A Component Selection Algorithm for High-Performance Pipelines †

Smita Bakshi and Daniel D. Gajski
Department of Information and Computer Science
University of California, Irvine, CA, 92717-3425, USA

Abstract

The use of a realistic component library with multiple implementations of operators, results in cost efficient designs; slow components can then be used on non-critical paths and the more expensive components on only the critical paths. This paper presents a cost-optimized algorithm for selecting components and pipelining a data flow graph, given a multiple-implementation library, and throughput and latency constraints. Experiments on a few benchmarks indicate that our algorithm gives results that are within 0.7% of the optimal result.

1 Introduction

An important factor in obtaining cost efficient designs is the ability to use multiple operator implementations in the datapath. Delay paths can then be balanced by using slow components where possible, and the faster components only when necessary. For high-performance applications, such as DSP systems, designers often combine pipelining with the use of a multiple-implementation library, so as to satisfy performance requirements at a reasonable cost.

This paper presents an algorithm that combines pipelining with component selection using such a realistic library. The aim of the algorithm is to pipeline a given data flow graph, and balance the use of slow and fast components, such that the delay of each pipe stage is equal to (or as close to) a given constraint, and the total cost of the data flow graph is minimized.

The paper is organized as follows. The next section outlines related research in the area of pipelined synthesis and explains how we compare with it. Section 3 gives a formal definition of the component selection and pipelining problems while Section 4 describes our proposed algorithm for solving these problems. Section 5 presents results demonstrating the quality of our algorithm and finally, Section 6 concludes the paper with a summary of our major contributions.

2 Previous work

We categorize related research into two classes based on pipelining and component selection. The first class consists of tools such as Sehwa [1], the tools from the GE Corporate R&D Laboratories [2], and PLS, a pipelined scheduler [3]. These tools pipeline a given DFG so as to optimize area or performance for given constraints, usually on the throughput or latency of the design. However, they all assume a single implementation for functional units which forces them to use the same component on non-critical and critical paths, resulting in designs that are inefficient and more costly. SLIMOS [4] and MOSP [5] differ slightly from the above approach - they start from a multiple implementation library and then select *one single implementation per operator*. Hence their final design also contains single implementations, leading to the same design inefficiencies mentioned above.

The second category contains algorithms such as TBS [6] and the module selection algorithm presented in [7]. Though these tools use unrestricted libraries that allow multiple physical implementations for the same operator, they combine component selection with non-pipelined scheduling, rather than with pipelined scheduling.

Our algorithm [8] spans both categories since it pipelines a data flow graph *and*, for each pipe stage, determines the best selection of components from a realistic library containing many different implementations per operator. For a given throughput and latency constraint, our algorithm thus produces cheaper designs over those produced by previous pipelining algorithms that use limited libraries with only one implementation per operator.

3 Problem statement and definitions

Given a data flow graph $\mathcal{DFG}(V, E)$ where V represents a set of vertices, and $E \subseteq V \times V$ a set of directed edges, a component library $\mathcal{CL}$ consisting of a set of three tuples $\langle ComponentType,\ Area \text{ and } Delay \rangle$, and constraints on Pipe Stage *(PS) delay* and *Latency*, find an *Assignment* of vertices to components and a *Partition* of $\lfloor Latency/PS\ delay \rfloor$ stages of delay *PS delay*, so as to minimize cost (given by the sum of the area of datapath components).

The terms *Latency*, *PS delay*, *Assignment* and *Partition* are defined as follows:

Definition 1: *PS delay* is the sample inter-arrival delay, that is the delay between the arrival of two consecutive input samples. This is also the clock cycle of the design. *Throughput*, which is often the prime constraint on DSP systems, is the inverse of the *PS delay*.

†This work was supported by the Semiconductor Research Corporation (grant #93-DJ-146).

Definition 2: *Latency* is the total execution time ($n \times$PS delay, for an n-stage pipeline), that is, the time between the arrival of an input sample and the availability of the corresponding output.

