A Component Selection Algorithm for High-Performance Pipelines

Smita Bakshi
Daniel D. Gajski

Technical Report #94-01
June 15, 1994

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717-3425
(714) 856-7063

sbakshi@ics.uci.edu
gajski@uci.edu

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 report presents a cost-optimized algorithm for selecting components and pipelining a data flow graph, given a multiple-implementation library, and throughput and latency constraints. Results on several DSP examples indicate the importance of component selection as a parameter in design space exploration.

Contents

List of Figures

1 Introduction

In exploring the design space of high-performance pipelines, three design features play a significant role: *architecture*, *pipelining*, and *component selection*. A large number of design alternatives can be first evaluated by varying the component selection and the number of pipe stages of a given architecture. The exploration can be further increased by repeating the component selection and pipelining for a variety of different architectures.

Generally speaking, the architecture of a design refers to the type and number of its components and their interconnectivity, where the number of components in a design gives an indication of its "parallelism". A "parallel" architecture is one that exploits the inherent parallelism in a specification by computing several operations at the same time. While parallelism improves design performance, it also results in relatively expensive designs.

The parallelism of an architecture is illustrated with the help of a 4th-order (P=4) FIR filter shown in Figure 1. Consider the two designs in Figure 1(b). Design 1 has a higher degree of parallelism than Design 2, since it can compute four multiplications in parallel, while Design 2 can perform only one multiplication at a time. While Design 1 computes an output in just one iteration (or one pass through the datapath), Design 2 requires four iterations or passes. Thus, for this example, higher design parallelism results in higher costs and lower execution times.

The second design feature, pipelining, is another means of increasing design performance for a relatively small overhead in terms of pipelining register costs. This feature is all the more significant for DSP computations since they are regular and repetitive in nature, and yield well to pipelining techniques.

Pipelining is illustrated for the FIR filter example in Figure 1(c). We obtain a second level of exploration by pipelining each of the architectures in Figure 1(b) in different ways and into a different number of stages. For instance, Design 1 can be pipelined into 2 or 3 stages resulting in Designs 3 and 4, respectively.

The third design feature, component selection, adds yet another level of exploration. It involves selecting components from a realistic library with more than one implementation per operator, such that slow components are used on non-critical paths, while the faster and more expensive components are used only when necessary, on critical paths.

Component selection is illustrated for the FIR filter in Figure 1(d) where Designs 5 and 6 are obtained from Design 3 by using a different selection of adders and multipliers from a given library. Assume that the library consists of 2 multipliers, $M1$ and $M2$, (area of 200 and 400 gates and delay of 80 and 60 ns, respectively) and 2 adders, $A1$ and $A2$, (area of 50

