Cone-Based Clustering Heuristic for List-Scheduling Algorithms *

Sriram Govindarajan and Ranga Vemuri †

Laboratory for Digital Design Environments
Department of ECECS
P.O. Box 210030
University of Cincinnati
Cincinnati, OH 45221–0030

Abstract

*List scheduling algorithms attempt to minimize latency under resource constraints using a priority list. We propose a new heuristic that can be used in conjunction with any priority function. At each time-step, the proposed clustering heuristic tries to find a best match between ready operations and the resource set. The heuristic arbitrates among equal priority operations based on operation-clusters formed from the dependency graph. Based on this heuristic we have presented a new Cone-Based List Scheduling (*CBLS*) algorithm. Results presented in this paper compare* CBLS *with the well-known Force Directed List Scheduling (*FDLS*) algorithm, for several synthesis benchmarks. In cases where* FDLS *produces sub-optimal schedules,* CBLS *produces better schedules and in other cases* CBLS *performs as good as* FDLS*. Moreover, in conjunction with a simple priority function (namely the self-force of an operator),* CBLS *results in considerable improvement in latency when compared to* FDLS *that has the same priority function. Finally, we show that* CBLS *with the simple priority function performs better in execution time as well as latency when compared to the original* FDLS *that has a relatively complex priority function.*

1 Introduction

The traditional view of high level or behavioral synthesis (HLS) [1] involves transforming the behavioral specification of a design into a register transfer level (RTL) specification which usually consists of a data path and a controller. Scheduling, as stated by many of our peers [2, 3, 4], is an important step in HLS. Scheduling can be described as the process of dividing the DFG into time steps that correspond to clock cycles at the RTL level. Therefore, scheduling directly controls the throughput rate of the RTL design produced. However, for large designs the task of finding optimal schedules is a bottleneck in terms of synthesis time. Therefore, *there exists a tradeoff between the scheduling time and design performance.* A designer would try to exploit this tradeoff using good scheduling algorithms that need to be computationally simple, at the same time produce high-quality schedules.

A wide variety of algorithms [2, 3, 4] exist in the current literature to perform scheduling. In this paper we are primarily concerned about the *List Scheduling* (LS) algorithm [4, 2] that takes resource constraints (design area and component library) and tries to optimize the latency (or throughput) of the design. In short, a basic list scheduling algorithm [4] maintains a priority list of operations (*ready list*) at each time step, from which it schedules operations until resources become insufficient, and defers the rest. Some of the priority functions (PFs) suggested in the literature are the *mobility range* [2], the *critical path* [4] and the *force of an operator* which is used in the well known Force Directed List Scheduling (FDLS) algorithm [3], suggested by Paulin and Knight. In this paper we will consider FDLS for comparison purposes.

The following section presents an example to show the performance of list scheduling algorithm and the improvement that can be done. Section 3 describes our heuristic which is the basic idea behind the Cone-Based List Scheduling algorithm (CBLS) that we are proposing. Section 4 gives a detailed presentation of the CBLS algorithm. In Section 5 we compare the performance of CBLS with the well known Force-Directed List Scheduling (FDLS) algorithm, for several synthesis benchmarks. Finally, we present some improvements that can be made to CBLS, in Section 6.

2 Motivation through an Example

The PFs mentioned in Section 1 need not clearly specify a total ordering of the ready list. They could generate equal priorities for some ready operations and hence they may be placed in any order. Thus a par-

*This research was partially supported by DARPA and monitored by the FBI, under contract number J-FBI-93-116.

