A New Technique for Performance Constrained Scheduling in Data Path Synthesis

Yuan Hu, *Ahmed Ghouse* and *Bradley S. Carlson*

The Department of Electrical Engineering
State University of New York at Stony Brook
Stony Brook, NY 11794-2350
Phone: (516) 632-8474
Fax: (516) 632-8494
Technical Report 662
College of Engineering and Applied Science

Abstract— An algorithm is presented to solve the problem of *performance constrained* scheduling in data path synthesis. Given a time constraint T for completion of the operations in the data flow graph (DFG), the algorithm minimizes the number of functional units required in the hardware implementation of the DFG. The algorithm achieves superior results by computing a lower bound on the number of functional units required to execute the operations of the DFG under the performance constraint T, and then employing a simple resource constrained scheduling algorithm to schedule the operations into the best control steps. The lower bound not only greatly reduces the size of the solution space, but also provides a means to measure the proximity of the final solution to an optimal one. Since the bound is computed in polynomial time this scheduling algorithm is very effective, especially for large DFGs. Experiments indicate that the lower bound is very tight. For all of the test cases the difference between our solution and the optimal solution is not greater than one. The solution of the performance constrained scheduling problem is presented for data flow graphs which contain multi-cycle, structural pipelined operations and functional pipelined operations. This technique can be extended to other types of DFGs.

I INTRODUCTION

In the behavioral synthesis of integrated circuits (ICs) the fundamental steps are the scheduling and allocation of operations on functional (hardware) units. The synthesis is commonly divided into a data path design and a control path design. Operation scheduling and allocation are the most important steps in behavioral synthesis [1], since they can effectively determine the time-cost trade-offs. The scheduling problem can be approached from two perspectives: *resource constrained* and *performance constrained.* Resource constrained scheduling is the task of minimizing the completion time given a fixed amount of resources, and performance constrained scheduling is the task of minimizing the amount of resources given a fixed completion time. Most scheduling techniques focus on the former variant of the problem. Presented in this paper is an algorithm to solve the performance constrained scheduling problem using lower and upper bounds on the number of functional units required. We address the performance constrained scheduling problem because it is more complex than the resource constrained scheduling problem. Furthermore, it is a more realistic problem given the advancement of integrated circuit technology (e.g., increased circuit densities).

The new technique is superior to existing performance constrained scheduling algorithms because it is more efficient in design space search, and the lower bound derived in this paper quantifies the quality of the solution.

The research on data path scheduling and allocation is addressed in [2, 3, 4, 5, 6, 7, 8, 9]. Early work (e.g., [8]) used exhaustive search and is therefore not practical for large designs. Some branch-and-bound techniques such as integer linear programming have been used by Hafer and Parker [7], Lee [6] and Balakrishnan [4]. These algorithms have exponential complexity by nature, and therefore are not practical for large designs. Most polynomial time heuristics are constructive (e.g., [2] [5]). They use a stepwise refinement approach in which the selection and assignment of operations to the control steps is one at a time until all operations are assigned.

The most notable of polynomial time heuristic scheduling algorithms are force-directed scheduling [2], neural network based scheduling (NNS) [19] and multiple exchanging pair selection algorithm [5]. Force-directed scheduling uses so-called *force* to guide scheduling, with computational complexity $O(Tm^3)$, where T is the height of the data flow graph and m is the number of operations in the graph. NNS [19] uses an energy function to guide scheduling by moving the cost function towards an equilibrium point such that the total

cost is minimized. Park's [5] multiple exchanging pair selection algorithm with a complexity of $O(Tm^2 log\ m)$ is based on a partitioning algorithm in which maximal cumulative gain is used to decrease the total cost. A comprehensive survey of various scheduling and allocation techniques can be found in [10].

A common characteristic of these polynomial time heuristics is that they give only an *upper bound* to the minimum number of functional units needed to meet the design constraints. Although they obtain near optimal or optimal schedules on a few test examples, these algorithms in the general case have no way of knowing the proximity of their solution to the optimal solution. The lack of a tight lower bound is recognized as one of the weaknesses in current scheduling algorithms [20]. Generally, the quality of a heuristic solution for an NP-complete problem is proportional to the time spent in search for a good solution; however, a quantitative measure of the quality of a heuristic solution can only be obtained by measuring its proximity to the optimal solution. Therefore, tight lower and upper bounds are essential to providing a good measure of the quality of an algorithm when the optimal solution is too expensive to compute.

The purpose of our paper is to present a new polynomial time algorithm to solve the performance constrained scheduling problem such that it gives both a lower and upper bound to the optimal number of functional units required to finish all the operations in a data flow graph under time constraint T. Since our algorithm computes a lower bound on the optimal solution, it is possible to observe the proximity of the solution to the optimal one. The design space is well defined by the lower and upper bounds, and when used in conjunction with a simple resource constrained scheduling algorithm, the functional units can be successfully allocated to operations to meet the time constraints in short search time. We have tested our method on 25 different data flow graphs for various time constraints and types of operations, and our solutions differ from the optimal solutions by at most one.

The organization of our paper is as follows. Presented in Section II are the bounds on the number of functional units for non-pipelined, structural pipelined and functional pipelined cases. Presented in Section III is the "cut the longest queue" scheduling algorithm for non-pipelined, structural pipelined operations and scheduling algorithm for functional pipelined operations. These scheduling algorithms are used to assign operations to functional units and consequently give the upper bounds. Section IV contains the complexity analysis and presentation of test examples, and the paper is concluded in Section V.

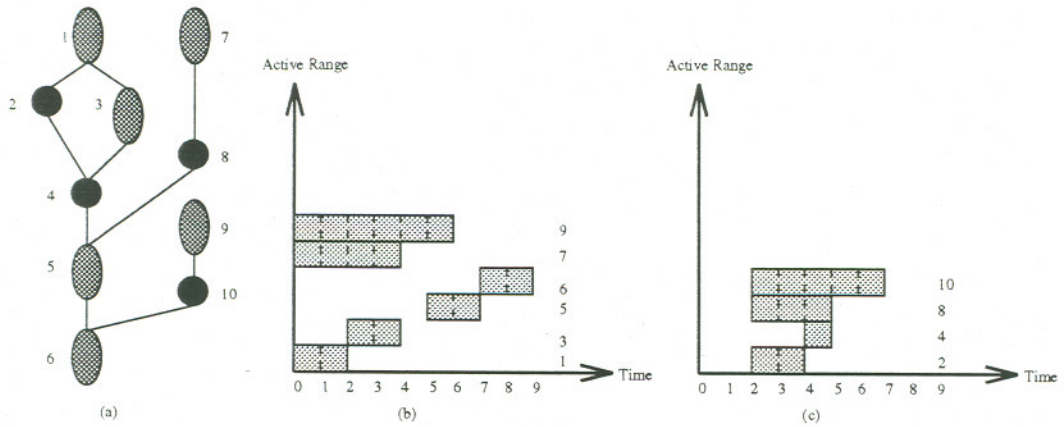

Fig. 1. (a) An example data flow graph, (b) the list of active ranges for the multiplications, and (c) the list of active ranges for the additions.

II Bounds On The Number Of Functional Units

In this section lower and upper bounds on the number of functional units required to execute the operations of a data flow graph within a specified time are derived for the non-pipelined operations, structural pipelined operations and functional pipelined operations. As a prerequisite for computing the bounds on the required number of functional units the as soon as possible (ASAP) and the as late as possible (ALAP) schedules[1] under the assumption of an unlimited amount of resources must be computed for every operation in the graph, and the critical path must be identified. The algorithms for processing the data flow graph to find the ASAP and ALAP schedules can be found in [22]. The run time complexities of these algorithms are $O(E)$, where E is the number of edges in the graph.

The ASAP and ALAP schedules are used to compute bounds on the number of functional units of each type required to execute the operations of the DFG within a specified time. The final solution is a specification of the number of functional units of each operation type and a schedule of all operations of the data flow graph. For example, Fig. 1(a) is a data flow graph with two types of operations: addition and multiplication. In our method the bounds for the number of multipliers and the bounds for the number of adders are determined separately and combined to form the final solution. The ASAP and ALAP schedules of each operation in Fig. 1(a) are shown in Table I. In Fig. 1(a) the operation identifier is shown adjacent to the node, an oval represents a multiplication with two units of delay and a circle represents

[1] A schedule is represented by the completion times of its operations.

Table I. The ASAP and ALAP schedules of operations in Fig.1(a).

Operations	1	2	3	4	5	6	7	8	9	10
ASAP	2	3	4	5	7	9	2	3	2	3
ALAP	2	4	4	5	7	9	4	5	6	7

an addition with one unit of delay.

A Non-pipelined Operations

Let m be the total number of operations in the data flow graph, τ_j be the time for *completing* the j^{th} operation in the graph ($j = 1, 2, ..., m$), and let integer s be the time required to execute an operation. An m-tuple $(\tau_1, \tau_2, ..., \tau_m)$ of completion times for the m operations in a data flow graph represents a schedule of the graph. Let $\overline{\tau_j}$ be the *completion* time of operation j in the as soon as possible (ASAP) schedule of the graph, and let $\underline{\tau_j}(T)$ be the *completion* time of operation j in the as late as possible (ALAP) schedule. Note that the ALAP completion time $\underline{\tau_j}(T)$[2] depends on the total completion time T.

The time interval $[\overline{\tau_j} - s, \underline{\tau_j}]$ is called the *active range* for operation j. The active range for each operation can be listed in a Cartesian coordinate system as shown in Figs. 1(b) and (c). For example, in the data flow graph in Fig. 1(a) the total completion time is $T = 9$. It is assumed that two units of time ($s = 2$) are needed to complete a multiplication operation; therefore, for multiplication 7 we have $\overline{\tau_7} = 2$ and $\underline{\tau_7(9)} = 4$. Thus the active range of operation 7 is $[\overline{\tau_7} - s, \underline{\tau_7(T)}] = [0, 4]$ as indicated in Fig. 1(b).

