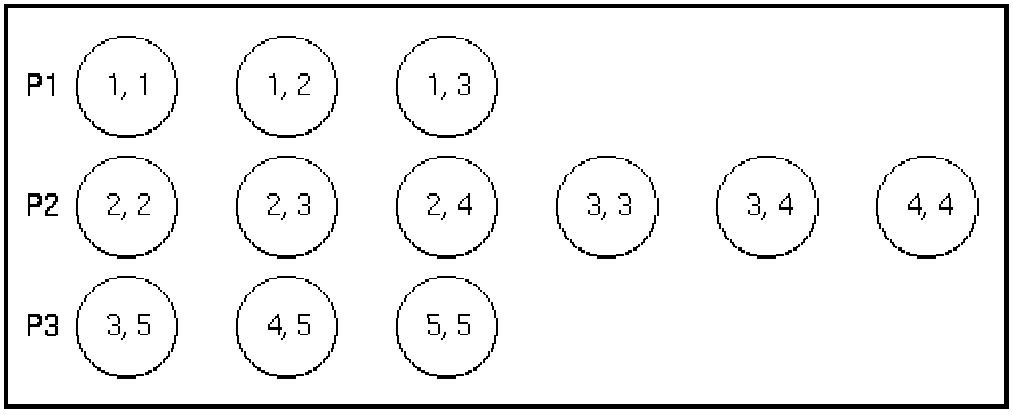

Figure 2.4. Nodes of the assignment graph.

Illustration of Bokhari's algorithm: Step 2

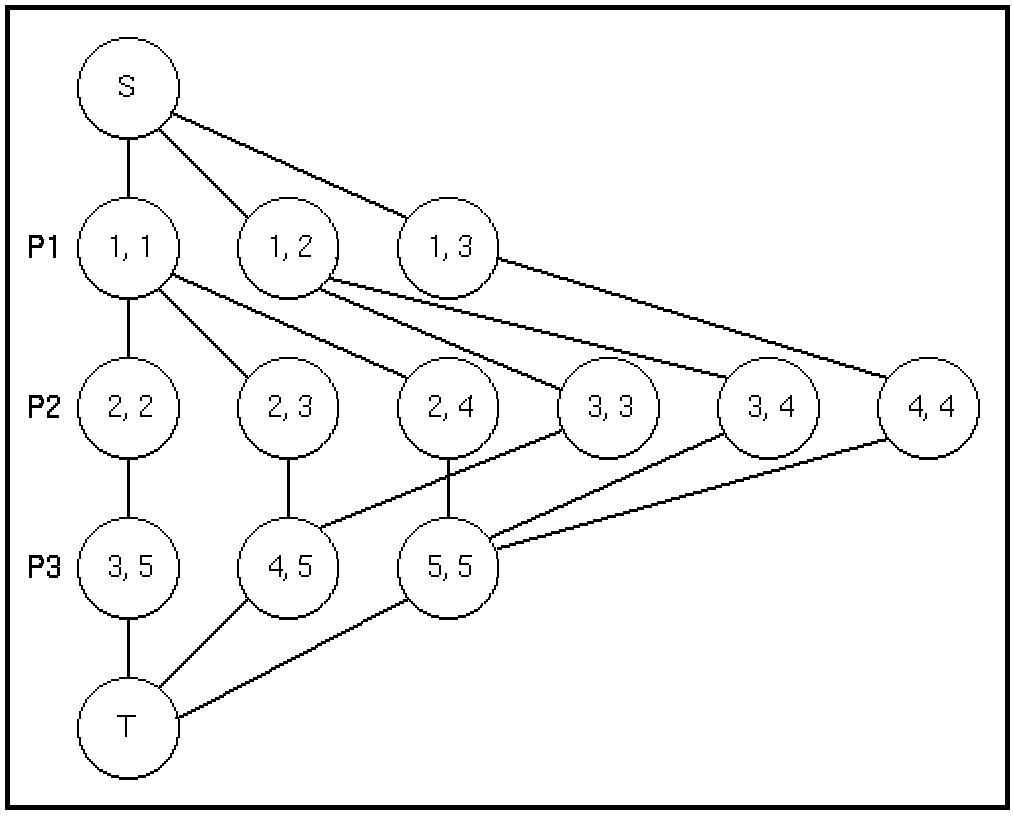

Figure 2.5. Edge creation for the assignment graph.

1. A node <*i, j*> on level *k=1,…n- 1* is connected to all nodes <*j+1,q*> (for any *q*) on level *k+1*.

2. All nodes <*1,j*> in the first layer are connected to a starting node *s*.

3. All nodes <*i, m*> in the final layer are connected to a terminating node *t*.

Any path connecting nodes *s* and *t* corresponds to an assignment of modules to processors.

Illustration of Bokhari's algorithm: Step 3

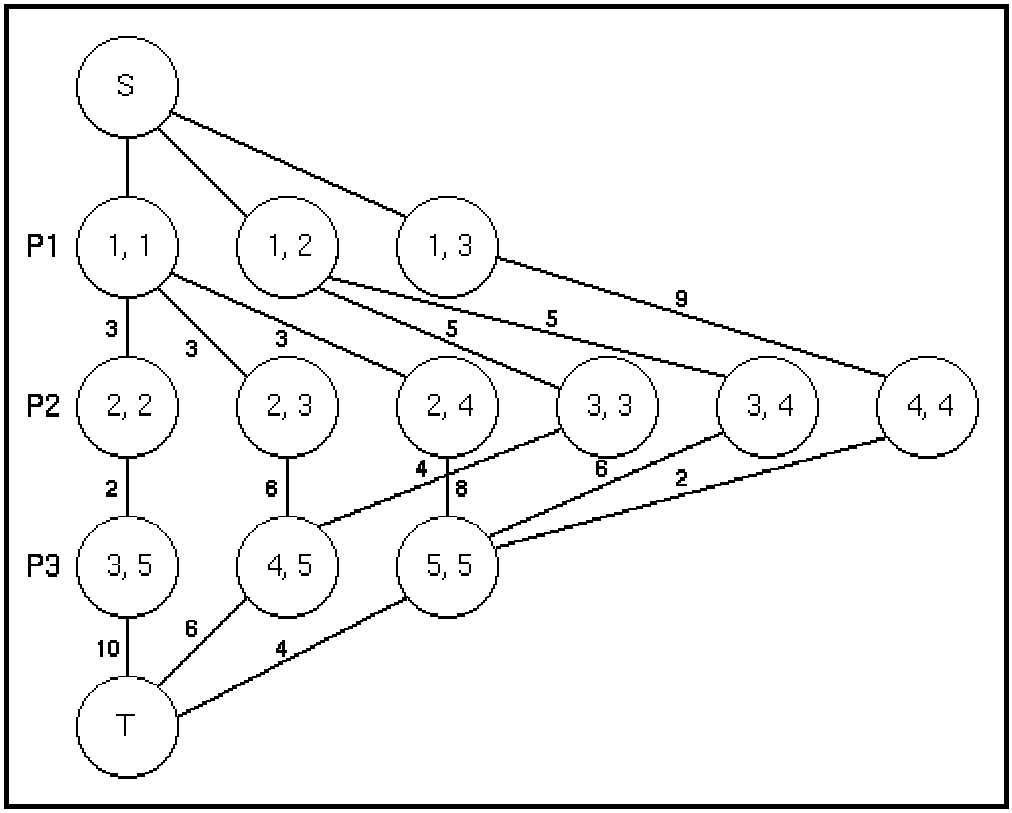

Figure 2.6. Weights on the nodes of assignment graph.

In layers *k=1,…n* each downward edge from node <*i*, *j*> is weighted with the time required for processor *k* to process modules *i* through *j* (this accounts for the total computation time on processor *k*) plus the communication time for modules residing on processor *k* to receive data from processor *k-1*. Figure 2.6 shows the nodes having weights representing execution times.

The paths in this graph represent every possible contiguous sub-chain assignment and the weight of the heaviest edge in a path corresponds to the time required by the most heavily loaded processor. Therefore, to find the optimal assignment we need to find the path in which the heaviest edge has minimum weight. This path is called the critical path.

Illustration of Bokhari's algorithm: Step 4

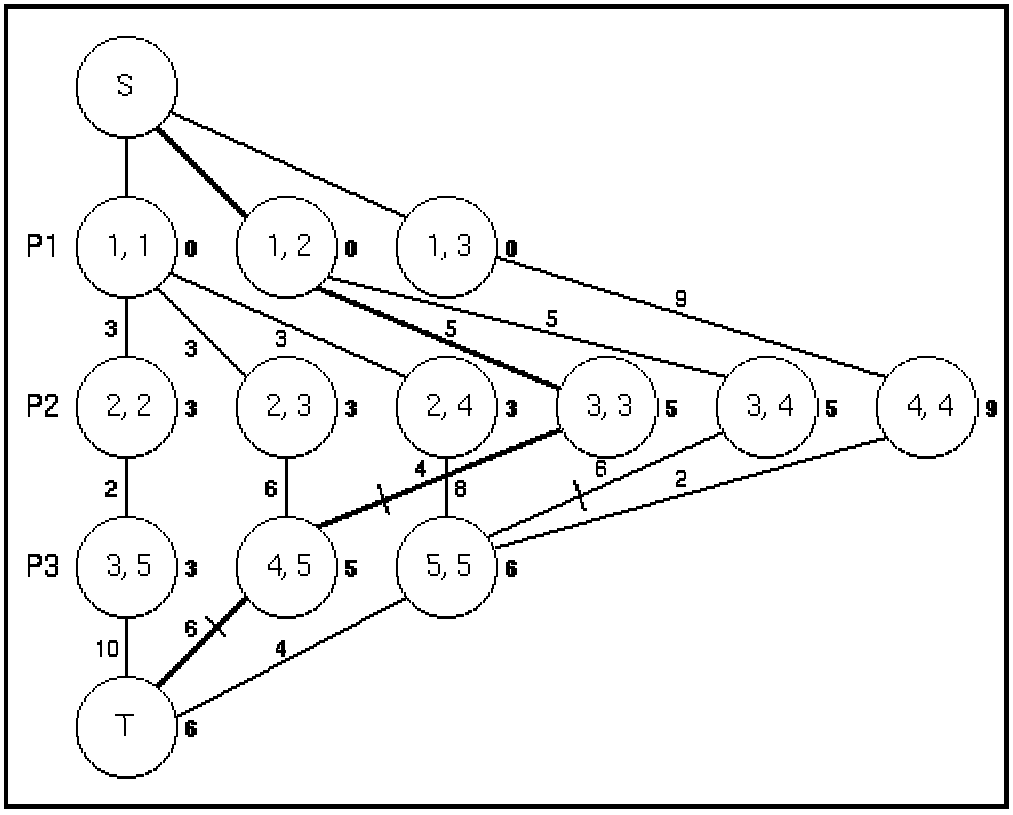

Figure 2.7. Highlighted critical path representing minimum bottleneck assignment.

We find the critical path as follows. The nodes in the first layer are labeled *L(i)=0*; all other nodes are labeled *L(i)=infinity or a large number*. Starting at the top and working downwards, examine each edge *e* connecting *node a (above)* to *node b (below)* then replace *L(b)* by

$$Min\ [max\ (W\ (e),\ L\ (a)),\ L\ (b)] \tag{2.22}$$

where *W (e)* is the weight of edge. If several edges need to be examined, mark the edge that contributed to the final label *L (b)*. The critical path is then found by starting at node *T* and following the marked edges (where appropriate). This is a modification of the Dijkstra's algorithm for finding minimum bottleneck path.

2.5.2 Basis for using Bokhari's method for CCMs

A common requirement in a communication system is to repeatedly apply a fixed sequence of operations (or transforms) to an essentially unending series of signals. For example, each arriving packet (or "window" or "frame") of data may have to be Fourier transformed, multiplied by a fixed frequency, filtered, clipped, and inverse-transformed as shown in Figure 2.8. This kind of application has a serial or chain-like structure to it and naturally lends itself to pipelining [3].

Should all these processes be carried on a uniprocessor, the maximum rate at which incoming data frames can be processed is determined by the time required for the processor to apply all the processing steps to each frame.