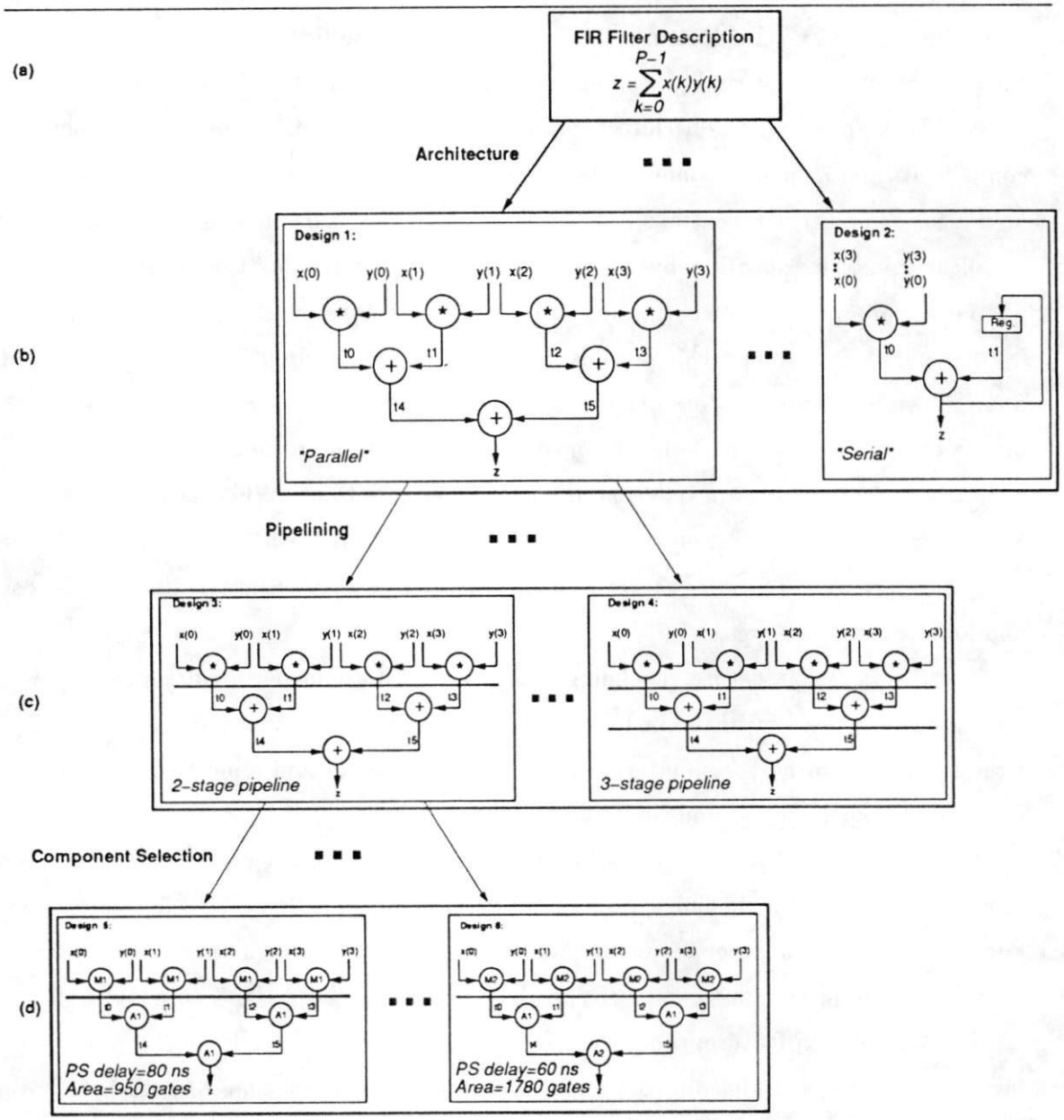


Figure 1: Illustrating the exploration of a 4th-order (P=4) FIR filter obtained by varying the architecture, pipelining, and component selection.

and 80 gates and delay of 40 and 20 ns, respectively). Using this library the cheapest design for a pipe stage (PS) delay constraint of 80 ns is obtained with the component selection shown in Design 5, and for a PS delay constraint of 60 ns by the component selection in Design 6. Note that a design may contain different implementations of the same operator. As an example, Design 6 has two instances of $A1$ and one of $A2$, instead of three instances of only $A2$, which would have resulted in a more costly design.

We have just illustrated that a designer can explore the design space in one, or a combination of three ways: (1) varying the architecture of the design, (2) selecting different components, and (3) pipelining the design in different ways and into a different number of stages. Our design strategy for varying these three parameters works as follows: the designer first specifies the architecture by using a mix of behavior and structure in his specification. He then invokes an automated algorithm to take over the tasks of component selection and pipelining. This report presents an input format for specifying an architecture, and an algorithm for component selection and pipelining.

The report 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 describes our architecture specification with the help of an example. Section 4 gives a formal definition of the component selection and pipelining problems while Section 5 describes our proposed algorithm for solving these problems. Section 6 presents results demonstrating the quality of our algorithm and its application in exploring the design space of two industrial strength examples. Finally, Section 7 concludes the report with a summary of our major contributions.

2 Previous Work

Previous research in the area of pipelined synthesis has resulted in the development of tools such as Sehwa [10], the tools from the GE Corporate R&D Laboratories [6], and PLS, a pipelined scheduler [5]. These systems pipeline a given DFG so as to optimize a cost function while satisfying constraints on performance or area. For instance, the GE tools attempt to minimize the number of pipe stages and the component area while satisfying clock period and throughput constraints. Sehwa contains several resource and performance constrained scheduling algorithms, providing the user different options of arriving at a pipelined design. For example, the user could specify a throughput and latency constraint, as well as allocate resources, and Sehwa would produce a schedule, if feasible. Or the user could specify a clock cycle limit and ask for the shortest or most expensive schedule.