Given a schedule $(\tau_1, \tau_2, ..., \tau_m)$ the activity of operation j in the graph can be described by the following function.

$$f(\tau_j, t) = \begin{cases} 1 & t \in [\tau_j - s, \tau_j] \\ 0 & \text{otherwise} \end{cases}$$

$f(\tau_j, t)$ is called the *active function*. $f(\tau_j, t)$ is the actual representation of an operation in a schedule $(\tau_1, \tau_2, ..., \tau_m)$, while the active range $[\overline{\tau_j} - s, \underline{\tau_j}]$ represents the freedom of an operation among all possible schedules. Furthermore, the active function of the entire data flow graph can be described as the sum of the active functions of the individual operations.

$$F(\tau_1, \tau_2, ..., \tau_m, t) = \sum_{j=1}^{m} f(\tau_j, t)$$

[2] $\underline{\tau_j}(T)$ is abbreviated as $\underline{\tau_j}$ in the sequel.

F is the active function of the entire data flow graph with respect to the schedule $(\tau_1, \tau_2, ..., \tau_m)$. F is an indication of how many operations are being executed *simultaneously* at an instant of time t given a schedule $(\tau_1, \tau_2, ..., \tau_m)$. The problem of finding the minimum number of functional units n to finish the computational task within time T can be formulated as finding an integer n and an optimal schedule $(\tau_1^*, \tau_2^*, ..., \tau_m^*) \subset D$ such that

$$n = \max_{\forall t \in [0,T]} F(\tau_1^*, \tau_2^*, ..., \tau_m^*, t) = \min_{\forall (\tau_1, \tau_2, ..., \tau_m) \in D,} \max_{\forall t \in [0,T]} F(\tau_1, \tau_2, ..., \tau_m, t) \text{ [11]},$$

where D is the schedule space. That is, the minimum number of functional units required is the minimum over all feasible schedules of the maximum activity during each unit time interval over the time interval $[0, T]$.

To find a feasible schedule under these conditions is an NP-complete problem, but there are possibilities of finding a tight lower bound and one way to obtain a good lower bound is to relax the constraints. There are three major constraints, the precedence relationships between operations, the active ranges and the completion time T. We construct a relaxation by eliminating constraints for the precedence relationships, but keep the active ranges of operations. As a result of this relaxation each operation can be considered *independently.* We define the *minimum load* for each operation j in the interval $[t_1, t_2]$, denoted $\phi_j(t_1, t_2)$, as

$$\phi_j(t_1, t_2) = \min_{\forall \tau_j} \int_{t_1}^{t_2} f(\tau_j, t) dt. \tag{1}$$

The integral indicates how much of the active function $f(\tau_j, t)$ of operation j will be present in the interval $[t_1, t_2]$ for a given completion time τ_j. Therefore, $\phi_j(t_1, t_2)$ is the minimum portion of the active function which *must* be present in the interval $[t_1, t_2]$ among *all* schedules. The collective effect of the $\phi_j(t_1, t_2)$'s of all the operations is an indication of the minimum number of functional units needed to meet the time constraint.

Given the ASAP and ALAP schedules, let $\overline{n_j(t_1, t_2)}$ be the overlap between the active function $f(\overline{\tau_j}, t)$ and $[t_1, t_2]$ for the ASAP schedule, and $\underline{n_j(t_1, t_2)}$ be the overlap between the active function $f(\underline{\tau_j}, t)$ and $[t_1, t_2]$ for the ALAP schedule. The overlap between two time intervals A and B is denoted by $|A \cap B|$. The computation of minimum load $\phi_j(t_1, t_2)$ is given in the following lemma.

Lemma 1

$$\phi_j(t_1, t_2) = min\{\overline{n_j(t_1, t_2)}, \underline{n_j(t_1, t_2)}\}, \tag{2}$$

where

$$\overline{n_j(t_1,t_2)} = |[\overline{\tau_j} - s, \overline{\tau_j}] \cap [t_1,t_2]| \text{ and } \underline{n_j(t_1,t_2)} = |[\underline{\tau_j} - s, \underline{\tau_j}] \cap [t_1,t_2]|.$$

Proof:

We partition the time axis into five regions by four time instances, $\overline{\tau_j} - s$, $\overline{\tau_j}$, $\underline{\tau_j} - s$ and $\underline{\tau_j}$. If $\underline{\tau_j} - s \geq \overline{\tau_j}$, the partition is called a *non-overlapped* partition; otherwise it is called an *overlapped* partition. Given a partition of the time axis, there are $\begin{pmatrix} 5 \\ 2 \end{pmatrix} + \begin{pmatrix} 5 \\ 1 \end{pmatrix} = 15$ possible positions for an arbitrary time interval $[t_1,t_2]$ on the time axis depending on the locations of t_1 and t_2. We prove the lemma for one of the positions of $[t_1,t_2]$, and the proofs for the others are similar. Let us assume that we are given a non-overlapped partition of the time axis and

$$t_1 < \overline{\tau_j} - s, \quad \underline{\tau_j} - s < t_2 < \underline{\tau_j}.$$

According to the definition of $\phi_j(t_1,t_2)$, it is the minimum integral (overlap) possible between $[t_1,t_2]$ and $f(\tau_j,t)$. Therefore,

$$\phi_j(t_1,t_2) = \min_{\forall \tau_j} \int_{t_1}^{t_2} f(\tau_j,t)dt = t_2 - (\underline{\tau_j} - s).$$

On the other hand,

$$\overline{n_j(t_1,t_2)} = |[\overline{\tau_j} - s, \overline{\tau_j}] \cap [t_1,t_2]| = s$$

and

$$\underline{n_j(t_1,t_2)} = |[\underline{\tau_j} - s, \underline{\tau_j}] \cap [t_1,t_2]| = t_2 - (\underline{\tau_j} - s) < s.$$

Thus,

$$min\{\overline{n_j(t_1,t_2)}, \underline{n_j(t_1,t_2)}\} = s - (\underline{\tau_j} - t_2) = \phi_j(t_1,t_2).$$

Similarly, we can prove for the other positions of $[t_1,t_2]$ that

$$\phi_j(t_1,t_2) = min\{\overline{n_j(t_1,t_2)}, \underline{n_j(t_1,t_2)}\}.$$

■

Consider operation 7 in Fig. 1(b). If $[t_1,t_2] = [0,3]$, then

$$\overline{n_7(t_1,t_2)} = |[\overline{\tau_7} - s, \overline{\tau_7}] \cap [t_1,t_2]| = |[0,2] \cap [0,3]| = 2,$$

$$\underline{n_7(t_1,t_2)} = |[\underline{\tau_7} - s, \underline{\tau_7}] \cap [t_1,t_2]| = |[2,4] \cap [0,3]| = 1,$$

and

$$\phi_7(t_1,t_2) \;=\; min\{\overline{n_7(t_1,t_2)}, \underline{n_7(t_1,t_2)}\} = min\{2,1\} = 1.$$

Consequently, the relationships between the minimum load (ϕ_j), minimum number of functional units (n) and the lower bound (n_L) of n is given in the following theorem. The time interval $[t_1,t_2]$ is a sub-interval of $[t_3,t_4]$, denoted as $[t_1,t_2] \subseteq [t_3,t_4]$, if $t_3 \le t_1 \le t_2 \le t_4$.

Theorem 2 *For n to be the optimal number of non-pipelined functional units needed to complete the computational task of a data flow graph within time T, it is necessary that for* **any possible** *time interval $[t_1,t_2] \subseteq [0,T]$,*

$$\sum_{j=1}^{m} \phi_j(t_1,t_2) \le n(t_2-t_1), \tag{3}$$

where $\phi_j(t_1,t_2)$ is defined in Eq. (1). Furthermore, a lower bound on n is

$$n_L = \lceil \max_{\forall [t_1,t_2] \subseteq [0,T]} \frac{\sum_{j=1}^{m} \phi_j(t_1,t_2)}{(t_2-t_1)} \rceil.$$

Proof: Since it is the proof of a necessary condition, it must be shown that Eq. (3) is true for all time intervals $[t_1,t_2]$. Suppose n is the minimum number of functional units required to implement an optimal schedule $(\tau_1^*, \tau_2^*, ..., \tau_m^*)$, then $n \ge F(\tau_1^*, \tau_2^*, ..., \tau_m^*, t)$ over $[t_1,t_2]$ since n is the maximum over $[0,T]$. Hence,

$$\begin{aligned} \sum_{j=1}^{m} \phi_j(t_1,t_2) &= \sum_{j=1}^{m} \min_{\forall \tau_j} \int_{t_1}^{t_2} f(\tau_j,t)dt \le \int_{t_1}^{t_2} \sum_{j=1}^{m} f(\tau_j^*,t)dt \\ &= \int_{t_1}^{t_2} F(\tau_1^*, \tau_2^*, ..., \tau_m^*, t)dt \le n(t_2-t_1). \end{aligned}$$

Therefore,

$$n \ge \frac{\sum_{j=1}^{m} \phi_j(t_1,t_2)}{(t_2-t_1)}$$

for every sub-interval $[t_1,t_2]$ of $[0,T]$. Thus, every $\frac{\sum_{j=1}^{m} \phi_j(t_1,t_2)}{(t_2-t_1)}$ is a lower bound on n. Since n is an integer, the tightest lower bound n_L is

$$n_L = \lceil \max_{\forall [t_1,t_2] \subseteq [0,T]} \frac{\sum_{j=1}^{m} \phi_j(t_1,t_2)}{(t_2-t_1)} \rceil.$$

■

The number of possible sub-intervals in the time interval $[0,T]$ is $T(T+1)/2$; therefore, there are $T(T+1)/2$ different $\sum_{j=1}^{m} \phi_j(t_1,t_2)$'s to compute in order to determine n_L.

