A DESIGN METHODOLOGY FOR ALGORITHM-SPECIFIC ARCHITECTURES[1]

Sanjay R. Deshpande[2] and James C. Browne

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-87-06 February 1987

[1]This research was supported by Space and Naval Warfare Systems Command Contract N00039-86-C-0167, and by the Department of Energy under contract number DE-FG05-85ER25010.

[2]Department of Electrical and Computer Engineering.

Abstract

The optimality criterion of keeping the utilization of the resources of a dedicated, special purpose architecture to the maximum is interpreted as being equivalent to keeping them busy at all times. This is in turn understood in terms of the concept of *conservation of tokens*. The problem of designing an optimal architecture is expressed in terms of a system of simultaneous equations. It is shown that it is possible to build an optimal architecture, if and only if, an integer solution exists for the set of equations. A particular instance of the Generalized Linear Pipeline is illustrated as an example of such an architectural design. It is also shown that solutions exist for trees and graphs which are combinations of trees. Furthermore, certain conditions are specified, which when satisfied ensure integer solutions for non-tree computation graphs.

1 Introduction

One of the metrics of goodness of a special purpose architecture is the utilization of its individual elements. Although the all-important metric of an architecture is its speed of processing, it often depends entirely of the technology employed and the formulation of the problem itself. When these two factors are prespecified, it is the utilization of the resources that becomes important. The process involved in designing a systolic array for a particular algorithm is that of formulating the algorithm in a manner suitable for pipelining the basic operations, with multi directional data flow. The operation of simple linear pipelines is well understood [1], and as a result, they are employed heavily in designing systolic arrays. It is known that, by designing each stage of the pipeline to have the same delay, and by feeding data items after unit delay, it is possible to achieve high throughput and high utilization (utilization = 1). In designing systolic arrays, multiple independent computations are often interleaved to satisfy dependencies which conflict with this basic principle of pipelines [2].

In most instances, a dedicated architecture is sought for repetitively executing algorithms, that is, for algorithms which repeat a set of operations infinitely many times over differing data sets. We also will restrict the discussion in this report to such algorithms. Typically, steady state solutions are obtained, which, given an infinite repetition of the computation involved, are most interesting.

The rest of the report is organized as follows. In Section 2 we define the notation used to capture computations. In Section 3 are given characteristics of the solution space. The problem of designing algorithm-specific architectures is also defined there. Finally, in Section 4, results are obtained regarding certain classes of computation graphs. An example of Generalized Linear Pipeline is also included in Section 4.

2 Specification of Computations

The computations expressed by an algorithm are specified using directed acyclic graphs called the Data Dependency Graphs (DDG's). The nodes of a DDG represent computations and the edges represent data dependencies between pairs of nodes. The computations specified by the nodes are considered to be atomic; that is, the computations are not to be split into simpler operations. Also by convention, the node upon which an edge is incident is "data-dependent" on the node from which the edge originates. Several edges may originate at a node (output edges), and several may be incident upon it (input edges). The nodes of a DDG which have no input edges are called *input nodes*, and those

with no output edges are called *output nodes*. The rest of the nodes are called *intermediate nodes*. The input and output nodes represent interfaces to input and output devices.

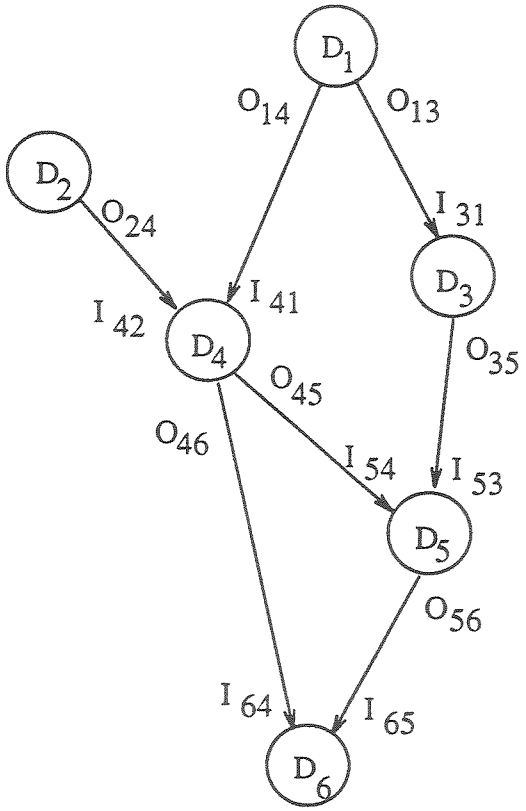

Figure 1: An Example Of A DDG

With each node of a DDG is associated an atomic unit of computation of arbitrary complexity. There is also associated a delay of execution of the computation. A node executes its computation by *firing*. When a node fires, it absorbs input data and produces output data after a certain delay, at which point it is ready to fire again.[2] Upon firing, the node absorbs a prespecified number of data items (input

[2]The firing of a node for absorbing input can be conceptually separated from its firing for producing output. But under the condition of steady state, the two rates are the same and are considered synonymous. Under the pipelined behavior of nodes, however, the two firings have to be separately viewed.

tokens) from each of its input dependencies and produces a prespecified number of data items (output tokens) for each of its output dependencies. (The number of tokens is assumed to be an integer.) This is represented by assigning the corresponding numbers of tokens to input and output edges. An example of a DDG is shown in Figure 1.

3 Solution Space and Problem Statement

Computation Blocks: We have stated above that the computations expressed by the nodes of a DDG are atomic, and that it is not possible to resolve them into simpler subcomputations. The result of this on the solution space is that the nodal computations are assumed to be implemented as dedicated hardware modules and these modules are to be utilized as building blocks to construct the architecture. The delays mentioned in Section 2 refer to the delays of these hardware modules. The hardware modules are referred to as computation or function blocks or elements.

We further assume that the computation elements have finite buffer sizes. In general, we will allow large buffer sizes, but the sizes will not be allowed to grow without bound.

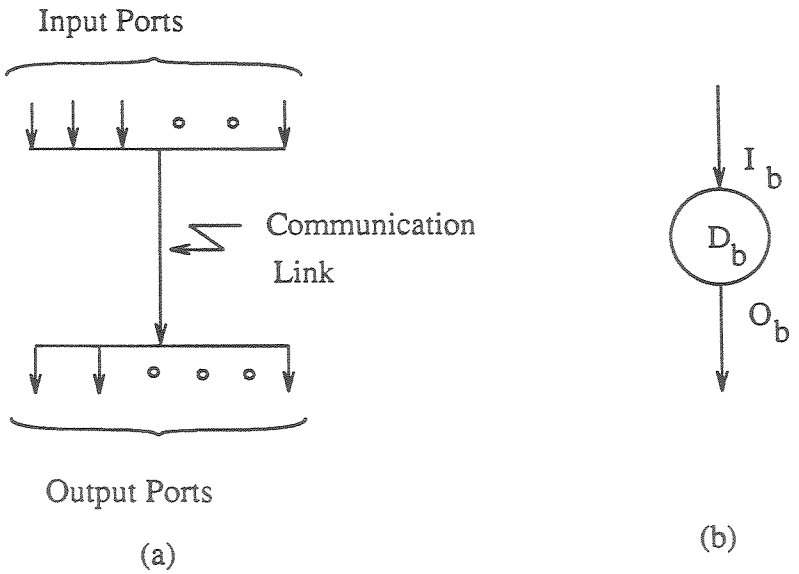

Figure 2: Model Of A Bus

Communication Elements: The dependencies represented by the edges of the DDG result in an

intermodule communication network. The basic building block for the intermodule communication network will be a bus (also referred to as a communication block or element). A bus is defined here as a unidirectional serial communication medium (wires with driver-receiver pairs, for example) with multiple input and output ports (see Figure 2a). At any given time, a bus connects one of its input ports to one of its output ports. The switching between pairs of modules can be achieved by turning on an appropriate driver-receiver pair at the appropriate time.