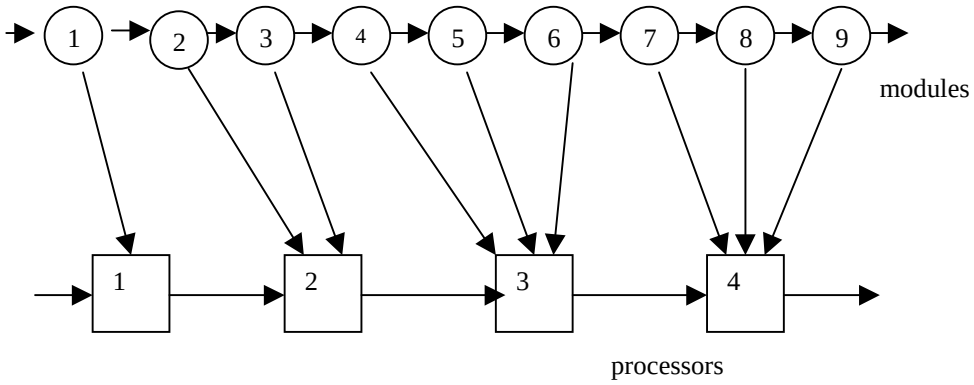

Figure 2.8. A nine-module chain mapped to a four-processor chain.

Putting each process on a separate processor can easily pipeline this computation. Because the inter-connection pattern of the processes is serial, the processors need only be connected in a chain. The processor that takes the longest amount of time to execute the modules (the bottleneck processor) now determines the maximum rate of processing. This is an expensive solution since it requires as many processors as processes. It is

inefficient because many processors may be lightly loaded and spend most of their time waiting for the bottleneck processor to finish.

The following problem then emerges. Given a set of m modules connected in a chain-like fashion and a multiprocessor chain of size $n < m$, find the assignment of sub-chains of processes to processors that minimize the load on the most heavily loaded processor. The contiguity constraint ensures that two modules that communicate with each other lie on directly connected processors. The assumption is that all the processors are to be utilized. The optimal assignment of sub-chains to processors is influenced by the following factors: a) the time required to execute each module (which may vary across processors, if the processors are dissimilar), b) the amount of inter-module communication (which can be nonuniform; because once a frame of data has been transformed, it may have a different number of data points), and c) the speeds of the links between pairs of connected processors.

Very similar problems arise in the field of image analysis, where the requirement is to take an image or a set of images and to apply various operators to it [12]. In the case of parallel solutions of partial differential equations, a straightforward technique is to partition the (possibly nonuniform) mesh into vertical strips. During each iteration, an estimate is made of the values within the strip. Because strips only influence adjacent strips, the communication pattern required is chain-like and the problem can be run on a chain or ring of processors [13]. There again emerges the problem of optimal assignment of a chain of processors. The structure of this problem is the same as that shown in the model of Figure 2.8, except that the edges connecting the modules are undirected (communication takes place in both directions). The time required to complete one step of the computation is equal to the time required by the most heavily loaded processor to complete its computation and to communicate with its neighbors. The important difference between this case and the signal-processing example is that this is parallel not pipelined processing. The assignment algorithm is insensitive to this difference [1].

In addition, the application of the various constraints such as the area, the pin limitation, and the power limitation is very conveniently supported by Bokhari's method. Any constraint can be applied during the time of creating nodes and edges. Only those

nodes and edges in the graph that correspond to legal assignment and are within the limitation of the constraints are created.

Chapter 3

Generalization of Bokhari's Algorithm for the Case of CCMs

This chapter deals with the generalization of Bokhari's algorithm to the case of CCMs. The chapter contains a description of what changes are to be made to the algorithms and to the building of the assignment graph. The proof of correctness for the modified algorithm for CCMs is also presented. Finally, two methods are presented to reduce the complexity of the algorithm: one is suggested by Nicol [10] and another one is suggested in this thesis.

3.1 Partitioning chain structured modules onto a chain of PEs

As pointed out by Bokhari [1], the problem of mapping module chains onto different types of architectures frequently arises in image, signal processing applications, and parallel solution of partial differential equations. His algorithm has been adapted to the case when a series task graph is to be mapped onto a series of PEs comprised of re-configurable devices and CPUs. Let v_1, $v_2 \ldots v_m$ denote a series of m modules, which may be executed concurrently. The graph representing the series of modules is G_m. The n PEs w_1, $w_2 \ldots w_n$, are also connected in series. We will constrain the space of mappings by considering only those which satisfy the *contiguity constraint*; i.e., the set of modules assigned to a PE forms a contiguous sub-chain of v_1, $v_2 \ldots v_m$. The model of inter-connection for the PEs is a chain represented by the graph G_p where the cost of communication for each link may be different. The different PEs have different capabilities.

3.1.1 The problem

The task is to partition a chain of modules of length m onto a chain of n processing elements (PEs) which may include CPUs, FPGAs, and DSPs. The partitioning problem is defined by Equation 2.1

3.1.2 Assumptions

The assumptions made for the following sections of this chapter are as follows. a) The Processing Elements (PEs) are relatively small in number compared to the number of modules, i.e., the number of PEs is of *O(1)*; b) FPGAs have limited area, so area constraints have to be applied to FPGAs but not to CPUs and DSPs; c) modules assigned to FPGAs execute concurrently; and d) a communication link once assigned for communication in one direction is meant to be that way throughout the phase of assignment.

3.2 The solution

The objective functions and constraints mentioned in Section 2.1 apply to the following sections of this chapter. All equations that are mentioned in Section 2.1 are equally applicable to the following sections.

3.2.1 Layered assignment graph G_a

The layered assignment graph G_a is given by a directed graph $G_a = (A, B)$, where A is a set of nodes, and B is a set of edges in the assignment graph $B \subseteq A \times A$. The number of nodes is $O(m^2n)$ and the number of edges is $O(m^3n)$. G_a is a layered graph of $n+2$ layers. Hence its nodes are divided into $n+2$ disjoint sets given by

$$A = \bigcup_{l=0}^{n+1} A_l. \tag{3.1}$$

Similarly all its edges are also divided into $n+1$ disjoint sets given by

$$B = \bigcup_{l=0}^{n} B_l. \tag{3.2}$$

All nodes in layer A_k correspond to PE w_k. Note that layer $A_0 = \{s\}$, where s is the start node of the graph G_a, and that layer $A_{n+1} = \{t\}$, where t is the end node of the graph

G_a. The nodes in layer A_k have a set of edges B_k connecting them to layer A_{k+1} according to the following rule: if $<(i,j),k> \in A_k$ and $<(j+1,b),k+1> \in A_{k+1}$ then there exists an edge $e \in B_k$ connecting them. The node s connects to all nodes in layer A_1. All nodes in layer n connect to node t. This rule ensures that the assignment is based on the contiguity constraint. Also, by varying i and j for each w_k as

$$k \leq i \leq m \text{ and,} \tag{3.3}$$

$$i \leq j \leq m, \tag{3.4}$$

all possible nodes $<(i,j),k>$ are created in the layer. The assignments made using the rules in this section apply to the case when assignments are to be made to all PEs for utilizing all resources. If assignments are to be made for the case when some PEs need not be assigned modules, the same rules apply with the modification that an additional *m-1* empty modules with zero execution and communication times are added at either end of the chain of the input graph G_m [1]. This is done before the creation of the assignment graph. Each of these empty modules has zero area and zero power consumption.

3.2.2 Minimum bottleneck assignment

The procedure followed to obtain the minimum bottleneck assignment is obtained by first constructing a layered assignment graph G_a as follows. First the nodes are built according to the rules specified in Figure 3.1, then the edges of the graph G_a are built as specified in Figure 3.2. The weights are assigned as specified in Section 3.2.3. Then the algorithm as described in Figure 3.3 is executed on the graph G_a to get the minimum bottleneck path.

Each of the n layers in the graph G_a represents a PE. Each of the nodes in a layer represents different possible module assignments to the PE of that layer. Any path from s to t represents a possible assignment of modules to the PEs. The path obtained after the execution of the algorithm represents the assignment of the modules to the corresponding processors. The algorithm is based on finding the shortest paths based on Dijkstra's

algorithm, with a slight modification for minimizing bottleneck values. But Dijkstra's algorithm can be directly applied to minimize the total execution time. The algorithm given in Figure 3.3 is for minimizing the bottleneck values. The path obtained by the execution of the algorithm represents the minimum bottleneck assignment of modules to PEs, i.e., the assignment with maximum throughput. The algorithm always produces the best path, and has a run time of $O(m^3 n)$.

```
Create node s
For layer k=1 to n do
For each <(i,j),k> defined in Equations 3.3 and 3.4 do
        if (area, pin, and power constraint satisfied for modules <(i,j),k>) then,
         create node<(i,j),k>
end do
Create node t.
```

Figure 3.1. Node creation for graph G_a.

3.2.3 Assignment of weights to the nodes of graph G

Every node $<(i, j), k>$ is assigned a weight given by Equations 2.6 and 2.7. Every node in the graph has a label with an initial value of infinity.

Node *s* is connected to all nodes in layer *1*

For layer *k=1* to *n-1* do

For all pair of nodes *<(i, j), k>* and *<(j+1, b), k+1>* between layers *k* and *k+1*,

create an edge between nodes *<(i, j), k>* and *<(j+1, b), k+1>*.

end do

All nodes in the layer *n* are connected to node *t*.

Figure 3.2. Edge creation for graph G_a.

Input: A weighted assignment graph G_a and starting vertex *s*. Weight of any node *x* is *weight(x)* and its label is denoted by *label(x).* Every label is initialized to infinity.

Algorithm: Use a variant Dijkstra's algorithm to find the minimum bottleneck path from *s* to *t* as follows.

Examine each pair of nodes, *(x, y)* connected by an edge in layers *k* and *k + 1* and replace label *y* by,

min (label (y), max (label (x), weight (y))).

The iteration continues until all pairs of nodes connected by edges are looked at once and labeled similarly.

For finding smallest total execution time, apply Dijkstra's algorithm directly.

Output: A minimum bottleneck path from *s* to *t* or smallest total execution time.

Figure 3.3. Algorithm 3.1 for finding minimum bottleneck path or minimum total execution time from the assignment graph G_a.

3.3 Proof of correctness for Algorithm 3.1

Theorem 3.1: Algorithm 3.1 assigns a chain of modules v_1, $v_2 \ldots v_m$ to a chain of PEs w_1, $w_2 \ldots$ w_n achieving optimal throughput and total execution time, while meeting constraints on area, pin, and power.

Proof: All possible module assignments meeting the constraints are given by the creation of the nodes $\langle (i,j),k \rangle$ according to the Equations 3.3 and 3.4. No nodes that violate constraints are created. The use of the rule given in Section 3.2.2 for creating edges between any two nodes of layer k and $k+1$ ensures that any path taken from s to t gives a contiguous assignment of modules with no duplication of modules.

The constraints given by Equations 2.17, 2.18, and 2.19 are applied during the creation of the nodes and edges. The assignment graph G_a has a set of nodes and edges that represents all possible assignments after the constraints have been applied. Any path from s to t represents a possible valid assignment that satisfies all the constraints.

If the objective is to minimize the total execution time, the weights for the nodes are given by Equation 2.7. The objective is to minimize these weights. Applying Dijkstra's algorithm [36] to this graph G_a results in a path from s to t that has minimum total execution time. This path represents an assignment of modules to PEs such that the sum of the weights on the nodes is minimum.

If the objective is to maximize throughput, the weights for the nodes are given by Equation 2.6. Application of Algorithm 3.1 results in a minimum bottleneck path. Algorithm 3.1 is a modification of Dijkstra's algorithm to record minimum bottleneck paths from s to all nodes. This means that the bottleneck values of that assignment are minimum under the constraints. Hence an optimal assignment of modules to PEs in which the total execution time is minimum under the various constraints results.€

3.4 Improvement in the speed of execution

Method 1

The procedure can be accelerated using the method suggested by Nicol [10]. A simple technique will reduce the number of graph edges for all of these problems without affecting path costs. For the case of a chain of modules to be assigned to a chain of PEs, *n-2* layers are added, one between each of the layers of the old assignment graph (except between layers *1* and *2* where an additional layer provides no benefit). Figures 3.4 and 3.5 illustrate the old and the new modified assignment graph.

