HIGH-LEVEL SYNTHESIS OF PIPELINED ARCHITECTURES FOR MULTI-RATE ALGORITHMS[*]

Sanjay R. Deshpande[†]

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

TR-88-40 November 1988

Abstract

This technical report contains an analytical framework, based on which a high-level methodology for synthesis of pipelined architectures for multi-rate algorithms may be structured. Multi-rate functions are formally defined and a need for a non-systolic design approach for efficiently implementing algorithms containing them is pointed out. The problem of designing efficient architectures for multi-rate algorithms is defined and a solution approach is proposed. The concept of Token Conservation Equations (TCEs), which model the average behavior of an architecture, is introduced, and it is shown how the necessary number of hardware elements can be predicted by analyzing these TCEs. The issues of sharing the physical components and buffer sizes are also discussed briefly.

[*] This research was supported in parts by DARPA under grant N00039-86-C-0167 and by ONR-URI under grant N00014-86-K-0763.

[†] Dept. of Elect. & Comp. Engineering; The University of Texas at Austin, Austin, TX 78712. (512) 471-4520.

1 Multi-Rate Functions

In this section we formally define multi-rate functions and argue for a need to analyze them differently than the single-rate functions. We point out that the presence of multi-rate functions in an algorithm will demand a non-systolic approach to its efficient implementation in hardware.

1.1 Definition

Definition 1 *A relation* R *over sets* A *and* B *(from* A *to* B*, denoted as* $A \longrightarrow B$*) is a set of ordered pairs (a, b), where* a *and* b *are sets and* $a \in A$ *and* $b \in B$.

Definition 2 *A function* F *is a relation which satisfies the property:* $\forall a, \forall b_1, \forall b_2[(a, b_1) \in \mathrm{F} \wedge (a, b_2) \in \mathrm{F} \Longrightarrow b_1 = b_2]$.

Definition 3 *A function* F *from* $A \longrightarrow B$ *for which* $\forall(a,b) \in \mathrm{F}$, $|a| = m, |b| = n$. *is denoted as* $\mathrm{F}^{m,n}$.

Most image and signal processing functions can be defined as $\mathrm{F}^{m,n}$s. We will concentrate on a specific subset of these functions which satisfy further criteria.

Definition 4 *A function* $\mathrm{F}^{m,n}$ *is* p-input-unique *if* $\forall a_i, a_j \in Domain(\mathrm{F}^{m,n})$, $|a_i - \bigcup_{j \neq i} a_j| = p,\ p \geq 1$.

Definition 5 *A function* $\mathrm{F}^{m,n}$ *is* q-output-unique *if* $\forall b_i, b_j \in Range(\mathrm{F}^{m,n})$, $|b_i - \bigcup_{j \neq i} b_j| = q,\ q \geq 1$.

A p-input-unique, q-output-unique $\mathrm{F}^{m,n}$ is denoted by $\mathrm{F}^{m,n}_{p,q}$. We will consider functions which have m=p and n=q. For brevity these functions are denoted as simply $\mathrm{F}_{p,q}$. These functions effectively operate on unique sets of input data such that the individual data values are not shared between the domain sets a_i. Similarly, the output images b_is do not share data elements.

The examples of such signal processing functions are decimation and interpolation operations, series to parallel and parallel to series conversions and some operations involved in the beam-forming process.

Functions, according to their mathematical definition, are memory-less, and thus correspond to purely combinational circuits. In a combinational implementation of a function, an image in the range of the function is obtained by inputing all *antecedent* data values at once. In case of a function for which consecutive antecedents a_is are not disjoint, the shared data elements have to be input repeatedly. The number of inputs can be reduced significantly by allowing the function implementation to have *finite* storage. This permits the inputs to be restricted to only the unique data elements. However, this process changes the nature of the implementation from purely combinational to a sequential circuit.

Consider such sequential implementation of an $F_{p,q}$. Here we assume that the input arrives sequentially, at most one element per clock cycle. The same is assumed to be true for the output. Note that we allow certain cycles during which no elements may be input or output and during which the function is computing the image set bi for a previously input set a_i.

Definition 6 *We call a function implementation* single clock rate efficient *if it can accept a new input element and produce a new output element during every clock cycle. We will allow a finite latency before the implementation may begin to satisfy this criterion. However, once it starts to be clock rate efficient, it must remain so indefinitely.*

Consider a sequential finite-storage implementation of an $F_{p,q}$ which is single clock rate efficient. Let us first assume that p<q. It takes p cycles to input an a_i and it takes q cycles to output the corresponding image b_i. Since $\forall a_i a_j, i \neq j, a_i \cap a_j = \emptyset$, a b_i can be computed as soon as the corresponding a_i is input, and may be output after a finite latency; the a_i can be discarded. So we may analyze the storage considerations for only the b_is. Since a sequential circuit is causal, we can assume that $\forall a_i, a_j$, if a_i is input before a_j, b_i is output before b_j. Consider the p·q cycles starting with the cycle during which the first element of an a_i is input. During these cycles q a_is are input, and this results in q b_is. Now consider the p·q cycles starting with the cycle during which the first element of the first b_i so produced is output. Only p b_i elements are output during this interval. The remaining q-p b_is have to be stored and output during subsequent cycles, increasing the delay after which the next sequence of q b_is are output. Thus there is a net build-up of delayed b_i elements requiring unbounded storage and latency contradicting the finite storage and finite latency assumption.

If p>q, then similar analysis requires a diminishing latency for every subsequent b_i. Since latency is bounded below by zero (due to causality property of a sequential circuit), the implementation is forced to introduce cycles during which no output takes place.

The above arguments may be expressed as,

Lemma 1 *There does not exist a single clock rate efficient sequential implementation of an* $F_{p,q}$ *when* $p \neq q$. □

We call functions of the type $F_{p,q}$, $p \neq q$ as being multi-rate.