For analysis, however, a communication element is represented by a model similar to a computation element (see Figure 2b) except that it has a single input arc and a single output arc, and the numbers of tokens associated with these are always same ($I_b = O_b$). This model of a bus allows us to integrate it in the general framework required to analyze a DDG by letting us treat it like a computation block. The only difference is that a computation block modifies the data it absorbs, while the communication block does not. A communication block absorbs one token and produces one token when it fires, and has a delay value D_b[3].

Global System Clock: We will impose a requirement of synchronism over the entire architecture. That is, all intermodule data transfers take place at ticks of a system-wide clock. We do not make any assumptions about the internal clocks of the computation blocks. The existence of a single global clock implies that all relevant delays are integral multiples of the global clock period.

Problem Statement: Having described the architectural model, we can state the design problem as follows: Given a DDG as described above, we must construct an architecture which is optimal according to the criterion of maximum resource utilization. In this report, we will construe the condition of maximum utilization to mean full utilization (100%). Within the scope of this criterion we will attempt to minimize hardware cost by eliminating unnecessary hardware.

The problem of designing an architecture is that of finding appropriate numbers of individual types of computation blocks, defining a communication structure over them, and specifying the execution or firing schedule. Since the solution architecture will have 100% utilization, execution schedule is trivial - one of greedy scheduling, where a block fires as soon as its dependencies are satisfied and after it has completed its current execution; there is no idle time, except at the start of the algorithm. For

[3]It is not necessary to have uniform delay for buses corresponding to different dependencies; the delays can be allowed to vary, for example, to model different sizes of tokens.

communication elements the common link connecting the input and output ports is utilized 100%.

4 Analysis of Data Dependency Graphs

4.1 Principle of Conservation of Tokens

We assume that communication blocks (i.e. buses) have no storage capability and also that the communication blocks are strictly transmitive. That is, the number of tokens entering a communication block equals the number of tokens leaving it. This is held true at all times and is called the *Principle of Conservation of Tokens*. Thus, according to the principle, in a steady state of execution, the rate of tokens entering a communication block must equal the rate of tokens leaving it. Since the communication blocks realize dependencies, the said principle can be used to analyze the latter and the function blocks connected via them.

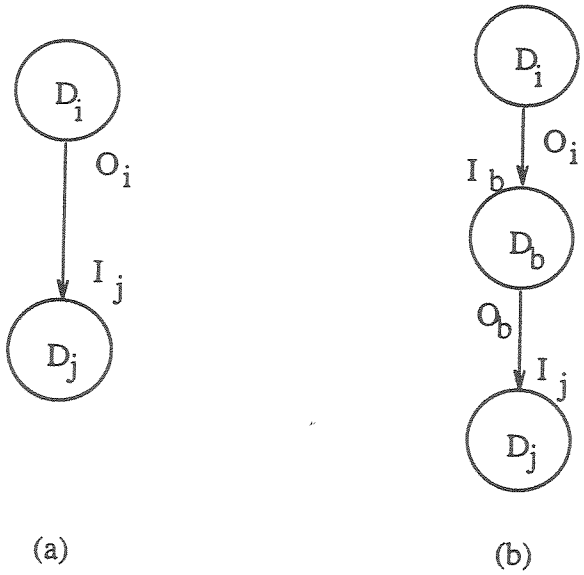

Figure 3: A Data Dependency

4.2 Analysis of DDGs

Consider nodes i and j connected by a dependency as shown in Figure 3a. The communication blocks, which realize the dependencies, interpose between every pair of function blocks connected by a dependency. Prior to the analysis, it is necessary to modify the DDG by replacing each dependency with a node representing the communication block in the desired architecture and a pair of dependencies connecting it to the original source and destination nodes, in a manner consistent with the original dependency. We call this new graph a Modified DDG (MDDG). See Figure 3b.

By the principle of conservation of tokens, the rate of influx of tokens equals the rate of efflux of tokens for the communication element. Under steady state conditions, the rate of influx must equal the rate production of tokens by node i. Similarly, the rate of efflux must equal the rate of absorption of tokens by node j. Assuming that n_i and n_j are numbers of blocks of types i and j respectively, and n_b are the number of communication blocks, we have the following relations:

$$n_i \cdot \frac{O_i}{D_i} = n_b \cdot \frac{I_b}{D_b} \quad , \quad and \quad n_b \cdot \frac{O_b}{D_b} = n_j \cdot \frac{I_j}{D_j} \; .$$

Given that $I_b = O_b$, we also have

$$n_i \cdot \frac{O_i}{D_i} = n_j \cdot \frac{I_j}{D_j}$$

Each dependency in a DDG gives rise to a pair of equations of the form shown above. Thus a DDG can be expressed in terms of |E| simultaneous equations, where E is the set of edges of the MDDG; these equations are henceforth referred to as the set of token-conservation equations for the DDG. If there are |V| nodes in the MDDG, the system of equations is of the form $\mathbf{Rn} = \mathbf{0}$, where **R** is an $|E| \times |V|$ matrix relating the token rates, and **n** and **0** are vectors of length |V|. The following theorem shows the relationship between the system of equations describing a DDG and the desired architecture. In what follows, whenever we refer to a solution to a set of token-conservation equations for a DDG, unless otherwise specified, we imply that we analyze the corresponding MDDG, and in fact refer to the solution to its token-conservation equations.

Theorem 1: It is possible to design a 100% utilized architecture for a DDG, if and only if, the system of equations, $\mathbf{Rn} = \mathbf{0}$, describing the DDG, has an integer solution.

Proof:

Only if:

Assume that an architecture can be built from primitives defined in Section 3 for a DDG, and that the utilization of the computation and communication elements is 100%. The fact that an architecture can be built implies that a consistent set of communication elements can be configured to implement the dependencies between nodes. Moreover, according to the assumptions made in Section 3, the computation elements have finite buffer sizes and the communication elements have no storage capability.

Consider a dependency like the one shown in Figure 4 which may be part of an MDDG. This part of the proof is unaffected by the fact that either A or B is a communication element, so we will ignore this fact. Let n_A and n_B respectively be the (integer) number of A and B nodes used in the architecture. Let the delays associated with them be D_A and D_B respectively. Since the nodes are utilized 100%, both types of nodes are periodic over the time period $D_A \cdot D_B$. During this time period, the number of tokens produced by A nodes is $n_A \cdot O_A \cdot D_B$, while the same consumed by B nodes is $n_B \cdot I_B \cdot D_A$. By the assumption of finite buffers, under steady state condition, the number of tokens consumed by B nodes must equal the number of tokens produced by A nodes. Thus we get $n_A \cdot O_A \cdot D_B = n_B \cdot I_B \cdot D_A$. Transposing D_A and D_B, we get the desired token conservation equation for the dependency between nodes A and B. This is true for every dependency of the MDDG. Thus the choice of n_i's used in the architecture satisfy the set of token conservation equations.