†Author for Correspondence, (513)-556-4784 (Voice), (513)-556-7326 (FAX), *Ranga.Vemuri@UC.EDU*

ED&TC '97 on CD-ROM

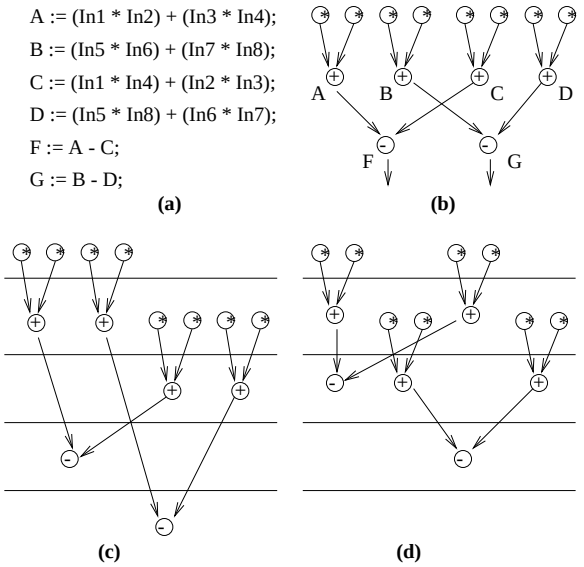

Figure 1: **(a) An Example (b) DFG (c) List Scheduled DFG (d) Improved Schedule**

ticular order of picking the equal priority operations for scheduling at any time step could finally lead to a sub-optimal schedule. For example, consider the behavioral specification and the corresponding DFG shown in Figures 1.a and 1.b. Consider a resource set with four multipliers, two adders and one subtractor. In this example, all paths starting from the eight multiply operations are of equal length and are critical. Therefore, any of the PFs mentioned in Section 1 would compute the priorities of the eight multiply operations to have the same value. Suppose at the first time step, the first four multiplications were picked for scheduling, it would finally result in a schedule (Figure 1.c) that has five time steps. However, for the same resource set there exists a better schedule shown in Figure 1.d that is one time step less. Note that, in Figure 1.c the subtractor is not utilized in the third time step. Whereas, in the new schedule the subtractor is utilized in the third time step, which accounts for the reduction of a time step. In the following section we will describe the basic idea behind producing better schedules for such cases.

3 Cone-Based Clustering Heuristic

Consider the DFG shown in Figure 1.b and the resource set with four multipliers, two adders and a subtractor. In the second stage of the DFG there are four addition operations (A,B,C and D) that are contributing to the two output nodes F and G. Suppose, at the second time step there were two adders available. Then, it is quite intuitive to *schedule those operations that contribute to the same output.* In this example,

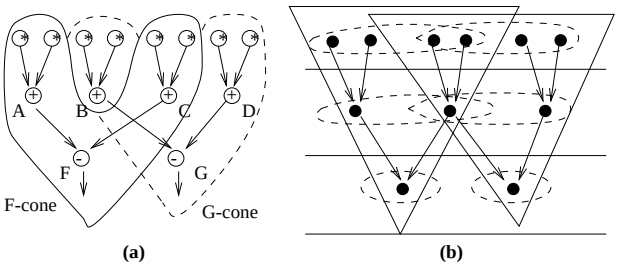

Figure 2: **(a) Cone formation (b) Cone and Cluster formation**

we should schedule either the operations A and C since they contribute to the same F output, or schedule the operations B and D since they contribute to same G output. The same strategy can be extended to the first time step. If we have only four multipliers, we should schedule those multiply operations that contribute to the same output operation. It is wise to schedule operations contributing to the same output because, this could finally help in the scheduling of their successor operations at earlier time steps. The effect of this approach is clearly shown in Figure 1.d, resulting in a shorter schedule.

An important point to note is that this heuristic should be followed, taking into account other features of the DFG. For example, if we neglect the mobility range (Alap_timestep − Asap_timestep + 1) [2] and try to schedule operations purely based on this heuristic, it might result in a poor schedule. However, if we use this heuristic in conjunction with a priority function that accounts for others factors in the DFG, the combination would produce better schedules wherever possible as shown in Figures 1.c and 1.d. We will now present some definitions that explain the methodology behind the heuristic.