The aim of our algorithm is to minimize the component area, given throughput and latency constraints; thus, the point of similarity in our work and the above mentioned tools is in the constraints and the cost function. The difference arises in the nature of the component library that is used for synthesis. The above mentioned tools use a library that contains a single implementation for all operators of a given type, that is, they start with a pre-selected set of components with only one implementation per operator. This forces them to use the same component on non-critical and critical paths, resulting in designs that are inefficient and more costly. This is unlike our algorithm which uses a library containing more than one implementation for each operator. Hence we are able to arrive at more efficient designs by using fast implementations for critical operations and slower implementations for the non-critical ones.

The authors of SLIMOS [7] and MOSP [8] define module selection as the process of selecting a single implementation for all operators of a given type, from a library that may contain several implementations corresponding to that operator type. The selected implementation is then used to perform *all* the operations of that type in the design. For instance, out of five different adder implementations the SLIMOS and MOSP algorithms may determine the carry-look-ahead adder to be best suited for the design, and then proceed to use this adder implementation for all the add operations in the design. Thus although, the algorithms start with a multiple-implementation library, their selected module set contains *just one implementation per operator.* This differs from our algorithm which, for instance, may select the carry-look-ahead implementation for those add operations that are on a critical path and need to be completed soon, and perhaps a ripple-carry-adder for other non-critical add operations.

The TBS [11] algorithm as well as the module selection algorithm presented in [12], use unrestricted libraries that allow multiple physical implementations for the same operator. However, these algorithms combine scheduling and component selection, whereas we combine pipelining and component selection. The difference in the two algorithms arises because stages in a pipelined datapath execute concurrently each on their own set of components, whereas "states" in a scheduled data flow graph execute sequentially on the same set of components. Thus, while selecting components for a state i, the TBS algorithm, for instance, attempts to use the components that it had already selected for previously scheduled states, $1 \cdots (i-1)$. Our algorithm does not have this constraint since components across a pipe stage are not shared.

The MASS Synthesis approach [9] generates a minimum cost pipelined design, given

a CDFG, a module library, and throughput constraints. The module library may contain more than one implementation per operator, hence, for instance, their final design could consist of three different adder implementations, two different multiplier implementations and so on. However, the authors have *not* demonstrated the ability to pipeline *and* select components for the same design. Their experimental results show (1) component selection for non-pipelined designs, and (2) pipelined synthesis with single implementation operators. Hence, they fall in the same category as TBS.

In summary, our algorithm [1] 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 Specifying Architectures

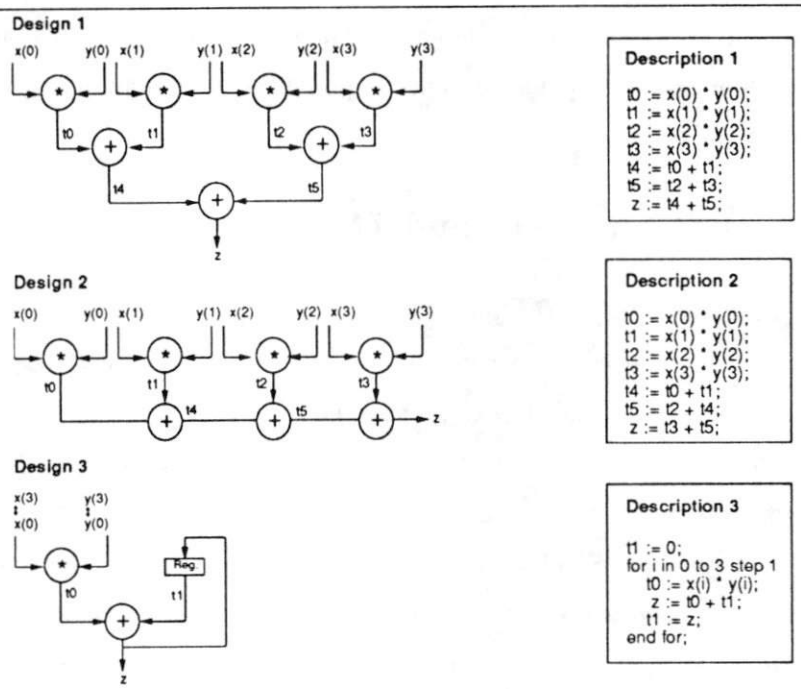


Figure 2: Illustrating the mix of behavior and structure in the input description of three 4th-order FIR filter designs.