If:

There are two parts to building an architecture: 1) Constructing topologically consistent connections between various nodes so that their dependencies can be satisfied. 2) Ensuring that the computation does indeed proceed at 100% utilization.

We prove this in two parts: First we prove that a consistent set of connections can be made between function and communication blocks. Later, we will use this fact along with others to conclude that the computation can proceed with 100% utilization.

Consider again two nodes in an MDDG, A and B, connected by a dependency, as shown in Figure 4. Without loss of generality, assume that B is the communication element. We know that the following relationship is true:

$$n_A \cdot \frac{O_A}{D_A} = n_B \cdot \frac{I_B}{D_B} \quad .$$

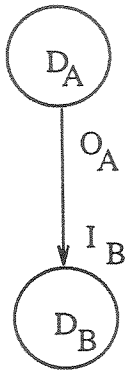

Figure 4: Data Dependency Between Nodes A and B

We define a "hypergraph" over the two types of nodes as follows. For each node of type A, we define $O_A \cdot D_B$ ports, giving us a total of $n_A \cdot O_A \cdot D_B$ ports on the A side. We similarly define $I_B \cdot D_A$ ports for each node of type B, resulting in $n_B \cdot I_B \cdot D_A$ ports on the B side. From the above relation, the total number of ports on either side are equal, and therefore can be matched one to one. This completes the first part.

By definition, node A is periodic with period D_A, and node B is periodic with period D_B. Thus both types of node are periodic with a common period $D_A \cdot D_B$. It can be verified that the expressions above, derived for total number of ports, also define the total number of tokens generated and absorbed by A nodes and B nodes respectively over the period $D_A \cdot D_B$. It can be interpreted that the links of the above hypergraph each carry exactly one token every period of $D_A \cdot D_B$. It remains to be shown that, given the consistent connectivity and exact definition of the source-destination pair for each token, it is possible to execute the computation at 100% utilization.

Assume, without any loss of generality that, the A nodes have no input dependencies; i.e., they are input nodes. Assume also that they are not blocked and can execute at 100% utilization. We now insist that every node of type B does not start firing until it finds at least a total of $I_B \cdot D_A$ tokens in (finite) buffers

of A nodes to which it connects[4]. Since it is periodic over $D_A \cdot D_B$, and the number of tokens it absorbs over that time period is $I_B \cdot D_A$, the same as the number of tokens delivered to it (by hypergraph construction), we conclude that a B node does not idle over the time period due to lack of tokens. Also, the number of tokens remaining in the buffers at the end of this period is same as at the beginning. We can transitively prove this for the rest of the nodes in the graph. □

Note 1: It is possible to reduce the total number of ports on both A and B side if we assign port connections in a "greedy" order. Since each edge of the hypergraph carries exactly one token every period, it is possible to skew temporally the edges of the hypergraph connecting the same two nodes. It is then possible to collapse these edges into one time-shared edge which carries the same number of tokens.

Note 2: Although we insisted that each node have a buffer size large enough to hold all tokens it needs during the period $D_A \cdot D_B$, not all nodes will require a buffer of that large a size; a much smaller buffer size will generally do.

Note 3: The buffer size, however, may be larger for nodes which have multiple input arcs to allow some parent nodes to continue firing while waiting for some of its other input dependencies to be satisfied. This is especially true during the start of the computation when the pipeline is being filled up.

Lemma 1: If **n** is a solution of the system of token-conservation equations for a DDG, and if k is a positive integer, then $k \cdot \mathbf{n}$ is also a solution.

Proof: Both sides of each equation get multiplied by k. □

The effect of multiplying the original solution by a positive integer is that of replicating the architecture the integer number of times. It is clear that the utilization of the resultant architecture is also 100%. Converse to the above lemma, it is obvious that the following is also true.

Lemma 2: If **n** ($\equiv \langle n_i \rangle$) is an integer solution of the system of token-conservation equations for a DDG, and if k is an integer such that every n_i can be expressed as $k \cdot n'_i$, where n'_i's are all integers, then $\langle n'_i \rangle$ are also a solution. □

Lemma 3: If there exists a solution for the set of token-conservation equations for a DDG, it is unique within a factor allowed by Lemmas 1 and 2.

Proof: Let $\langle n_i \rangle$ and $\langle n'_i \rangle$ be two integer solutions to the system of token-conservation equations. Consider a dependency similar to the one shown in Figure 4. It is clear that,

---

[4]If node A was a communication element, then we insist that it does not start firing until it has $I_B \cdot D_A$ tokens in its own input buffer.

$$n_A \cdot \frac{O_A}{D_A} = n_B \cdot \frac{I_B}{D_B} \quad ,$$

and also,

$$n'_A \cdot \frac{O_A}{D_A} = n'_B \cdot \frac{I_B}{D_B} \quad .$$

If we take ratios of the corresponding sides of the above two equations, we obtain

$$\frac{n_A}{n'_A} = \frac{n_B}{n'_B} \quad .$$

This can be shown to be true ∀ A and B pairs connected by a dependency, and therefore transitively ∀ i and j, where nodes i and j are any nodes in the DDG. Thus, ∀ i,

$$n'_i = n_i \cdot \frac{n'_A}{n_A} \quad .$$

The proof is completed by noting that n_A and n'_A are integers. □

4.3 Tree-Form DDGs

We will now analyze different classes of DDGs and specify conditions under which these graphs have integer solutions. We start with tree-form graphs.

Theorem 2: A DDG in the form of a rooted tree[5] has an integer solution.

Proof: Assume that the nodes in the DDG are labelled uniquely, 1 through m, with the root assigned the label "1". Since every node, except the root, has one input edge and since each node may have several output edges, except the leaf nodes which have no output edges, let us assign to each node a number I_i denoting the number of input tokens the node absorbs upon firing. Since the root has no input edges, it is not assigned any input tokens. Also, as before, if node i is connected by a dependency to node j, a number O_{ij} is assigned to the corresponding output port of node i. See Figure 5. Notice that for a tree-form DDG, if node j is dependent on node i, node i is j's only parent. We can now define a solution to the set of token-conservation equations.

We set

[5]The graph referred to here is also known in the literature as an *out-tree*.