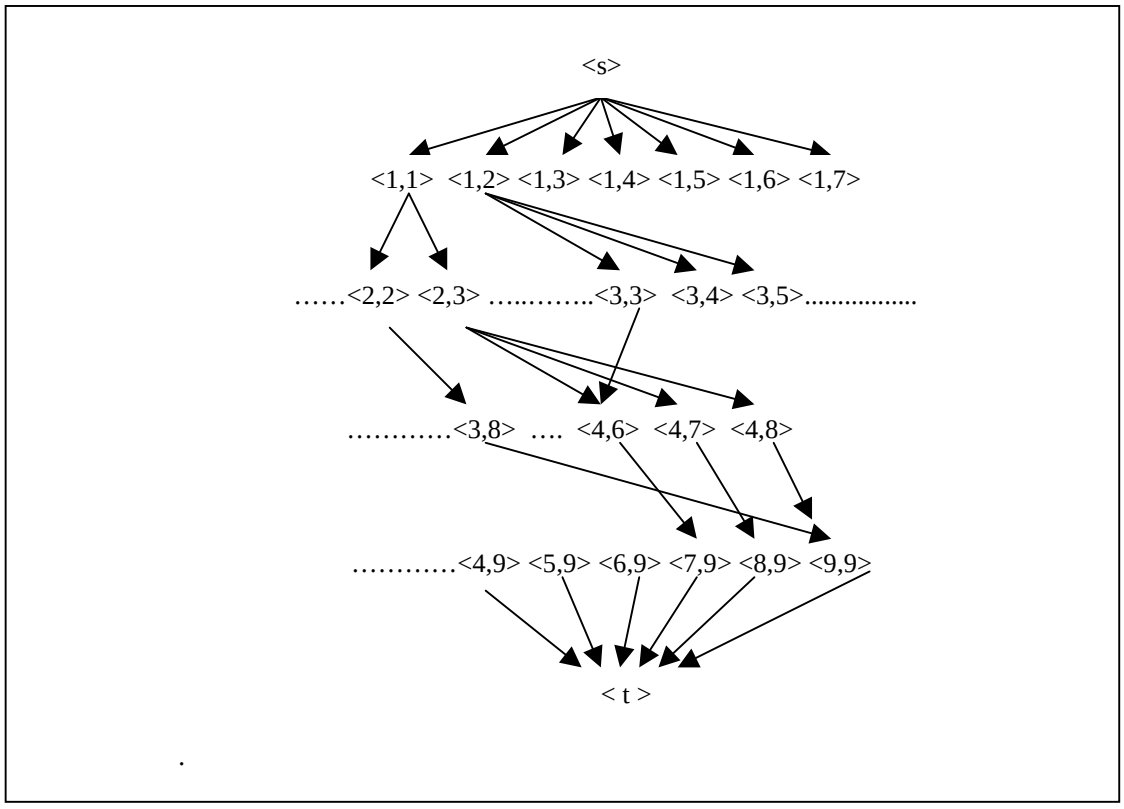

Figure 3.4. A layered graph for a linear array problem with nine modules and four processors.

Each new layer has *m* nodes, labeled *1* through *m*. To avoid confusion the *kth* layer will be referred to in the new graph as being identical to the *kth* layer in the original

graph. Node $<(i,j),k>$ in layer k directs a single edge to node $<j>$ in the new layer between layers k and $k+1$. This edge is labeled exactly as before. Node $<j>$ in the new layer in turn directs an edge to every node of the form $<(j+1,l),k+1>$ in layer $k+1$. Every such edge is labeled with weight zero. Again, many nodes and edges are not shown in order to avoid congestion. It is clear that any path from source to sink still defines a legal assignment and has a weight identical to that of the corresponding path in the original graph. The number of edges drops from $O(nm^3)$ to $O(nm^2)$, reducing the complexity by a factor of m.

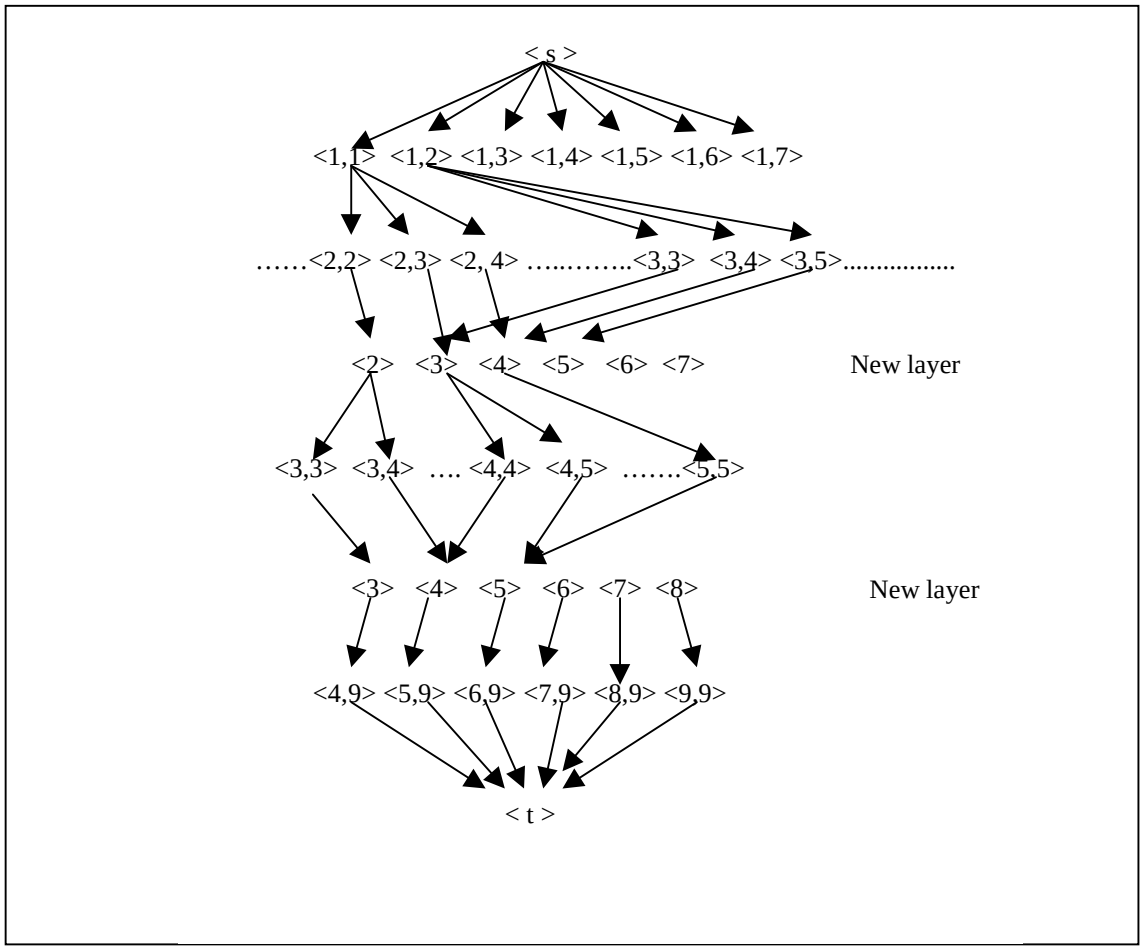

Figure 3.5. An improved layered graph for a linear array problem with nine modules and four processors.

Method 2

In the second method, an array of structures is used to hold the weights of the execution and communication times. Because the edges join nodes $<(i,j),k>$ to nodes $<(j+1,l),k+1>$, the weights of the communication costs can be included with the execution weights of the nodes. This makes it possible to not have any edges at all. The advantage of this method is that it reduces memory storage considerably. Also, the time to build the edges can be reduced. The disadvantage of this method is that constraints cannot be imposed on the communication links.

Chapter 4

Solution to Partitioning Series Module Graphs onto Arbitrary PE Graphs

This chapter contains a new algorithm, which is based on Bokhari's algorithm. The algorithm solves the partitioning problem for the case of an arbitrary PE graph with the module graph being a chain. The proof of correctness and the complexity of this algorithm are also presented.

4.1 Mapping chain structured modules onto a general PE graph

As pointed out in Chapter 3, Algorithm 3.1 solves the problem of partitioning a chain of modules across a chain of PEs. In this chapter that algorithm is modified to partition a chain of modules across a set of PEs whose inter-connection need not be a chain. The inter-connection pattern of the PEs can take any shape with the limitation that the number of PEs is relatively small in number (of order *O (1)*). Given a small set of PEs having some inter-connection, the algorithm partitions the modules over the PEs while applying the constraints and meeting the objective of optimal throughput.

The algorithm presented here is appropriate for a parallel implementation. Examples for both a module graph and a PE graph are as shown in Figures 4.1 and 4.2.

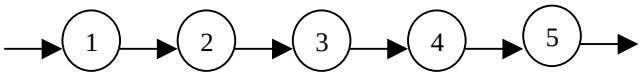

Figure 4.1. A chain module graph.

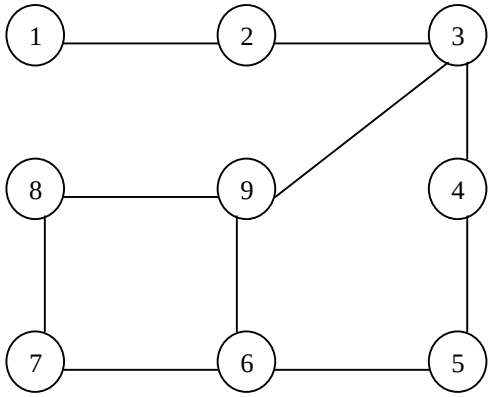

Figure 4.2. A PE graph with nine PEs and its links.

4.1.1 The problem

The task is to partition a chain of modules of length *m* onto a set of *n* processing elements (PEs) which may include CPUs, FPGAs, and DSPs. These PEs are inter-connected by a set of communication links whose inter-connection pattern may take any form such as a ring, star, square or combinations of these. The partitioning is given by Equation 2.1.1.

4.1.2 Assumptions

All assumptions mentioned in Section 3.1.2 also apply to the following sections in this chapter. An additional assumption is that the PE graph G_p representing the inter-connection pattern among PEs can take any shape such as a square, ring, star, or a combination of such inter-connections.

4.2 The solution

The module graph consists of a set of modules connected in series. To apply Algorithm 3.1, we need a set of PEs whose inter-connection pattern is linear. The central

idea of solving the problem of partitioning a chain of modules onto an arbitrary connection of PEs is to explore the links of the PE graph G_p using a depth first search to obtain a linear chain of PEs. The modules are partitioned over each of these chains. The best partition is the required solution.

Figure 4.3. Example of a trail.

Figure 4.3 contains one trail explored from the PE graph of Figure 4.2, starting from node 1, passing through node 6 and revisiting node 6 at the end to explore the link between node 9 and node 6. A sequence of depth first searches is performed on the PE graph starting from each node to enumerate all the trails in the PE graph. This procedure of depth first search is performed on all the nodes in the PE graph. The nodes in each explored trail represent the PEs. The edges in that same trail represent the links that have been encountered in the depth first search. In each of the trail, adjacent PEs are connected by a unique (to that trail) communication link. In the process of exploring all links uniquely, a trail may revisit a PE more than once. This is illustrated in Figure 4.3.

The chain of modules is then partitioned over each of these sets of PEs, which are connected in the trail. Algorithm 3.1 forms the basis for each partitioning of the chain of modules onto each trail of PEs. Each partitioning gives an assignment under the different constraints mentioned in Chapter 2. Selecting the best assignment out of the different assignments will give an optimal minimum bottleneck assignment of modules to the PEs.

Two different possible assignments to the trails of PEs (explored by depth first search from the original PE graph) are illustrated by Figures 4.4 and 4.5. Figures 4.4 and 4.5 show two possible partitionings of 25 modules over two different trails explored by depth first search on the PE graph.

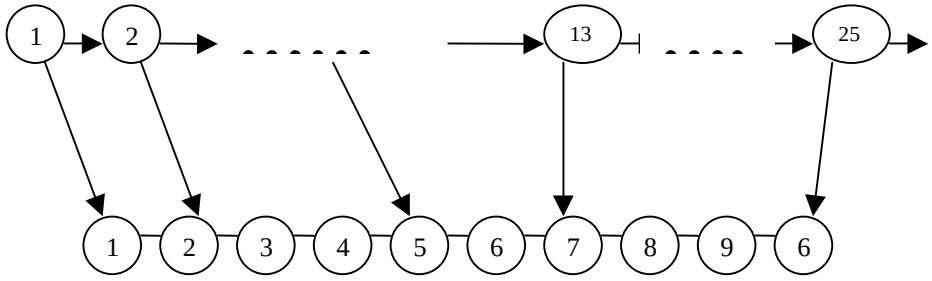

Figure 4.4. Partitioning problem A.