The basic idea is to schedule operations that belong to the same output operation. We define an **Output operation** to be a leaf node of the DFG (We will consider extension to cyclic DFGs later in Section 6). For example, in Figure 1.b there are two output operations F and G. We define a **Cone** to be the set of all operations that are ancestors of an output operation, including the output operation itself. A cone is marked by the id of the output operation. Figure 2.a shows the formation of F-cone and the G-cone (marked by the solid and the dashed lines). Operations A, C, F and their four ancestor multiplication operations *belong* to the F-cone and the rest belong to the G-cone. A DFG can be viewed as a set of cones that may be *intersecting*, in other words having one or more common operations, as shown for another DFG in Figure 2.b. Therefore, an operation may belong to more than one

cone. We define the **Cone_set** of an operation to be the set of all cones to which the operation belongs. The cone_set for each of the operations in the DFG shown in Figure 2.a will have only one element, either F or G.

We know that a ready list is a list of operations that are ready to be scheduled at the current time step. We now define a **Cluster** to be the set of all operations from the ready list that belong to the same cone. For example, Figure 2.b shows that there are two clusters at each time step (marked by the dashed ellipses). Since cones may intersect each other, clusters can also intersect as shown in Figure 2.b. *A cluster has a subset of operations from a single cone and has the same id as that cone.* We define the **size** of a cluster to be the number of operations in it. We define a **Cluster_set** to be the set of all clusters formed at a particular time step, from the ready list. Basically, after cluster formation, every operation in the ready list will belong to one or more clusters in the cluster_set. Note that the important difference between a cone and a cluster is that clusters are dynamically created at each time step, whereas cones are created only once for a DFG.

The basic idea behind our heuristic is to attempt to schedule full clusters at every time step, when there is a conflict among equal priority operations. The following section presents the algorithm that is based on the idea presented so far.

4 Cone-Based List Scheduling

The Cone-Based List Scheduling (CBLS) algorithm combines the basic List Scheduling algorithm [4, 2] with the Cone-Based Clustering heuristic presented in Section 3. The idea behind the CBLS algorithm (shown in Figure 3) is that, *whenever there is resource contention among operations whose priorities are equal, schedule those operations that belong to the same cluster, i.e., that contribute to the same output.* The algorithm guarantees that the priorities generated by the PF are never violated. Therefore, we are assured that the algorithm never produces a worse schedule than that produced by the list scheduling algorithm using the same PF. In the worst case, CBLS using a PF would perform as good as the original LS algorithm using the same PF.

There are two main parts to our CBLS algorithm: the formation of cones and the formation of clusters. The procedure *FORM_CONES* (in Figure 3) searches for leaf nodes in the DFG and forms a new cone with each leaf node as the output operation. The procedure *Upward_propagate_new_cone_id* adds the id of the output operation to the cone_set of the output operation and each of its ancestors, thereby creating the cone. It is a procedure that finds all ancestors by re-

```
CONE_BASED_LIST_SCHEDULING(DFG, R_set)
Begin
    ▷ Here, obtain the ASAP schedule of the DFG,
    ▷ if it is necessary for computing the PF.
    FORM_CONES(DFG)
    T_step ← 1  ▷ While loop iterates across T_step's
    while (not all DFG operations are scheduled)
        L_ready ← {Unscheduled operations all of
          whose predecessors have been scheduled }
        C_set ← { All Clusters from L_ready }
        C_set ← Sort_Clusters_by_Increasing_Size(C_set)
        while (R_set not sufficient)
        ▷ Need to defer an operation
            for each (operation Op ∈ L_ready)
              compute_PRIORITY_FUNCTION(Op)
            end for
            P_min ← Minimum Priority assigned to
                    any operation in L_ready
            for each (cluster C ∈ C_set in sorted order)
                for each (operation Op ∈ C)
                    if (Op.priority = P_min) then
                        ▷ Defer the operation
                        L_ready ← L_ready − {Op}
                        ▷ Exit from outer for loop
                        break 2
                    end if
                end for
            end for
        end while
        for each (operation Op ∈ L_ready)
            Schedule Op at T_step
        end for
        T_step ← T_step + 1
    end while
End
FORM_CONES(DFG)
▷Form cone with leaf node as output operation
Begin
    for each (operation Op ∈ DFG)
        Op.cone_set = ∅
    end for
    for each (operation Op ∈ DFG)
        if (Op is a leaf node) then
          Upward_propagate_new_cone_id(Op, Op.Id)
        end if
    end for
End
Upward_propagate_new_cone_id(Op, cone_id)
▷Propagate the cone_id to all ancestors
Begin
    Op.cone_set ← Op.cone_set ∪ {cone_id}
    for each (O_pred ∈ Op.predecessor_list)
        Upward_propagate_new_cone_id(O_pred, cone_id)
    end for
End
```