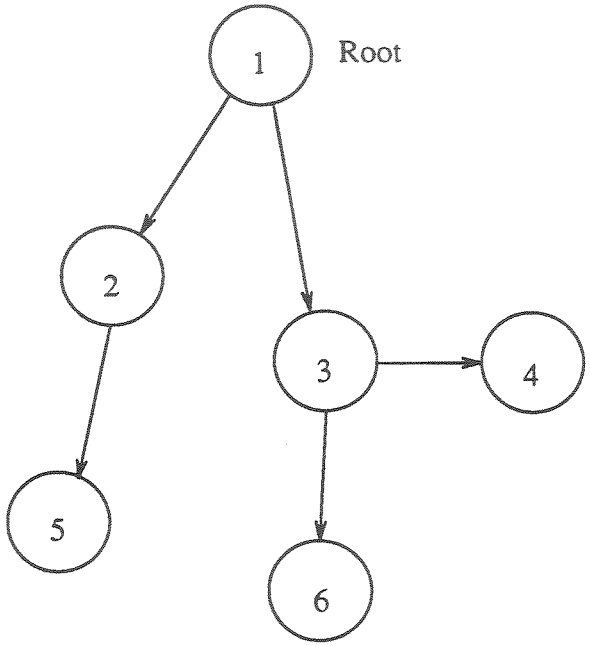

Figure 5: A Tree-Form DDG

$$n_1 = D_1 \cdot \prod_{i=2}^{m} I_i \ .$$

We can then set

$$n_i = \frac{D_i}{D_1} \cdot \frac{\prod_{root}^{parent\ of\ i} O_{jk}}{\prod_{child\ of\ root}^{i} I_l} \cdot n_1 \ ,$$

where the products are taken over nodes along the directed path from the root to node i.

To see that these assignments do satisfy the set of equations, we need to notice first that n_i is an integer $\forall$ i, since the terms in the product in the denominator are contained in the expression for n_1. Consider nodes p and q such that p is the parent of q. The path from the root to node q is unique and node p is in it. Let node o be the parent of p. Thus,

$$n_q = \frac{D_q}{D_1} \cdot \frac{\prod_{root}^{p} O_{jk}}{\prod_{child\ of\ root}^{q} I_l} \cdot n_1$$

$$= \frac{D_q}{D_1} \cdot \frac{\prod_{root}^{o} O_{jk}}{\prod_{child\ of\ root}^{p} I_l} \cdot \frac{O_{pq}}{I_q} \cdot n_1$$

$$= \frac{D_q}{I_q} \cdot n_p \cdot \frac{O_{pq}}{D_p}$$

□

Definition 1: An *inverted tree*[6] is a graph in which all nodes have at most one output arc. The leaf nodes have no input arcs and the root has no output arc. An inverted tree therefore looks like a tree shaped DDG, but the direction of its dependencies is rootward. See Figure 6.

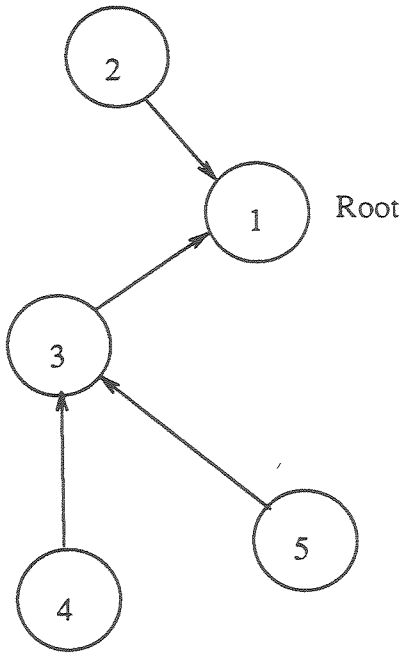

Figure 6: An Inverted Tree DDG

Theorem 3: A DDG with the form of an inverted tree has an integer solution.

[6]Also known in the literature as an *in-tree*.

Proof: The proof of this theorem is similar to the one for Theorem 2. We start by numbering the nodes in the DDG, root getting the label 1. Also assume that there are m nodes in the DDG. We let

$$n_1 = D_1 \cdot \prod_{j=1}^{m} O_j \ .$$

For any node i ($\neq 1$), we assign

$$n_i = n_1 \cdot \frac{D_i}{D_1} \cdot \frac{\prod_{child\ of\ i}^{root} I_j}{\prod_{i}^{parent\ of\ root} O_j} \ ,$$

where the products are taken along the directed path from node i to the root. It can be verified that these assignments satisfy the system of equations. It should also be noted that the terms in the denominator for the expression for n_i also appear in the numerator as part of n_1. Thus n_i's are integers. □

We can extend the above results to combination of multiple trees.

Theorem 4: A DDG formed by starting with a tree and incrementally appending a tree at a time to the existing DDG, such that, every appended tree overlaps the existing DDG in exactly one node, has an integer solution.

Proof: The DDG can be broken down into trees and inverted trees which share common nodes. These can be separately analyzed and solutions obtained. The solutions can be combined to obtain the final architecture in the same order the DDG was formed. At each step, we multiply individual solutions meeting at a node by appropriate integers so as to get a common number for the shared node. □

We will now illustrate the design methodology via an example of a Generalized Linear Pipeline.

4.4 Generalized Linear Pipeline

A Linear Dependency Graph (LDG) is a DDG in which the nodes form a linear array. The two end nodes are called input and output nodes and have an output edge and an input edge respectively. Each of the intermediate nodes has an input edge and an output edge.

If an LDG is analyzed in the manner described above, we obtain a Generalized Linear Pipeline architecture. It has the characteristics of a common multistage pipeline, but is general in that, each stage

has multiple identical computation elements and a delay which may be different from that of other stages. In fact, a common pipeline is a special case of the generalized pipeline under the condition of $I_i = O_i = D_i = 1, \forall\, i \in V \cup E$[7]. The following theorem shows that an architecture can always be designed for an LDG that achieves 100% utilization.

Corollary 1: The system of token-conservation equations for an LDG has an integer solution.

Proof: An LDG is a special case of a tree-form DDG, but we give a separate proof here. Assume that there are m nodes in an LDG, numbered 1 through m. We get the following set of equations:

$$n_1 \cdot \frac{O_1}{D_1} = n_2 \cdot \frac{I_2}{D_2}$$

$$n_2 \cdot \frac{O_2}{D_2} = n_3 \cdot \frac{I_3}{D_3}$$

$$n_{i-1} \cdot \frac{O_{i-1}}{D_{i-1}} = n_i \cdot \frac{I_i}{D_i}$$

$$n_i \cdot \frac{O_i}{D_i} = n_{i+1} \cdot \frac{I_{i+1}}{D_{i+1}}$$

$$n_{m-1} \cdot \frac{O_{m-1}}{D_{m-1}} = n_m \cdot \frac{I_m}{D_m}$$

These equations can be used to express n_i in terms of n_1 and constants.

$$n_i = D_i \cdot \frac{n_1}{D_1 \cdot \prod_{j=2}^{i} I_j} \cdot \prod_{j=1}^{i-1} O_j$$

If we choose

$$n_1 = D_1 \cdot \prod_{j=2}^{m} I_j \ ,$$

which is an integer value, we can obtain the following integer expression for n_i :

[7]Remember that communication and function blocks are analyzed similarly.

$$n_i = D_i \cdot \prod_{j=i+1}^{m} I_j \cdot \prod_{j=1}^{i-1} O_j$$

□

The foregoing proves that a solution exists for the system of equations describing an LDG. It does not however guarantee the smallest values for n_i's which satisfy the equations.

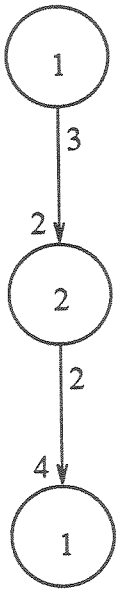

Figure 7: A Linear Dependency Graph

An Example

Figure 7 shows a computation with a Linear Dependency Graph. The numbers inside the nodes indicate the delays associated with their execution.