Designers of DSP systems typically know the basic design topology or architecture they wish to use; however, they require assistance in time-consuming tasks such as pipelining and component selection. A pure structural description is inappropriate since it would require

them to know the complete structure of the design. On the other hand, a pure behavioral description is also inappropriate since it would prevent them from specifying the design topology they have in mind. Thus, we propose an input format that allows a mixture of both structure and behavior in the specification. Figure 2 illustrates this feature by giving descriptions of three different 4th-order FIR filter architectures.

Designs 1 and 2 differ in their summation topology - one uses an adder "tree" and the other an adder "chain". This difference is brought out in the description by using appropriate assignment statements. Design 3 differs from Designs 1 and 2 in the number of multiply and add operators it contains. Designs 1 and 2, containing four multiply and three adder nodes each, require just one iteration per output, that is the four multiplications and three additions are computed in just one pass through the design. Design 3, however, contains just one multiply operator and hence it requires 4 iterations per output. In the description, this "behavior" (that is, the iterations) is specified by enclosing the design "structure" within a $for - loop$ statement. A designer can thus specify an architecture by mixing behavioral and structural constructs in this manner.

As stated previously, after a designer specifies an architecture manually, we utilize an algorithm for performing pipelining and component selection. This algorithm is discussed in the next few sections.

4 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 $\langle ComponentType$, $Area$ and $Delay \rangle$, and constraints on *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.

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 3 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.

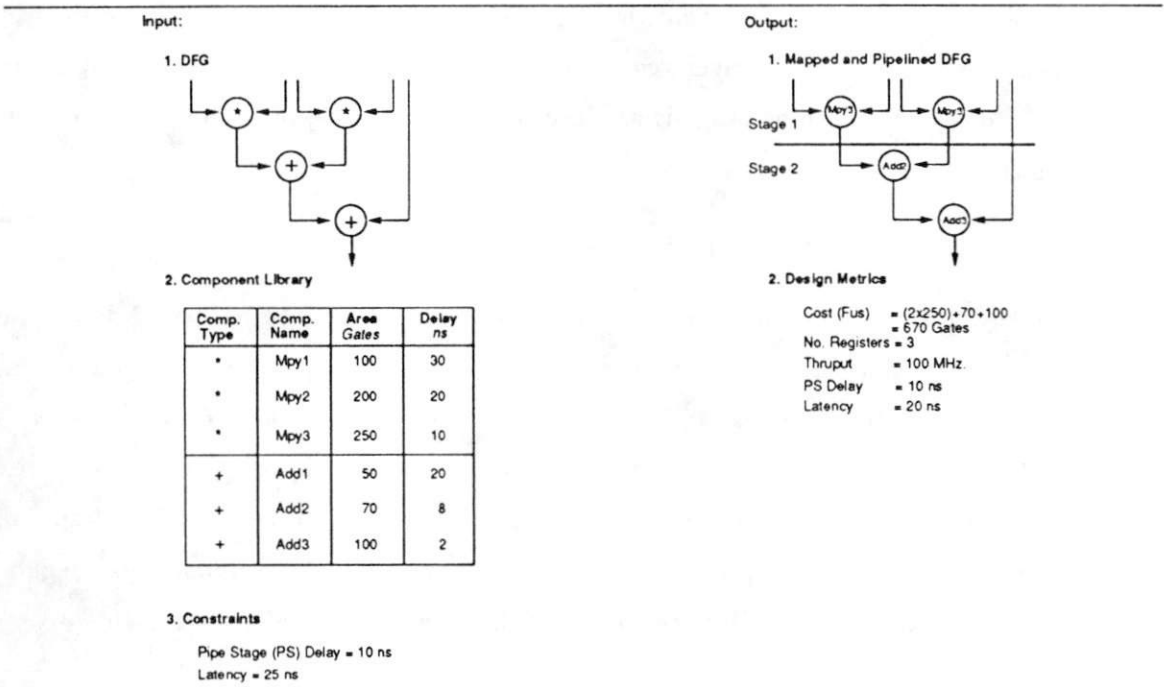


Figure 3: An example illustrating the inputs and outputs of the component selection and pipelining algorithms.