By going through $T(T+1)/2$ time intervals we obtain not only a lower bound, but also an upper bound on the number of functional units. The number of functional units needed to implement an *arbitrary* feasible schedule constitutes an upper bound. The ASAP and ALAP schedules are feasible schedules, and therefore can be used to compute an upper bound. The number of functional units needed by the ASAP and ALAP schedules are denoted by $\overline{n_U}$ and $\underline{n_U}$, respectively. The upper bound is

$$n_U = min\{\overline{n_U}, \underline{n_U}\},$$

where

$$\overline{n_U} = \max_{\forall t \in [0,T]} F(\overline{\tau_1}, \overline{\tau_2}, ..., \overline{\tau_m}, t), \quad \text{and} \quad \underline{n_U} = \max_{\forall t \in [0,T]} F(\underline{\tau_1}, \underline{\tau_2}, ..., \underline{\tau_m}, t).$$

Both the ASAP and ALAP schedules are used to derive the upper bound because both $F(\overline{\tau_1}, \overline{\tau_2}, ..., \overline{\tau_m}, t)$ and $F(\underline{\tau_1}, \underline{\tau_2}, ..., \underline{\tau_m}, t)$ are computed for every t during the computation of the lower bound, i.e., additional effort is not required to compute this upper bound.

The upper bound may not be tight, since only two different schedules are considered; however, it can save unnecessary computational effort in some cases. Any schedules for which $n < n_U$ is a better schedule than both the ASAP and ALAP schedules. The improvement of the upper bound is achieved by using a better scheduling algorithm. The details of scheduling will be discussed in a later section.

Example

In the example shown in Fig. 1(b) the completion time is $T = 9$, the total number of multiplication operations is 6 and the delay of multiplication is 2. The multiplication is a two-cycle operation. According to Theorem 2, among all $\sum_{j=1}^{10} \phi_j(t_1, t_2)$'s, $[t_1, t_2] = [0, 4]$ gives the maximum $\frac{\sum_{j=1}^{10} \phi_j(t_1,t_2)}{(t_2-t_1)}$. Thus, the lower bound on the number of multipliers is

$$n_L = \lceil \frac{\sum_{j=1}^{10} \phi_j(0,4)}{(4-0)} \rceil = \lceil \frac{\phi_1(0,4) + \phi_3(0,4) + \phi_7(0,4) + \phi_9(0,4)}{4} \rceil = \lceil \frac{2+2+2+0}{4} \rceil = 2.$$

The upper bound is derived from the ALAP schedule. Thus,

$$n_U = min\{\overline{n_U}, \underline{n_U}\} = \underline{n_U} = 2, \quad at \quad t = 3.$$

Since $n_L = n_U$, the optimum number of multipliers is 2 and the ALAP schedule is an optimal schedule.

B *Structural Pipelined Operations*

So far we have discussed the lower and upper bounds for multi-cycle operations. If the execution of an operation in a data flow graph can be divided into the executions of sub-operations and can be pipelined on available functional units, then the operation is called a *structural pipelined operation.* In a structural pipelined architecture the number of pipelined functional units[3] needed is different from that of the non-pipelined case. For example, a schedule of 4 multiplications is shown in Fig. 2, and each multiplication can be divided into sub-operations A B C and D. The number of pipelined multipliers needed to complete the schedule is not 4 which is the maximum number of multiplications active, but instead 3 since the execution of any sub-operation of multiplication 1 can be carried out by any of the free multiplier sub-units previously executing sub-operations of multiplications 2, 3 or 4. Therefore, the minimum number of pipelined multipliers needed is the *maximum* number of sub-units[4] of a *certain kind*, i.e. $n_p = max\{n_A, n_B, n_C, n_D\} = max\{3,3,3,3\} = 3$, where n_i is the minimum number of i^{th} sub-units of the pipelined multiplier needed during any unit time interval.

The idea of structural pipelining is to divide an operation into s sub-operations and pipeline these s sub-operations onto any pipelined functional sub-unit *as soon as* one is available, instead of waiting for the *entire* operation to be completed as in the non-pipelined case. The active function $f(\tau_j, t)$ for each operation can be divided into s *active sub-functions*, i.e.,

$$f(\tau_j, t) = \sum_{k=1}^{s} f^k(\tau_j, t).$$

Each active sub-function is defined for sub-operation k as

$$f^k(\tau_j, t) = \begin{cases} 1 & t \in [\tau_j - s + k - 1, \tau_j - s + k] \\ 0 & \text{otherwise.} \end{cases}$$

After the division of an operation into sub-operations, each sub-operation in a structural pipelined data flow graph can be treated as an independent operation with execution time equal to one. The active function of *all* the i^{th} sub-operations in a data flow graph is defined as

$$F^i(\tau_1, \tau_2, ..., \tau_m, t) = \sum_{j=1}^{m} f^i(\tau_j, t) \quad 1 \le i \le s, \tag{4}$$

where s is the number of sub-units of the pipelined functional unit and m is the total number of operations in the data flow graph. $F^i(\tau_1, \tau_2, ..., \tau_m, t)$ is the sum of the active

[3] A pipelined functional unit is an entire pipeline of hardware.

[4] A sub-unit is a stage of a pipeline hardware.

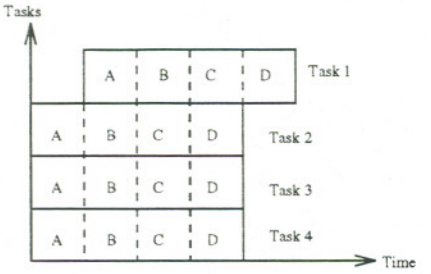

Fig. 2. Reservation table for pipelined operations in a data flow graph.

sub-functions of the i^{th} sub-operations *only*. The relationships between $F(\tau_1, \tau_2, ..., \tau_m, t)$, $F^i(\tau_1, \tau_2, ..., \tau_m, t)$, $f^i(\tau_j, t)$ and $f(\tau_j, t)$ can be demonstrated with the following equation.

$$F(\tau_1, \tau_2, ..., \tau_m, t) = \sum_{j=1}^{m} f(\tau_j, t) = \sum_{i=1}^{s} F^i(\tau_1, \tau_2, ..., \tau_m, t) = \sum_{i=1}^{s} \sum_{j=1}^{m} f^i(\tau_j, t)$$

Let n_p^i be the minimum number of i^{th} sub-units of a functional unit needed during any unit time interval. Then the minimum number of pipelined functional units n_p needed to complete a structural pipelined data flow graph is

$$n_p = max\{n_p^1, n_p^2, ..., n_p^s\}.$$

Since the computation of n_p^i is based on the non-pipelined theory, we define the *minimum load* similar to non-pipelined case for the k^{th} sub-operation of operation j in the interval $[t_1, t_2]$ as

$$\phi_j^k(t_1, t_2) = \min_{\forall \tau_j} \int_{t_1}^{t_2} f^k(\tau_j, t) dt. \tag{5}$$

The computation of ϕ_j^k is the same as for the non-pipelined case except the delay of each sub-operation is 1. From Lemma 1, we have

$$\phi_j^k(t_1, t_2) = min\{\overline{n_j^k(t_1, t_2)}, \underline{n_j^k(t_1, t_2)}\},$$

where

$$\overline{n_j^k(t_1, t_2)} = |[\overline{\tau_j} - s + k - 1, \overline{\tau_j} - s + k] \cap [t_1, t_2]|$$

and

$$\underline{n_j^k(t_1, t_2)} = |[\underline{\tau_j} - s + k - 1, \underline{\tau_j} - s + k] \cap [t_1, t_2]|.$$

$\overline{n_j^k(t_1, t_2)}$ is the overlap between the ASAP active function of the k^{th} sub-operation and $[t_1, t_2]$, and $\underline{n_j^k(t_1, t_2)}$ is the overlap between the ALAP active function of the k^{th} sub-operation and $[t_1, t_2]$.

The computation of the lower bound for structural pipelined data flow graphs is stated in the following theorem.

Theorem 3 *Given a data flow graph G and m operations which can be structurally pipelined into s sub-units, the minimum number of pipelined functional units (with s sub-units) is n_p. The lower bound for n_p is*

$$n_{pL} = \lceil \max_{\forall [t_1, t_2] \subseteq [0,T]} \frac{\max_{\forall k \in [1,s]} \sum_{j=1}^{m} \phi_j^k(t_1, t_2)}{(t_2 - t_1)} \rceil,$$

where $\phi_j^k(t_1, t_2)$ is defined in Eq. (5).

Proof: Because the computation of n_p^k is based on the theory for the non-pipelined case, according to the definition of n_p^k and Theorem 2

$$\sum_{j=1}^{m} \phi_j^k(t_1, t_2) \leq n_p^k (t_2 - t_1) \quad \forall k,$$

and the lower bound of n_p^k is,

$$n_{pL}^k = \lceil \max_{\forall [t_1, t_2] \subseteq [0,T]} \frac{\sum_{j=1}^{m} \phi_j^k(t_1, t_2)}{(t_2 - t_1)} \rceil.$$

On the other hand, from the definition of n_p,

$$\begin{aligned} n_{pL} &= \lceil \max_{\forall k \in [1,s]} n_{pL}^k \rceil = \lceil \max_{\forall k \in [1,s]} \max_{\forall [t_1, t_2] \subseteq [0,T]} \frac{\sum_{j=1}^{m} \phi_j^k(t_1, t_2)}{(t_2 - t_1)} \rceil \\ &= \lceil \max_{\forall [t_1, t_2] \subseteq [0,T]} \frac{\max_{\forall k \in [1,s]} \sum_{j=1}^{m} \phi_j^k(t_1, t_2)}{(t_2 - t_1)} \rceil. \end{aligned}$$

■

Similar to the non-pipelined case there are $T(T+1)/2$ different time intervals to be considered. By computing the lower bound, we also obtain an upper bound since the ASAP and ALAP schedules are computed. The upper bound for the structural pipelined case is

$$n_{pU} = min\{\overline{n_{pU}}, \underline{n_{pU}}\},$$

where

$$\overline{n_{pU}} = \max_{\forall t \in [0,T]} \max_{\forall k \in [1,s]} \{F^k(\overline{\tau_1}, ..., \overline{\tau_m}, t)\} \text{ and } \underline{n_{pU}} = \max_{\forall t \in [0,T]} \max_{\forall k \in [1,s]} \{F^k(\underline{\tau_1}, ..., \underline{\tau_m}, t)\}.$$

$F^k(\underline{\tau_1}, ..., \underline{\tau_m}, t)$ is defined in Eq. (4).