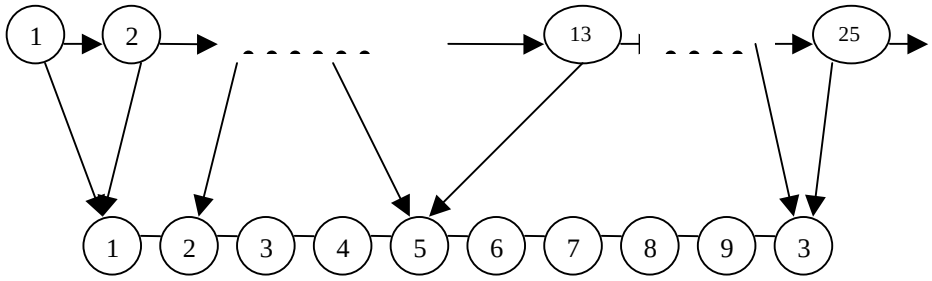

Figure 4.5. Partitioning problem B.

The enumeration of all contiguous links using depth first search on the PE graph G_p gives a set of graphs given by G_{p1}, G_{p2}, G_{p3},…G_{pq}, where q is some finite number. Each $G_{px} = (W_x, F_x)$ where, $W_x = \{w_{x1}, w_{x2}, w_{x3}\ldots. w_{xl}\} \subseteq W$ and, $F_x = \{f_{x1}, f_{x2}, f_{x3}\ldots. f_{xl-1}\} \subseteq W_x \times W_x \subseteq F$. Each w_{xi} and w_{xi+1} are connected by a link f_{xj}. The modules v_i have to be partitioned over PEs w_{xj} under the constraints to form

$$V_n = \bigcup_{i=1}^{xl} V_i \qquad (4.1)$$

Definitions:

Dummy PE: A PE w_l, may appear more than once in the trail. Any subsequent revisit to the node is referred to as a dummy PE. A dummy PE of w_l is denoted as $w_{d(l,r)}$ where r

denotes the revisit to the PE. For example, in Figure 4.1.3, PE 6 occurs twice in the trail. The original PE is w_6 and the first revisit is $w_{d(6,1)}$. Note that $w_{d(l,0)} = w_l$.

Dummy PE layer: When the chain of modules is partitioned onto PEs in a trail, a layered assignment graph G_a is created. This layered assignment graph will not only have layers representing the PEs in the trail, but also will have layers representing the dummy PEs in the trail. Those layers are referred to as the dummy PE layers. The set of nodes A_l correspond to w_l and the set of nodes $A_{d(l,r)}$ correspond to $w_{d(l,r)}$. Also, the number of layers h apart from the s and t layers in the assignment graph is equal to the number of nodes in the trail.

4.2.1 Construction of layered graph for each of the G_{px}

There will be as many layers in the assignment graph as the number of nodes in the chain graph G_{px}, which is a trail. This includes both the PEs w_l and their dummy PEs $w_{d(l,r)}$. Let A^c denote the set of layers in the assignment graph that have dummy layers. Let A^d denote the set of all layers in the assignment graph that are dummy layers of A^c and A^r denote the set of all remaining layers. A^r also includes the s and t single node layers. If the total number of nodes in a trail is h, then the number of layers in the assignment graph is $h+2$. The total number of layers A is given by

$$A = A^c \cup A^d \cup A^r = \bigcup_{l=0}^{h+1} A_l \qquad (4.2)$$

The main issue for the area constraint in this case is whether the paths in the assignment graph give legal assignments as regards to the area of the PE. Because all A_l and $A_{d(l,r)}$ correspond to the same w_l, the area constraint to be applied during the execution of the algorithm is given by

$$\sum_{q=0}^{r} Area(i_q, j_q)_{d(l,r)} \leq Size_l, \qquad (4.3)$$

where, i_q through j_q are modules assigned to layer l corresponding to $w_{d(l,q)}$. An area constraint for the CPU may have to be applied if the memory required to hold all the modules is limited. The power constraint is implemented in the same fashion as the area constraint.

Similarly, the pin constraints are implemented as follows. If the different links of a PE have different limitations on the number of terminals, then each of the inputs and outputs of the different sets of assignments from the layers A^c and A^d have to be applied separately as

$$\sum_{q=0}^{r} \quad Count(i_q, j_q)_{d(l,r)} \leq T_l, \tag{4.4}$$

where, i_q through j_q are modules assigned to layer l corresponding to $w_{d(l,q)}$. T_l is the pin limitation of PE w_l.

The *contiguity constraint* is applied only during creation of edges in the assignment graph so that any layer in the assignment graph will have only contiguous sub-chains of modules. The contiguity constraint is applicable only to the assignment graph and not to the PEs. This is because any PE that appears more than once in the trail may get an assignment of more than one sub-chain of modules. These sub-chains of modules assigned to the same PE may be mutually non-contiguous disjoint sets.

4.2.2 Assignments of weights to the nodes of the graph G_a

The weights to be assigned to the nodes vary depending on whether the nodes belong to layers in A^c or A^d, and on whether they are CPUs or FPGAs. The weight $T_{i,j,k}$ for the nodes in both layers in A^c and A^d is

$$max(S_{i,j,k} , C_{i,j,k}), \tag{4.5}$$

for the case when the PE is an FPGA and the objective is to minimize throughput.

However, the weight $T_{i,j,k}$ for the nodes in both layers in A^c and A^d is

$$S_{i,j,k} + C_{i,j,k}, \tag{4.6}$$

for the case when the PE is either an FPGA or a CPU, and the objective is to minimize for total execution time.

The weights on the nodes of the assignment graph have to reflect the modified total time cost in the case of a PE representing a CPU and the objective is to minimize the throughput. If $S_{i,j,k}$ and $C_{i,j,k}$ are the execution and communication costs as defined by Equations 2.3 and 2.5 respectively for a node $<(i,j),k>$, then

$$(S_{i,j,k})_{d(l,u)} = (S_{i,j,k})_{d(l,u-1)} + S_{i,j,k}, \tag{4.7}$$

$$(C_{i,j,k})_{d(l,u)} = max((C_{i,j,k})_{d(l,u-1)}, C_{i,j,k}), \tag{4.8}$$

where $(S_{i,j,k})_{d(l,u)}$ and $(C_{i,j,k})_{d(l,u)}$ represent the modified execution and communication costs for that node in layer $A_{d(l,u)}$. Hence the total time cost $T_{i,j,k}$ of that node is

$$T_{i,j,k} = max((S_{i,j,k})_{d(l,u)}, (C_{i,j,k})_{d(l,u)}). \tag{4.9}$$

The modification is not applied for $A_{d(l,0)}$, and the execution and communication costs are given by Equations 2.3 and 2.5 respectively.

Input: Assignment graph G_a.

Algorithm:

For each combination $C \in S$ do

Remove all nodes temporarily in all layers $A_i \in A^c$ having dummy nodes, except for the set of nodes in C.

For each layer $A_{d(l,r)} \in A^d$ do

For each node in layer $A_{d(l,r)}$ do

if area, or pin, or power constraints not satisfied,

Temporarily remove the node from graph G_a

else, compute weights from Equations 4.5 through 4.9.

end do

end do

Return minimum bottleneck path using Algorithm 3.1

Restore all temporarily removed nodes into G_a

end do

Return

Output: A minimum bottleneck assignment of modules to the trail of PEs

Figure 4.6. Algorithm 4.1.

Input: PE graph and module graph

Algorithm: For every trail, G_{px}, in the PE graph do

Construct assignment graph G_{ax}

Call Algorithm 4.1 on G_{ax} to find the shortest path between s and t.

Store minimum of the bottleneck paths.

end do

Return

Output: Minimum bottleneck path under the constraints.

Figure 4.7. Algorithm 4.2

Algorithm 4.2 uses a recursive depth first search and for each explored trail calls Algorithm 4.1. A recursive depth first search on the PE graph gives all the trails in the PE graph. Each trail G_{px} consists of the vertices obtained from a recursive depth first search, and its edges are given by the edges between the corresponding vertices in graph G_p.

A combination, C, is a set of nodes where exactly one node is selected from each of the layers $A_l \in A^c$. Let S represent the set of such combinations. For example, let A_1={<(1,1),1>, <(1,2),1>, <(1,3),1>}, A_2={<(2,2),2>, <(2,3),2>, <(2,4),2>} and A_3={<(3,3),3>, <(3,4),3>, <(3,5),3>, <(3,6),3>} be three layers (each assignment <(i,j),k> is a node). Let A^c include the layers A_1, A_2 and A_3. Let A_3 be the dummy layer of A_1. Therefore A_3 belongs to both A^c and A^d. One possible combination is C={<(1,2),1>,<(2,4),2>, <(3,5),3>}. The set S contains a total of $3 \times 3 \times 4 = 36$ possibilities.

4.3 Proof of correctness of the algorithm

Theorem 4.1: For the assignment graph G_a built for any trail obtained from Algorithm 4.2, applying Algorithm 4.1 partitions the modules onto the trail of PEs giving the minimum bottleneck path or minimum total execution time under area, pin, and power constraints.

Proof: The proof can be stated in three steps. The first step is to prove that only paths that give valid assignments are considered and that all valid assignments are taken into account. The second step involves proving that the constraints are satisfied. The final step is to prove that the best partitioning is obtained under the constraints.

The use of Equations 3.3 and 3.4 for creating assignments to layers ensures that all possible assignments of modules are created for all layers including the dummy layers A^d. The use of the contiguity constraint, as mentioned in Section 4.2 on the creation of edges of the graph G, ensures that any path in the graph G_m having module assignments from both a layer $A_i \in A^c$ and its dummy layer $A_{d(i)} \in A^d$ are disjoint sets of modules with no modules reassigned to the same PE. By considering each $C \in S$ separately, all paths

from *s* to *t* passing through that combination of nodes is considered (if such a path through *C* exists). Hence, applying Dijkstra's algorithm for finding the shortest path from *s* to *t* gives the shortest path through *C*. Only those paths which are valid are taken into account. By considering all combinations of $C \in S$, it is ensured that paths from *s* to *t* passing through all possible paths through *C* are considered. The best of the paths gives the best partitioning under no constraints.

The application of the constraints during the creation of the nodes and edges as given by Equations 2.16, 2.17, and 2.18 ensures only valid assignments are created in the layers of the assignment graph. The areas of the modules, number of terminals, and power consumed by the modules in the layer $A_l \in A^c$ and that of modules in its dummy layers $A_{d(l,r)} \in A^d$ should together satisfy the constraints of that PE w_i. This is done by temporarily including only those nodes in the layers $A_{d(l,r)} \in A^d$ that satisfy the area, pin, and power constraints given by Equations 4.3 and 4.4 for each combination *C*. By applying these constraints for all combinations of $C \in S$, all paths considered from *s* to *t* satisfy the constraints.

The assignment of weights to the nodes is given by Equations 4.5 through 4.10. Applying Dijkstra's algorithm to find the shortest (minimum bottleneck) path from *s* to *t* gives the best path through each combination *C*. Repeating this to all $C \in S$ gives the best path through each *C* under constraints. Selecting the best of these assignments gives the best partitioning of the chain of modules to the trail of PEs.

Corollary 4.1: Algorithm 4.2 will partition a chain of modules onto the PEs, which can have any inter-connection pattern to get optimal throughput/total execution time under the area, pin, and power constraints.

Proof: Algorithm 4.2 uses recursive depth first search to enumerate all possible trails. It then the uses Algorithm 4.1 to find the best of the minimum bottleneck paths for each trail under constraints. By choosing the best of the minimum bottleneck paths obtained from all trails, all possible partitions are considered and they are optimal under the constraints.

4.4 Complexity

Theorem 4.2: Algorithm 4.1 has a complexity of $O(m^c)$ where c is a constant independent of m.

Proof: Algorithm 3.1 is called for all combinations of C. This is equal to the number of possible combinations for choosing only one node from each of the layers of A^c. The number of nodes in a layer is $O(m^2)$. The number of combinations is bounded by $O((m^2)^q)$ where q is the number of layers.

Algorithm 3.1 has a complexity of $O(m^3n)$. The total complexity is given by $O(m^3n) \times O((m^2)^q)$.The number of layers is bounded by $O(n^2)$. This is because the number of times an edge is visited in the PE graph is utmost once and the order of the edges is $O(n^2)$. However, n is assumed to be $O(1)$. Hence the overall complexity is given by $O(m^c)$.