5 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.

5.1 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 4) 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.

5.2 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 5(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 ($[3\times100]+[2\times50]$) gates. Let the constraint on the PS delay be 50 ns.

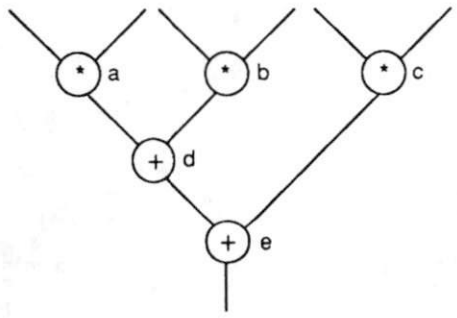
For the purposes of this explanation, let us assume that we slow down a vertex by replacing it with the next slower component in the library. We now have to pick the first vertex to slow down. Intuitively speaking, this should be the one that gives the highest cost

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 its performance.
8. **If** (*this slow down meets performance constraints*), accept it
9. **else**, reject it.
10. **Until** (*no vertex can be slowed down without violating constraints*).
11. **End if**

Figure 4: An overview of the component selection and pipelining algorithm.

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[1], 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.

DFG:



(a)

Component Library–1:

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

(b)

Component Library–2:

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

(c)

Figure 5: $\mathcal{DFG}$ and component libraries used to illustrate the vertex weight and the need for a "commonality factor".

If we had 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

[1]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 5 has the following I-O paths: $a - d - e$, $b - d - e$, and $c - e$.

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).

The area-delay gain

The components considered in the library in Figure 5 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 5, if *Add*2 had a delay of 12 ns instead of 20 ns, the cost benefit of vertices d and e should really be [Area(A1)-Area(A2)]/[Delay(A2)-Delay(A1)] = (50-30)/(12-10) = 20/2, and of vertices a, b, and c it should be [Area(M1)-Area(M2)]/[Delay(M2)-Delay(M1)] = (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 5(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:

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

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

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.

This method is illustrated with the help of an example (Figure 6). As an initialization step all input nodes, $i1$ to $i5$, are assigned a FW of 0, and all operator nodes with *only* input

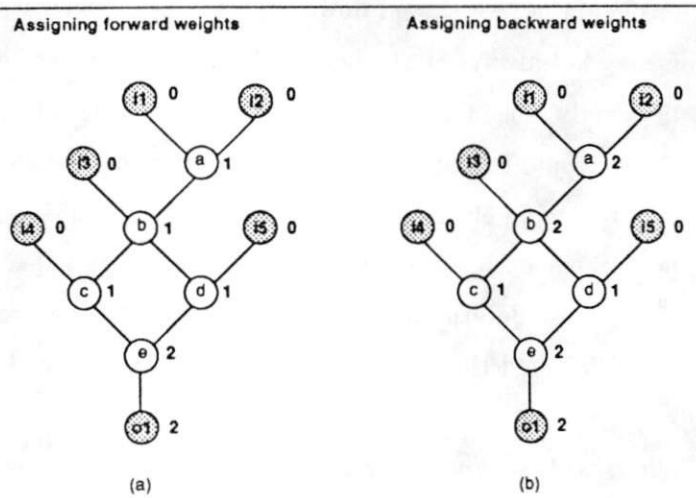


Figure 6: Determining the commonality factor by assigning a forward and backward weight to vertices.

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. As a first step in the example, the FW of a is propagated to b, since b is the only successor of a. The FW of b is then split equally amongst vertices c and d, resulting in a value less than 1. After rounding off, vertices c and d each get a FW of 1. Finally vertex e is assigned a FW of 2, one each from c and d.

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 (that is, FW(c):FW(d)), resulting in a BW assignment of 1 each. The BW of c, split amongst $i4$ and b in the ratio 0:2, results in assigning $i4$ a BW of 0, and b, a partial BW of 1. Similarly, when the BW of d is split amongst b and $i5$ in the ratio 2:0, it results in assigning b a value of 1, bringing its total BW to 2. The BW of b is finally propagated to $i3$ and a in the ratio 0:1.