The above lemma is important because it implies that for systems involving $F_{p,q}$ $p \neq q$ functions, it is not possible to obtain an implementation which has a single clock and which at the same time is efficient in the sense of definition 6.

Since systolic implementations assume a single global clock with which all registers are clocked, these too cannot efficiently implement the $F_{p,q}$, $p \neq q$, functions. We can therefore conclude that the following is true.

Corollary 1 *Systolic architectures cannot implement* $F_{p,q}$, $p \neq q$, *functions such that the implementation is clock rate efficient.* □

We can further conclude that,

Corollary 2 *Systolic implementation of functions which have* $F_{p,q}$, $p \neq q$, *as subfunctions cannot be clock rate efficient.* □

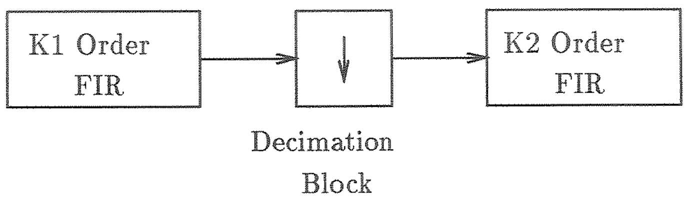


Figure 1: A two stage FIR

The last two corollaries suggest that a non-systolic approach, or an approach with multiple clocks, should be used to implement multi-rate functions or functions that incorporate multi-rate subfunctions. This is illustrated with an example in the following subsection.

1.2 Illustration Of The Need For Multiple Clocks

Consider a two stage filter shown in Figure 1. The filter is composed of two FIR filter blocks of orders K1 and K2 with a decimation block in the middle. We assume, for the purpose of illustration, that the decimation factor is 2 and that K2 is an even number.

An FIR filter of order K is described by the following recurrence relation:

$$\begin{aligned} y_n &= a_0x_n + a_1x_{n-1} + a_2x_{n-2} + \cdots + a_{K-1}x_{n-K+1} && (1) \\ &= \sum_{i=0}^{K-1} a_i x_{n-i} \, . && (2) \end{aligned}$$

One possible systolic implementation of an FIR filter presented in [An 85] uses a cell structure shown in Figure 2. Using this cell structure the systolic implementation uses a chain of K cells to implement a K-order filter. Thus using a single clock, we will need K1+K2 cells to implement the two stage filter.

Now if we take into account the existence of the rate change operation between the two FIR blocks, we can obtain a filter that is lower in cost.

We will use three clocks C1, C2 and C3 such that C1 has twice the frequency as C2 and C3, and C3 has the phase opposite to C2. The K1 order FIR is implemented as before using the C1 clock. We can implement the K2 order FIR as shown in Figure 3. Important difference in this design is that there are only half as many systolic cells as in the single clock design, but they continue to be clocked by C1. However, during alternate cycles, the cells compute different halves of two different sums which are eventually combined to produce the correct output.

To compute the cost differential between the two approaches we assume the following:

A Cost of an adder.

M Cost of a multiplier.

R Cost of a register.