Example

In Fig. 1(b) the execution of the multiplication requires 2 units of time and the total completion time is $T = 9$. If a 2-stage pipelined multiplier is used, the multiplication operation is divided into 2 sub-operations with $k = 1, 2$. After computing $\sum_{j=1}^{10} \phi_j^1(t_1, t_2)$ for sub-operation 1 and $\sum_{j=1}^{10} \phi_j^2(t_1, t_2)$ for sub-operation 2 of the multiplication for all time intervals, it is determined that the maximum $\frac{\sum_{j=1}^{10} \phi_j^k(t_1,t_2)}{(t_2-t_1)}$ is at $[t_1, t_2]$=[0,1] when $k = 1$. Therefore, the lower bound on the number of pipelined multipliers is

$$n_{pL} = \lceil \frac{\sum_{j=1}^{10} \phi_j^1(0,1)}{(1-0)} \rceil = \lceil \frac{\phi_1^1(0,1) + \phi_7^1(0,1) + \phi_9^1(0,1)}{1-0} \rceil = \lceil \frac{1+0+0}{1-0} \rceil = 1.$$

The upper bound is derived from the ALAP schedule.

$$n_{pU} = min\{\overline{n_{pU}}, \underline{n_{pU}}\} = \underline{n_{pU}} = 2, \quad at \quad t = 3$$

Note that the time intervals that give the maximum may not be unique. Any one of them will suffice for the solution.

C A Simplified Approach for Structural Pipelined Operations

The algorithm presented in the last section involves keeping track of which sub-operation of an operation has the most frequent appearance. In order to avoid this kind of situation we also present a simpler algorithm to calculate the lower bound. In this simple alternative approach we suggest that all sub-operations occurring in the time interval $[t_1, t_2]$ be counted indiscriminately, i.e., we treat an s-stage pipelined operation as an s stage multi-cycle operation.

Given a data flow graph with operations that can be structurally pipelined, we examine the total number of *sub-operations* in the time interval $[t_1, t_2]$ *indiscriminately.* The relationship between the lower bound on the number of structural pipelined functional units n_{pL} and the total number of sub-operations is stated in the following Lemma.

Lemma 4 *Given an arbitrary number r of sub-operations active in a time interval $[t_1, t_2]$, a lower bound n_{pL} on the number of s-stage pipelined functional units n_p required to complete all sub-operations in $[t_1, t_2]$ is*

$$n_{pL} = \frac{r}{(t_2 - t_1) \times s} \leq n_p.$$

Proof: Each structural pipelined functional unit active during the time interval $[t_1, t_2]$ can execute up to $s \times (t_2 - t_1)$ sub-operations. These $s \times (t_2 - t_1)$ sub-operations must be in certain positions in order to be covered by one structural pipelined functional unit. If these $s \times (t_2 - t_1)$ sub-operations are in arbitrary positions then more structural pipelined functional units may be needed. Therefore, for r arbitrary sub-operations, a lower bound n_{pL} on the number n_p of pipelined functional units needed is

$$n_{pL} = \frac{r}{(t_2 - t_1) \times s} \leq n_p.$$

■

Note that we assume each sub-operation has a delay of one unit. r also represents the total area occupied by the sub-operations in the interval $[t_1, t_2]$.

Based on Lemma 4 a simpler approach to compute the lower bound on the minimum number of pipelined functional units required to complete a structurally pipelined data flow graph is stated in the following theorem.

Theorem 5 *Given a data flow graph G and m operations which can be structurally pipelined into s sub-units, the minimum number of structural pipelined functional units (with s sub-units) required is n_p. The lower bound for n_p is*

$$n_{pL} = \lceil \max_{\forall [t_1, t_2] \subseteq [0, T]} \frac{\sum_{j=1}^{m} \phi_j(t_1, t_2)}{s \times (t_2 - t_1)} \rceil,$$

where $\phi_j(t_1, t_2)$ is defined in Eq. (1).

Proof: Let r^* be the total number (area) of sub-operations of an optimal schedule $(\tau_1^*, \tau_2^*, ..., \tau_m^*)$ in the interval $[t_1, t_2]$. Since $\sum_{j=1}^{m} \phi_j(t_1, t_2)$ is the sum of the minimum load of all operations, from the proofs of Theorem 2 and Lemma 4,

$$\begin{aligned} \sum_{j=1}^{m} \phi_j(t_1, t_2) &\leq \int_{t_1}^{t_2} F(\tau_1^*, \tau_2^*, ..., \tau_m^*, t) dt \\ = r^* &\leq n_p \times (t_2 - t_1) \times s. \end{aligned}$$

Hence, the lower bound is

$$n_{pL} = \lceil \max_{\forall [t_1, t_2] \subseteq [0, T]} \frac{\sum_{j=1}^{m} \phi_j(t_1, t_2)}{s \times (t_2 - t_1)} \rceil.$$

■

This theorem concludes that the same algorithm used for non-pipelined functional units can be used to derive the solution of the minimum number of functional units for structural pipelined operations. The only modification is that $\sum_{j=1}^{m} \phi_j(t_1, t_2)$ is divided by $(t_2 - t_1) \times s$ instead of $t_2 - t_1$. Experiments show this lower bound is nearly as tight as the bounds from Theorem 3, but avoids keeping track of the individual sub-operations and thus requires less computation time.

As a by-product of the lower bound computation, the upper bound for the simplified method is

$$n_{pU} = min\{\overline{n_{pU}}, \underline{n_{pU}}\},$$

where

$$\overline{n_{pU}} = \max_{\forall t \in [0,T]} F(\overline{\tau_1}, \overline{\tau_2}, ..., \overline{\tau_m}, t) \text{ and } \underline{n_{pU}} = \max_{\forall t \in [0,T]} F(\underline{\tau_1}, \underline{\tau_2}, ..., \underline{\tau_m}, t).$$

$F(\overline{\tau_1}, \overline{\tau_2}, ..., \overline{\tau_m}, t)$ and $F(\underline{\tau_1}, \underline{\tau_2}, ..., \underline{\tau_m}, t)$ are the activity functions for non-pipelined operations instead of pipelined operations.

Example

In Fig. 1(b), $s = 2$ and $T = 9$. $[t_1, t_2] = [0, 4]$ is determined to generate the maximum $\frac{\sum_{j=1}^{10} \phi_j(t_1,t_2)}{s \times (t_2 - t_1)}$, and the lower bound is

$$n_{pL} = \lceil \frac{\sum_{j=1}^{10} \phi_j(0,4)}{(4-0) \times 2} \rceil = \lceil \frac{\phi_1(0,4) + \phi_3(0,4) + \phi_7(0,4) + \phi_9(0,4)}{(4-0) \times 2} \rceil = \lceil \frac{2+2+2+0}{8} \rceil = 1.$$

The upper bound is derived from the ALAP schedule

$$n_{pU} = min\{\overline{n_{pU}}, \underline{n_{pU}}\} = \underline{n_{pU}} = 2, \quad at\ t = 3.$$

These are the same results that are obtained using Theorem 3.

The relationship between the bounds of Theorem 3 and those of Theorem 5 are stated in the following theorem.

Theorem 6 *The bounds of Theorem 3 are tighter than those of Theorem 5.*

Proof:

1. Lower bound

   Let n_{pL1} and n_{pL2} be the lower bounds derived from Theorems 3 and 5, respectively. Given any time interval $[t_1, t_2]$, let $\psi_1(t_1, t_2)$ be the count of the *maximum* occurence (load) in $[t_1, t_2]$ of a *single* sub-operation as in Theorem 3, and let $\psi_2(t_1, t_2)$ be the count of the *total* occurences (load) of *all* sub-operations in $[t_1, t_2]$ as in Theorem 5. That is,

$$\psi_1(t_1, t_2) = \max_{\forall k \in [1,s]} \sum_{j=1}^{m} \phi_j^k(t_1, t_2)$$

   and

$$\psi_2(t_1, t_2) = \sum_{j=1}^{m} \sum_{k=1}^{s} \phi_j^k(t_1, t_2) \leq \psi_1(t_1, t_2) \times s.$$

   This means $s \times \psi_1(t_1, t_2)$ is the maximum of all sub-operations that can possibly occur in $[t_1, t_2]$. Thus, from

$$s \times \psi_1(t_1, t_2) \geq \psi_2(t_1, t_2)$$

   we have

$$\frac{\psi_1(t_1, t_2)}{(t_2 - t_1)} \geq \frac{\psi_2(t_1, t_2)}{s(t_2 - t_1)},$$

   and from Theorems 3 and 5,

$$n_{pL1} \geq n_{pL2}.$$

2. Upper bound

   Since

$$max\{F^1(\tau_1, \tau_2, ..., \tau_m, t), ..., F^s(\tau_1, \tau_2, ..., \tau_m, t)\} \leq F(\tau_1, \tau_2, ..., \tau_m, t),$$

$$\overline{n_{pU1}} \leq \overline{n_{pU2}} \text{ and } \underline{n_{pU1}} \leq \underline{n_{pU2}}.$$

   Hence,

$$n_{pU1} \leq n_{pU2}.$$

■

D *Functional Pipelined Operations*

A behavioral description often consists of loop statements and the loop execution usually dominates the total execution time. Optimizing the execution of the loop body is critical to the performance of the design.

Scheduling a loop body is quite different from scheduling a straight-line code segment, since parallelism beyond the iteration boundaries can be exploited by executing several iterations of the loop concurrently. The type of pipelining where the entire data flow graph is pipelined is called functional pipelining. The objective of this section is to find a lower bound on the minimum number of functional units needed given the task Initiation Latency (IL), the loop body data flow graph and the iteration time (completion time) T.