Definition 3: If we associate a type (such as $\times$, $\div$, $+$ etc.) called $VertexType(v)$, with every vertex, v, then an *Assignment* is defined as a function from $V \rightarrow \mathcal{CL}$, such that if $Assignment(v) = c$, then $VertexType(v) = ComponentType(c)$. This just states that vertices can only be mapped to components of the same type.

Definition 4: A *Partition* is a collection of subsets of vertices, such that the union of all subsets is the complete vertex set, V, and the intersection of any two subsets is the empty set. Stated mathematically, a partition is a collection of subsets, V_i, such that $V_i \subset V$, $\bigcup_{\forall i} V_i = V$, and $V_i \bigcap V_j = \emptyset$, $\forall i, j$ where $i \neq j$.

The example in Figure 1 illustrates the problem. Given are a $\mathcal{DFG}$, a $\mathcal{CL}$, and constraints on *PS delay* (10 *ns*) and *Latency* (25 *ns*). The $\mathcal{DFG}$ is partitioned into two stages of delay 10 *ns* each and mapped to components so that the total cost is minimized. The output consists of a mapped and pipelined $\mathcal{DFG}$ and a set of design metrics as shown.

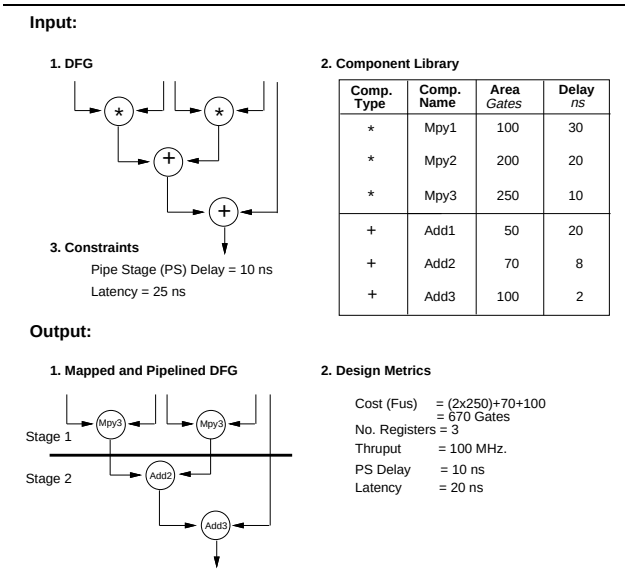


Figure 1: An example illustrating the inputs and outputs of the component selection and pipelining algorithms.

4 Component selection and pipelining

Having stated the problem, we now present the algorithms for component selection and pipelining. We first give an overview of the complete algorithm and then individually explain the tasks of finding an *Assignment* and a *Partition*. Finally, we explain the pseudo-code of the complete algorithm with the help of a walk-through example.

1. Map vertices to fastest components, pipeline $\mathcal{DFG}$, and evaluate performance.
2. **If** (*fastest design does not satisfy constraints*)
3. exit the program.
4. **Else**
5. **Loop**
6. Select the "best" vertex to slow down.
7. Pipeline the $\mathcal{DFG}$, and evaluate performance.
8. **If** (*performance constraints met*),
9. accept this slow down, **else**, reject it.
10. **Until** (*no vertex can be slowed down without violating constraints*).
11. **End if**

Figure 2: An overview of the component selection and pipelining algorithm.

4.1 An overview of the algorithm

The algorithm takes as input a non-pipelined $\mathcal{DFG}$, a component library, and a constraint on the *PS delay* and *Latency*. It outputs a mapped $\mathcal{DFG}$ partitioned into $\lfloor Latency / PS\ delay \rfloor$ stages, such that the delay of each pipe stage is less than or equal to the *PS delay* and the total area of the $\mathcal{DFG}$ is minimized.

The algorithm (Figure 2) starts by mapping each vertex of the $\mathcal{DFG}$ to the fastest available component. It then slows down vertices by mapping them to progressively slower components. At each slow down the $\mathcal{DFG}$ is pipelined and if constraints are violated, the slow down is not accepted. This process is repeated until no vertex can be slowed down without a violation of constraints.