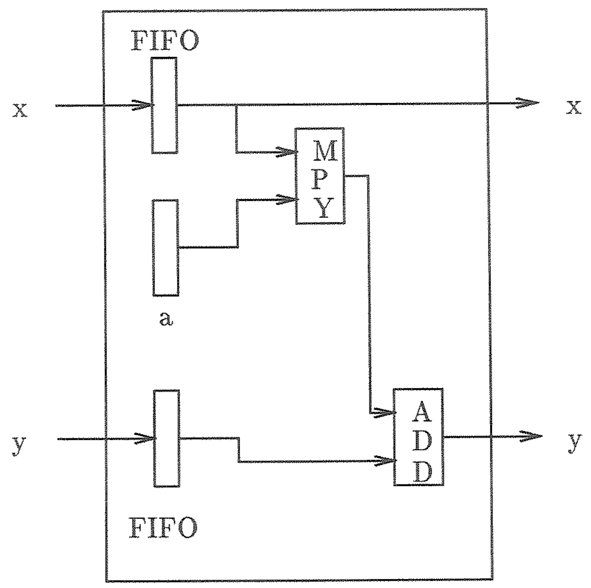


Figure 2: A systolic cell for FIR computation

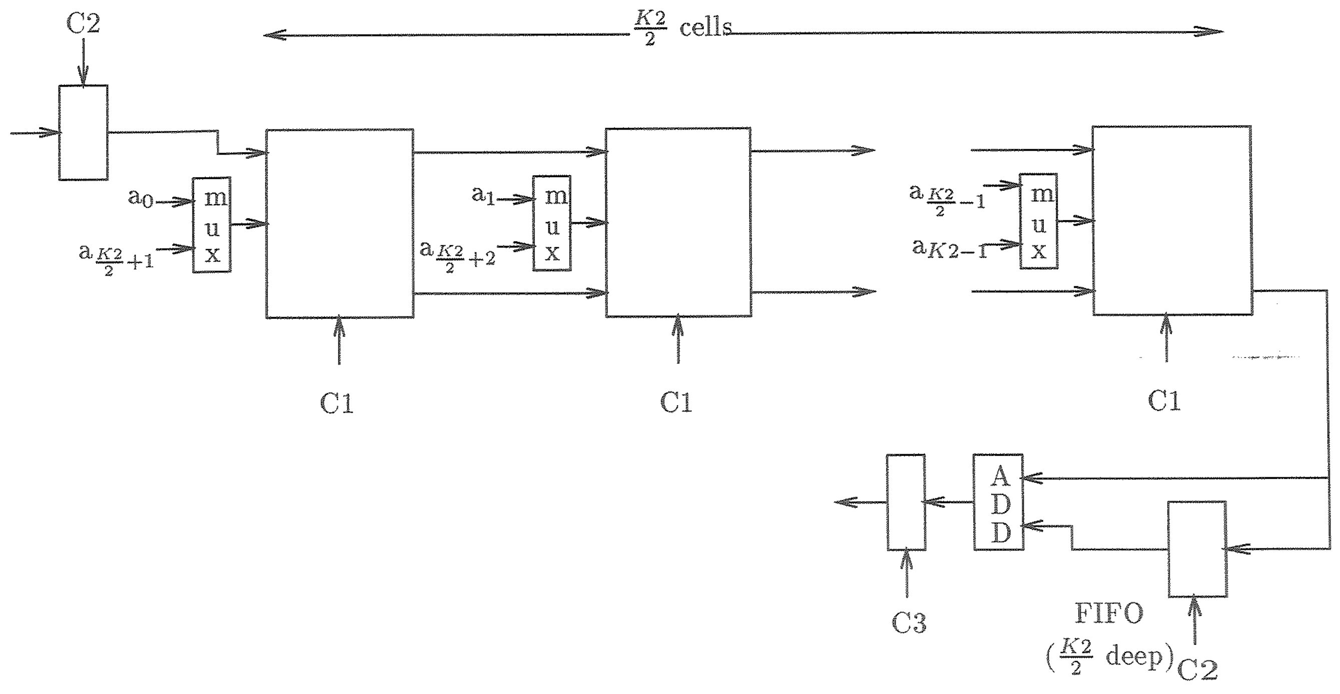


Figure 3: A multi-clock implementation of an FIR

m Cost of a multiplexer.

c Number of registers per systolic cell.

The cost differential between the two designs is given by,

$$\begin{aligned}\Delta Cost &= \frac{K2}{2}(A+M+cR) - \frac{K2}{2}(m+R) - A - 2R \quad (3)\\ &= \frac{K2}{2}(A+M+(c-1)R-m) - A - 2R \quad (4)\end{aligned}$$

The $\Delta Cost$ is positive if A + M + (c–1)R > m + $\frac{2}{K2} \cdot (A+2R)$, that is, if (1 - $\frac{2}{K2}$)A + M + $(c-1-\frac{4}{K2})$R > m, which is always true in practice.

2 Modelling Multi-rate Hardware Elements

As seen in the previous section, multi-rate computational elements behave differently than single-rate elements, which have to be taken into account while obtaining hardware efficient architectures. To accommodate multi-rate elements we first look at the modelling aspect of these elements. In this section we show how modelling multi-rate components as multi-delay elements provides a uniform basis for handling both types of components. We start by proposing a behavioral model framework for elements of the solution architecture.

2.1 Architectural Characteristics

The architectural elements will be of three types: computation, storage and communication. Examples of node types are shown in Figure 4. The computation elements may be either combinational or sequential, but their behavior is always described within the framework of a synchronous system, i.e., in terms of a system wide clock.

2.1.1 Clock

The architecture generated will be synchronous with a single, fine granularity clock. All events in the execution of an algorithm will be assumed to happen at ticks of such a clock.

2.1.2 Computation Nodes

With each computation element is associated an atomic unit of computation of arbitrary complexity. Its physical realization in hardware executes the computation by *firing*. Typically, when a node fires, it absorbs input data, after a delay produces output data, and is ready to fire again. A node may absorb or produce a different number of data units for each of its input or output dependencies, and this is represented by assigning the corresponding integer numbers to its input and output edges, as in Figure 4. The node can fire only after there are corresponding number of tokens in the input buffer for each input dependency.

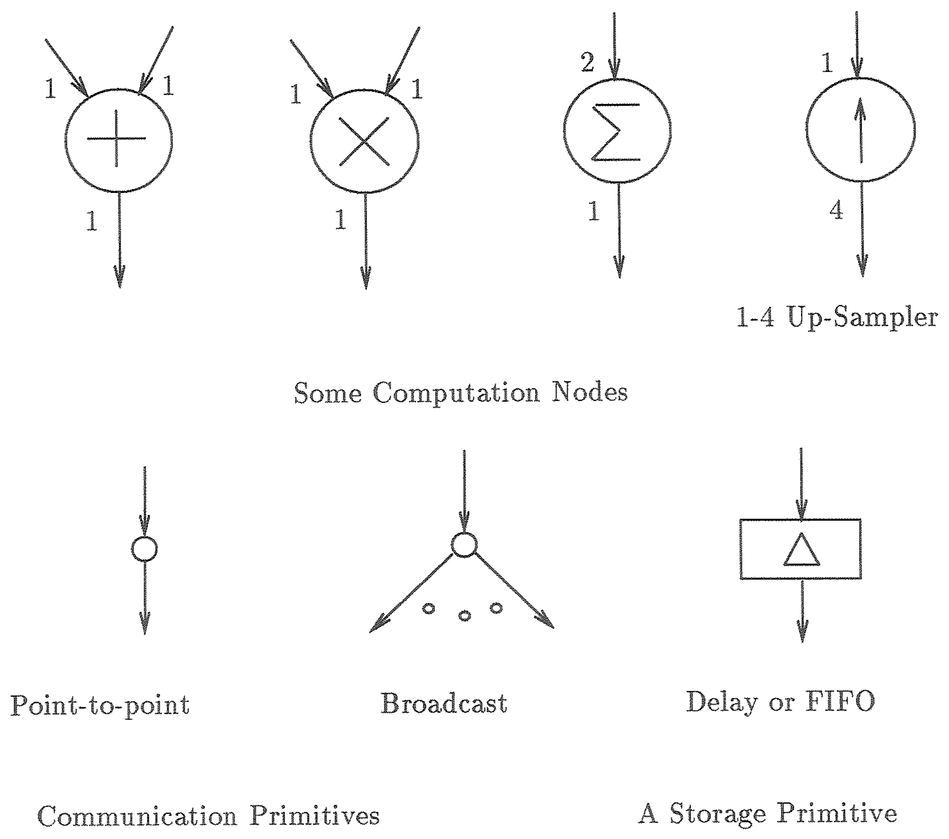
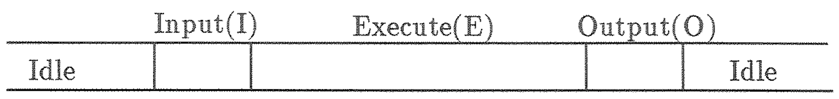
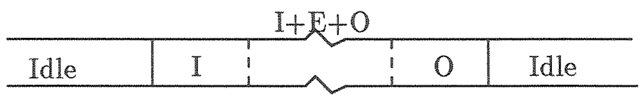