To illustrate the idea of functional pipelining we use the same example as in Fig. 1(a), except the data flow graph is viewed as the loop body data flow graph in a functional pipelining situation. In the example successive iterations begin their executions two control steps apart.

In functional pipelining different initiations of tasks share the same control steps. If we look at the loop body data flow graph, these operations are located IL control steps apart in the data flow graph. In other words, operations in a loop body data flow graph that are IL control steps apart will be executed in the same control step when functional pipelining is utilized. In our example $IL = 2$, therefore after every two units of time a new task is introduced, and operations that are two control steps apart will be executed in the same control step in the final schedule.

A partition is formed on the operations of the loop body data flow graph. Operations that are IL control steps apart belong to the same partition, and there are IL different partitions. The active function of partition i is $\sum_{i=0}^{\lceil \frac{T}{IL} \rceil} F(\tau_1^*, \tau_2^*, ..., \tau_m^*, t + i \times IL)$, where $0 \leq t < IL$. The active function is the same as the previous cases, except it reflects the consideration of all the operations in a partition. Formally, given the task Initiation Latency IL and iteration time T, we are to find n_{fp} and an optimal schedule $(\tau_1^*, \tau_2^*, ..., \tau_m^*)$ such that

$$\begin{aligned} n_{fp} &= \max_{\forall t \in [0,\mathrm{IL}]} \sum_{i=0}^{\lceil \frac{T}{IL} \rceil} F(\tau_1^*, \tau_2^*, ..., \tau_m^*, t + i \times IL) \\ &= \min_{\forall (\tau_1, \tau_2, ..., \tau_m) \in D} \max_{\forall t \in [0,\mathrm{IL}]} \sum_{i=0}^{\lceil \frac{T}{IL} \rceil} F(\tau_1, \tau_2, ..., \tau_m, t + i \times IL). \end{aligned} \tag{6}$$

In this formulation an entire partition is taken into consideration; therefore, instead of calculating the minimum load in a single time interval, we must calculate the minimum load for all the time intervals IL units apart. Φ_j is defined as the minimum overlap between the active function $f(\tau_j, t)$ of operation j and the time intervals $\cup_{i=0}^{\lceil \frac{T}{IL} \rceil} [t_1 + i \times IL, t_2 + i \times IL]$,

```
algorithm Φ_j(t_1, t_2, τ_j, τ̄_j)
    Φ_j = maximum integer;
    for each position of τ_j between τ_j and τ_j + IL do
        SUM = 0;
        for i = 0 to ⌈T/IL⌉ do
            SUM += |[t_1 + i × IL, t_2 + i × IL] ∩ [τ_j − s, τ_j]|;
        end_for
        Φ_j = min(Φ_j, SUM);
    end_for
    return(Φ_j);
end_algorithm
```

Fig. 3. Algorithm to compute ϕ_j.

and is given by

$$\Phi_j(t_1, t_2) = \min_{\forall \tau_j} \sum_{i=0}^{\lceil \frac{T}{IL} \rceil} \int_{t_1 + i \times IL}^{t_2 + i \times IL} f(\tau_j, t) dt. \tag{7}$$

The computation procedure of Φ_j is illustrated in Fig. 3. Consequently, the lower bound on the number of functional units required to complete the functional pipelined data flow graph given the initiation latency IL and iteration time T is stated in the following theorem.

Theorem 7 *Given a functional pipelined data flow graph, the task initiation latency IL and the loop body data flow graph iteration time T, the minimum number of functional units required to complete the functional pipelined data flow graph is n_{fp}. The lower bound of n_{fp} is*

$$n_{fpL} = \lceil \max_{\forall [t_1, t_2] \subseteq [0, IL]} \frac{\sum_{j=1}^{m} \Phi_j(t_1, t_2)}{t_2 - t_1} \rceil.$$

Proof: If τ_j^* represents the completion time for operation j in an optimal schedule, then from the definition of Φ_j

$$\Phi_j(t_1, t_2) = \min_{\forall \tau_j} \sum_{i=0}^{\lceil \frac{T}{IL} \rceil} \int_{t_1 + i \times IL}^{t_2 + i \times IL} f(\tau_j, t) dt \leq \sum_{i=0}^{\lceil \frac{T}{IL} \rceil} \int_{t_1 + i \times IL}^{t_2 + i \times IL} f(\tau_j^*, t) dt.$$

Thus,

$$\begin{aligned} \sum_{j=1}^{m} \Phi_j(t_1, t_2) &\leq \sum_{j=1}^{m} \sum_{i=0}^{\lceil \frac{T}{IL} \rceil} \int_{t_1 + i \times IL}^{t_2 + i \times IL} f(\tau_j^*, t) dt \\ = \sum_{i=0}^{\lceil \frac{T}{IL} \rceil} \int_{t_1 + i \times IL}^{t_2 + i \times IL} \sum_{j=1}^{m} f(\tau_j^*, t) dt &= \sum_{i=0}^{\lceil \frac{T}{IL} \rceil} \int_{t_1 + i \times IL}^{t_2 + i \times IL} F(\tau_1^*, \tau_2^*, ..., \tau_m^*, t) dt. \end{aligned}$$

Using a variable transformation with $t' = t - IL \times i$, we have $dt' = dt$ and $t = t' + IL \times i$. Therefore, from Eq. (6),

$$\begin{aligned}\sum_{j=1}^{m} \Phi_j(t_1, t_2) &\leq \sum_{i=0}^{\lceil \frac{T}{IL} \rceil} \int_{t_1}^{t_2} F(\tau_1^*, \tau_2^*, ..., \tau_m^*, t + IL \times i)dt \\ &= \int_{t_1}^{t_2} \sum_{i=0}^{\lceil \frac{T}{IL} \rceil} F(\tau_1^*, \tau_2^*, ..., \tau_m^*, t + IL \times i)dt \\ &\leq n_{fp}(t_2 - t_1).\end{aligned}$$

Hence, the lower bound is

$$n_{fpL} = \lceil \max_{\forall [t_1, t_2] \subseteq [0, IL]} \frac{\sum_{j=1}^{m} \Phi_j(t_1, t_2)}{t_2 - t_1} \rceil.$$

■

Since the total number of partitions is IL, only time intervals $[t_1, t_2]$ that are subsets of $[0, IL]$ need to be considered. The number of sub-intervals in the time interval $[0, IL]$ is $IL(IL+1)/2$; therefore there are $IL(IL+1)/2$ different $\sum_{j=1}^{m} \Phi_j(t_1, t_2)$'s to compute in order to determine n_{fpL}.

The computation of the upper bound n_{fpU} is accomplished at the same time the lower bound is computed. An upper bound on the number of functional units required to complete the data flow graph given the initiation latency IL and iteration time T is given by

$$n_{fpU} = min\{\overline{n_{fpU}}, \underline{n_{fpU}}\},$$

where

$$\overline{n_{fpU}} = \max_{\forall t \in [0, IL]} \sum_{i=0}^{\lceil \frac{T}{IL} \rceil} F(\overline{\tau_1}, \overline{\tau_2}, ..., \overline{\tau_m}, t + i \times IL)$$

and

$$\underline{n_{fpU}} = \max_{\forall t \in [0, IL]} \sum_{i=0}^{\lceil \frac{T}{IL} \rceil} F(\underline{\tau_1}, \underline{\tau_2}, ..., \underline{\tau_m}, t + i \times IL).$$

Example

In our example the task initiation latency $IL = 2$. For multiplication the maximum $\frac{\sum_{j=1}^{10} \Phi_j(t_1, t_2)}{t_2 - t_1}$ occurs at $[t_1, t_2] = [0, 1]$. The following $\Phi_j(0, 1)$'s are needed[5] to obtain the lower bound.

$$\Phi_1(0,1) = \Phi_3(0,1) = \Phi_5(0,1) = \Phi_6(0,1) = \Phi_7(0,1) = \Phi_9(0,1) = 1$$

[5] All possible time intervals need to be computed in order to determine the maximum.

Thus, the lower bound is

$$n_{fpL} = \lceil \frac{\sum_{j=1}^{10} \Phi_j(0,1)}{1-0} \rceil = \lceil \frac{6}{1} \rceil = 6.$$

Since $\underline{n_{fpU}} = \overline{n_{fpU}} = 6$, the upper bound is

$$n_{fpU} = min\{\overline{n_{fpU}}, \underline{n_{fpU}}\} = min\{6,6\} = 6.$$

Similarly, for addition

$$n_{fpL} = 2, \quad n_{fpU} = min\{\overline{n_{fpU}}, \underline{n_{fpU}}\} = min\{4,3\} = 3.$$

III SCHEDULING

In the previous section we derived the lower and upper bounds on the number of functional units required to execute the operations of a data flow graph in time T, but the problem of assigning the operations to functional units after we obtain the bounds is not addressed. There is no guarantee that a feasible schedule exists for a design which contains hardware units equal in number to the lower bound. On the other hand, the ASAP and ALAP are two feasible schedules, but the number of functional units required by them may not be close to optimal.

Since the resource constrained scheduling problem is NP-hard, only an exhaustive search strategy can be guaranteed to produce an optimal schedule. However, the lower bound is of significant importance even if a feasible schedule does not exist, because it can be used with a branch-and-bound algorithm to solve the scheduling problem *optimally*. The lower bound is crucial if such a technique is employed, since the execution time of the branch-and-bound algorithm depends on the quality of the lower bound. From a practical point of view it is not always necessary to generate an optimal solution if the solution obtained is close to an optimal one [5]. Therefore, a good heuristic is in more demand in a real world design environment.

Our objective is to employ a simple yet effective algorithm to generate a feasible schedule for the data flow graph after we obtain the lower and upper bounds. The scheduling algorithm should assume that the number of resources is limited by the bounds, and it must produce a feasible schedule for the data flow graph. The new and feasible schedule produced by the scheduling algorithm will replace both the ASAP and ALAP schedules, and the new number of functional units required to schedule the data flow graph feasibly will become the new

upper bound. The general procedure for finding a feasible schedule and improving the upper bound is outlined below.