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 4). 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.

5.3 Pipelining

Given a $\mathcal{DFG}$, an *Assignment* for the $\mathcal{DFG}$, and a Pipe Stage (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.

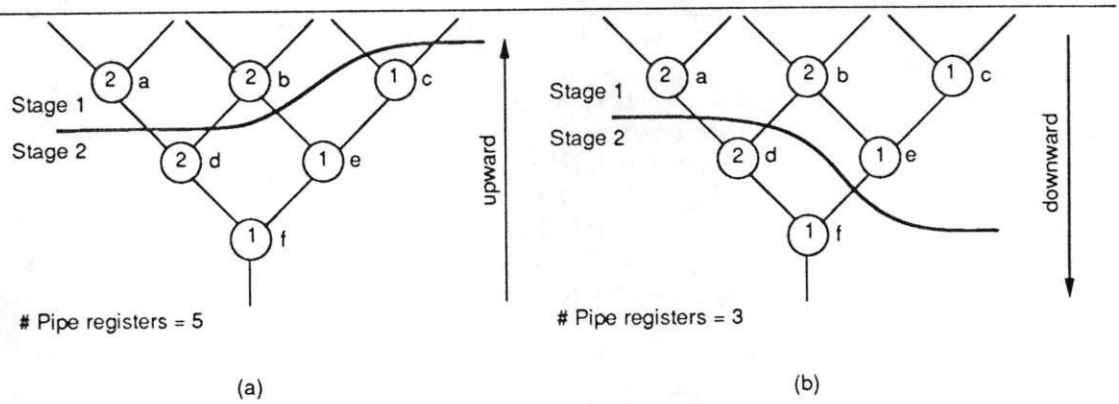


Figure 7: Downward and upward traversal for pipelining a $\mathcal{DFG}$.

The algorithm is illustrated on the mapped $\mathcal{DFG}$ in Figure 7. Let the PS delay constraint be 3 *ns* (the number in a vertex indicates its delay). We start the upward traversal from vertex f and place the pipe stage partition after vertices d, e and c as shown, since the accumulated delay exceeds 3 *ns* after these vertices. Traversal continues from vertex a and b, but since the delay does not exceed 3 *ns* a new partition is not placed.

In the downward traversal, we start from the input nodes and place a partition between vertices a and d (and b and d) since including d would give a delay of 4 *ns*, which violates the constraint. Similarly, the partition is placed between vertices e and f since the cumulative delay of b and e is 3 *ns*.