Figure 4: Examples Of Node Types

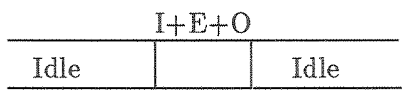


A Computation Node With Internal Sequencing



A Multi-delay Combinational

Computation Node



A Communication Node

Figure 5: Execution Models Of Nodes

The internal execution models of some node types are shown in Figure 5. All nodes have up to three phases: input, execution and output, each of which is an integer number of cycles long. During the input phase, a node inputs one token per cycle, and during the output phase, the node outputs, one token per cycle.

For internally sequential computation elements, it is possible to clearly distinguish between different phases. For combinational computation nodes like an adder, however, it is sufficient to specify the total delay in terms of number of clock cycles. It is assumed, nevertheless, that the node inputs data during the first cycle of its execution and outputs data during the last cycle.

2.1.3 Communication Nodes

The delay of a communication node, either point-to-point or broadcast, is one clock cycle. It operates by transferring data from a source to one or more destinations. In terms of delay analysis, however, it behaves like any single-delay combinational element.

2.1.4 Storage Nodes

The only types of storage node we will concern ourselves with in this paper are FIFOs and buffers. FIFOs are used to realize delay elements specified in an algorithm, while buffers are inserted in the architecture by the methodology. Throughout the paper it will be assumed that FIFOs with buffers are implemented as circular buffers.

In general, buffers may be inserted before and/or after every computation and FIFO element.

In presence of multi-rate operations it can be shown that buffers are necessary at both input and output of each of the computation and FIFO elements. The buffer requirement is lower for single-rate algorithms; only single buffers at the input or output of computation and FIFO elements are required as shown later. Buffers are also inserted to equalize delays along two paths between a pair of nodes.

The models assumed here are sufficient for analyzing multi-rate algorithms. In a multi-rate algorithm, there are nodes for which input and output tokens are not equal in number, as is the case in the $\sum$ node in Figure 4. In reference to the node models shown in Figure 5, it means that number of input cycles are different than the number of output cycles.

3 An Architecture Design Approach For Multi-rate Algorithms

The preceeding sections demonstrated a need for an architecture design approach for algorithms which incorporate multi-rate functions. In this section we propose such an approach, which will not result in solutions similar to the one obtained in Section 1.2 for the two stage filter, but which will, however, handle multi-rate and multi-delay components. Sub-sections 3.1 of this section defines the representation basis for algorithms. Sub-section 3.2 gives a precise definition of the properties of the designs which result from application of the synthesis procedure.

3.1 Specification Of Algorithms

Computations: Algorithms are expressed using a type of directed graphs called Data Dependency Graphs (DDGs).

Definition 7 *A DDG is a directed graph over three types of node: computation, storage and communication. Computation nodes represent arithmetic, logic, serialization/parallelization, and decimation/interpolation operations, storage nodes represent sample delays, and communication nodes represent the desired communication operations between nodes of the first two types. The directed edges represent* data *dependencies between pairs of nodes. Nodes which have no input edges are called* input nodes, *and those with no output edges are called* output nodes.

Input Latency: Another important part of the specification of a signal processing algorithm is the *sampling rate.* The sampling rate of the algorithm is the rate at which new input arrives, which determines the rate at which new computations are initiated. The inverse of the sampling rate is called the *input latency.*

3.2 Problem Definition and Approach

Definition 8 *Let u_{ij}, $j = 1 \ldots n_i$, be the utilization of an element e_{ij} of type i, where $u_{ij} = \frac{D_i}{T_{ij}}$, where D_i is the delay of the component of type i, and T_{ij} is the sum of delay and the average idle time (see Figure 5), and n_i is the number of elements of type i occurring in the architecture. The average utilization of elements of type i is computed as $\frac{\sum_{j=1}^{n_i} u_{ij}}{n_i}$.*

An architecture is utilization-efficient with respect to an element of type i, if it maximizes its average utilization. The average utilization is maximized when $n_i = \lceil \sum_{j=1}^{n_i} u_{ij} \rceil$

Formally, the problem of designing utilization-efficient architectures can be stated as:

- *Given a specification of an algorithm in terms of a DDG,*
- *Given the input latency,*
- *Given one-to-one mapping between node types in the DDG and the hardware component types to be used in the architecture,*
- *Given the specification of individual types of component in terms of the delay models described in the previous section,*

Obtain an architecture which is utilization-efficient with respect to specified types of hardware element.

An architecture is defined in two parts: a hardware organization and an execution schedule.

Solution Approach:

The sampling rate of an algorithm is the rate at which new input arrives, which determines the rate at which new computations are initiated. This in turn imposes a repetitive pattern (or schedule) of execution with period equal to $\frac{1}{sampling\ rate}$. It is possible to schedule the computations dynamically, but we find that for the problem defined above, it is sufficient to obtain a static schedule, which also has the practical advantage of being less expensive to implement.

Presence of multi-delay hardware elements in the architecture will, in general, necessitate insertions of buffers in the data paths. Buffers, due to the static scheduling discipline, will also be clocked cyclically, although not simultaneously as in systolic approach, giving a pipelined behavior.

Thus, the approach we take to solve the above stated problem is the one to find a communication structure over the minimum necessary set of hardware components and buffers, and to obtain a static schedule for executing the computation specified by the DDG.