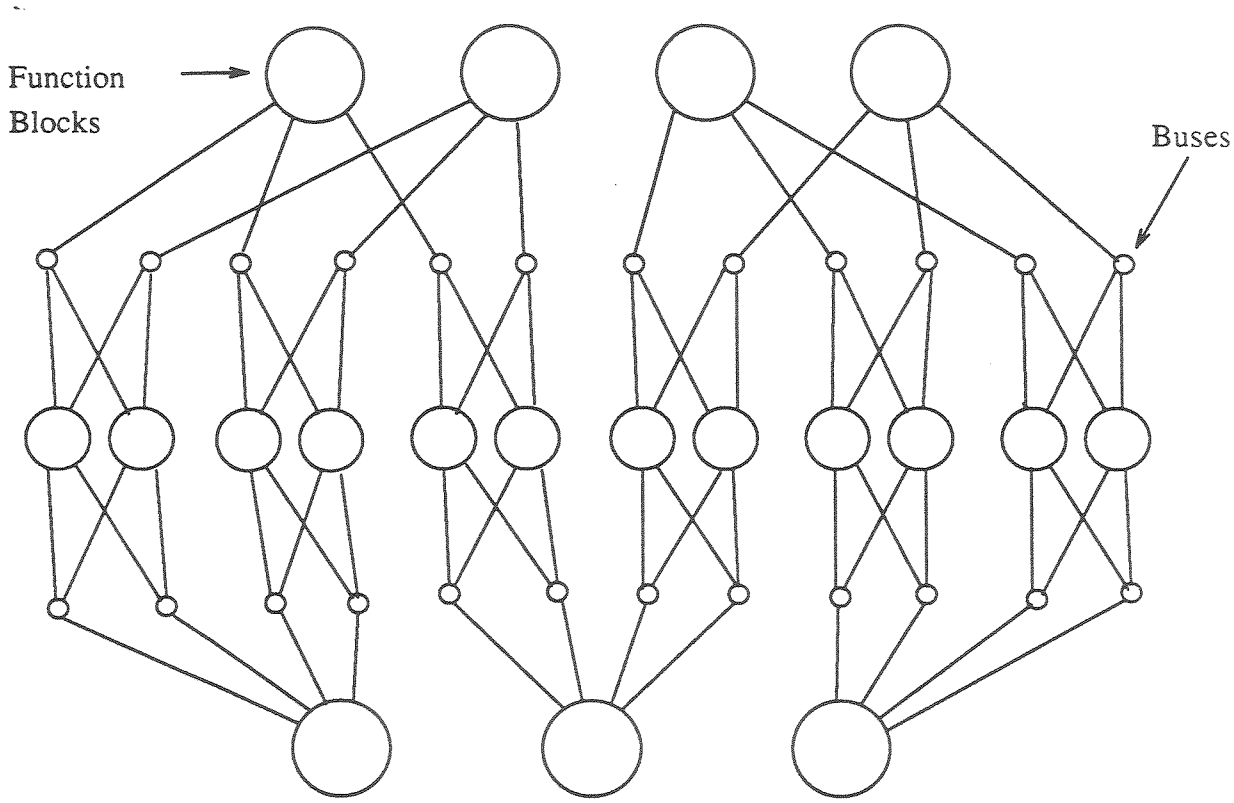

Figure 8: Solution Architecture For The LDG

The instances of bus elements are computed the same way as the function elements. For the bus elements, however, $I_b = O_b = D_b = 1$. Figure 8 shows the solution architecture.

The architecture in Figure 8 has both temporal and spatial parallelism. The architecture is minimally optimum for the given computation graph. That is, it is possible to simply duplicate the architecture and still produce an optimum architecture which provides double the throughput of the original architecture, but elimination of one or more architectural elements will result in less that 100%

utilization of one or more elements. It is important to note that for an arbitrary combinations of I_i's, O_i's and D_i's, it is possible that the resultant architecture is very complex and large. In practice, however, these combinations are not likely to be arbitrary, and one may expect a reasonable structure.

4.5 Non-Tree DDGs

Definition 2: A DDG is said to be a Consistent DDG (CDDG), if and only if,

1) For every pair of nodes p and q connected by multiple paths, there exists a rational constant C such that, for every path connecting the two nodes,

$$\frac{\prod_{p}^{parent\ of\ q} O_j}{\prod_{child\ of\ p}^{q} I_j} = C \ ,$$

where the products are taken along the path from p to q. See Figure 9.

If there are multiple input nodes in the DDG, then the following two conditions must also hold:

2) For every pair of *input* nodes p1 and p2, there exists a rational constant C' such that, for every node q that is a common descendant of p1 and p2,

$$\frac{\prod_{p1}^{parent\ of\ q} O_j}{\prod_{child\ of\ p1}^{q} I_j} = C' \cdot \frac{\prod_{p2}^{parent\ of\ q} O_j}{\prod_{child\ of\ p2}^{q} I_j} \ ,$$

where the products on the left hand side are taken along a path from p1 to q and the ones on the right hand side are taken along a path from p2 to q. Furthermore,

3) If there is another *input* node p3, such that it has common descendents with p1 and p2, and if C1, C2 and C3 are rational constants, as defined by Condition 2 above, between p1 and p2, between p1 and p3, and between p2 and p3, respectively, then C1 = C2 / C3.

If a DDG satisfies the first condition, then it means that, if p were to fire a certain number of times, it results in generating tokens on the input edges of q in numbers proportional to the numbers associated with the corresponding dependencies of q. This is necessary to ascertain that when q fires to absorb all

tokens on one of its input edges, it also finds the exact numbers of tokens on edges belonging to the other paths between the nodes p and q. In absence of such a condition, there will be an infinite build-up of tokens along certain paths, resulting in a meaningless computation. All real computations must satisfy this condition.

The proof of Corollary 1 outlines the method we may use to analyze a path within a DDG. Given the number n_p for a node p, we can uniquely define the the number n_q for a descendant q of p by analyzing a path from p to q.

It is possible that we may get different values for n_q by following different paths from p to q. However, if the graph is a CDDG, we can state the following:

Lemma 4: In a CDDG, given two nodes, p and q, connected by multiple paths, if integers n_p and n_q satisfy the set of token-conservation equations along a path connecting p and q, then they satisfy the set of token-conservation equations along all paths connecting the two nodes.

Proof: We first define what is meant by satisfying the set of token-conservation equations along a path. Assume that a path from node p to q is formed by nodes p,1,2, . . . ,i,q, as shown in Figure 9. If n_p is chosen to be the number of p nodes, then the following must hold true:

$$n_p \cdot \frac{O_p}{D_p} = n_1 \cdot \frac{I_1}{D_1}$$

$$n_1 \cdot \frac{O_1}{D_1} = n_2 \cdot \frac{I_2}{D_2}$$

:

$$n_i \cdot \frac{O_i}{D_i} = n_q \cdot \frac{I_q}{D_q}$$

We can back substitute to obtain the following relationship:

$$n_q = n_p \cdot \frac{D_q}{D_p} \cdot \frac{\prod_i^p O_j}{\prod_q^1 I_j} .$$

When n_p and n_q satisfy the above relationship, they are said to satisfy the set of token-conservation equations along the path p,1,2 . . . ,i,q.