It is to be noted that an upward traversal yields a pipeline with 5 registers, and a downward traversal yields a pipeline with 3 registers. Hence, the latter is selected.

```
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 the weight of all vertices.
    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 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 8: Pseudo code of the combined component selection and pipelining algorithm.

5.4 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 8) 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 9. 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 9(b)). As an example, consider vertex d: its commonality factor is 2 since it appears on two distinct I-O paths, $a-d-e$ and $b-d-e$, and its *BestAssignment* is $Add3$ since that gives the highest area-delay gain of 70/20, as opposed to 20/10 for $Add2$, and 75/30 for $Add4$. The weight of vertex d evaluated according

to equation (5) is then $70/(20\times2) = 1.75$.

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 9(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. *Add*2 satisfies these properties. We update *weight*(d) to 1.0 ($20/(10\times2)$) and return it to the list.

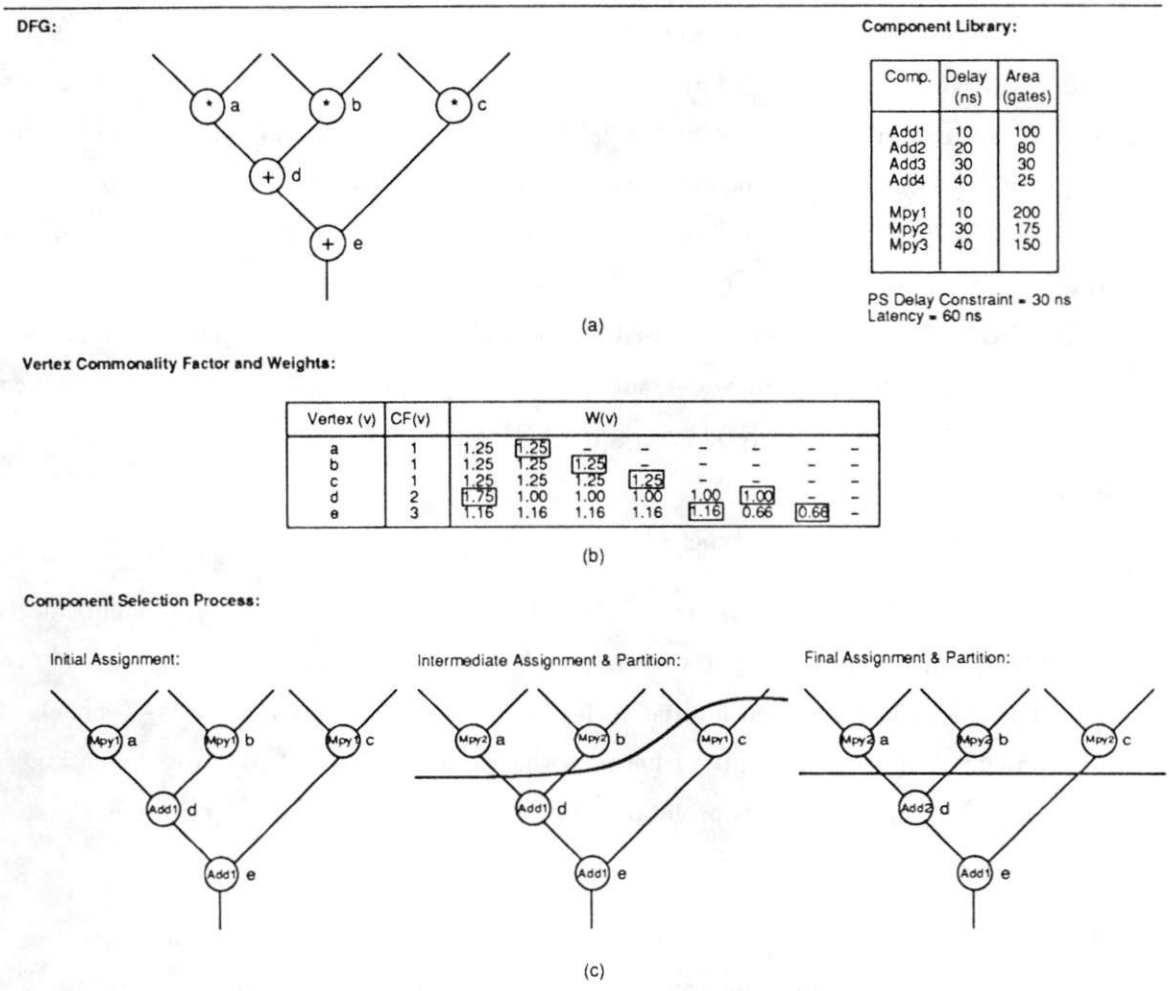


Figure 9: A walk-through example to illustrate the component selection and pipelining algorithm.

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 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 9(c).

6 Experimental Results

We have implemented the component selection and pipelining algorithms using 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 [3] 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 three 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.

Experiment #3 demonstrates the results of applying our exploration strategy and algorithms for varying the architecture, pipelining and component selection of two industrial-strength DSP systems, a Beamformer and an Inverse Discrete Cosine Transform ($IDCT$). This experiment indicates that a large design space can be explored

within a matter of seconds and a designer can quickly narrow down to the most desirable design by studying tradeoffs between throughput, latency and cost.

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

6.1 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 [11] 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.

Similarly, an attempt to compare our results with MASS [9] fails since they have only provided results of pipelined designs using reduced libraries, that is libraries with single implementation components.

TABLE 2
OUR ALGORITHM VS. AN EXHAUSTIVE ALGORITHM

Example	PS Delay Constraint (ns)	PS Delay (ns)		Cost ($ND2gates$)		% error in Cost $\frac{our-exh}{exh} \times 100$
		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