Figure 3: **The CBLS Algorithm**

cursively traversing through a list of immediate predecessors (*predecessor_list*), starting from the output operation of the cone. A cluster in the cluster_set ($\mathcal{C}_{set}$) is formed by looking at the cone_sets of the operations in the ready list, to group operations that belong to the same cone. We can achieve a simple implementation of the cluster_set creation, that makes only one pass of the ready list. On the other hand, the procedure *FORM_CONES* is a relatively complex procedure. However, *FORM_CONES is executed only once, before the actual list scheduling is done*. Therefore, the only overhead in CBLS compared to the basic LS algorithm, is the formation of clusters at each time step, which is also not computationally expensive. We will now proceed to explain the rest of the algorithm.

CBLS, like any other list scheduling algorithm works by scheduling operations at each time step ($\mathcal{T}_{step}$), starting from the first. In each iteration of the outer while loop (corresponding to a time step), the ready list ($\mathcal{L}_{ready}$) and the cluster_set ($\mathcal{C}_{set}$) are formed. The cluster_set is then sorted in the ascending order, based on the size of a cluster. This is because, the largest cluster is picked for scheduling so that maximum resources can be allocated to operations in a single cluster. The inner while loop defers an operation in each iteration, until the resource set ($\mathcal{R}_{set}$) is sufficient to schedule all operations in the ready list. First, priorities for the ready operations are re-computed by calling the user defined priority function. Note that the cluster_set is not re-constructed inside the inner while loop, but is done only once in each time step. From the sorted cluster_set, the smallest cluster ($\mathcal{C}$) is selected for deferring operations. The inner most *for* loop finds a least priority ($\mathcal{P}_{min}$) operation from smallest cluster selected. If none of the operations in $\mathcal{C}$ have the least priority, then the next smallest cluster is selected. In this way the two nested for loops ensure that always a least priority operation is deferred in each iteration of the inner while loop. Thus, all clusters in the cluster_set are searched to exhaust all the least priority operations before moving to the next smallest priority. The body of the inner most if-condition deletes the selected operation from the ready list and breaks out of the two enclosing nested for loops.

After the inner while loop finishes, all the remaining operations in the ready list are scheduled at the current time step. The $\mathcal{T}_{step}$ is then incremented and the outer while loop continues until all the operations in the DFG are scheduled.

5 Results

As explained in Section 4, CBLS can be used with any priority function. For comparisons, we have chosen the well known Force-Directed List Scheduling algorithm [3] that uses a rigorous priority function called the Force of an operator. As discussed by Paulin and Knight, the Force of an operator is the sum of the Self-force and the Successor-forces. To see the effect of our clustering heuristic with other priority functions, we have considered a simple priority function similar to this Force, but with no successor force calculation, i.e., computes only the Self-force. We will call these two priority functions as follows:

FDLS/p PF = PF in Paulin and Knight's FDLS = Self-force + Successor-forces

FDLS/s PF = Self-force of an operation

We have taken results by executing CBLS and FDLS with these two PFs. We will call these algorithms as follows:

FDLS/p Algorithm = FDLS = The original Paulin and Knight's Algorithm (with FDLS/p PF)

FDLS/s Algorithm = FDLS with FDLS/s PF

CBLS/p Algorithm = CBLS with FDLS/p PF

CBLS/s Algorithm = CBLS with FDLS/s PF

In the following sections we will compare the performance of the CBLS algorithms versus the FDLS algorithms, for a variety of design examples.