Intuitively speaking, the aim of the algorithm is to slow down as many vertices by as much as possible, and this is achieved by balancing the use of slow and fast components so that the delay of each pipe stage is as close to *PS delay* as possible, and the total cost is minimized.

4.2 Component selection

The key to the algorithm lies in judiciously selecting vertices to be slowed down in each iteration, since slowing down one vertex may prevent slowing down others due to graph dependencies. Thus, the desirability of slowing down a vertex has to be evaluated with respect to all the vertices that would be affected by its slow down. With every vertex we thus associate a value, called the vertex *weight*, which is a measure of its "desirability" or priority in the selection process. In each iteration of the algorithm, vertex weights are evaluated and the vertex with the highest weight is selected to be slowed down.

The vertex weight

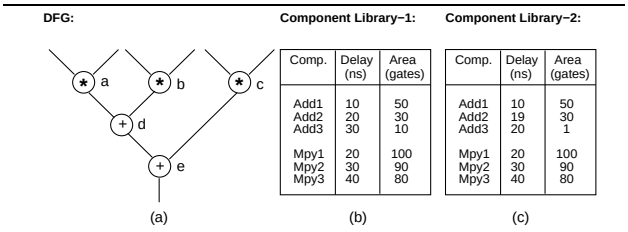
We first give an intuitive explanation of the vertex weight by using an example, and then formally define the terms in the vertex weight formula.

An example

Consider the $\mathcal{DFG}$ and $\mathcal{CL}$ in Figures 3(a) and (b). Let the vertices of the $\mathcal{DFG}$ be initially mapped to the "fastest components" (that is all the $\star$ vertices to $Mpy1$ and all the $+$ vertices to $Add1$). This results in a total delay of

40 ns and a cost of 400 ($[3 \times 100]+[2 \times 50]$) gates. Let the constraint on the *PS delay* be 50 ns.

For the purposes of this explanation, let us assume that we slow down a vertex by replacing it with the next slower component in the library. We now have to pick the first vertex to slow down. Intuitively speaking, this should be the one that gives the highest cost benefit or, in other words, the greatest area reduction. In the example, vertices d and e give an area reduction of 20 gates, as opposed to 10 gates for the $\star$ vertices, a, b, and c. Let us slow down any one vertex, say e. Since e exists on all I-O paths[†], slowing down any other vertex would violate the constraint of 50 ns. Thus by slowing down e, we have prevented slowing down any of the other vertices, a to d, and the final design has a cost of 380 gates and a delay of 50 ns. If we had



Comp.	Delay (ns)	Area (gates)
Add1	10	50
Add2	20	30
Add3	30	10
Mpy1	20	100
Mpy2	30	90
Mpy3	40	80

Comp.	Delay (ns)	Area (gates)
Add1	10	50
Add2	19	30
Add3	20	1
Mpy1	20	100
Mpy2	30	90
Mpy3	40	80

Figure 3: **$\mathcal{DFG}$ and component libraries used to illustrate the vertex weight and the need for a "commonality factor".**

first slowed down node a instead of node e we could still have slowed down nodes b and c in the next two iterations. Instead of replacing node e with $Add2$, we could thus have replaced each of the nodes a, b and c with $Mpy2$, resulting in a cheaper design of 370 gates. Even though individually the vertices a, b and c give an area reduction of just 10 gates each, together their reduction is greater than that of vertex e. Thus the comparison we should be making is:

Area Reduction(e) vs. Area Reduction(a)+Area Reduction(b)+Area Reduction(c)

Assuming, for the time being, that a, b, and c give the same area reduction, the only reason why we would choose to slow down e over a, b, and c is if:

$$\frac{\text{Area Reduction(e)}}{3} > \text{Area Reduction(a)}$$

Thus it is clear that the area reduction by itself is not a good measure of the vertex weight. Rather, it is the area reduction weighted by a factor, roughly equal to the number of unique I/O paths containing that vertex. We call this factor the vertex *commonality factor*. The weight of a vertex, v, is then given by:

$$W(v) = \frac{\text{Area Reduction}(v)}{\text{Commonality Factor}(v)} \quad (1)$$

We refine this formula after formally defining the two terms, area reduction, also called the area-delay gain (ADG), and commonality factor (CF).