Chapter 5

Module Graphs with a Fork

In the following sections the algorithm for partitioning a modules graph G_m with a single fork over a PE graph G_p that is a chain is presented, followed by proof of correctness for that algorithm and its complexity. The algorithm presented is appropriate for a parallel implementation.

5.1 Partitioning a module graph having a fork onto a chain of PEs

A module graph G_m with a single fork contains a main set of modules denoted as $V_M=\{v_1, v_2,\ldots v_f\}$. The fork gives rise to two sets of modules. The two chains of modules branching from the fork are denoted as $V_{a1}=\{v_{\alpha 1}, v_{\alpha 2}\ldots v_{\alpha a1}\}$ and $V_{a2}=\{v_{\beta 1}, v_{\beta 2}\ldots v_{\beta a2}\}$. The total number of modules is m and therefore we get

$$f+a1+a2=m. \tag{5.1}$$

Examples of both the module graph and the PE graph dealt with in this chapter are given in Figures 5.1 and 5.2, respectively. For example, $V_M=\{1,2,3,4\}$, $V_{a1}=\{5,6\}$, and $V_{a2}=\{7,8,9\}$ for Figure 5.2.

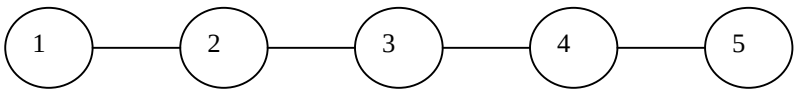

Figure 5.1. A PE graph with five PEs connected as a chain.

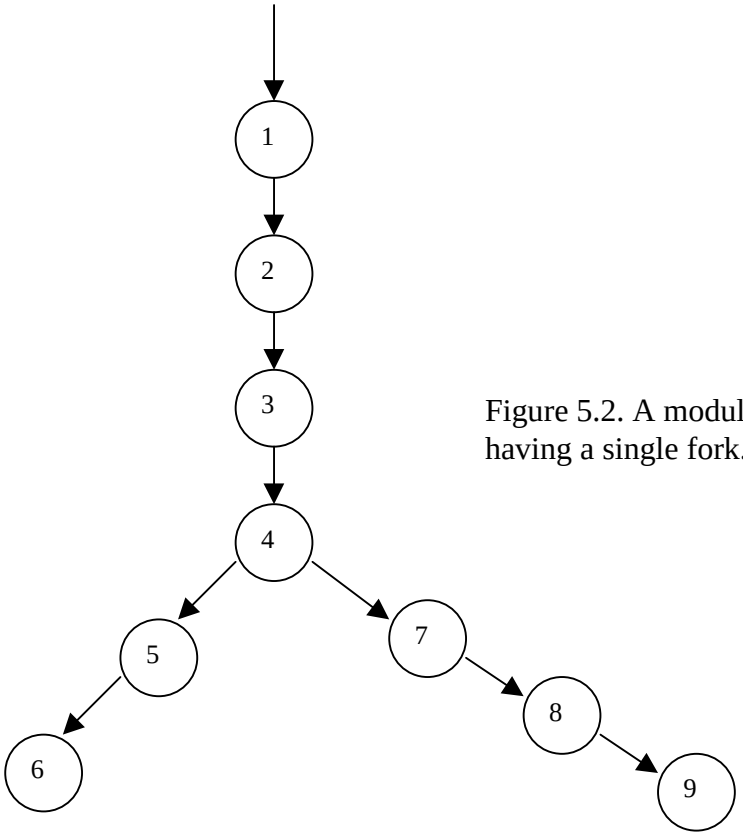

Figure 5.2. A module graph having a single fork.

5.1.1 The problem

The problem is to partition a module graph with a single fork onto a set of n processing elements (PEs) which may include CPUs, FPGAs, and DSPs, where the PEs are connected by a set of communication links whose inter-connection pattern is a chain. The partitioning is as described by Equation 2.1.1.

5.1.2 Assumptions

The assumptions are the same as given in Section 3.1.2. An additional assumption is stated as follows. Communicating modules need not be on adjacent PEs. However, there is no backward communication allowed. For example, module v_a has a data

dependency on module v_{a-1}; if module v_{a-1} is assigned to PE w_k. then module v_a can be assigned to any of the Pes following w_k but not to its preceding PEs.

5.2 The solution

The module graph has a single fork, which means that both V_{a1} and V_{a2} of the module graph G_m as shown in Figure 5.3 must receive the output of v_f before either of them can execute. There is no backward communication allowed. Hence if v_f is assigned to PE w_f, then all sub-chains of modules of V_{a1} and V_{a2} have to be assigned to a PE w_k such that

$$f \leq k \leq n. \tag{5.2}$$

In the following sections the term dummy PE is used to represent the following. The dummy PE of this section is similar to that in Section 4.1.1. The assignment of modules might utilize some or all of the resources of a PE. The dummy PEs are those PEs which have been assigned some set of modules. The dummy PE represents the available resources such as area, pin, and power of the original PE onto which another set of modules can be assigned.

The technique used to partition the modules onto the PEs consists of the following steps. The set of modules of V_M and V_{a1} are regarded as a contiguous chain of modules V_{C1}. The chain of modules, V_{C1} is partitioned onto the PEs, while, the other chain of modules V_{a2} is partitioned onto the available resources of their dummy PEs. Alternately, V_M and V_{a1} can be regarded as a contiguous chain V_{C2} and partitioned onto the PEs. The set of modules V_{a2} is then partitioned onto the available area, pin, and power of the dummy PEs. Both methods are equivalent. The interpretation of V_{C1} and V_{a2} of the module graph G_{m1} is illustrated in Figures 5.3 and 5.4. Interpretation of V_{C2} and V_{a1} of the second module graph G_{m2} is similar and is not illustrated.

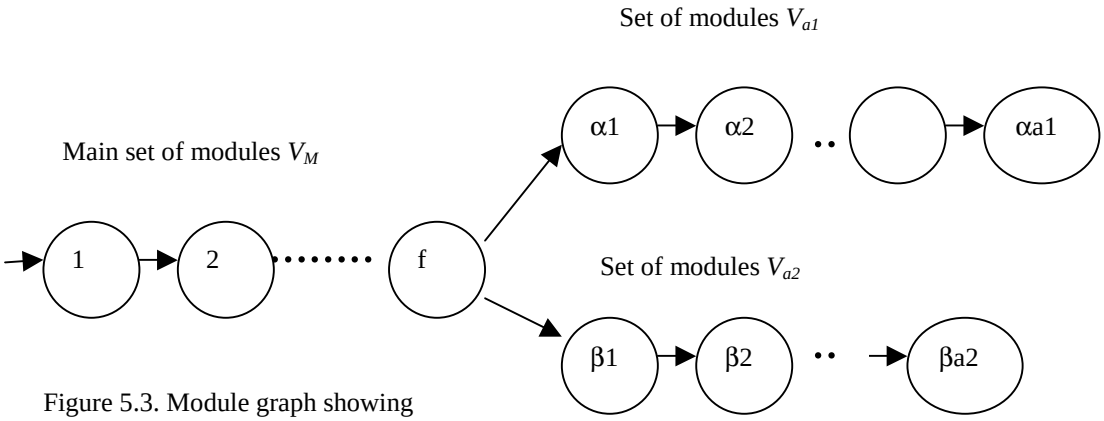

Figure 5.3. Module graph showing the set of modules V_{a1}, V_{a2}, and V_M.

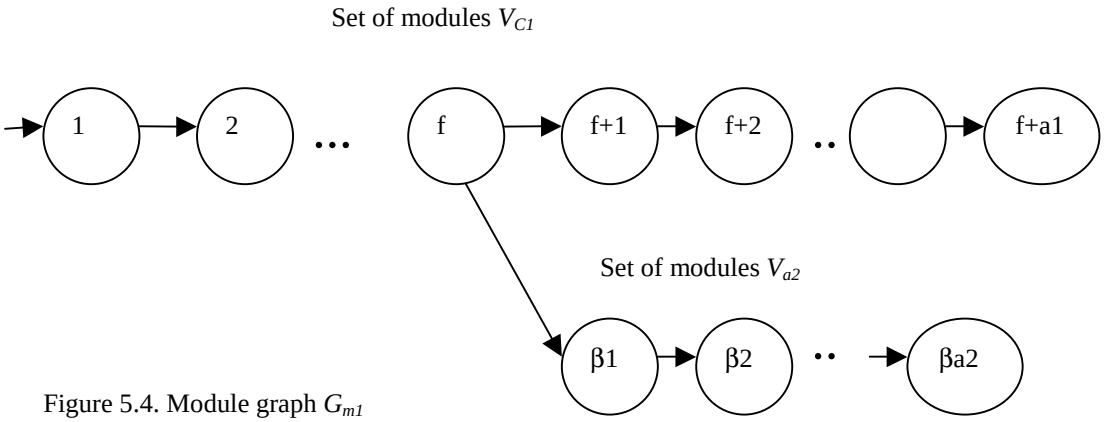

Figure 5.4. Module graph G_{m1} showing the set of modules V_{C1} and V_{a2}.

5.2.1 Construction of the assignment graph

Definition: A fork node a_f in an assignment graph is defined as a sub-chain of modules containing v_f in the sub-chain.

The modules of the graph G_m have been numbered as shown in Figure 5.4. Once the module graph G_{m1} is identified as illustrated in Figures 5.3 and 5.4, an assignment graph is built. The same procedure is involved for set of modules in the modules graph

G_{a2}. Either of the assignment graphs G_{a1} and G_{a2} are built for each G_{m1} and G_{m2} using the same rules as illustrated in the previous section. The procedure mentioned for G_{a1} in the following sections applies to G_{a2} as well.

The assignment graph G_{a1} is built as follows. There are *n* layers, each of them representing the *n* PEs. All these layers have module assignments that are sub-chains of $V_{C1}=\{v_1, v_2 \ldots v_f, v_{f+1} \ldots v_{f+a1}\}$. These assignments are based on exactly the same rules described in Section 3.2.1. Building of the nodes, the edges, and the assignment of weights are discussed in the following sections in detail. All modules in the *nth* layer are connected to a node *t0*.

All layers having only the sub-chains of $V_f = \{v_f, v_{f+1} \ldots v_{f+a1}\}$ have a dummy layer. These dummy layers have module assignments that are sub-chains of V_{a2}. The rules for building the nodes and edges are described in Sections 5.2.2 and 5.2.3.

5.2.2 Construction of nodes in the assignment graph G_{a1}

Node *s* is the first node. Nodes in layers *1* to *n* are created, as in Equations 3.3 and 3.4 with the exception that the number of modules assigned to these *n* layers is *f+a1*. Node *t0* is created after the creation of the *n* layers. Next, the dummy layers in A^d are built. The number of dummy layers depends on the number of PEs that can have an assignment of the sub-chains of $V_f = \{v_f, v_{f+1} \ldots v_{f+a1}\}$. The first dummy layer contains only sub-chains of nodes starting with the first module $v_{\beta 1}$ i.e., it contains all nodes $\langle (i,j),k \rangle$ such that

$$i = \beta_1, \text{ and } i \le j \le \beta_{a2} . \tag{5.3}$$

All the rest of the dummy layers from will contain all sub-chains of modules $\langle i,j \rangle$ such that

$$\beta_1 \le i \le \beta_{a2}, \text{ and } i \le j \le \beta_{a2} . \tag{5.4}$$

The last dummy layer will contain only the sub-chains of modules, which end with β_{a2} given by

$$\beta_1 \leq i \leq \beta_{a2} \text{ and } j = \beta_{a2}. \quad (5.5)$$

Finally node *t* is created.

5.2.3 Construction of edges for the assignment graph G_{a1}

The restriction that all PEs need to be utilized is not applicable to this algorithm. Hence empty modules are added to the module graph. This makes it possible to have some PEs not getting any assignment of modules. The set of modules V_{c1} is added *m-1* number of empty modules at the end. The set of modules V_{a2} is added *m-1* number of empty modules on both ends of the chain.