Design Examples	Maximal $\mathcal{R}_{set}$ {[+], [*], [÷]}	Tot. Ops.	$\mathcal{R}_{Sets}$	CPL
1.DCT8x8	{448, 512, 64}	1984	$1.4 * 10^7$	10
2.DCT4x4	{48, 64, 16}	240	49152	8
3.DFT	{48, 48, 0}	96	2304	8
4.TN-1	{24, 48, 0}	60	1152	6
5.TN-2	{18, 36, 0}	54	648	6
6.FFT	{24, 24, 0}	48	576	6
7.LSS4x4	{12, 16, 16}	44	3072	4
8.LSS3x3	{9, 12, 9}	30	972	3

Table 1: **Synthesis Benchmarks**

5.1 Design Examples

For obtaining the results, we have considered a number of synthesis benchmarks listed in Table 1. The examples are listed in the decreasing order of the number of operations in the DFG. For each example we have shown the maximal resource set (maximum number of operations of each type that appear in parallel, in the ASAP schedule), the total number of operations, the total number of possible resource sets ($\mathcal{R}_{Sets}$) and the critical path length (CPL). We use a parameterized RTL component library using which the scheduler generates the resource sets for a given design.

Our first example, the Discrete Cosine Transform (DCT) [5] 8x8 version is the largest among those in the table. It has more than ten million possible resource sets and 1984 operations in the DFG, of which

a maximum of 448 additions, 512 multiplications and 64 divisions appear in parallel in its ASAP schedule. The DCT4x4 is a smaller version of discrete cosine transform. The Discrete Fourier Transform (DFT) [6] example consists of two 8-point DFTs with thirty two design inputs sixteen design outputs. The Threshold Network (TN) [8] example is widely known as the perceptron network in neural systems. Each node in the network takes a variable number of inputs and generates their weighted sum as the output. We have considered two versions of the TN, each having nodes that take two inputs, but the number of nodes and their connectivity are different. The Fast Fourier Transform (FFT) [6] example consists of an 8-point FFT-butterfly network constructed using two 4-point FFTs. The Linear System Solver (LSS) [7] example is a popular method of solving a linear system of equations using matrix inversion. This LSS example computes the solution to a system of four equations with four variables each.

5.2 CBLS/p Versus FDLS/p

In this section, we compare the results taken by executing CBLS with the FDLS/p PF and the original FDLS algorithm as discussed by Paulin and Knight [3]. Table 2 shows the latency (number of $\mathcal{T}_{steps}$) of schedules generated by these two algorithms, for the synthesis benchmarks described earlier. Column-A and column-B show the number of time steps for the schedules generated by FDLS/p and CBLS/p algorithms and column-C shows the reduction achieved by CBLS/p. The resource sets for each of these examples are listed in the order of increasing areas (in terms of the sum of the areas of all the components in the resource set).

All the design examples have a large number of possible resource sets. Therefore, we identified selected points in their design space and ran the algorithms for those resource sets. Typically, we selected the resource sets to yield the maximally serial and maximally parallel schedules and several in between. For some smaller area resource sets ($R_1 - R_4$) of DCT8x8, CBLS produced schedules that are seven time steps shorter compared to that of FDLS. For DCT4x4, CBLS showed a maximum reduction of three time steps. In the case of DFT which is a smaller example compared to DCT4x4, CBLS reduced the schedule by four time steps. In the case of FFT and the TN examples, the CBLS showed a maximum improvement of three time steps. So far for all the examples, as the area of the resource set increases the number of time steps saved decreases. However, this need not be true always, as we can see in the case the first and the second resource sets for both the LSS examples. This is because, in this case, the number of division operations chosen in