[†] An I-O path is defined as a set of operator nodes connecting an input node to an output node. Thus, the example in Figure 3 has the following I-O paths: $a-d-e$, $b-d-e$, and $c-e$.

The area-delay gain

The components considered in the library in Figure 3 are very "evenly" spread out, that is each component differs from the previous one by a delay of 10 ns. If this is not the case, as is most likely in a realistic component library, then both the area *and* delay changes caused by replacing the current assignment of a vertex with a slower component, should be taken into account in the selection process. We should really be comparing the area reduction per unit change in delay, rather than just the area reduction. For instance, for the example in Figure 3, if $Add2$ had a delay of 12 ns instead of 20 ns, the cost benefit of vertices d and e should really be [Area(Add1)-Area(Add2)]/[Delay(Add2)-Delay(Add1)] = (50-30)/(12-10) = 20/2, and of vertices a, b, and c it should be [Area(Mpy1)-Area(Mpy2)]/[Delay(Mpy2)-Delay(Mpy1)] = (100-90)/(30-20) = 10/10. The weight of d and e would then be higher than that of a, b, and c, resulting in their slow down (rather than the slow down of a, b or c), and hence in the less costly design.

The area delay gain (ADG) of a vertex is defined as follows:

Definition 5: Let the current assignment of a vertex, v, be the component c'. If the new assignment of the vertex is c'', then the area-delay gain of v with respect to c'', $ADG(v, c'')$ is defined as:

$$ADG(v, c'') = \frac{Area(c') - Area(c'')}{Delay(c'') - Delay(c')} \quad (2)$$

In the previous example, we slowed down a vertex by replacing it with the next slower component. This may not always be the best choice to make. Consider the component table in Figure 3(c). If we only consider $Mpy2$ and $Add2$ as possible replacements for $Mpy1$ and $Add1$ respectively, we would end up replacing nodes a, b and c with $Mpy2$. However, we get a cheaper design by replacing either of d or e with $Add3$ (area 351 vs. 370 gates). We could have obtained the cheaper design had we conducted a more global search of the component table and for all vertices, determined the component with the greatest area-delay gain. This component is also called the $BestAssignment$ for a vertex v. It is formally defined as follows:

Definition 6: The $BestAssignment$ for a vertex v, is the unique component c'' that satisfies the following properties:

$$ComponentType(c'') = VertexType(v) \quad (3)$$

$$ADG(v, c'') \geq ADG(v, c), \quad \forall c \in \mathcal{CL} \text{ satisfying (3)} \quad (4)$$

In other words, c'' is the component that gives the maximum ADG.

Refining the vertex weight definition

We now refine the vertex weight definition given in (1), to include all the factors mentioned above, namely, the area-delay gain, the $BestAssignment$ component and the vertex commonality factor.

Definition 7: The weight of a vertex, v, is given by:

$$W(v) = \frac{ADG(v, c'')}{CF(v)} \quad (5)$$

where c'' is its $BestAssignment$ (i.e. the unique component satisfying the properties (3) and (4) listed above).

Next, we give a method of obtaining the commonality factor of all vertices in the $\mathcal{DFG}$.

The commonality factor

The commonality factor is determined by making two traversals of the $\mathcal{DFG}$. In the first traversal (from input to output), we assign a *forward weight* (FW) to every node. The forward weight of an output node indicates the number of unique paths from input nodes to that output node. In the second traversal (from output to input nodes) we propagate the forward weight of nodes to their predecessors and assign a *backward weight* (BW) to every node. The backward weight of a node is also its commonality factor.