The *contiguity constraint* is applied only during creation of edges in the assignment graph so that any layer in the assignment graph will have only contiguous sub-chains of modules. The contiguity constraint is applicable only to the assignment graph and not to the PEs. The construction of the edges from *s* to *t0* is similar to the edge creation as described in Figure 3.2.1. The edges are built from node *t0* to all nodes in the first dummy layer. For all other layers, Equations 3.3 and 3.4 are applied in the creation of edges. All nodes in the last layer are connected to the node *t.*

5.2.4 Application of constraints

Because both A_l and $A_{d(l,1)}$ correspond to the same w_l , the area constraint to be applied during the execution of the algorithm is given by

$$Area(i, j)_l + Area(a,b)_{d(l,1)} \leq Size_l \quad (5.6)$$

where, *i* through *j* are modules assigned to a layer $A_l \in A^c$ and *a* through *b* are modules assigned to the dummy layer $A_l \in A^d$. The power constraint is implemented in the same fashion as the area constraint.

Similarly, the pin constraints are implemented as follows. If the different links of a PE have different limitations on the number of terminals, then each of the inputs and outputs of the different sets of assignments from the layers A^c and A^d has to be applied separately as

$$Count(i, j)_l + Count(a, b)_{d(l,1)} \leq T_l \quad (5.7)$$

where, i through j are modules assigned to a layer $A_l \in A^c$ and a through b are modules assigned to the dummy layer $A_l \in A^d$. T_l is the maximum number of pin limitation of PE w_l.

5.2.5 Assignment of weights

Weights are applied similarly to Equations 4.5 through 4.9 in Section 4.2.2. In particular, there can be only one dummy layer for any layer in this case.

Input: Module graph G_m.
Algorithm:
Identify G_{m1} as described in Section 5.2.
Construct the assignment graph G_{a1} as described in Sections 5.2.1 through 5.2.5.
Find minimum bottleneck path P from s to t using Algorithm 4.1.
Output: An optimal assignment of chain of modules having a fork, to the n PEs under constraints.

Figure 5.5. Algorithm 5.1.

The algorithm for finding minimum throughput and minimum total execution time is based on Algorithm 4.1. The combination of nodes $C \in S$ is selected from all the nodes in A^c, similar to Algorithm 4.1. So this makes it possible to explore all possible combinations of paths through the assignment graph such that the dummy nodes in the

dummy layers satisfy the constraints for each of the nodes represented by *C*. Consequently, it is possible to assign only valid modules by setting only those modules in the dummy layers which satisfy all the constraints with the nodes in the combination path *C*.

5.3 Proof of Correctness of Algorithm 5.1

Theorem 5.1: Algorithm 5.1 partitions the modules G_{m1} onto the chain of n PEs giving an optimal throughput and total execution time assignment under area, pin, and power constraints.

Proof: The proof can be stated in three steps. The first step is to prove that only paths that give valid assignments are considered and that all valid assignments are taken into account. The second step involves proving that the constraints are satisfied. The final step is to prove that the optimal partitioning is obtained under the constraints.

The use of Equations 3.3 and 3.4 for creating assignments to layers ensures that all possible assignment of modules are created for the first n layers. The use of the Equations 5.3, 5.4, and 5.5 ensures that all assignments are made to the dummy layers A^d. The use of the contiguity constraint mentioned in Section 5.2.3 on the creation of edges ensure that no same modules are reassigned to the same PE. The use of the condition in Section 5.2.1 that all layers having only the sub-chains of $V_f = \{v_f, v_{f+1} \ldots v_{a1}\}$ have a dummy layer each ensure that there is no backward communication between V_{a2} and V_M. Hence applying Algorithm 4.1 results in a shortest path from s to t with only valid assignments and covering all possible assignments.

The application of the constraints during the creation of the nodes and edges as given by Equations 2.16, 2.17, and 2.18 ensures only valid assignments are created in the layers of the assignment graph. The use of the constraints given by Equations 5.9 and 5.10 during the execution of Algorithm 4.1 ensures that all constraints are met for all paths from *s* to *t*.

The assignment of weights to the nodes is given in Section 5.2.5. Applying Algorithm 4.1 to find the shortest path from s to t gives the optimal partitioning of the

modules G_{m1} to the chain of PEs for either minimum bottleneck or minimum total execution time.

Note: Algorithm 5.1 can be extended to partition two disjoint sets of chains of modules on to the same chain of PEs.

Chapter 6

Results

This chapter contains a description of the experiments that were performed to illustrate the runtime of the Algorithms 3.1, 4.1, 4.2, and 5.1 with and without the application of constraints. A description of the verification procedure undertaken to check the correctness of the implementation of these Algorithms 3.1, 4.2 and 5.1 is also given.

6.1 Implementation of algorithms 3.1, 4.1, 4.2, and 5.1

Algorithms 3.1, 4.1, 4.2 and 5.1 were implemented in C++ language to run on a SUN SPARCstation. The program contains three major components. The first component reads in the number of processors, number of modules, input module graph, and input PE graph. The input module graph and PE graph were created for each implementation separately. Other parameters required by the program, such as the time of execution of the modules, communication times between links, the area limitation of the PEs, and the area of the modules were either made uniform with some constant value or generated randomly within a certain range by the program. The second component of the program builds the assignment graph required for execution of all the algorithms. Building of the assignment graph consists of building the nodes and edges and then assigning weights. The third component consists of the shortest path algorithm, which is executed on the assignment graph and the results obtained.

6.1.1 Results for Algorithm 3.1 implementation

The program was executed and results were obtained, as regards to the time taken to build the assignment graph and the time needed for execution of the algorithm. The

number of nodes and edges created were also recorded. Timings were taken from the implementation of Algorithm 3.1 under no constraints and the results are as shown in Table 6.1. The number of PEs was kept constant at 10. It can be seen that as the size of the input modules increases, the number of nodes and edges of the assignment graph increases and so does the runtime.

Table 6.1. Table containing the results for Algorithm 3.1 implementation.

NUMBER OF MODULES	NUMBER OF NODES	NUMBER OF EDGES	TIME FOR NODE CREATION (SEC)	TIME FOR EDGE CREATION (SEC)	RUNTIME OF ALGORITHM 3.1 (SEC)
10	45	68	~0	~0	~0
15	235	688	2.85	~0	~0
20	575	2483	3.06	0.01	~0
25	1065	6078	3.30	0.02	~0
30	1705	12098	3.52	0.05	~0
35	2495	21168	3.77	0.10	0.01
40	3435	33913	4.02	0.17	0.02
45	4525	50958	4.38	0.27	0.02
50	5765	72928	4.81	0.43	0.03
55	7155	100448	5.35	0.63	0.04
60	8695	134143	5.69	0.89	0.07
65	10385	174638	6.14	1.16	0.08
70	12225	222558	6.63	1.63	0.10
75	14215	278528	7.21	2.16	0.13
80	16355	343173	8.16	2.76	0.15
85	18645	417118	8.93	3.60	0.20
90	21085	500988	9.69	4.46	0.23
95	23675	595408	10.72	5.64	0.28
100	26415	701003	11.68	6.96	0.35
130	46005	1604248	15.13	10.3	0.83
140	53735	2023063	27.88	14.56	1.14
150	62065	2509078	49.63	20.40	1.53
160	70995	3067293	82.12	28.67	2.45
170	80525	3702708	126.6	38.72	2.57
200	112715	6122153	315.1	58.70	6.38
225	143665	8800878	556.15	63.87	6.95
250	178365	12165228	907.36	121.72	11.87

The second set of results for the same program recorded the results when a constraint was implemented. For this purpose the areas of all modules were made the same. The maximum number of modules *300* was divided by the number of PEs *10* to get an average of *30*. So the limitation of the number of modules on these PEs was varied

from *b* to no limitation in four steps as *30, 50, 100,* and *no limitation.* This procedure was carried out for *m* ranging from *150* to *300* in steps of *50.* These results are shown in Table 6.2. Note that as the limitation value on the number of modules is increased, the number of nodes and edges created decreased significantly and consequently the runtimes were reduced as well.

Table 6.2. Results for Algorithm 3.1 under increasing constraints.

NUMBER OF MODULES	LIMITATION ON NUMBER OF MODULES PER PE	NUMBER OF NODES CREATED	NUMBER OF EDGES CREATED	RUNTIME OF ALGORITHM 3.1 (SEC)
150	30	30423	707460	0.33
150	50	46703	1615100	0.79
150	100	73403	3184341	1.92
150	None	80373	3360641	2.15
200	30	42423	1022460	0.48
200	50	66703	2490100	1.20
200	100	113403	6460716	4.55
200	None	147073	8294366	6.59
250	30	54423	1337460	0.62
250	50	86703	3365100	1.65
250	100	153403	9960200	7.47
250	None	233773	16893091	41.84
300	30	66423	1652460	0.76
300	50	106703	4240100	2.04
300	100	193403	13460200	18.68

The third set of results consisted of a plot of throughput versus the constraints on the number of modules on any PE. Throughput values versus the constraint on the number of modules on any PE were recorded for *m=100* with the limitation on the PEs, t_c, ranging from *5* to *50*. These results were plotted in Figure 6.1. It can be seen that as the limitation value on the number of modules residing on any PE is increased, the bottleneck value increases. And for some value of limitation less than the average, *200/10=20*, the

bottleneck becomes infinity which means that for $t_c < 10$, no feasible solution exists. The number of PEs was kept constant at 10.

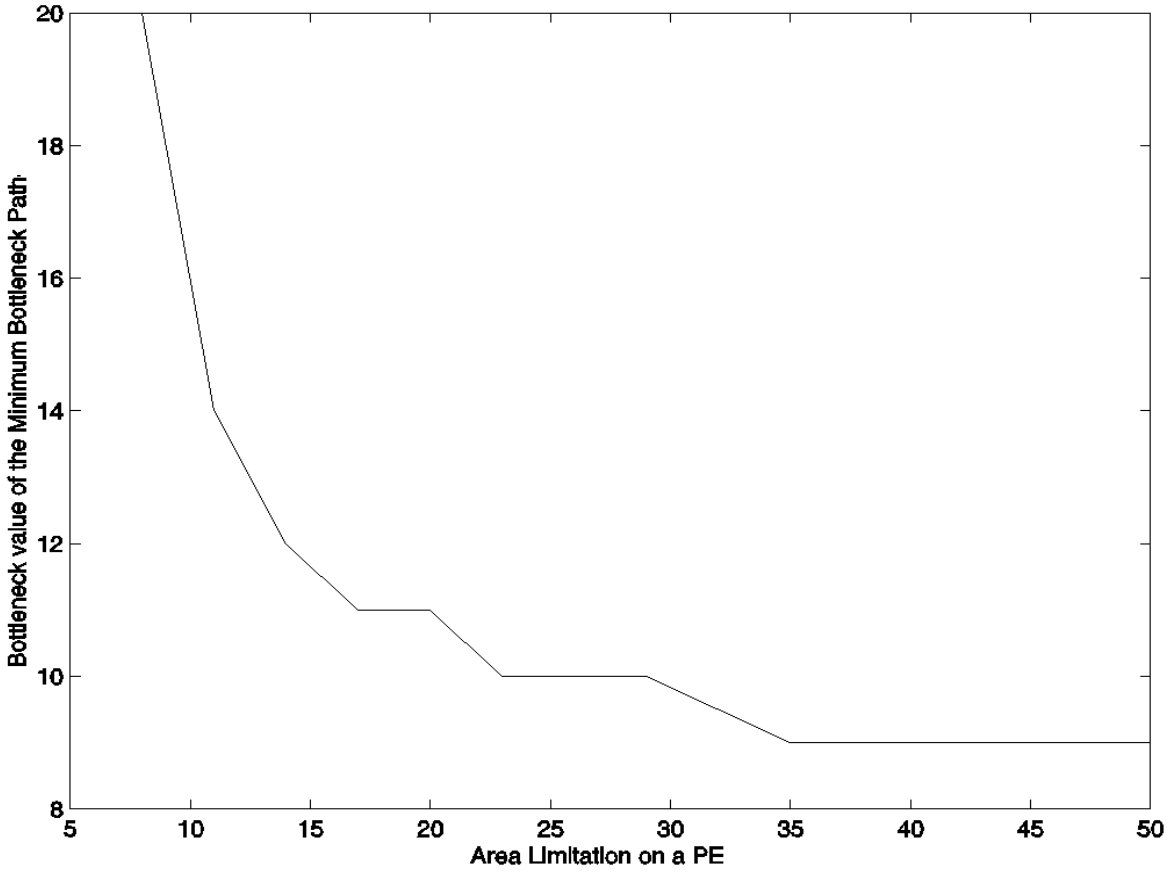

Figure 6.1. Plot of bottleneck values versus the limitation of modules on a PE.