Now consider Figure 10, which shows two paths, r and s, connecting nodes p and q. Let n_p and n_q satisfy the token-conservation equations along path r. Thus,

$$n_q = n_p \cdot \frac{D_q}{D_p} \cdot \frac{\prod_{i_r}^{p} O_j}{\prod_{q}^{1_r} I_j} ,$$

where the products are taken along r. However, by the definition of a CDDG,

$$\frac{\prod_{i_r}^{p} O_j}{\prod_{q}^{1_r} I_j} = \frac{\prod_{i_s}^{p} O_j}{\prod_{q}^{1_s} I_j} .$$

Thus,

$$n_q = n_p \cdot \frac{D_q}{D_p} \cdot \frac{\prod_{i_s}^{p} O_j}{\prod_{q}^{1_s} I_j} ,$$

where the products are now taken along s. Thus n_p and n_q satisfy token-conservation equations along s.

In the above, we made no assumption about whether or not r and s share one or more nodes. Thus the lemma holds for intersecting paths as well. □

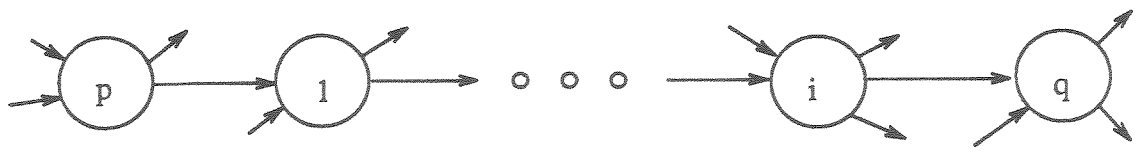

Figure 9: A Path From p to q

Notation: In what follows we will often encounter ratios like

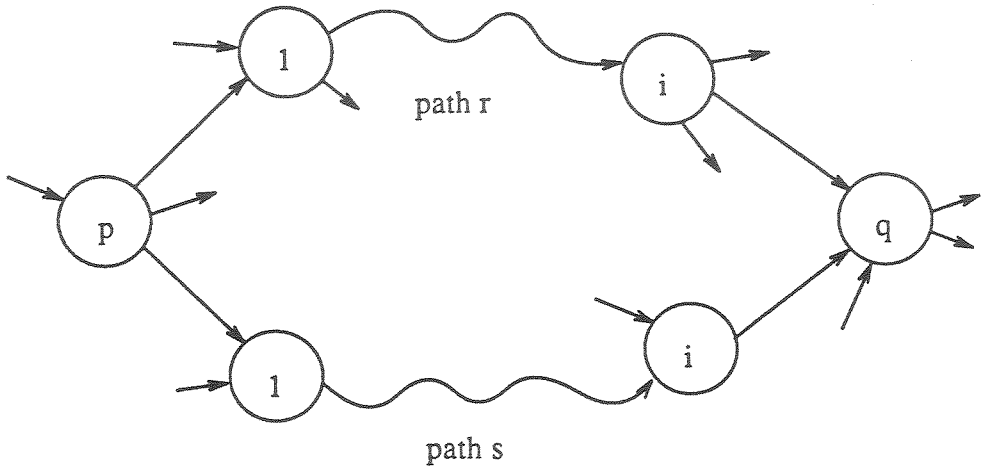

Figure 10: Multiple Paths Within a CDDG

$$\frac{\prod_{i_r}^{p} O_j}{\prod_{q}^{1_r} I_j},$$

used in Lemma 4 above. To make algebraic manipulations more readable, we will use the notation R_r to mean the same ratio.

Consider Figure 11. From Lemma 4 above, we know that an integer solution exists if the DDG shown in the figure is in fact a CDDG. Consider the subgraph resulting from removal of edge c from the CDDG, which is a tree rooted at node p. By Theorem 2 the tree subgraph has an integer solution. We now prove the following corollary.

Corollary 2: A solution for the tree subgraph of the CDDG formed by removal of edge c as shown in Figure 11, is also a solution of the CDDG.

Proof: Let n_p, n_q and n_r be the solution values for nodes p, q and r respectively corresponding to the tree subgraph. From the proof to Theorem 2,

$$n_q = n_p \cdot \frac{D_q}{D_p} \cdot R_a$$

and,

$$n_r = n_p \cdot \frac{D_r}{D_p} \cdot R_b$$

By condition 1 of a CDDG, we obtain the following:

$$n_q = n_p \cdot \frac{D_q}{D_p} \cdot R_{b \cup c}$$

That is,

$$n_q = n_p \cdot \frac{D_q}{D_p} \cdot R_b \cdot \frac{O_c}{I_c}$$

$$= n_r \cdot \frac{D_q}{D_r} \cdot \frac{O_c}{I_c} \quad .$$

Thus, a solution for the tree subgraph also satisfies the token-conservation equation for the dependency associated with edge c. It is therefore a solution of the CDDG. □

The second condition is a generalization of the first to handle the case of multiple input nodes. To get an intuitive idea behind the condition consider Figure 12. Nodes p1 and p2 have common descendents q1 and q2 connected by paths shown in the figure. Assume that node p1 fires a certain number of times so as to generate tokens on the corresponding input edges of q1 and q2. If p2 were to fire an appropriate number of times so as to allow q1 to fire and absorb all tokens on its input arcs, it is impossible for q2 to fire and absorb all tokens on its input arcs unless condition 2 was met. In fact we will prove below that condition 2 is a necessary and sufficient condition for a DDG of the form of Figure 12 to have an integer solution.

Lemma 5: A DDG of the form of Figure 12 has an integer solution, if and only if, condition 2 in Definition 2 is satisfied.

Proof: In order to prove this lemma, we label paths from p1 and p2 to nodes q1 and q2 as shown in the

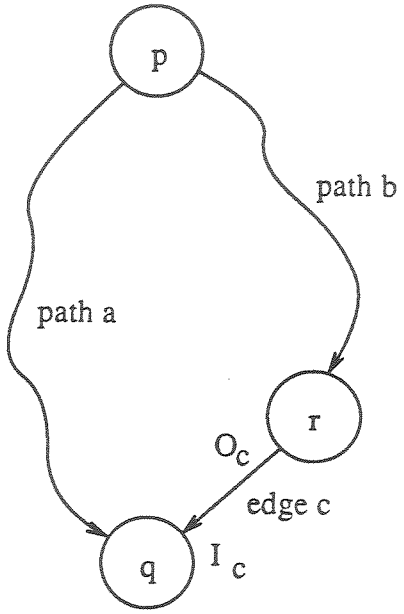

Figure 11: A CDDG And Its Tree-Subgraph

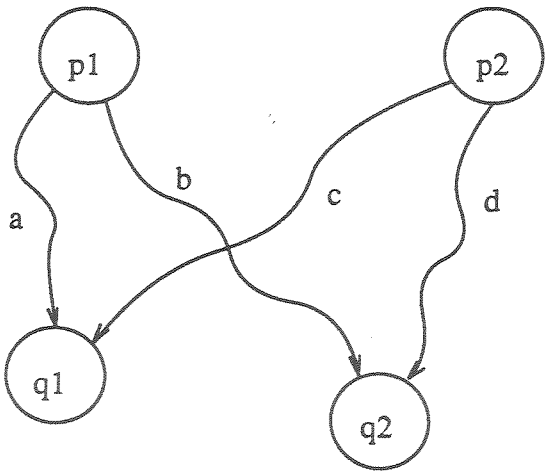

Figure 12: Illustration Of Condition 2

figure, and we use techniques used in proving Lemma 4.