Design Examples	$\mathcal{R}_{Set}$ Information $R_i\{[+],[*],[\div]\}$	Latency($\mathcal{T}_{steps}$) A	B	C
1.DCT8x8	$R_1\{7,8,64\}$	140	133	7
	$R_2\{14,16,64\}$	76	69	7
	$R_3\{28,32,64\}$	44	37	7
	$R_4\{56,64,64\}$	28	21	7
	$R_5\{112,128,64\}$	16	13	3
	$R_6\{224,256,64\}$	12	11	1
	$R_7\{448,512,64\}$	10	10	0
2.DCT4x4	$R_1\{3,4,16\}$	39	36	3
	$R_2\{6,8,16\}$	23	20	3
	$R_3\{12,16,16\}$	15	12	3
	$R_4\{24,32,16\}$	10	9	1
	$R_5\{48,64,16\}$	8	8	0
3.DFT	$R_1\{4,4,0\}$	27	23	4
	$R_2\{7,7,0\}$	25	21	4
	$R_3\{14,14,0\}$	15	13	2
	$R_4\{28,28,0\}$	10	9	1
	$R_5\{48,48,0\}$	8	8	0
4.TN-1	$R_1\{4,8,0\}$	18	15	3
	$R_2\{12,24,0\}$	11	9	2
	$R_3\{20,40,0\}$	8	7	1
	$R_4\{24,48,0\}$	6	6	0
5.TN-2	$R_1\{4,8,0\}$	13	11	2
	$R_2\{10,20,0\}$	10	8	2
	$R_3\{18,36,0\}$	6	6	0
6.FFT	$R_1\{3,3,0\}$	19	16	3
	$R_2\{4,4,0\}$	17	14	3
	$R_3\{7,7,0\}$	15	13	2
	$R_4\{14,14,0\}$	10	9	1
	$R_5\{24,24,0\}$	6	6	0
7.LSS4x4	$R_1\{3,4,2\}$	13	11	2
	$R_2\{3,4,4\}$	10	7	3
	$R_3\{6,8,4\}$	8	7	1
	$R_4\{6,8,8\}$	6	5	1
	$R_5\{12,16,16\}$	4	4	0
8.LSS3x3	$R_1\{3,4,2\}$	8	7	1
	$R_2\{6,8,4\}$	6	4	2
	$R_3\{9,12,9\}$	3	3	0

Table 2: **Latencies for FDLS/p Vs. CBLS/p**

$\mathcal{R}_{Set}$	1. DCT8x8 FDLS/s	CBLS/s	A	2. DCT4x4 FDLS/s	CBLS/s	B
R_1	189	137	52	48	39	9
R_2	97	72	25	26	22	4
R_3	51	39	12	16	12	4
R_4	28	21	7	12	11	1

Table 3: **Latencies for FDLS/s Vs. CBLS/s**

the resource set plays a crucial role in the quality of the schedule. From Table 2, we can see that for all the examples, when the largest resource set was chosen CBLS/p performed as good as FDLS/p. This is because, the largest resource set is sufficient to schedule all the ready list operations at any time step, never leading to a resource contention.

5.3 CBLS/s Versus FDLS/s

In this section, we will compare the results of the CBLS/s algorithm (CBLS with FDLS/s PF) and the FDLS/s algorithm that does not look at successor forces, when computing the force of an operator. The DCT examples are the largest among those listed in Table 1. Therefore, we have presented the latency (number of $\mathcal{T}_{steps}$) results for these DCT examples in Table 3. Columns A and B show the reduction in $\mathcal{T}_{steps}$ achieved by the CBLS/s algorithm for these examples.

From the results we can clearly see that CBLS/s performs extremely well compared to FDLS/s. For the smallest resource set (R_1) of DCT8x8, CBLS/s has reduced more than 50 time steps. There are two reasons to this: (i) FDLS/s without the calculation of successor forces does not perform very well for smaller resource sets, (ii) However, CBLS/s has some knowledge about the structure of the entire DFG and is able to perform much better. The results also show that CBLS *can be used with any simple* PF *and would perform in the worst case as good as the original* LS *with that* PF. Note that, although CBLS/s computes global information about the DFG, much of this computation (cone_sets) is done only once. Only a small amount of additional computation (cluster_sets) is necessary during each iteration of the algorithm.