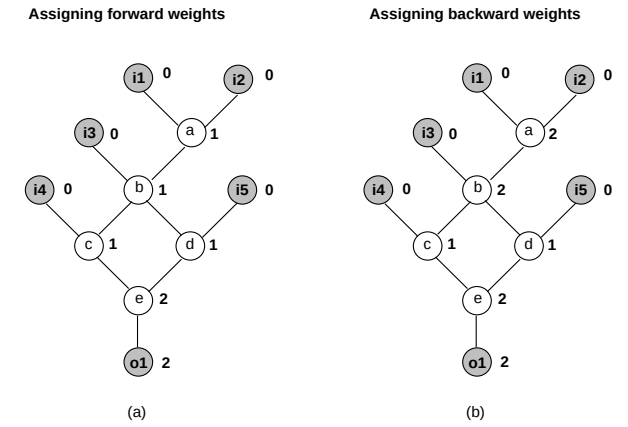


Figure 4: Determining the commonality factor by assigning a forward and backward weight to vertices.

This method is illustrated with the help of an example (Figure 4). As an initialization step all input nodes, $i1$ to $i5$, are assigned a FW of 0, and all operator nodes with *only* input node predecessors (node a in the example) are assigned a FW of 1. The forward weight of a node is then split equally amongst its successors. This split value is rounded up to 1, if it is less than 1. The FW of a vertex is then the sum of the split values from all its predecessors. Using this method, nodes b, c, d and e get assigned a FW of 1, 1, 1 and 2 respectively,

In the backward traversal, we distribute the backward weight of a node to its predecessors in the ratio of their forward weights. The backward weight of a node is then the sum of these partial BWs from all its successors. As an initialization step, we equate the backward weight of all output nodes to their forward weights. Thus BW($o1$)=FW($o1$)=2. We then assign e a BW of 2 since e is the only predecessor of $o1$. The BW of e is then distributed amongst c and d in the ratio of 1:1, resulting in a BW assignment of 1 each. This process is continued resulting in BW(b)=2 and BW(a)=2.

Thus far we have explained how to associate a weight with every vertex, which is used as a priority function in selecting the most favorable vertex to slow down in each iteration of the loop (step 6 in Figure 2). We now explain the next step (step 7) of the combined component selection and pipelining algorithm, namely the algorithm for partitioning or pipelining the $\mathcal{DFG}$ into equal delay stages.

4.3 Pipelining

Given a $\mathcal{DFG}$, an *Assignment* for the $\mathcal{DFG}$, and a *PS delay* constraint, the pipelining algorithm partitions the $\mathcal{DFG}$ into a minimal number of stages that meet the *PS delay* constraint. It traverses the graph in two directions, downward (from the input to the output nodes), and upward (from output to input nodes). As it traverses the graph it keeps accumulating the delay from the boundary of the last pipe stage. A new boundary is set when the performance constraint can no longer be satisfied. The traversal is repeated for both directions, and the pipeline with the fewer number of "cuts" is selected. A "cut" refers to the intersection of an edge of the $\mathcal{DFG}$ with the pipe stage partition, and it corresponds to a pipeline register. Hence, the fewer the number of cuts, the fewer the pipeline registers.

4.4 Pseudo-code of the combined algorithm

Having defined the vertex weight and the algorithm for pipelining (steps 6 and 7), we now present the pseudo-code of the complete algorithm (Figure 5) and walk through it by using a simple example.

We wish to select a design with a *PS delay* of 30 ns and a *Latency* of 60 ns (or 2 pipe stages) for the $\mathcal{DFG}$ and a $\mathcal{CL}$ in Figure 6. We first determine the commonality factor of all vertices and map each vertex to the fastest component, i.e. all multiplier vertices to $Mpy1$ and all adder vertices to $Add1$. Next, we determine the $BestAssignment$ and the $weight$ of all vertices (shown in the table in Figure 6(b)).

After evaluating all vertex weights, the vertices are arranged in the order of decreasing weights, and the first vertex, that is, the one with the highest weight is selected to be slowed down. This is node d in the example (shown as the boxed entry in Figure 6(b)). However, with this slow down and with a *PS delay* constraint of 30 ns, the DFG can only be pipelined in 3 stages of delay 10, 30 and 10 ns each. Since this is not acceptable, the slow down is rejected and we look for the next $BestAssignment$ for vertex d with a delay less than 30 ns. $Add2$ satisfies these properties. We update $weight(d)$ to 1.0 ($20/(10\times2)$) and return it to the list.