If:

There are two tree-form DDGs with their roots at p1 and p2. First consider the tree rooted at p1. By Theorem 2, we can find an integer solution for it. It is also possible to find an integer solution for the tree rooted at p2. These trees overlap at q1 and q2. Let $n_{q1}(p1)$ and $n_{q1}(p2)$, and $n_{q2}(p1)$ and $n_{q2}(p2)$, be the numbers of q1 and q2 nodes for trees rooted at p1 and p2 in their respective integer solutions.

We can find two integers k_1 and k_2 such that, $k_1 \cdot n_{q1}(p1) = k_2 \cdot n_{q1}(p2)$. Thus we multiply the solution for tree rooted at p1 by k_1 and the tree rooted at p2 by k_2. We must now show that $k_1 \cdot n_{q2}(p1) = k_2 \cdot n_{q2}(p2)$.

From Theorem 2 we know that,

$$k_1 \cdot n_{q1}(p1) = k_1 \cdot n_{p1} \cdot \frac{D_{q1}}{D_{p1}} \cdot R_a \ ,$$

$$k_1 \cdot n_{q2}(p1) = k_1 \cdot n_{p1} \cdot \frac{D_{q2}}{D_{p1}} \cdot R_b \ ,$$

$$k_2 \cdot n_{q1}(p2) = k_2 \cdot n_{p2} \cdot \frac{D_{q1}}{D_{p2}} \cdot R_c \ ,$$

and,

$$k_2 \cdot n_{q2}(p2) = k_2 \cdot n_{p2} \cdot \frac{D_{q2}}{D_{p2}} \cdot R_d \ .$$

The left hand sides of the first and the third equations are equal by construction, and thus we have,

$$k_1 \cdot n_{p1} \cdot \frac{D_{q1}}{D_{p1}} \cdot R_a = k_2 \cdot n_{p2} \cdot \frac{D_{q1}}{D_{p2}} \cdot R_c \ .$$

But by assumption,

$$\frac{R_a}{R_c} = Rational\ Constant = \frac{R_b}{R_d} .$$

Therefore,

$$k_1 \cdot n_{p1} \cdot \frac{D_{q2}}{D_{p1}} \cdot R_b = k_2 \cdot n_{p2} \cdot \frac{D_{q2}}{D_{p2}} \cdot R_d .$$

It thus follows that, $k_1 \cdot n_{q2}(p1) = k_2 \cdot n_{q2}(p2)$.

Only if:

Assume that an integer solution for the DDG shown in Figure 12 exists. Thus,

$$n_{q1} = n_{p1} \cdot \frac{D_{q1}}{D_{p1}} \cdot R_a = n_{p2} \cdot \frac{D_{q1}}{D_{p2}} \cdot R_c .$$

Similarly,

$$n_{q2} = n_{p1} \cdot \frac{D_{q2}}{D_{p1}} \cdot R_b = n_{p2} \cdot \frac{D_{q2}}{D_{p2}} \cdot R_d .$$

From the above, it can be seen that,

$$\frac{R_a}{R_c} = \frac{D_{p1}}{D_{p2}} \cdot \frac{n_{p2}}{n_{p1}} = \frac{R_b}{R_d} .$$

Using the property of uniqueness of a solution according to Lemma 3, we conclude that

$\frac{n_{p2}}{n_{p1}}$

is a rational constant, and hence this lemma. □

Notice that we have made no assumptions about the nodes on paths a, b, c and d in the above lemma, so the paths may share nodes and edges. Consider Figure 13 in which we have added an input node p3 which has common descendents, of which q3 and q4 are two, with each of p1 and p2 such that, condition 2 is satisfied in the combined DDG. We will now prove the following lemma.

Lemma 6: If CDDG A in Figure 13 overlaps with another CDDG B such that condition 3 is satisfied (at the nodes at which the two CDDGs overlap,) for the combined DDG, then the resultant DDG (which is also a CDDG) has an integer solution.

Proof: To prove this lemma it is sufficient to show that a common set of integers values can be found for nodes in CDDG A and CDDG B such that the two CDDGs have the same numbers of corresponding nodes at which they overlap.

Consider two nodes q1 and q2 at which CDDG B overlaps CDDG A. Let p1 and p2 be two input nodes in CDDG A such that, q1 is a descendant of both p1 and p3, and q2 is a descendant of both p2 and p3. Let q3 be a common descendant of p1 and p2 within CDDG A.

According to the definition,

$$C1 = \frac{R_a}{R_b} \; ; \; C2 = \frac{R_c}{R_d} \; ; \; and \; C3 = \frac{R_e}{R_f} \; .$$

Let $n_{q1}(A)$ and $n_{q2}(A)$ be the solution values for q1 and q2 within CDDG A, and let $n_{q1}(B)$ and $n_{q2}(B)$ be the corresponding solution values within CDDG B. For the combined CDDG to have an integer solution, making $n_{q1}(A) = n_{q1}(B)$ must imply $n_{q2}(A) = n_{q2}(B)$. To begin with, we state the following relations using the same techniques as used for Lemma 5.

$$n_{q1}(A) = n_{p1} \cdot \frac{D_{q1}}{D_{p1}} \cdot R_c \; , and \quad n_{q1}(B) = n_{p3} \cdot \frac{D_{q1}}{D_{p3}} \cdot R_d \; ;$$

and,

$$n_{q2}(A) = n_{p2} \cdot \frac{D_{q2}}{D_{p2}} \cdot R_e \; , and \quad n_{q2}(B) = n_{p3} \cdot \frac{D_{q2}}{D_{p3}} \cdot R_f \; .$$

Letting $n_{q1}(A) = n_{q1}(B)$, and using the relation C2 = C1·C3, we obtain:

$$n_{p3} = n_{p1} \cdot \frac{D_{p3}}{D_{p1}} \cdot \frac{R_c}{R_d} = n_{p1} \cdot \frac{D_{p3}}{D_{p1}} \cdot \frac{R_a}{R_b} \cdot \frac{R_e}{R_f} = n_{p2} \cdot \frac{D_{p3}}{D_{p2}} \cdot \frac{R_e}{R_f} = n_{q2}(A) \cdot \frac{D_{p3}}{D_{q2}} \cdot \frac{1}{R_f} \; .$$

Thus,

$$n_{q2}(B) = n_{p3} \cdot \frac{D_{q2}}{D_{p3}} \cdot R_f = n_{q2}(A) \quad .$$

This result holds for every pair of q1 and q2, and thus for all nodes at which CDDG A and CDDG B overlap. The result also holds if q1 and q2 are identical. □