1. Assign $n = n_L$ for all types of operations initially, where n_L is the lower bound.
2. Use a resource constrained scheduling algorithm to schedule the data flow graph. The scheduling algorithm chosen depends on the type of the data flow graph.
3. If the schedule meets the time constraint T, then replace the upper bound by the current n (i.e., we found a feasible schedule better than the ASAP and ALAP schedules) and terminate.
4. If the schedule does *not* meet the constraint T (i.e., extra control steps are needed to obtain a feasible schedule), the number of functional units needed is increased by 1 for those types of operations which need extra control steps to be completed; if $n < n_U$, go to step 2; otherwise, if $n = n_U$ the ASAP or ALAP schedule (whichever was used for deriving n_U) is the best schedule we can find.

Experimental results indicate that n_L is a tight lower bound, and in many cases it is actually the *optimal*. Therefore, assuming $n = n_L$ from the start is an educated choice. If the current schedule meets the time constraint, then the current n is a new upper bound for the optimal solution. In the case $n_L = n_U$ the algorithm has determined that n_L is the optimal solution. If the current n cannot satisfy the time constraint T, then the numbers of certain types of functional units which require extra control steps are increased by 1 as long as $n < n_U$. The data flow graph is then rescheduled, and this process is repeated until a feasible schedule with completion time T is obtained.

Note that the upper bound will never exceed n_U, since the best known schedule could be either the ASAP or ALAP schedules from which n_U is derived. Test cases indicate that the procedure will generate an optimal or near optimal schedule under time constraints as well as an improved upper bound on the minimum number of functional units required. The detailed scheduling algorithms for the different types of data flow graphs will be discussed in the following sub-sections.

A Scheduling For Non-pipelined DFGs

For non-pipelined and structural pipelined operations in a data flow graph the "Cut The Longest Queue" (CTLQ) [12] scheduling algorithm is chosen. The name "Cut The Longest Queue" is first introduced in Hu's research [12] on scheduling a rooted tree of operations

where the queue (or priority function) refers to the distance of an operation to the exit node. The original CTLQ is a simple but very effective method. The merits of the CTLQ algorithm are determined based on the results of extensive empirical studies [13]. Empirical studies compared force-directed scheduling (FDS) [2], force-directed list scheduling (FDLS) [2] and MAHA [3] over one hundred graphs, some of which are randomly generated. Ranking the methods in terms of the number of non-inferior schedules produced, CTLQ scheduling with complexity $O(mlog(m))$ produced the maximum number of non-inferior schedules, followed by FDS, FDLS, and finally MAHA.

The basic CTLQ scheduling algorithm first assigns functional units to operations (nodes) which are farthest from the exit node, then removes those nodes that have been assigned and repeats the process until all the nodes have been scheduled. For example, in Fig. 4 the number of operations is 5 and the number of functional units is 2. Operations 1 and 2 are scheduled in control step one since operations 1 and 2 are farther from the exit node than 3.

In order to schedule several different types of operations in a data flow graph, a slightly modified queue is more efficient [21] than the original queue. The modified queue is a *pseudo-ALAP* value of an operation and is derived from the control steps assigned by using backward ASAP list scheduling with a subset of the resource constraints. The algorithm is briefly described as follows.

1. Perform backward list scheduling with the resource constraint of type k. For other operation types assume unlimited resources. Let the *pseudo-ALAP* schedule of operations be the control step assignment under such a schedule.
2. Perform forward CTLQ scheduling using *pseudo-ALAP* as the queue. All the resource constraints on the number of functional units are used in the forward scheduling.
3. Repeat steps 1 and 2 for each operation type and find the best schedule.

The modified CTLQ algorithm outperforms the simple CTLQ algorithm in the area of resolving the congestion of operations. The modified CTLQ algorithm has the speed of the simple CTLQ algorithm yet is effective for node-distribution when operations are congested.

In the scheduling of non-pipelined operations most of the operations in a data flow graph are *multi-cycle operations*; therefore, we must distribute them into *consecutive* control steps such that the *same* hardware is dedicated to a particular operation[6]. For example, in Fig. 1(a) the lower bounds for multiplication and addition are 2 and 1, respectively. The corresponding

[6]In some other architectures (e.g., pipelining) hardware sub-units can be shared by different operations.

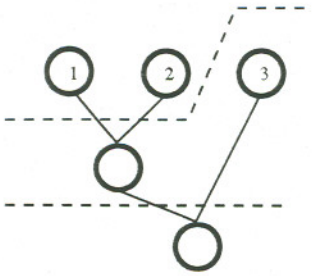

Fig. 4. A data flow graph scheduled by the CTLQ algorithm.

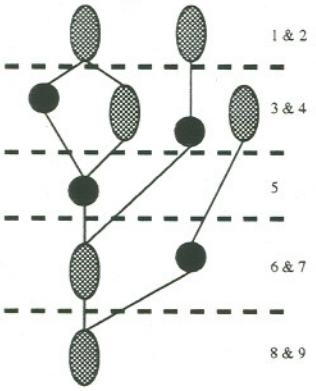

Fig. 5. A schedule for the data flow graph in Fig. 1 assuming non-pipelined functional units.

schedule is shown in Fig. 5. Since a feasible schedule requiring 2 multipliers and 1 adder is produced by CTLQ, the lower bound is the optimal solution.

B Scheduling For Structural Pipelined DFGs

The same CTLQ scheduling algorithm with some modifications is used for scheduling structural pipelined DFGs. The essential differences are

- Sub-operations whose predecessor sub-operations have already been scheduled should be scheduled immediately onto *available* sub-units of *any* pipelined functional unit
- An operation has to follow precedence between sub-operations, but functional sub-units do not

Consider the example in Fig. 1. If the multiplication which takes 2 time units is pipelined, we will need fewer multipliers. In Fig. 1 $s = 2$ for the multiplications and from the previous discussion in our example on the lower bound, we have $n_L = 1$. Thus, we start scheduling with the assumption that the minimum number of multipliers is 1. The number of adders remains unchanged since it is not pipelined. The scheduling result is shown in Fig. 6. The

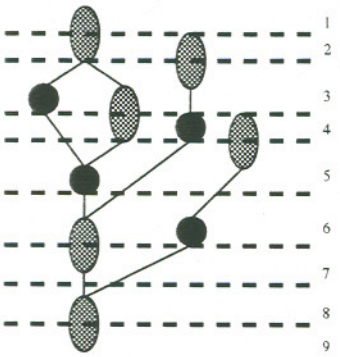

Fig. 6. A schedule for the data flow graph in Fig. 1 assuming pipelined multipliers.

completion time T is still 9, but only 1 multiplier is needed. Since we can successfully schedule the data flow graph to complete in time T using one pipelined multiplier and one non-pipelined adder, the lower bounds are our optimal solutions ($n_L = n_U$).

C Scheduling For Functional Pipelined DFGs

The scheduling algorithm for functional pipelined data flow graphs is more complicated than non-pipelined and structural pipelined, since the minimum number of functional units needed is the minimum of the sum of the number of functional units needed for multiple control steps. We can not simply use the "cut the longest queue" scheduling approach.

The objective of scheduling here is to make sure that all control steps that are IL control steps apart are considered together. Since normal scheduling is not capable in such cases, we present a scheduling algorithm tailored specifically to this problem.

Since the task initiation latency is IL, there are IL partitions among all operations starting from partition 0 to partition $IL - 1$. The operations are partitioned according to the control step in which they belong. Each partition contains operations that are IL control steps apart. For example, if $IL = 2$ then partition 1 contains operations in control steps 1, 3, 5, 7, ... By computing the lower bound, we have already determined the minimum number of functional units allowed n_{fp} (initially it is n_{fpL}) for each partition. The idea of scheduling is to distribute the operations as evenly as possible into each of the control steps in the same partition such that no more than n_{fL} operations exist in a partition and these operations can be successfully executed by n_{fp} functional units.

We propose an algorithm called *folded CTLQ scheduling* as illustrated in Fig. 7. The algorithm can be described as follows.

1. Operations with no freedom should be assigned to control steps immediately.
2. Schedule the operation with the smallest ALAP value to the earliest possible control step k, usually the earliest step is $\overline{\tau_j}$.
3. Check if the partition that step k belongs to has exceeded the total number of operations n allowed in it.
4. If it has not exceeded the preset hardware constraints n, then fix the schedule.
5. If it has exceeded the hardware constraints for the partition, then delay the schedule of the operation by one control step at a time until the assignment of the operation to a control step meets the hardware constraints.
6. After an operation has been fixed to a control step, adjust the active ranges of operations that are affected by such an assignment.
7. Delete the assigned operation from the list of operations which need to be scheduled.
8. Repeat the process until all operations are scheduled.

In our example the final schedule where a new task is introduced every 2 units of time is shown in Fig. 8(a), and the assignments of operations based on the algorithm in Fig. 7 are in Figs. 8(b) and (c), where the dark gray areas correspond to the final schedule of the operations. Six multipliers and two adders are required to successfully execute the functional pipelined data flow graph with $IL = 2$ and $T = 9$. In our example the lower bounds for the number of multipliers and adders are the same number required in the optimal schedule.

The scheduling algorithms introduced here for non-pipelined, structural pipelined and functional pipelined data flow graphs are heuristics aiming to finalize the assignment of operations to functional units, to measure the tightness of the lower bound and at the same time to improve the upper bound computed by the ASAP or ALAP schedules. Our methods are not bound to the scheduling techniques employed here, any resource constrained scheduling algorithm can be used.