In the next iteration, either of nodes a, b, and c can be selected since they all have the same weight of 1.25. First node a is selected to be replaced by $Mpy2$. Since the graph is successfully pipelined into 2 stages, one of delay 30 ns and the other 20 ns, the move is accepted. The next $BestAssignment$ for a, $Mpy3$, has a delay (40 ns) greater

Algorithm Component_Selection_and_Pipelining
Determine commonality factor of all vertices.
Map each vertex, v, to the fastest (or least delay component) of type $VertexType(v)$.
Determine the $BestAssignment$ and all vertex weights.
Make a list of vertices in order of decreasing weights.
Loop until (empty list)
Assign the first vertex in the list to $current\_vertex$.
Pipeline the $\mathcal{DFG}$, and evaluate performance.
Are performance constraints met?
If (no)
Do not "accept" this change.
Else If (yes)
"Accept" this change.
End If
Update_Vertex_List($current\_vertex$).
End Loop
End Algorithm

Procedure Update_Vertex_List($current\_vertex$)
Find the next $BestAssignment$ for $current\_vertex$.
If ($not\ found$ OR $not\ acceptable$)
Remove $current\_vertex$ from list.
Else if ($found$ AND $acceptable$)
Update $current\_vertex$ weight and return to list, maintaining the sorted order.
End if
End Procedure

Figure 5: Pseudo code of the combined component selection and pipelining algorithm.

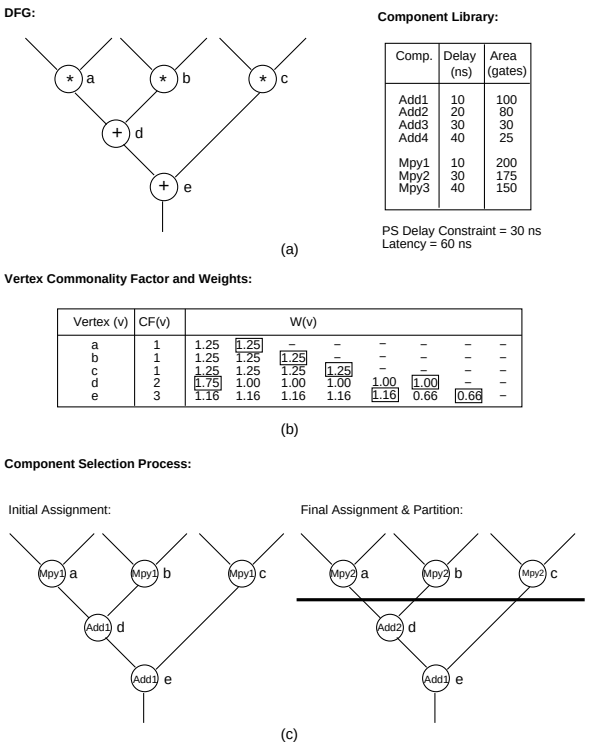


Figure 6: A walk-through example to illustrate the component selection and pipelining algorithm.

than the $PS\ delay$ ($30\ ns$), hence node a is dropped from the list (indicated by a "-" in the table). Vertices b and c undergo the same process. In the fifth iteration, vertex e is selected to be replaced with $Add3$ - this too is not accepted since it violates constraints. Next, node d is replaced with Add2 and removed from the list, and in the final iteration, node e is also removed from the list. The algorithm then terminates, since there are no nodes left to consider. The final $Assignment$ and $Partition$ is shown in Figure 6(c).

5 Experimental results

We have implemented the component selection and pipelining algorithms in C on a SUN SPARC station. The component selection algorithm has a complexity of $O(N^2C)$ where N is the number of vertices in the $\mathcal{DFG}$, and C is the maximum number of implementations of any operator type in the given component library. The pipelining algorithm has a complexity of $O(N)$ and the combined algorithm for component selection and pipelining also has a complexity of $O(N^2C)$.

In all our experiments we have used a modified version of the DTAS library [9] shown in Table 1 for multiplier and adder/subtractor components. Component cost is in terms of the number of equivalent ND2 (2-input NAND) gates from the LSI Logic Library, while delay is in ns.