To achieve minimum resource requirement (MRR), the operations need to be distributed uniformly over this period. We achieve this by *sliding* some of the operations forward in time. The sliding may result in increasing the *graph latency*, which is the time delay between the arrival of an input and the production of corresponding output. Insertion of delays to improve input latency has been discussed in [Ko 81]. On the other hand, here we use the same technique for improving resource requirements.

In the following sections we will show how we can analyze the DDG representation of an algorithm and thereby obtain important design information.

4 Analysis of Data Dependency Graphs

In this section we show how DDGs can be analyzed to obtain architectural design information such as number of hardware elements required and sharability of these physical elements. In the end we also look as the special case of single-rate algorithms to show that only single input buffering is sufficient.

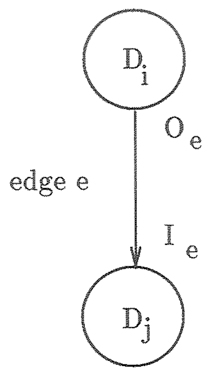


Figure 6: A Dependency

4.1 Token Conservation Equations

In this section we introduce the concept of Token Conservation Equations and use them to analyze the behavior of DDGs.

Consider nodes i and j connected by a dependency as shown in Figure 6. Since there is no loss of data during transfer between the two nodes, under steady state conditions, the average rate of production of tokens (data items) by node i must equal the rate of absorption of tokens by node j. Assuming that p_i and p_j are numbers of elements of types i and j respectively, we have the following relation:

$$p_i \cdot \frac{O_e}{D_i} = p_j \cdot \frac{I_e}{D_j}$$

Each dependency in a DDG gives rise to a *token conservation equation* (TCE) of the form shown above. Note that the equation remains unchanged even if the direction of the dependency is reversed. Thus the steady state behavior of a DDG can be expressed in terms of |E| TCEs, where E is the set of edges of the DDG. For the pipeline to function properly, the set of TCEs must be simultaneously satisfied by the set $< \mathbf{p}_i >$.

The above equation can be transformed to read

$$\frac{O_e}{D_i} \cdot p_i - \frac{I_e}{D_j} \cdot p_j = 0$$

Thus, the system of TCEs is of the form $\mathbf{R} \cdot \mathbf{p} = \mathbf{0}$. Matrix $\mathbf{R}$ can be factored into two matrices $\mathbf{T}$ and $\mathbf{D}^{-1}$ such that, matrix $\mathbf{T}$ involves only token numbers I_e and O_e, the subscripts refering to the corresponding edge, and $\mathbf{D}^{-1}$ is the inverse of the diagonal delay matrix $\mathbf{D}$.

Clearly, since a DDG is a graph with |V| nodes, the rank of its $\mathbf{R}$ matrix must be $\leq$ |V|, the number of variables. However, for the set of TCEs to have a non-trivial solution (the trivial solution is $\mathbf{p} = \mathbf{0}$), the rank of $\mathbf{R}$ must be less than |V| [St 80]. It will be shown that the condition we will formulate below, if satisfied by a DDG, makes the rank of its $\mathbf{R}$ matrix |V|–1 and therefore guaranties a non-trivial solution.

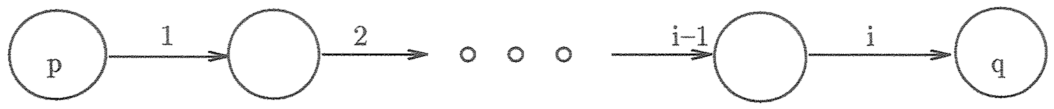


Figure 7: A Path s From p To q

But before we can prove the above, we will establish that a tree-form DDG always has a non-trivial solution.

Definition 9 *An undirected graph obtained by ignoring the directions on edges of a DDG is called an Undirected DDG (UDDG).*

From the definition, the dependencies of a DDG become edges in the UDDG. The TCEs for the edges in a UDDG are those for the corresponding dependencies in the DDG. Thus the set of TCEs describing the UDDG is the same as that of the original DDG and the solution to one is the solution to the other.

Notation: Consider a path s from some node p to some other node q within the UDDG of a given DDG. (See Figure 7.) We form a ratio ρ_s by traversing the path s. Starting at p and with $\rho_s = 1$, when we exit a node, we multiply the denominator by the token marking on the edge leaving the node, and when we enter a node, we multiply the numerator by the token marking on the edge entering that node. For convenience, let us denote factors multiplied into the numerator as O_j and those multiplied into the denominator as I_j. Thus,

$$\rho_s = \frac{\prod\limits_s O_j}{\prod\limits_s I_j}.$$

Consider the set of TCEs corresponding to the dependencies along the path s. Assume that a path s from node p to q is formed by nodes p,1,2, ... ,i,q, as shown in Figure 7. If the set <p$_i$> is chosen such that the set of TCEs is simultaneously satisfied, then the following must hold true:

$$\begin{aligned}
p_p \cdot \frac{O_1}{D_p} &= p_1 \cdot \frac{I_1}{D_1} \\
p_1 \cdot \frac{O_2}{D_1} &= p_2 \cdot \frac{I_2}{D_2} \\
&\vdots \\
p_i \cdot \frac{O_i}{D_i} &= p_q \cdot \frac{I_i}{D_q}
\end{aligned}$$

We can back substitute to obtain the following relationship:

$$p_q = p_p \cdot \frac{D_q}{D_p} \cdot \frac{\prod_{j=1}^{i} O_j}{\prod_{j=1}^{i} I_j}.$$

When p_p and p_q satisfy the above relationship, they are said to satisfy the set of TCEs along the path s.

Notice that,

$$\frac{\prod_{i}^{p} O_j}{\prod_{q}^{1} I_j} = \rho_s \ .$$

Thus, the above relation simplifies to $p_q = p_p \cdot \frac{D_q}{D_p} \cdot \rho_s$.