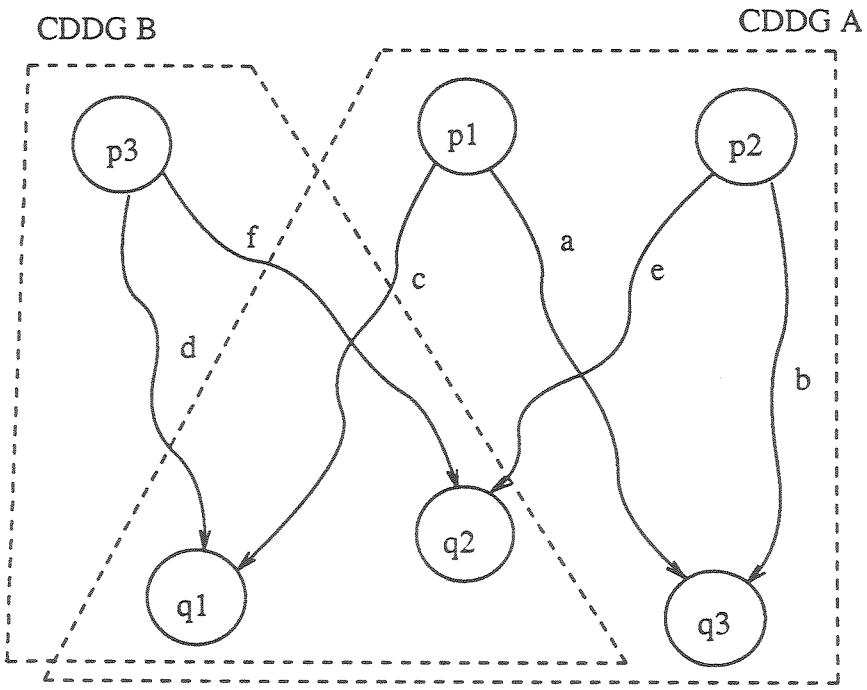

Figure 13: Illustration Of Condition 3 For A CDDG.

We are now in a position to analyze a CDDG.

Theorem 5: A CDDG with a single input node has an integer solution.

Proof: For a CDDG with a single input node we can find a single tree subgraph with the input node as root. By Theorem 2, we can find an integer solution for this tree subgraph. And by Corollary 2, this solution will satisfy the token-conservation equations of the edges of the CDDG not included in the tree-subgraph, and is therefore also a solution of the CDDG. □

Theorem 6: A CDDG (with multiple input nodes) has an integer solution.

Proof: We can partition the CDDG in the following manner:

Choose an input node and assign to it all its descendents to create a single input sub-CDDG. Mark the input node and its descendant nodes from the original CDDG. Call this sub-CDDG, sub-$CDDG_1$.

Choose next input node and assign it all its unmarked descendents. Also assign to it all marked *children* of unmarked descendents. Call this component sub-$CDDG_2$. The marked children assigned to this sub-CDDG are the nodes at which sub-$CDDG_2$ overlaps sub-$CDDG_1$. Mark the members of this sub-CDDG as well.

Repeat the last step with every input node.

With the final input node, all nodes of the original CDDG will be exhausted. The above process creates a set of single input sub-CDDGs which overlap to form the original CDDG.

From Theorem 5 above, each of these individual sub-CDDGs has an integer solution. Consider sub-$CDDG_1$ and sub-$CDDG_2$. By Lemma 5, we conclude that it is possible to find a single pair of integer multipliers for these two overlapping CDDGs such that the numbers of all nodes at which the two CDDGs overlap, match. We can thus obtain an integer solution for the union of the two sub-CDDGs and create a larger sub-CDDG with an integer solution. This process of accretion can be repeated with every single input sub-CDDG obtained above until the original CDDG is obtained. Lemma 6 guarantees an integer solution at each step. Therefore, an integer solution exists for the original CDDG. □

5 Cyclic Dependency Graphs

We introduced Data Dependency Graphs as being acyclic. On the other hand, there are many instances of computation graphs for repetitive computations where there are cycles within the graphs, such that, the results of certain sub-computations are fed back into the computation stream. If there are no additional semantic requirements put on the sub-computation executions, we can obtain integer solutions for cyclical graphs as well, if certain conditions are met for the loops. Here we give a necessary and sufficient condition for graphs with cycles (or loops) to have an integer solution. As before, integer solutions imply an architecture with 100% utilization.

Definition 3: A Cyclic Dependency Graph (CDG) is a Consistent CDG (CCDG), if and only if,

1) For every cycle l in the CDG, $R_l = 1$, and

2) The CDG satisfies the conditions listed in Definition 2.

Consider the dependency loop shown in Figure 14, which may be part of a CCDG. It consists of a path a from node p to node q and an edge b from q to p. If we remove edge b from the loop, only path a is left and the sub-graph becomes acyclic. We can demonstrate the following:

Lemma 7: If a set of integers satisfies the set of token-conservation equations for path a, they also satisfy the token-conservation equation for edge b.

Proof: Let $n_p, \ldots, n_q$, be the set of integers satisfying the token-conservation equations along path a. As in Lemma 5,

$$n_q = n_p \cdot \frac{D_q}{D_p} \cdot R_a \quad .$$

By condition 1 in Definition 3 above,

$$R_l = R_a \cdot \frac{O_b}{I_b} = 1 \quad .$$

Thus,

$$n_q = n_p \cdot \frac{D_q}{D_p} \cdot \frac{I_b}{O_b} \quad .$$

□

We can now prove the following theorem.

Theorem 7: A CCDG has an integer solution.

Proof: For the CCDG we identify a minimal set of edges E_m which when eliminated creates an acyclic subgraph G'. First we analyze this sub-graph and ignore the set E_m. G' is a CDDG since it is acyclic and satisfies the conditions specified in Definition 2. Therefore by Theorem 6 it has an integer solution. Now

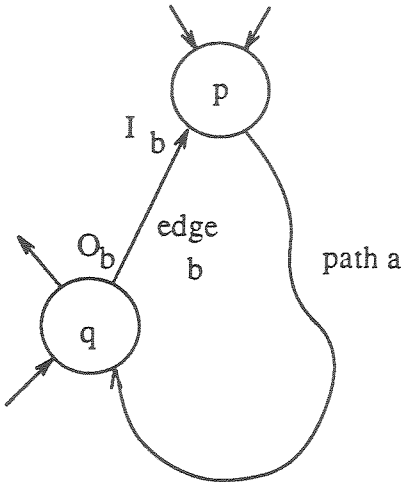

Figure 14: A Cycle Within A CCDG

we reconstruct the original CCDG by adding one edge from the set E_m at a time. By Lemma 7 the solution for G' also satisfies the token-conservation equation for the added edge. This holds for every edge in E_m. Thus the solution for G' satisfies the set of token-conservation equations for the original CCDG, and hence the theorem. □

Comments: In practice many cyclic dependency graphs have other semantics associated with them, such as, the value being fed back to a node bears a fixed temporal relationship to the current firing of the node. For example, in Signal Flow Graphs the value being fed back may be an output of the algorithm delayed by exactly m clock periods. Such considerations were not dealt with in Lemma 7 or Theorem 7, and therefore these results are not directly applicable to such situations.

References

1 The Architecture of Pipelined Computers; P. M. Kogge, Hemisphere Publishing Corporation, McGraw-Hill Book Company, 1981.

2 "Warp Architecture and Implementation - Preliminary Version"; M. Annaratone et al, Dept. of Computer Science, Carnegie-Mellon University, 1985.