We have conducted two types of experiments:

Experiment #1 demonstrates the quality of results produced by the component selection algorithm by comparing it with optimal results produced by an exhaustive search. These results have been limited to fairly small sized examples (the HAL benchmark and an 8th-order FIR filter) because the exhaustive search takes exponential time ($O(C^N)$), which becomes prohibitive for larger examples (even after pruning the search space).

Experiment #2 demonstrates the importance of the commonality factor during the vertex weight assignment for component selection. This experiment has been conducted on the 5th-order elliptical-wave filter benchmark.

TABLE 1
MODIFIED DTAS COMPONENT LIBRARY

Component Type	Component Name	Delay (ns)	Cost (eqv. ND2 gates)
*	Mpy1	57.97	2368
*	Mpy2	44.21	2400
*	Mpy3	36.21	2600
*	Mpy4	32.98	2710
*	Mpy5	28.57	2978
*	Mpy6	25.00	3500
*	Mpy7	22.50	4000
*	Mpy8	20.50	4500
+/-	Add1/Sub1	25.80	62
+/-	Add2/Sub2	20.00	125
+/-	Add3/Sub3	13.50	187
+/-	Add4/Sub4	10.00	250
+/-	Add5/Sub5	5.50	375
+/-	Add6/Sub6	3.00	500

5.1 Experiment #1: Quality of results

In order to measure the quality of results produced by the component selection algorithm, we coded an exhaustive algorithm that gives the optimal solution since it tries all possible combinations of vertices and components, and

selects the one with minimum cost within performance constraints.

We executed the two algorithms for the HAL benchmark and the FIR filter. While our algorithm took a few seconds on a SUN SPARC, the exhaustive algorithm took several days on some of the examples. The results of both algorithms are presented in Table 2. The "PS Delay Constraint" column gives the constraint we specified to the two algorithms, while the "PS Delay" columns give the *PS delay* of the designs produced by the two algorithms. Each example was evaluated for 8 different *PS delay* constraints. For the HAL benchmark, our algorithm produced designs with an area that was, at worst, 0.1% higher than those produced by the exhaustive algorithm. For the FIR filter the two algorithms gave identical results except in two cases, one in which the design produced by our algorithm was 0.01% more costly and the other in which it was 0.7% more costly.

We have been unable to compare our results with those produced by other algorithms since most algorithms assume a single implementation of components. Though TBS [6] is an exception, it combines component selection with scheduling rather than with pipelining. We attempted to compare our results for the elliptical filter benchmark; whereas our algorithm produces designs with a *PS delay* of as low as 200 *ns*, the fastest design that TBS produces has a delay of 1700 *ns*. This is an unfair comparison - it simply serves to corroborate the efficiency of pipelined designs over non-pipelined designs.

TABLE 2
OUR ALGORITHM VS. AN EXHAUSTIVE ALGORITHM

Example	PS Delay Constraint (*ns*)	PS Delay (*ns*)		Cost (*ND2gates*)		% error in Cost
		Our Alg.	Exh. Alg.	Our Alg.	Exh. Alg.	
HAL	71	70.5	70.5	28062	28062	0.0
	90	88.6	89.5	20452	20438	0.06
	110	109.3	109.5	17525	17525	0.0
	130	129.9	129.9	16222	16207	0.1
	150	149.4	149.4	15567	15567	0.0
	170	169.9	169.9	15054	15054	0.0
	200	199.5	199.4	14709	14709	0.0
	240	237.6	237.6	14488	14488	0.0
FIR Filter	40	37.6	37.6	13912	13912	0.0
	50	47.7	47.7	12150	12150	0.0
	70	68.7	68.7	10724	10724	0.0
	90	87.7	89.0	10287	10286	0.01
	100	97.0	97.0	10098	10098	0.0
	110	109.3	109.3	9973	9973	0.0
	130	121.6	129.6	9848	9783	0.7
	140	135.4	135.4	9720	9720	0.0