Theorem 1 : *A DDG in the form of a tree has a non-trivial solution.*

Proof: Since a TCE is insensitive to the direction of the dependency, we may consider the UDDG of the DDG, knowing that a solution to the TCEs of the UDDG will also be a solution to the TCEs of the DDG.

Assume that the nodes in the UDDG are relabelled uniquely, 1 through m, with the root being assigned the label '1'. (The choice of the root node is quite arbitrary; any node may be made the root.) See Figure 8. For each edge e, the number associated with the end closer to the root is referred to as O_e, and the number associated with its other end is referred to as I_e. Also, of the two nodes an edge connects, the one that is nearer the root is considered the parent of the other.

Since the UDDG is a tree, there is a unique path from the root to every node in the graph. We can choose the following values as solutions to the set of TCEs for the UDDG. For a node i ($i \neq 1$), we assign

$$p_i = \frac{D_i}{D_1} \cdot p_1 \cdot \rho_i \ ,$$

where ρ_i is the product as defined earlier taken along the (unique) path from the root (node 1) to node i. p_1 can be chosen to be any rational number other than zero to obtain a non-trivial solution. □

To see that these assignments do satisfy the set of TCEs, consider nodes p and q connected by an edge e such that p is the parent of q. Since the UDDG has the form of a tree, the path from the root to node q is unique and node p and edge e are in it. Therefore, $\rho_q = \rho_p \cdot \frac{O_e}{I_e}$. We must verify that the assignments to p_p and p_q satisfy the TCE for the edge e. We have

$$p_q \ = \ \frac{D_q}{D_1} \cdot p_1 \rho_q$$

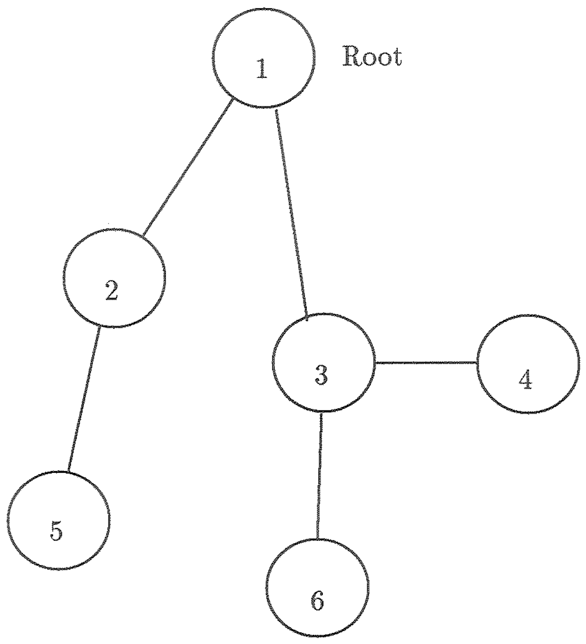


Figure 8: A Tree-Form UDDG

$$
\begin{aligned}
&= \frac{D_q}{D_1} \cdot p_1 \cdot \rho_p \cdot \frac{O_e}{I_e} \\
&= \frac{D_q}{D_1} \cdot \frac{D_p}{D_p} \cdot p_1 \cdot \rho_p \cdot \frac{O_e}{I_e} \\
&= \frac{D_q}{D_p} \cdot \frac{D_p}{D_1} \cdot p_1 \cdot \rho_p \cdot \frac{O_e}{I_e} \\
&= \frac{D_q}{D_p} \cdot p_p \cdot \frac{O_e}{I_e}
\end{aligned}
$$

That is,

$$p_q \cdot \frac{I_e}{D_q} = p_p \cdot \frac{O_e}{D_p}$$

Notice that the solution $< p_i >$ is uniquely determined by the choice of p_1. That is, the solution has exactly one free variable, and not more than one variable can be independently chosen. Put in matrix terms, the corresponding **R** matrix has the rank of one less than the number of variables. We can therefore state the following lemma:

Lemma 2 *The* **R** *matrix of a tree-form DDG with V nodes has the rank of* $|V|-1$.

Since any DDG contains its spanning tree with all its nodes, we can conclude that,

Lemma 3 *The* **R** *matrix of a DDG with* $|V|$ *nodes has the rank* $\geq |V|-1$.

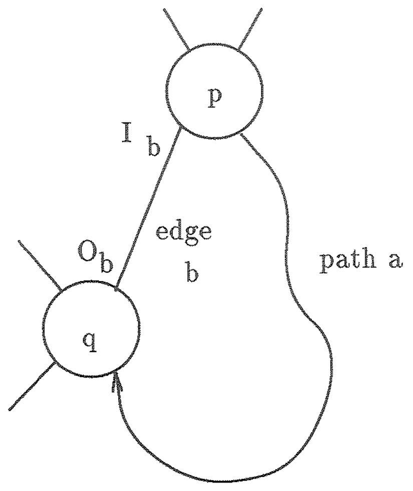


Figure 9: A Cycle Within A CDDG

The aforementioned condition is expressed in the form of a definition as follows:

Definition 10 *: A DDG is said to be a Consistent DDG (CDDG), if and only if, for every cycle l in its UDDG, $\rho_l = 1$.*

Consider the dependency loop shown in Figure 9, which may be part of a CDDG. 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 4 *: If a set of p_is satisfies the set of TCEs for path a, they also satisfy the TCE for edge b.*

Proof: Let p_p, ..., p_q, be the set of values satisfying the TCEs along path a. Then,

$$p_q = p_p \cdot \frac{D_q}{D_p} \cdot \rho_a.$$

By condition 1 in Definition 10 above,

$$\rho_l = \rho_a \cdot \frac{O_b}{I_b} = 1.$$

Thus,

$$p_q = p_p \cdot \frac{D_q}{D_p} \cdot \frac{I_b}{O_b}.$$

□

This lemma implies that in order to obtain a solution for the set of TCEs representing dependencies that form a cycle in a UCDDG, we need to consider all but one of the dependencies. The solution obtained will satisfy the TCE for the remaining dependency. In algebraic terms, it means that the TCE for the last dependency is linearly dependent on the equations for the other dependencies forming the cycle. Or, in other words, if there are m dependencies which form a cycle, then at most m–1 of the TCEs are linearly independent. Put differently, if the set of TCEs for a cycle are expressed in a matrix form, the corresponding **R** matrix has the rank of m–1. We will state this in the form of the following corollary.