5.4 CBLS/s Versus FDLS/p

The time spent by FDLS/p in computing the successor forces for large design examples is quite high. Table 4 shows the execution times measured using the Quantify commercial tool on a Sun sparc-20 with a 256 Megabytes of memory. We have shown results only for the DCT examples, since the scheduling time for all other examples are relatively small (less than a few seconds) compared to these numbers. These tables show that the CBLS algorithms yield better schedules, although at the expense of some execution time, compared to the corresponding FDLS algorithms.

The graphs in Figures 4.a and 4.b compare results for the first four resource sets of the DCT8x8 example. From Figure 4.a, we can see that CBLS/s, shows only a small increase in latency when compared to CBLS/p, and is still better than the original FDLS/p algorithm. From Figure 4.b, we can see that both the CBLS algorithms, incur only a small overhead in time (even for small resource sets) with respect to the corresponding FDLS algorithms. Also, we can see that with the simpler priority function (FDLS/s PF) there is a considerable amount of reduction in time for both CBLS and FDLS, compared to the FDLS/p PF.

1. DCT8x8: Scheduling time (seconds)				
$\mathcal{R}_{set}$	FDLS/p	CBLS/p	FDLS/s	CBLS/s
R_1	275.38	341.48	45.20	117.90
R_2	173.13	225.97	31.37	63.52
R_3	73.13	90.29	25.78	38.37
R_4	43.96	53.82	23.76	26.23
R_5	26.74	27.03	19.93	19.05
R_6	18.80	20.14	16.01	16.22
R_7	15.75	16.23	15.59	16.10
2. DCT4x4: Scheduling time (seconds)				
$\mathcal{R}_{set}$	FDLS/p	CBLS/s	FDLS/s	CBLS/s
R_1	1.354	1.535	0.863	0.886
R_2	0.865	0.958	0.603	0.669
R_3	0.621	0.686	0.542	0.562
R_4	0.537	0.565	0.513	0.526
R_5	0.533	0.527	0.513	0.522

Table 4: **Scheduling Time for the DCT examples**

The important point to note from both the graphs is that CBLS*/s with the* FDLS*/s priority function performs better in latency as well as execution time when compared to* FDLS*/p that uses the considerably more complex* FDLS*/p* PF. There are two reasons to this: (i) FDLS/p incurs a lot of overhead in execution time because of the relatively complex priority function, which gathers information about the entire DFG during *each* iteration of the scheduling algorithm. (ii) Since CBLS has knowledge about structure of the entire DFG, much of this knowledge in the form of cone_sets gathered once at the beginning of the algorithm, it is able to produce better schedules even with a relatively simple priority function.

6 Improvements

In the CBLS algorithm, the cones are always formed from the leaf nodes of the DFG. We will now show a case where we can further improve the schedule, if cones are formed from intermediate operations in the DFG.

Consider the DFG shown in Figure 5.b that corresponds the behavioral specification shown in Figure 5.a. Note that there is only one cone shown as a solid triangle. If we choose a resource set that has two multipliers, one adder and one subtractor and perform the CBLS, it could result in the schedule shown in Figure 5.c. Now consider the two intermediate addition operations as output operations and form two cones