Figure 6.2 contains two different implementations of Algorithm 3.1. One of the implementations is as described in Section 3.2. This consists of building the edges in the assignment graph. Hence more memory storage is required in this case. Also some time is required to create the edges. In the other implementation, described in Section 3.4, no edges are built. Hence there is no overhead in building the edges. It can be seen from Figure 6.2 that the runtimes of the implementation without the edges are much smaller than the implementation with edges for the assignment graph.

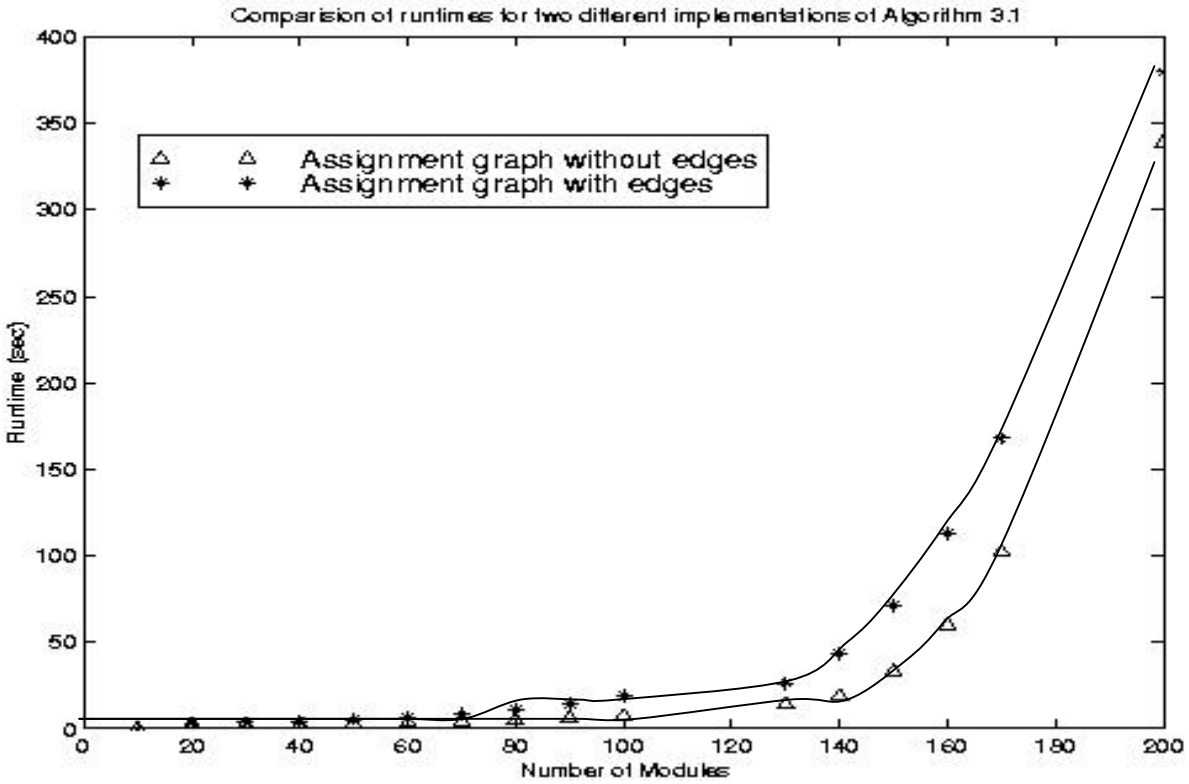

Figure 6.2. Comparison of two plots representing the runtimes of two different implementations of Algorithm 3.1.

Verification procedure for Algorithm 3.1

A procedure for checking Algorithm 3.1 was implemented. This consisted of a procedure for generating all possible assignments one by one without concern for runtime efficiency. For each assignment of modules to PEs, the bottleneck value or total execution time was recorded if that was the minimum. For the same set of inputs, Algorithm 3.1 was executed and the bottleneck value or total execution time was obtained. The results of both were compared. For all cases the bottleneck or total execution times matched perfectly. For many cases the actual assignments differed, but the bottleneck values and total execution times matched for all cases. This is because there can be more than one possible assignment for which the bottleneck values and total execution times are the same.

6.1.2 Results for Algorithm 4.1 and 4.2 implementation

Algorithm 4.1 was executed for two cases. In the first case, the PEs had an inter-connection pattern that was a ring. This required the creation of one dummy node in the assignment graph. The results for the implementation of Algorithm 4.1 are given in Table 6.3. These readings contain the number of nodes and edges created, the time required for their creation, and the runtime of Algorithm 4.1. These results were recorded for the case when no constraint was applied and the number of PEs was 10. It can be seen that as the size of the graph increases, the runtime also increases.

The second case consisted of four PEs with an inter-connection pattern as shown in Figure 6.3. This case required two dummy layers in the assignment graph. All PEs were assumed to be uniform. Only one trail was explored and Algorithm 4.1 was used to partition chain of modules onto this trail of PEs. The trail chosen was *1, 2, 3, 1, 5,* and *3*. The results for this case are given in Table 6.4. It can be observed that for a small trail, the time for creation of the graph is small, but the runtime of Algorithm 4.1 is large and the number of times Algorithm 3.1 is called also large for large values of *m*.

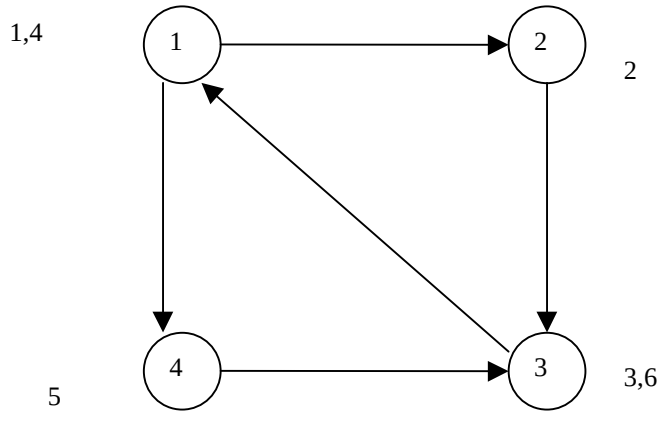

Figure 6.3. Directed arrows showing a trail in a PE graph.

Table 6.3. Results of Algorithm 4.1 for the case of PE inter-connection being a ring of 10 PEs.

NUMBER OF MODULES	NUMBER OF NODES	NUMBER OF EDGES	TIME FOR NODE CREATION (SEC)	TIME FOR EDGE CREATION (SEC)	RUNTIME OF ALGORITHM 4.1 (SEC)	NUMBER OF TIMES ALGORITHM 3.1 CALLED
25	1113	5710	0.00	0.02	0.03	15
50	7463	93560	3.01	0.39	1.89	40
75	19438	387660	4.17	1.87	12.47	65
100	37038	1013010	7.30	5.6	46.72	90
125	60263	2094610	10.24	10.93	130.92	115
150	89113	3757460	58.37	23.86	336.02	140
175	123588	6126560	198.41	39.9	738.75	165
200	163688	9326910	430.86	65.8	1397.28	190

Table 6.4. Results for the two dummy PE case for the trail shown in Figure 6.3.

NUMBER OF MODULES	NUMBER OF NODES	NUMBER OF EDGES	TIME FOR NODE CREATION (SEC)	TIME FOR EDGE CREATION (SEC)	RUNTIME OF ALGORITHM 4.1 (SEC)	NUMBER OF TIMES ALGORITHM 3.1 CALLED
10	73	145	~0	~0	~0	75
15	243	790	2.77	0.01	0.12	550
20	513	2310	2.99	0.01	1.07	1800
25	883	5080	3.24	0.02	5.46	4200
30	1353	9475	3.5	0.05	24.77	8125
35	1923	15870	3.8	0.08	77.72	13950
40	2593	24640	4.04	0.12	235.61	22050
50	4233	50805	4.34	0.12	1121.14	46575

The second set of results for the implementation of Algorithm 4.1 was to record the results when a constraint was implemented. For this purpose the areas of all modules were made the same, and the maximum number of modules considered $m=200$ was divided by the number of PEs $n=10$ to get an average of $b=20$. The limitation of the number of modules on these PEs was varied from b to no limitation in three steps as *50, 100,* and *200.* This procedure was carried out for m values of *50, 100,* and *200.* These results are shown in Table 6.5. Note that as the limitation on the number of modules

increased, the number of nodes and edges created decreased significantly with a corresponding decrease in runtime. The number of PEs was kept constant at 10. Verification of Algorithm 4.1 was similar to the verification done for Algorithm 3.1.

Table 6.5. Results showing reduction in graph size and execution time of Algorithm 4.1 under increasing constraints.

NUMBER OF MODULES	LIMITATION ON NUMBER OF MODULES PER PE	NUMBER OF NODES CREATED	NUMBER OF EDGES CREATED	RUNTIME OF ALGORITHM 4.1 (SEC)
50	20	5533	68040	0.65
50	50	7463	93560	1.88
50	100	7463	93560	1.88
50	200	7463	93560	1.88
100	20	14533	228040	2.23
100	50	29578	828970	20.07
100	100	37038	1013010	46.74
100	200	37038	1013010	46.74
200	20	32533	548040	5.27
200	50	74578	2825100	68.58
200	100	126653	7301070	517.43
200	200	163688	9326910	1397.28

Algorithm 4.2 was executed for a chain of modules of $m=100$ and for a PE graph as shown in Figure 6.4. The results are recorded in Table 6.6. A limitation of *50* modules per PE is imposed. All PEs are assumed to be uniform. But the links are not uniform. All unique trails are shown in Table 6.6. We can see that for the sequences where a PE is revisited, Algorithm 4.1 is called more that once. It can be observed that as the constraints are increased, the graph size reduces and hence the runtime also reduces.

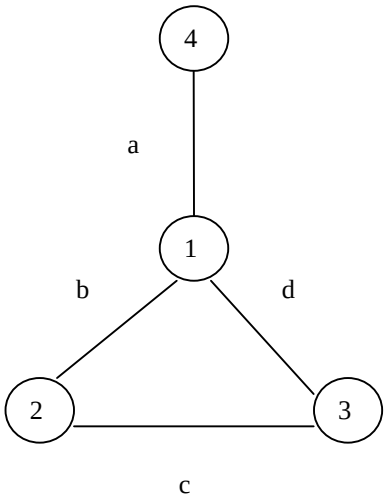

Figure 6.4. A PE graph with four PEs and four links

Table 6.6. Results for Algorithm 4.2.

TRAIL SEQUENCE	NUMBER OF NODES CREATED	NUMBER OF EDGES CREATED	CREATION TIME FOR NODES AND EDGES (SEC)	RUNTIME FOR ALGORITHM 4.1(SEC)	NUMBER OF TIMES ALGORITHM 4.1 CALLED
4,1,2,3,1	10828	2400096	4.3	428.08	3575
1,4	7353	125095	2.4	3.089	1
1,2,3,1,4	10828	240096	4.31	5.45	50
2,1,4	3778	5098	0.39	0.01	1
2,3,1,4	7353	125095	2.33	0.06	1
2,1,3,2	7353	125095	2.33	3.08	50
4,1,2,3,1	10828	2400096	4.35	428.00	3575
1,3,2,1,4	10828	240096	4.32	5.40	50
2,3,1,2	7353	125095	2.33	3.10	50
3,1,2,3	7353	125095	2.32	3.09	50
3,2,1,3	7353	125095	2.32	3.10	50
3,1,4	3778	5098	0.40	0.01	1
3,2,1,4	7353	125095	2.33	0.06	1

Results for a CCM of type Figure 1.1:

The structure of the CCM of Figure 1.1 considered for implementation of Algorithm 4.1 and 4.2 is as follows. Each host PC was a general purpose CPU. The embedded single PE attached to the host PC was an FPGA. This FPGA was connected to the host PC with a communication link of cost α. The number of such units was *8*. The total number of PEs was 16. All the host CPUs and the embedded FPGAs were connected to each other by a crossbar link with a communication cost of β, where $\alpha=\beta/10$. The constraint on the number of modules per PE was $1/24^{th}$ the total number of modules in the chain. There were three types of modules, *A, B,* and *C,* whose weights are given in Table 6.7 for the case of CPUs and FPGAs.

Table 6.7. Weights of the modules.