Corollary 3 *The* **R** *matrix corresponding to a set of TCEs for the edges of a loop with m nodes of a UCDDG has the rank of m–1; each TCE is linearly dependent on the other m–1 TCEs.*

Theorem 2 *The* **R** *matrix of a CDDG with $|V|$ nodes has the rank of $|V|$–1.*

Proof: We have already shown that the rank of the **R** matrix is greater than or equal to |V|–1 and that it cannot be greater than |V|.

Consider the UCDDG of the CDDG being analyzed. Let the edges of the UCDDG be divided into two disjoint sets E_t and E_l, such that E_t is the set of edges in the spanning tree of the UCDDG and E_l is the set of remaining edges of the UCDDG. Thus $E_t \cup E_l$ is the set of all edges in the UCDDG. The rank of the **R** matrix corresponding to the edges in E_t is |V|–1 (by Lemma 2).

By the definition of a spanning tree, every edge e in E_l creates a simple cycle when added to the set E_t. But by Lemma 3, the edge e does not add to the rank of the **R** matrix. This is true for every edge in E_l. Thus the rank of the **R** matrix of the UCDDG, and therefore of CDDG, is the rank of the **R** matrix corresponding to the edges in E_t, which is |V|–1. □

The consequence of the above theorem is that,

Theorem 3 : *The set of TCEs for a CDDG has a non-trivial solution.*

The satisfaction of the condition for CDDG can verified by using the following algorithm.

Algorithm 1 : Algorithm to check for a CDDG.

- *Obtain the UDDG from the given DDG.*
- *Use a Depth First Search algorithm*[1].
- *Assign $\rho_{root} = 1$.*
- *On traversing an edge (u, v) from node* u *to node* v*, compute $\rho_v = \rho_u \cdot \frac{O_u}{I_v}$, where O_u and I_v are markings on the edge (u, v) at* u *and* v *respectively.*
- *For each back edge (u, v), verify that $\rho_v = 1$.*

Since the rank of **R** of a CDDG is |V|–1, the solution has one free variable. Thus given a solution vector **p**, $\theta \cdot$**p** is also a solution, where θ is a scalar. Typically, the free variable is bound by the input rate, which is often determined externally by the application.

[1]As described in [Ah 74]

4.2 Utilization Factors

In general, the solution of the TCEs, $< \mathbf{p}_i >$, are rational numbers. However, the actual number of elements of type i in an architecture must be an integer. One may choose $\mathbf{n}_i = \lceil \mathbf{p}_i \rceil$, or any larger integer.

Given $\mathbf{n}_i \geq \mathbf{p}_i$, we interprete $\mathbf{u}_i = \frac{\mathbf{p}_i}{\mathbf{n}_i}$ as the average utilization of $\mathbf{n}_i$ elements of type i. Effectively, for every element of type i, the delay is changed from D_i to $\mathrm{T}_i = \frac{D_i}{\mathbf{u}_i}$. Thus, each element idles for $\mathrm{T}_i - \mathrm{D}_i$ time units. We refer to T_i as the *effective latency* of element of type i.

4.2.1 Single-rate Case

From the analysis presented in the foregoing sections, it is clear that a spanning tree of the UDDG is all that needs to be analyzed to obtain the solution to the TCEs of the DDG. So consider such a tree of the given DDG, with input node of the DDG as its root.

Since the DDG is single-rate, each dependency has the same number of tokens associated with its head and tail ends. From the proof of Theorem 1 we have

$$p_i = \frac{D_i}{D_1} \cdot p_1 \cdot \rho_i \; .$$

Remember that ρ_i is formed by multiplying token markings of edges along the path from the root to node i into the numerator and the denominator. One of the two markings associated with an edge is multiplied into the numerator and the other into the denominator. Since for a single-rate algorithm the two markings are equal, $\rho_i = 1$ for all i.

It is reasonable to assume that there is a single physical input source, and thus $p_1 = 1$. The input latency (T) is the delay of the input node; i.e., $\mathrm{T} \equiv \mathrm{D}_1$. Thus we have, $p_i = \frac{D_i}{T}$. With the above interpretation of p_i as $u_i \cdot n_i$ and comparing the denominators, we see that for single-rate algorithms $\mathrm{T}_i = \mathrm{T}$, for all i for which $\mathrm{D}_i \leq \mathrm{T}$. For operations which have $\mathrm{D}_i > \mathrm{T}$, the effective latency T_i is an integer multiple of T ($= \mathrm{n}_i \cdot \mathrm{T}$). It is also easy to see that for an operation of type i, regardless of where it appears in the DDG, p_i is the same.

4.3 Sharable Resources For Single-rate Algorithms

For a single-rate algorithm, the algorithm graph executes once for each input, resulting in a cyclic execution schedule with a period defined by the input latency (T). We are interested in *static* schedules[2], which repeat every T clock cycles[3]. To distinguish between a clock cycle and an execution cycle which repeats every T clock cycles, we refer to the latter as a *schedule cycle.*

Every operation represented in the DDG is executed exactly once every schedule cycle, although executions of different elements may correspond to different input values. The position within a schedule cycle at which a particular operation is executed is obtained by taking modulus (T) of its relative time position with respect to the input node of the DDG.

A hardware element may be used to execute more than one operation per schedule cycle, which leads to sharing of hardware. For a given execution schedule, one can find the number of

[2]Since the architecture is heterogeneous, we cannot use cyclo-static schedules described in [Sc 85].