5.2 Experiment #2: Effectiveness of commonality factor

In Section 5 we gave an intuitive explanation of the importance of the commonality factor in assigning vertex weights during component selection. To get a quantitative measure of this importance we conducted an experiment to compare two cases for the 5th-order elliptical wave filter benchmark: *Case 1*, which uses the commonality factor as described in Section 5, and *Case 2*, which assigns all vertices a commonality factor of 1, thereby removing its effect from the vertex weight formula given by equation (5). Table 3 presents results obtained for several different *PS delay* and pipe stage constraints. For most constraints, *Case 1* produces results that are far superior than those produced by *Case 2*, and in some cases the ratio of *Case 1:Case 2* is even as high as 2.5, indicating the importance of the commonality factor.

TABLE 3
EFFECT OF COMMONALITY FACTOR (CF)

Example	PS Delay (*ns*)/ Pipe Stage Constraint	Cost (*ND2gates*)		% difference in Cost
		Case 1: With CF	Case 2: Without CF	
5th-order Elliptical Wave Filter	35 / 2	6809	6999	2.8
	50 / 2	3806	5498	44.5
	75 / 2	1680	3680	119.0
	100 / 2	1428	3365	135.6
	125 / 2	1178	2802	137.9
	150 / 2	1302	1428	9.7
	175 / 2	1364	1364	0.0
	35 / 3	5932	6373	7.4
	50 / 3	2056	4308	109.5
	75 / 3	1553	3178	104.6
	100 / 3	1178	2053	74.3
	125 / 3	1302	1302	0.0
	150 / 3	1364	364	0.0

6 Conclusions

To summarize, we have presented a cost-optimized algorithm for pipelining a data flow graph and selecting components, given a multiple implementation library and throughput and latency constraints. This allows our designs to use fast components only for critical operations and the slower components for less critical ones. To test our component selection algorithm, we have compared its results with optimal results produced by exhaustively enumerating all possible designs. For the examples considered our algorithm gave results that were no more than 0.7% off from the optimal result. Whereas the exhaustive algorithm has an exponential time-complexity of $O(C^N)$ and took several days to execute on some of these examples, our algorithm has a polynomial time-complexity of $O(N^2C)$ and executed in less than a second for these examples.

References

[1] N. Park and A. C. Parker, "Sehwa: A software package for synthesis of pipelines from behavioral specifications," *IEEE Transactions on Computer Aided Design*, vol. 7, pp. 356–370, Mar. 1988.

[2] K. S. Hwang, A. E. Casavant, C.-T. Chang, and M. A. d'Abreu, "Scheduling and hardware sharing in pipelined data paths," in *Proceedings of the IEEE International Conference on Computer Aided Design*, pp. 24–27, 1989.

[3] C.-T. Hwang, Y.-C. Hsu, and Y.-L. Lin, "PLS: A scheduler for pipeline synthesis," *IEEE Transactions on Computer Aided Design*, vol. 12, pp. 1279–1286, Sept. 1993.

[4] R. Jain, A. Parker, and N. Park, "Module selection for pipelined synthesis," in *Proceedings of the 25th Design Automation Conference*, pp. 542–547, 1988.

[5] R. Jain, A. Parker, and N. Park, "MOSP: Module selection for pipelined designs with multi-cycle operations," in *Proceedings of the IEEE International Conference on Computer Aided Design*, pp. 212–215, 1990.

[6] L. Ramachandran and D. D. Gajski, "An algorithm for component selection in performance optimized scheduling," in *Proceedings of the IEEE International Conference on Computer Aided Design*, pp. 92–95, 1991.

[7] A. H. Timmer, M. J. M. Heijligers, L. Stok, and J. A. G.Jess, "Module selection and scheduling using unrestricted libraries," in *Proceedings of the European Design Automation Conference*, pp. 547–551, 1993.

[8] S. Bakshi and D. D. Gajski, "A component selection algorithm for high-performance pipelines," Tech. Rep. 94-01, Dept. of Information and Computer Science, University of California, Irvine, 1994.

[9] N. D. Dutt and J. R. Kipps, "Bridging high-level synthesis to RTL technology libraries," in *Proceedings of the 28th Design Automation Conference*, 1991.