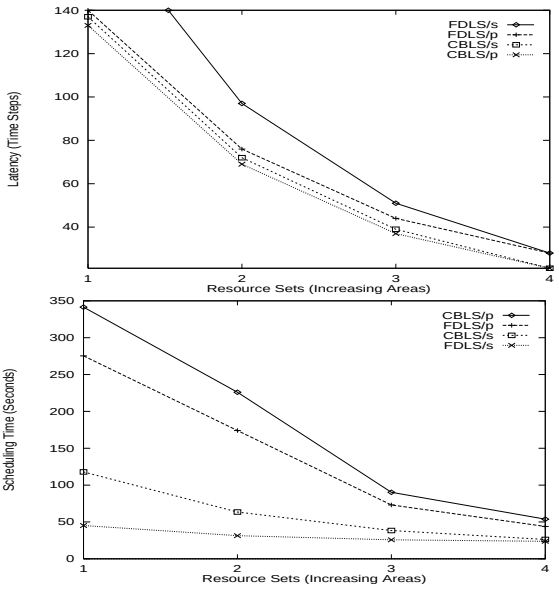

Figure 4: **DCT8x8: (a) $\mathcal{R}_{Sets}$ Vs. Latency (b) $\mathcal{R}_{Sets}$ Vs. Scheduling Time**

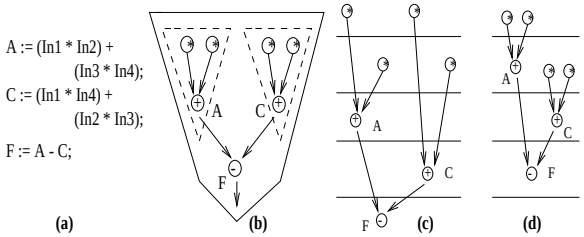

Figure 5: **Improving Schedule by intermediate Cone formation**

instead of a single cone. These cones are shown by the dashed triangles in Figure 5.b. If we run CBLS with this kind of cone formation then we will get the improved schedule shown in Figure 5.d, that is one time step less. Therefore, if we cannot clearly order operators using the cones formed from leaf nodes, we can use the intermediate cones to further resolve the conflict.

There are two advantages to this approach. First is in the case of cyclic graphs, where cycles are broken at a natural point, operations from the broken cycle (based on the new dependency relations) can be easily grouped into a cone, since cones can now be formed from any intermediate operation. Second is the case of a design output point that is not always a leaf node of the DFG. Rather, it is the point where the execution cycle of the behavioral specification ends for that particular design output. This can only be identified by the user, and can be specified as a pragma. From such pragmas, we can locate the output operations and form cones from them.

7 Conclusion

We have presented a new Cone-Based List Scheduling algorithm in this paper. It is based on a heuristic that tries to utilize resources efficiently by selecting carefully among equal priority operations. The selection criteria is based on the knowledge of the entire DFG. CBLS can be used with any priority function and will never violate the priorities generated by the PF. Therefore, CBLS would always perform either as good or better than the original LS algorithm with that PF. Results for several synthesis benchmarks have shown that CBLS improves the schedule when FDLS produces a sub-optimal schedule. In our experience, CBLS yields definite gains for data flow and arithmetic dominated designs and works at least as good as the original LS algorithm for the control dominated designs. The effect of the schedule quality on the overall area of the design is yet to be studied. As shown in the results, CBLS is computationally inexpensive and therefore incurs very little overhead compared to the original LS algorithm. Therefore, CBLS can be used in place of any other list scheduling algorithm by simply plugging-in the priority function.

References

[1] Raul Camposano, Wayne Wolf, "High-Level VLSI Synthesis", Kluwer Academic Publishers, 1991.

[2] Daniel Gajski, Nikil Dutt, "High-Level Synthesis", Kluwer Academic Publishers, 1992.

[3] Pierre G. Paulin and John P. Knight, "Force Directed Scheduling for the behavioral synthesis of ASICs," IEEE Trans. Computer Aided Design, Vol.8, pp. 661-679, June 1989.

[4] Jan Vanhoof et. al., "High-Level Synthesis for Real-Time Digital Signal Processing", Kluwer Academic Publishers, 1993.

[5] Phillip E. Mattison, "Practical Digital Video with Programming Examples in C", John Wiley & Sons, Inc., 1994.

[6] Leland B. Jackson, "Digital Filters and Signal Processing", Second Edition, Kluwer Academic Publishers, 1989.

[7] Michael Wolfe, "High Performance Compilers for Parallel Computing", Addison-Wesley Pub., 1996.

[8] Jacek M. Zurada, "Introduction to Artificial Neural Systems", West Publishing Company, 1992.