[3]This property has practical importance in that it simplifies the control structure for the architecture.

elements of a type required by finding the maximum number of operations of its type occurring in any clock cycle. The MRR is simply $\lceil \frac{n_i \cdot D_i}{T} \rceil$, where n_i is the number of occurrences of operation of type i and D_i is the delay of the hardware element executing the operation.

Theorem 4 *In a single-rate algorithm implementation with the static cyclic scheduling discipline, a hardware element of type i is sharable only if $p_i \leq 0.5$.*

Proof: For a statically scheduled execution, an operation is scheduled to execute at the same relative time within the schedule cycle for all schedule cycles. In effect, one execution of an element during a single schedule cycle represents a *footprint* of all similar executions of that element for all schedule cycles. Thus we need to look at only a single schedule cycle to determine sharability of hardware elements. An element that is shared is used multiple times within a schedule cycle, and those multiple uses will be repeated for every cycle. Clearly, for an element to be used more than once, its footprints cannot overlap. Therefore an element can be shared only if we can pack more than one of its footprints in a single cycle. This is possible only if $D_i \leq \frac{T}{2}$, that is, if $p_i \leq 0.5$. □

4.4 Buffer Requirements

The process of designing an architecture inserts buffers for holding tokens until they are ready to be used. In this section we will obtain estimates of the number of buffers required and the size of each of them.

4.4.1 Buffer Model

All buffers behave as FIFOs. It is assumed that these are implemented as circular buffers with separate read and write pointers. The following behavior is assumed of all circular buffers.

A buffer is composed of a set of registers, each of the size of a token it is expected to store.

A read or write request starts at the beginning of a clock cycle and terminates at the end of it.

When a register is written into, it is marked full until read from.

When a register is read from, it is marked empty until written into.

A write request to a full register is considered a conflict.

A read request to an empty register is considered a conflict.

Simultaneous reads or writes of buffer registers are not permitted.

Simultaneous read and write of different registers of a buffer is permitted.

Simultaneous read and write of the same register of a buffer is not permitted and is considered a conflict.

An example of such a device is the Am 29338 from Advanced Micro Devices. We will use similar circuits tailored to desired sizes in the synthesized architectures.

4.4.2 Buffer Sizes

Given the model of a buffer and the model of node behavior, we can determine the size of buffers that are inserted at the computational element types connected by dependencies. In general, we will assume that a buffer is inserted for each dependency. Thus for each dependency in the DDG, there exists an output buffer in the node at its tail or an input buffer in the node at its head, depending on whether the communication element is at the head or at the tail of the particular dependency, respectively[4].

For the sake of discussion here, we will refer to a dependency as "input dependency" if the node at its tail is a communication node, and as "output dependency" if the node at its head is a communication node. Consider an input dependency e. Let us denote the node at its head by B, and the communication node at its tail as b. Notice that for an input dependency like e, $O_e = 1 = D_b$, according to the specification of communication primitives.

It is impractical to estimate the buffer sizes for the most general case in which the token markings on an edge are arbitrary integers. It is more realistic to estimate the buffer sizes with certain restrictions put on the combinations of token markings. We will restrict our discussion here to the special case of single-rate, that is, every computational or storage element in the architecture absorbs one input token and output token upon firing.

Consider the scheduling discipline in which every communication element fires to coincide with the output phase of its parent node in the DDG. Thus the data output by the parent node is transmitted directly to the destination nodes and may be latched in their input buffers. We assume that a computation element has a physical port for every dependency that the operation it implements in the DDG has. This, along with the single-rate constraint on the algorithm implies that the lengths of input and output phases of all computation elements are one.

We can argue that under these assumptions, a single input buffer for each computation and storage element is necessary and sufficient and that no output buffers are necessary. Consider two nodes A and B, in a DDG, connected by a dependency such that the output of A is input by B. Since it is a single-rate algorithm, each firing of an input node results in one firing of each node in the algorithm DDG. The effective latency for each node is the same and is equal to the input latency T. That is, each node of the DDG fires exactly once every T time units. Thus if A fires and produces a token at the end of that firing, B must fire once before A produces the token again. That is, A will always find an empty buffer. Similarly, after B fires and absorbs the input token, A must complete one execution within the next T time units and produce an output which is stored in B's input buffer. Thus when B is ready to fire again after the T time units it always finds a token in its input buffer.

The coincidence of the execution of the child communication operation and the "write" of the destination buffer with the output phase of computation element is crucial to the sufficiency of only single input buffers; if this coincidence is compromised, output buffers are also needed.

References

[Ah 74] The Design and Analysis of Computer Algorithms; A. V. Aho, J. E. Hopcroft and J.

[4] Note that for each dependency in the DDG, one of the two nodes at its ends is always a communication node and the other is a computation or storage node.

D. Ullman, Addison Wesley Publishing Company, 1974.

[An 85] "Warp Architecture and Implementation - Preliminary Version"; M. Annaratone et al, Dept. of Computer Science, Carnegie-Mellon University, 1985.

[Cr 83] Multirate Digital Signal Processing; R. E. Crochiere and L. R. Rabiner, Prentice-Hall, 1983.

[Gr 85] Discrete and Combinatorial Mathematics: An Applied Introduction; R. P. Grimaldi, Addison Wesley Publishing Company, 1985.

[Ko 81] The Architecture of Pipelined Computers; P. M. Kogge, Hemisphere Publishing Corporation, McGraw-Hill Book Company, 1981.

[Le 81] "Optimizing Synchronous Systems"; C. E. Leiserson and J. B. Saxe, *22nd IEEE Symposium On Foundations Of Computer Science, 1981, pp 23-36.*

[Sc 85] "Synchronous Multiprocessor Realizations of Shift-Invariant Flow Graphs"; D. A. Schwartz, *Ph.D. Dissertation, July 1985, School of Electrical Engineering, Georgia Institute of Technology, Atlanta, Georgia 30332.*

[St 80] Linear Algebra and Its Applications; G. Strang, Academic Press Inc., 1980.