	WEIGHT OF THE MODULE ASSIGNED TO CPU	WEIGHT OF THE MODULE ASSIGNED TO FPGA
Module A	2	2
Module B	1	4
Module C	4	1

One of the trails for this CCM graph was chosen to demonstrate Algorithm 4.1. The longest trail containing 8 CPUs and 8 FPGAs in an alternating sequence was selected. The modules *A, B,* and *C* were generated randomly and their average proportion was $1/3^{rd}$ of the total number of modules. The runtimes are given in Table 6.8.

A chain of modules was partitioned onto the structure of the CCM using Algorithm 4.2. For each chain, 256 possible trails were explored using Algorithm 4.2. Algorithm 4.1 was called for each trail. The total runtime for Algorithm 4.2 was recorded as 7.84, 9.86, and 15.44 minutes on average for chains of 50, 75, and 100 modules respectively. It can be observed that the runtimes are tractable for this practical system.

Table 6.8. Runtimes for implementation of Algorithm 4.1 for CCM of Figure 1.1.

NUMBER OF MODULES	NUMBER OF NODES CREATED	NUMBER OF EDGES CREATED	TOTAL EXECUTION TIME (SEC)
50	4933	16100	0.19
75	14115	69460	2.83
100	28015	184239	6.51
125	46633	383850	14.73
150	69969	691706	57.92
175	98023	1131220	180.61
200	130795	1725805	395.67

6.1.3 Results for the implementation of Algorithm 5.1

The runtimes for the implementation of Algorithm 5.1 are as shown in Table 6.9. The number of PEs was kept constant at 4. The number of modules was varied from 13 to 23 in steps of 2. As the number of modules increases, it can be seen that the number of times Algorithm 4.1 called, increases drastically. The fork was at module number 2. One of the branches of the fork had 5 modules while the number of modules of the other branch were varied.

Table 6.9. Results for the implementation of Algorithm 5.1.

TOTAL NUMBER OF MODULES	NUMBER OF NODES CREATED	NUMBER OF EDGES CREATED	TIME FOR CREATION OF NODES AND EDGES	RUNTIME OF ALGORITHM 5.1	NUMBER OF TIMES ALGORITHM 4.1 CALLED
13	90	153	~0	0.29	5625
15	120	232	~0	1.84	38416
17	158	351	~0	11.43	164025
19	204	518	~0	62.50	527076
21	258	741	~0	211.64	1399489
23	320	1028	0.01	680.36	2400000

Chapter 7

Conclusions and Suggestions

7.1 Conclusions on the thesis

Adaptation of Bokhari's algorithm proved to be very useful for partitioning in the case of CCMs. It has been observed that for the case of a chain of modules partitioned over a chain of PEs, Bokhari's algorithm can be applied successfully. The algorithm can be easily implemented for obtaining optimal throughput and total execution time under constraints. There is flexibility in the application of the area, pin, and power constraints. The results for the same show that as constraints are applied, the algorithm takes less time to execute. In practice, one expects significant constraints, thus these runtimes are more realistic. A point to be noted is that the model for the modules and PEs is a chain; this restricts the direct application of the algorithm to fewer cases.

The method of implementing the assignment graph without constructing edges reduces time to construct the graph and also to execute the algorithm. There is a reduction in the memory storage because of not creating the edges.

The second algorithm presented in Chapter 4 is quite useful because this algorithm can handle any inter-connection of the PEs. The complexity of this algorithm is a relatively large polynomial, but with the application of constraints the practical execution times are reduced drastically.

The third algorithm presented in Chapter 5 provides an extension in the model of the module graph to allow branches. A large number of data dependencies come under this category of module graph and hence it is quite useful. The complexity of this algorithm is also a very large polynomial, but as the constraints are applied, the practical execution times are reduced drastically.

7.2 Suggestions for future work

A number of improvements can be made to the set of partitioning algorithms in this thesis in order to solve more general partitioning problems. By developing effective heuristics based on the results obtained for particular types of data dependencies and PE inter-connections, the algorithms presented in this thesis would help in obtaining near optimal solutions to more general partitioning problems which are NP-complete.

Adoption of Bokhari's [1] doubly weighted graph algorithm for optimizing a combination of throughput and execution time for the second and the third algorithm presented in Chapters 4 and 5 respectively, can be done. The doubly weighted graph algorithm works for the Algorithm 3.1 directly with few changes.

Generalizing the single fork module graph algorithm to work for a PE graph that is arbitrarily connected, is very similar to the idea presented in Chapter 4. This can be done by creating more dummy nodes and applying the constraints. However, the run time of that algorithm will be very high.

As mentioned in Chapter 4 and 5, Algorithms 4.2 and 5.1 have inherent parallelism. Further work can be done to implement these algorithms in parallel.

References

[1] S. H. Bokhari, "Partitioning Problems in Parallel, Pipelined, and Distributed Computing," IEEE Transactions on Computers, vol. 37, no. 1, pp. 48-56, Jan 1987.

[2] C. M. Fiduccia and R. M. Mattheyeses, "A Linear-time Heuristic for Improving Network Partitioning," Proceedings of the 19th Annual Design Automation Conference, pp. 241-247, July 1982.

[3] B. Krishnamurthy, "An Improved Min-Cut Algorithm for Partitioning VLSI Networks," IEEE Transactions on Computers, vol. C-33, no. 5, pp. 438-446, 1984.

[4] S. H. Bokhari, "A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in a Distributed Processor System," IEEE Trans. Software Eng., vol. SE-7, no. 6, pp. 583-589, Nov 1981.

[5] D. Towsley, "Allocating Programs Containing Branches and Loops Within a Multiple Processor System," IEEE Trans. Software Eng., vol. SE-12, no. 10, pp. 1018-1024, Oct 1986.

[6] H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," IEEE Trans. Software Eng., vol., SE-3, pp. 85-93, Jan. 1977.

[7] H. S. Stone and S. H. Bokhari, "Control of Distributed Processes," IEEE Comput. Mag., vol. 11, pp. 97-106, July 1978.

[8] S. M. Shatz, J-P. Wang, "Task Allocation for Maximizing Reliability of Distributed Computer Systems," IEEE Transactions on Computers, vol. 41, no. 9, pp. 1156-1168, Sep 1992.

[9] S. H. Bokhari, "Assignment Problems in Parallel and Distributed Computing," Kluwer Academic Publishers, Norwell, MA, 1987.

[10] D. M. Nicol, "Parallel Algorithms for Mapping Pipelined and Parallel Computations," ICASE Rep. 88-2, NASA Contractor Rep. 181655, Apr. 1988.

[11] C-H. Lee, D. Lee, and M. Kim, "Optimal Task Assignment in Linear Array Networks," IEEE Transaction on Computers, vol. 41, no 7, pp. 295-306, July 1992.

[12] S. R. Sternberg, "Biomedical Image Processing," IEEE computer, vol. 16, pp. 22-34, Jan. 1983.

[13] J. H. Saltz, "Parallel and Adaptive Algorithms for Problems in Scientific and Medical Computing," Ph.D. dissertation, Dep. Comput. Sci., Duke Univ., 1985.

[14] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal, "The RAW Benchmark Suite: Computation Structures for General Purpose Computing," Technical Report TR-709, MIT LCS, Mar. 1997.

[15] H-K. Yun, H. F. Silverman, "A Distributed Memory MIMD Multi-Computer with Re-configurable Custom Computing Capabilities," IEEE Workshop for FPGAs for Custom Computing Machines, IEEE Computer Society Press, Napa Valley, CA, pp. 8-13, 1997.

[16] P. Bertin and H. Touati, "PAM Programming Environments: Practice and Experience," Proc. IEEE Workshop FPGAs for Custom Computing Machines, CS Press, Los Alamitos, Calif., pp. 133-139, 1994.

[17] M. Gokhale, W. Homes, A Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti "Building and Using a Highly Parallel Programmable Logic Array," IEEE Computer, vol. 24, no. 1, pp. 81-89, Jan. 1991.

[18] B. Box. "Field Programmable Gate Array Based Re-configurable Preprocessor," Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, CS Press, Los Alamitos, Calif., pp. 40-49, 1994.

[19] J.M. Arnold, D.A. Buell, and E.G. Davis, "Splash 2," ACM Symposium on Parallel Algorithms and Architectures, ACM Press, New York, pp. 316-322, 1992.

[20] P. M. Athanas and H. F. Silverman, "Processor Reconfiguration Through Instruction Set Metamorphosis", IEEE Computer, vol. 26, no. 3, pp. 11-18, Mar. 1993.

[21] M. R. Garey and D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," W. H. Freeman, San Francisco, CA, 1979

[22] D. Fernandez-Baca, "Allocation Modules to Processors in a Distributed System," IEEE Trans. on Software Eng., vol. 15, no. 11, pp. 1427-1436, Nov. 1989.

[23] N. Sherwani, "Algorithms for VLSI Physical Design Automation," Second edition, Kluwer Academic Publishers, Norwell, MA, 1995.

[24] B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," Bell System Technical Journal, vol. 49, pp. 291-307, 1970.

[25] A. Chatterjee and A. Hartley, "A New Simultaneous Circuit Partitioning and Chip Placement Approach Based on Simulated Annealing," Proceedings of the 27th Design Automation Conference, Orlando, FL, pp. 36-39, 1990.

[26] J. Greene and K. Supowit, "Simulated Annealing Without Rejected Moves," Proceedings of International Conference on Computer Design, Port Chester, NY, pp. 658-663, Oct. 1984.

[27] D. G. Schweikert and B. Kernighan, "A Proper Model for the Partitioning of Electrical Circuits," Proceedings of the 9th Design Automation Workshop, Port Chester, NY, pp. 57-62, 1972.

[28] C. Kring and A. R. Newton, " A Cell-Replication Approach to Mincut Based Circuit Partitioning," Proceedings of IEEE International Conference on Computer-Aided Design, Santa Clara, CA, pp. 2-5, Nov. 1991.

[29] M. K. Goldberg and M. Burstein, "Heuristic Improvement Technique for Bisection of VLSI Networks," Proceedings of IEEE International Conference on Computer Design, Port Chester, NY, pp. 122-125, 1983.

[30] Y. Wei and C. Cheng, "Towards Efficient Hierarchical Designs by Ratio Cut Partitioning," Proceedings of IEEE International Conference on Computer-Aided Design, Santa Clara, CA, pp. 298-301, 1989.

[31] C. J. Alpert, J-H. Huang, and A. B Kahng, “Multilevel Circuit Partitioning,” Proceedings of the 34th Design Automation Conference, Anaheim, CA, pp. 530-533, June 1997.

[32] V. C. Chan and D. Lewis, "Hierarchical Partitioning for Field-Programmable Systems," IEEE Int. Conf. Computer-Aided Design-97, pp 428-435, Nov. 1997.

[33] Xilinx, Inc., "The Programmable Logic Data Book," San Jose, CA, pp. 2-4, April 1995.

[34] D.A. Buell and K.L. Pocek, "Custom Computing Machines: An Introduction," Journal of Supercomputing, vol. 9, pp. 219-230, 1995.

[35] D. A. Buell, J. M. Arnold, and W. J. Kleinfeldner, "Splash 2: FPGAs in a Custom Computing Machine," IEEE Computer Society Press, Los Alamitos, CA, 1996.

[36] D. B. West, "Introduction to Graph Theory," Prentice-Hall, Inc., Upper Saddle River, NJ, 1996.

[37] R. D. Hudson, D. L. Lehn, and P. M. Athanas, "A Run-Time Reconfigurable Engine for Image Interpolation," Proceedings of the Workshop of FPGAs for Custom Computing Machines, Napa Valley, CA, 1998.

[38] Virginia Tech Configurable Computing, "Tower of Power," http://www.ee.vt.edu/~ccm/top/top.html, 1998.

[39] I. S. I. East, " System Level Applications of Adaptive Computing," http://www.east.isi.edu/slaac_intro.html, 1998.

Vita

Suresh Chandrasekhar was born on June 12, 1974 in Bangalore, India. Suresh attended school in the same city. He attended school at St. Joseph's Boys High School, Bangalore till 1990, after which he continued his education at M.E.S College of Arts, Science, and Commerce, Bangalore till 1992 for his P.U.C. in Science (eleventh and twelfth grade). Suresh graduated from R. V. College of Engineering, Bangalore in August 1996 with a B. E. in Electronics and Communication.

After completing his undergraduate degree, Suresh started his graduate study at Virginia Tech in August 1996. In August 1998 he graduated from Virginia Tech with an M.S. in Electrical Engineering.

Later in August 1998 he plans to work for Intel Corporation, Portland, Oregon as a Component Design Engineer.