IV Computational Complexity and Performance

A Complexity

The computational complexity consists of two parts: complexity of the scheduling algorithm and that of the computation of the lower bound. The worst case computational complexity of the scheduling algorithm is $O(mlog(m) + Elog(E) + nE)$ [13], where m is the

```
algorithm Folded_Schedule(n,T,IL,DFG)
O = {all operations};
for each operation j in O do
    if τ̄_j == τ_j
      schedule operation j into control step τ̄_j;
      O = O - {operation j};
    end_if
end_for
do while O is not empty
    find the operation j with the least ALAP value;
    schedule operation j to the earliest possible step k;
        do while the total number of operations in partition k mod IL > n
            increase control step k by one;
            schedule operation j to step k;
        end_do
    fix the schedule for operation j;
    O = O - {operation j};
    adjust the active ranges of operations that are
    affected by the assignment of operation j;
end_do
end_algorithm
```

Fig. 7. Scheduling algorithm for functional pipelined operations

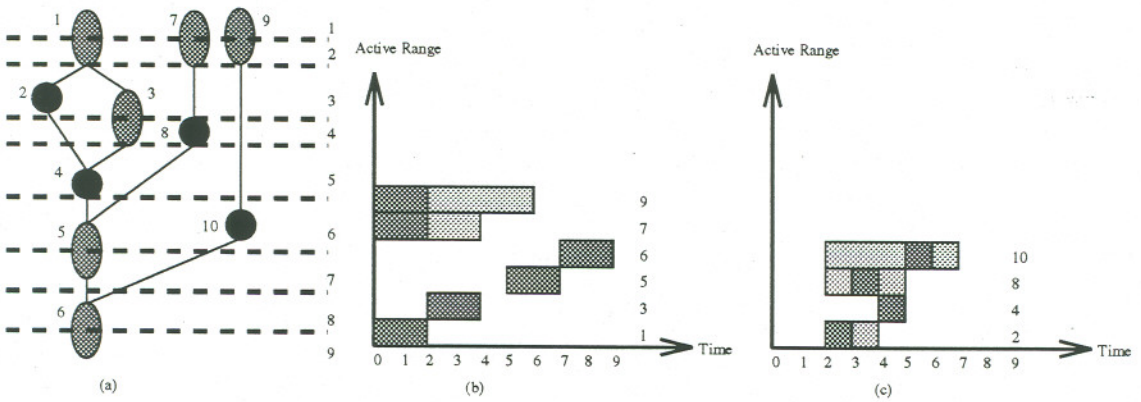

Fig. 8. (a) Schedule for the functional pipelined example, (b) the control step assignment for each multiplication, and (c) the control step assignment for each addition.

number of operations in the data flow graph, E is the number of edges denoting precedence constraints in the data flow graph and n is the maximum number of resources of any type. The worst case computational complexity of the lower bound computation is $O(T^2m)$, where T is the height of the graph and m is the total number of operations. The complexity is calculated as follows.

1. We have to go through $1 + 2 + ... + T = T(T + 1)/2 = (T^2 + T)/2$ time intervals to derive the lower and upper bounds.
2. At each step we may calculate up to m different $\phi_j(t_1, t_2)$'s in the worst case for the algorithms in Theorems 2 and 5, so the complexity is $O(T^2m)$.
3. For Theorem 3, we may calculate at most i $\phi_j^k(t_1, t_2)$'s for the sub-operations in the interval i and there may be m operations at each step. Thus the total calculation of $\phi_j^k(t_1, t_2)$'s at each step requires $i \times m$ units of time. The total number of constant time computations is $\sum_{i=1}^{T}(i \times m) \times (T - i + 1) = O(T^2m)$. Therefore, the worst case complexity is $O(T^2m)$.
4. For Theorem 7 we have to go through $IL(IL + 1)/2$ sub-intervals and the worst case complexity of computing Φ_j is $\lceil \frac{T}{IL} \rceil \times T$, so the complexity is $T^2(IL + 1)m = O(T^2m)$.

The worst case performance is a conservative estimate, because normally at each step the number of computations is much less than m. The dominant term is the computational complexity of the lower bound.

The computational complexity of force-directed scheduling is $O(Tm^3)$ [2], and in [5] the complexity is $O(Tm^2log(m))$, while the computational complexity in our algorithm is $O(T^2m)$. Since T is typically of the same order as m, our algorithm is of less complexity and is the only one that can generate a lower bound. Since we consider more time intervals, our bounds are also much tighter than that of [14]. The lower bound given in [14] is $\lceil \frac{m}{T} \rceil$ and is computed by considering only one time interval (namely, $[0, T]$) which is considered in our technique.

B Performance

Our experiments consist of two parts. We tested some large randomly generated graphs followed by some tests on benchmark examples to compare with existing algorithms.

B1 Randomly Generated Examples

The use of large randomly generated graphs for testing our technique is significant, since most other known performance constrained scheduling algorithms only tested a few benchmark examples. Our test examples are randomly generated similar to the test cases used in [13]. The test graphs are randomly generated with 50 operations each and can have up to 5 different types of operations. A total of 25 graphs are considered and they are evenly distributed by the number of different types of operations. All possible time/functional unit trade-offs are tested for all 25 graphs. The difference between the lower and upper bounds is either 0 or 1 for 100% of the test cases. Some of the differences are due to the scheduling algorithm, since it becomes increasingly difficult to schedule when the number of different types of operations increases. Nevertheless, for a simple algorithm like "cut the longest queue" the results are excellent. Due to the limitations in the length of this paper only some of the results are presented in Tables II to VIII. The results which are not shown here are of the same quality. Each table contains the number of edges of the graph, the delay and count of each type of operation, the time constraints, the lower and upper bounds and the difference between the lower and upper bounds.

For test graphs with one type of operation our lower bound is the optimal solution for *all* the test cases as shown in Table II. For test cases with multiple types of operations, the number of functional units of *one type* has to be increased no more than *once* to obtain a feasible schedule as indicated in Tables III to VI. The structural pipelined cases are shown in Table VII in which the adder is a pipelined functional unit. Shown in Table VIII are the results of a comparison of the lower bounds using Theorem 3 and Theorem 5. The small differences between the lower and upper bounds in our examples indicate that our bounds are very tight and the scheduling results are very good.

B2 Benchmark Examples

In order to compare our algorithm with existing algorithms we also use the same benchmark examples from the literature. The results are compared to HAL [2] and [5], two of the most notable scheduling techniques. Results in Table IX are for the differential equation example from [2]. We compare both the non-pipelined and the pipelined cases for $T = 6, 7$. Results in Table X are for a fifth order elliptic wave-filter adopted from [16] which has been selected as a benchmark test example for high-level synthesis algorithms. Results in Table XI are for the benchmark examples used in testing the functional pipelined

scheduling technique. They are a 16-point digital FIR filter borrowed from [17], and a fifth order elliptic filter from [16]. The results shown in Tables IX to XI indicate our algorithm produces optimal results for these benchmark examples with less complexity compared to existing scheduling techniques, and is the only one that can measure the optimality using a lower bound.

V Conclusion and Future Work

In this paper we have presented a new algorithm for the problem of performance constrained scheduling in data path synthesis which utilizes a lower bound on the number of functional units and a simple resource constrained scheduling technique. The lower bound is critical when the number of different operation types is large or the data flow graph is large. Obviously the results obtained here are not limited to the technique used in this paper. Since the bounds are very tight (shown by experiments and comparison to upper bounds), the lower and upper bounds can be used to develop an efficient branch-and-bound algorithm for the performance constrained problem. Moreover, the lower bounds can be used to evaluate the quality of heuristic algorithms for the performance constrained scheduling problem. Without the lower bound only a relative comparison between heuristics is possible.

The algorithm tends to a global optimization since it is not an iterative approach by nature, and only has a complexity of $O(T^2m)$. The algorithm achieves excellent results as well as being the only one so far to provide a *tight* lower bound while maintaining low complexity compared to other scheduling algorithms. The algorithm has the advantage of well-defined bounds for optimal solutions such that it can effectively guide the design process and also provide a means to measure the quality of the solution while other methods can not.

It is unknown whether or not a feasible schedule exists for the lower bound; however, since the bounds are tight, the adjustment required to find a feasible schedule is usually minimal. Because the quality of our technique depends on the resource constrained scheduling algorithm employed, future work may include the development of an improved heuristic for the resource constrained scheduling problem.

References

[1] D.D. Gajski, N.D. Dutt and B.M. Pangrle, "Silicon Compilation," In *Proc. IEEE 1986 Custom Integrated Circuits Conf.*, pp. 102-110, May 1986.

Table II. Results for data flow graphs with 1 type of operation.

Edges	Delay/Count For (+)	Time	Lower Bound (+)	Upper Bound (+)	$n_U - n_L$ (+)
92	1/50	T_c=8	7	7	0
		T=13	4	4	0
		T=25	2	2	0
89	1/50	T_c=8	7	7	0
		T=10	5	5	0
		T=17	3	3	0
79	1/50	T_c=8	6	6	0
		T=17	3	3	0
		T=50	1	1	0
90	1/50	T_c=7	8	8	0
		T=13	4	4	0
		T=25	2	2	0
90	1/50	T_c=8	7	7	0
		T=17	3	3	0
		T=25	2	2	0

Table III. Results for data flow graphs with 2 types of operations.

Edges	Delay/Count		T_c	Lower Bound		Upper Bound		$n_U - n_L$	
	(+)	(-)		+	-	+	-	+	-
90	9/28	14/22	88	4	4	4	5	0	1
97	7/25	2/25	37	6	2	6	2	0	0
97	2/23	7/27	41	2	5	2	6	0	1
90	1/25	2/25	15	2	4	2	4	0	0
91	1/23	2/27	12	2	5	2	5	0	0

Table IV. Results for data flow graphs with 3 types of operations.

Edges	Delay/Count			T_c	Lower Bound			Upper Bound			$n_U - n_L$		
	(+)	(-)	(*)		+	-	*	+	-	*	+	-	*
91	3/10	7/16	1/24	31	2	5	1	2	5	1	0	0	0
95	10/11	3/26	2/13	33	4	3	1	4	4	1	0	1	0
82	5/22	13/16	11/12	68	2	4	3	3	4	3	1	0	0
79	6/24	11/16	9/10	87	2	3	2	2	3	2	0	0	0
91	1/20	2/10	2/20	16	2	3	3	2	3	3	0	0	0

Table V. Results for data flow graphs with 4 types of operations.

Edges	Delay/Count				T_c	Lower Bound				Upper Bound				$n_U - n_L$			
	(+)	(-)	(*)	(/)		+	-	*	/	+	-	*	/	+	-	*	/
83	8/19	9/17	3/8	12/6	56	4	4	1	2	4	4	1	2	0	0	0	0
83	5/9	4/12	1/14	3/15	21	4	4	1	3	4	4	1	3	0	0	0	0
81	10/13	7/9	12/14	14/14	82	3	1	3	3	3	1	3	3	0	0	0	0
86	7/11	2/12	2/12	7/15	34	3	1	1	4	3	1	1	5	0	0	0	1
89	12/19	5/17	14/8	8/6	68	4	2	2	1	5	2	2	1	1	0	0	0

Table VI. Results for data flow graphs with 5 types of operations.

Edges	Delay/Count					T_c	Lower Bound					Upper Bound					$n_U - n_L$				
	(+)	(-)	(*)	(/)	(<)		+	-	*	/	<	+	-	*	/	<	+	-	*	/	<
92	9/8	5/5	12/11	7/14	5/12	89	1	1	2	2	1	1	1	2	2	1	0	0	0	0	0
90	6/11	1/12	5/11	10/10	13/6	58	2	1	2	2	2	2	1	2	2	2	0	0	0	0	0
93	10/13	4/6	14/12	1/10	5/9	32	3	1	2	1	1	3	1	3	1	1	0	0	1	0	0
84	7/7	14/8	4/13	2/7	4/15	95	2	3	2	1	2	2	4	2	1	2	0	1	0	0	0
84	12/12	2/10	1/10	3/12	3/6	45	8	1	1	2	2	8	2	1	2	2	0	1	0	0	0

Table VII. Results for data flow graphs with pipelined and non-pipelined operations.

Edges	Delay/Count				T_c	Lower Bound				Upper Bound				$n_U - n_L$			
	$+^1$	$-^2$	$*^2$	$/^2$		$+^1$	$-^2$	$*^2$	$/^2$	$+^1$	$-^2$	$*^2$	$/^2$	$+^1$	$-^2$	$*^2$	$/^2$
97	7/25	2/25	-	-	37	2	2	-	-	2	2	-	-	0	0	-	-
90	1/25	2/25	-	-	15	2	4	-	-	2	4	-	-	0	0	-	-
91	3/10	7/16	1/24	-	31	1	5	1	-	1	5	1	-	0	0	0	-
79	6/24	11/16	9/10	-	87	1	3	2	-	1	3	2	-	0	0	0	-
83	8/19	9/17	3/8	12/6	56	1	4	1	2	1	4	1	2	0	0	0	0
86	7/11	2/12	2/12	7/15	95	2	1	1	4	2	1	1	4	0	0	0	0

[1] Pipelined operation
[2] Non-pipelined operation

Table VIII. Comparison of the lower bounds from Theorem 3 and 5[3].

Edges	Delay/Count				T_c	n_{L1} by Theorem 3				n_{L2} by Theorem 5				$n_{L1} - n_{L2}$			
	$+^1$	$-^2$	$*^2$	$/^2$		$+^1$	$-^2$	$*^2$	$/^2$	$+^1$	$-^2$	$*^2$	$/^2$	$+^1$	$-^2$	$*^2$	$/^2$
97	7/25	2/25	-	-	37	2	2	-	-	2	2	-	-	0	-	-	-
90	1/25	2/25	-	-	15	2	4	-	-	2	4	-	-	0	-	-	-
91	3/10	7/16	1/24	-	31	1	5	1	-	1	5	1	-	0	-	-	-
79	6/24	11/16	9/10	-	87	1	3	2	-	1	3	2	-	0	-	-	-
83	8/19	9/17	3/8	12/6	56	1	4	1	2	1	4	1	2	0	-	-	-
86	7/11	2/12	2/12	7/15	95	2	1	1	4	2	1	1	4	0	-	-	-

[1] Pipelined operation
[2] Non-pipelined operation
[3] Only pipelined operations are compared

Table IX. Differential equation example from [2].

system	non-pipelined			complexity
	cycles(T)	(*)	fu(+,-)	
FDS	6	3	2	$O(Tm^3)$
	7	2	2	
Parker's	6	3	2	$O(Tm^2 log\ m)$
	7	2	2	
Ours(n_L)	6	3	2	$O(T^2m)$
	7	2	2	

system	pipelined			complexity
	cycles(T)	(*)	fu(+,-)	
FDS	6	2	2	$O(Tm^3)$
	7	1	2	
Parker's	6	n/a	n/a	$O(Tm^2 log\ m)$
	7	n/a	n/a	
Ours(n_L)	6	2	2	$O(T^2m)$
	7	1	2	

Table X. Fifth order elliptical filter.

system	non-pipelined			complexity
	cycles(T)	(+)	(*)	
FDS	17	3	3	$O(Tm^3)$
	18	3	2	
	19	2	2	
	21	2	1	
FDL	17	3	3	$O(Tm^3)$
	18	2	2	
	19	n/a	n/a	
	21	2	1	
Parker's	17	3	3	$O(Tm^2 log\ m)$
	18	2	2	
	19	n/a	n/a	
	21	2	1	
Ours(n_L)	17	3	3	$O(T^2m)$
	18	2	2	
	19	2	2	
	21	2	1	

system	pipelined			complexity
	cycles(T)	(+)	(*)	
FDS	17	3	2	$O(Tm^3)$
	18	3	1	
	19	2	1	
	21	n/a	n/a	
FDL	17	3	2	$O(Tm^3)$
	18	3	1	
	19	2	1	
	21	n/a	n/a	
Parker's	17	n/a	n/a	$O(Tm^2 log\ m)$
	18	n/a	n/a	
	19	n/a	n/a	
	21	n/a	n/a	
Ours(n_L)	17	3	2	$O(T^2m)$
	18	3	1	
	19	2	1	
	21	2	1	

Table XI. Results for data flow graphs with functional pipelined operations.

Examples	IT	II	Adds			Mults		
			Sehwa	HAL	Ours(n_L)	Sehwa	HAL	Ours(n_L)
FIR[17]	6	3	6	6	6	3	3	3
Elliptic[16]	19	17	n/a	3	2	n/a	2	2

[2] P.G. Paulin and J.P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Trans. on Computer-aided Design*, pp. 661-679, vol.8, no.6, June 1989.

[3] A.C. Parker, J. Pizarro and M.J. Mlinar, "MAHA: A program for data path synthesis," In *Proceeding of the 23rd ACM/IEEE Design Automation Conference*, pp. 461-466, June 1986.

[4] M. Balakrishnan and B. Marnedel, "Integrated scheduling and binding: A synthesis approach for design space exploration," In *Proceeding of 26th ACM/IEEE Design Automation Conference*, pp. 68-74, June 1989.

[5] In-cheol Park and Chng-min Kyang, "Fast and near optimal scheduling in automatic data path synthesis," *In Proceedings of 28th ACM/IEEE Design Automation Conference*, pp. 680-685, June 1991.

[6] J.H. Lee, Y.C. Hsu and Y.L. Lin, "A new integer linear programming formulated for the scheduling problem in data path synthesis," In *Proc. of IEEE ICCAD* , pp. 20-23, 1989.

[7] L.J. Hafer and A.C. Parker, "A Formal Method for Specification, Analysis, and Design of Register-Transfer Level Digital Logic," *IEEE Trans. on CAD*, vol. 2, no. 1, pp. 4-18, Jan. 1983.

[8] M.R. Barbacci, "Automated Exploration of the Design Space for Register Transfer System," Ph.D. Thesis, CMU, 1973.

[9] J.A. Nester, "Specification & Synthesis of Digital System with Interface," *CMUCAD-87-10, Dept. of ECE, CMU*, Apr. 1987.

[10] M.C. McFarland, A.C. Parker and R. Camposano, "The high-level synthesis of digital systems," In *Proceeding of the IEEE*, pp. 301-318, 78(2), Feb. 1990.

[11] A.B. Barskiy, "Minimizing the number of computing devices needed to realize a computational process with a specified time," *Engineering Cybernetics*, No. 6, pp. 59-63, 1968.

[12] T.C. Hu, "Parallel sequencing and assembly line problems," *Operation Research*, Vol 9, pp. 841-848, Nov. 1961.

[13] Rajiv Jain, *et al*, "Empirical evaluation of some high-level synthesis scheduling heuristics," In *Proceeding of the 28th ACM/IEEE Design Automation Conference*, pp. 686-689, June 1991.

[14] Miodrag Potkonjak and Jan Rabaey, "A scheduling and resource allocation algorithm for hierarchical signal flow graph," In *26th ACM/IEEE Design Automation Conference*, pp. 7-12, June 1989.

[15] Ki-soo Huang, et. al., "Scheduling and hardware sharing in pipeline data paths," In *Proceeding of IEEE*, pp. 24-27, 1989.

[16] S.Y. Kung, H.J. Whitehouse and T. Kailath, *VLSI modern signal processing*, Prentice Hall, pp. 258-264, 1985.

[17] N. Park and A.C. Parker, "Sehwa: A Software Package for Synthesis of Pipeline from Behavioral Specifications," *IEEE Trans. Computer-Aided Design,* pp. 356-370, Mar. 1988.

[18] D.J. Mallon and P.B. Denyer, "A New Approach to Pipeline Optimization," *Proc. European Conf. on Design Automation,* pp. 83-88, Mar. 1990.

[19] M. Nourani-Dargiri, C.A. Papachriston and Y.Takefuji, "A Neural Network Based Algorithm for the Scheduling Problem in High-level Synthesis," *Private Communication of SRC Group, SRC Pub C 92422* 1992.

[20] Roni Potasman, Joseph Lis, Alexandru Nicolau and Daniel Gajski, "Percolation Based Synthesis," *In Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 444-449, June 1990.

[21] Minjoong Rim and Rajiv Jain, "RACALS II: A New Approach for Scheduling Data Flow Graphs," Working paper, 1992.

[22] Robert Tarjan, *Data Structures and Network Algorithms*, SIAM, 1983.