6.2 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 the following two cases for the 5th-order elliptical wave filter benchmark:

Case 1: uses the commonality factor, as described in Section 5.

Case 2: assigns all vertices a commonality factor of 1, thereby removing its effect from the vertex weight formula given by equation (5).

Table 3 and Figure 10 present 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. However, in cases where the PS Delay constraint is high, the results produced by both cases are about the same. This is because, for large PS Delay values, most of the nodes in the $\mathcal{DFG}$ are mapped to the slowest components in the library, and the order in which the nodes are mapped (as determined by the commonality factor) is then unimportant.

TABLE 3
DEMONSTRATING THE EFFECT OF COMMONALITY FACTOR (CF)

Example	PS Delay Constraint (ns)	Pipe Stage Constraint	*Case 1*: With CF		*Case 2*: Without CF		% cost difference $\frac{Case2-Case1}{Case1} \times 100$
			PS Delay (ns)	Cost ($ND2gates$)	PS Delay (ns)	Cost ($ND2gates$)	
5th-order Elliptical Wave Filter	35	2	34.0	6809	34.5	6999	2.8
	50	2	49.3	3806	47.0	5498	44.5
	75	2	74.8	1680	74.0	3680	119.0
	100	2	98.1	1428	92.1	3365	135.6
	125	2	116.7	1178	116.7	2802	137.9
	150	2	129.0	1302	129.0	1428	9.7
	175	2	154.8	1364	154.8	1364	0.0
	35	3	34.3	5932	33.5	6373	7.4
	50	3	39.3	2056	49.3	4308	109.5
	75	3	65.1	1553	72.8	3178	104.6
	100	3	77.4	1178	77.4	2053	74.3
	125	3	103.2	1302	103.2	1302	0.0
	150	3	129.0	1364	129.0	1364	0.0

6.3 Experiment #3: Design exploration

We now apply the general exploration strategy discussed in Section 1, and the component selection and pipelining algorithm presented in Section 6, on two fairly large DSP systems, a 2-Dimensional 8×8 IDCT [4], and a 4-element, 4-beam Beamformer [2]. For both

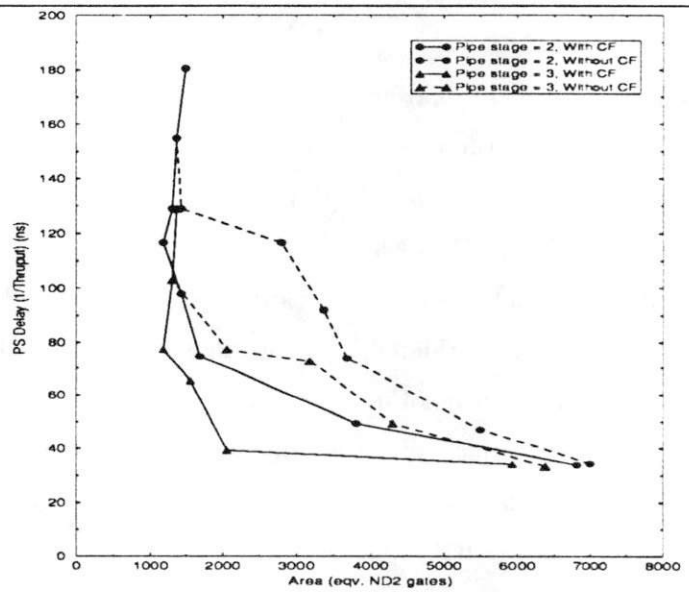


Figure 10: Demonstrating the effectiveness of the commonality factor (CF) on the 5th-order Elliptical Wave filter by considering two cases: *Case 1*, with CF, and *Case 2*, without CF (or CF=1).

examples, we wrote three descriptions representing different architectures. Each of the descriptions was then pipelined into a different number of stages and components selected such that throughput constraints were satisfied and the cost/area was minimized.

The results of the exploration have been presented as a trade off between throughput and area for (1) different architectures, with a fixed latency, and for (2) a fixed architecture, with varying latency. Thus, for instance if a designer has a "hard" or fixed constraint on the throughput of the system that he must satisfy, he can get a good idea of the architecture he should be considering, by looking at the first graph. After narrowing down to one, or possibly two architectures, he can then study the effect of latency, and pick a design point that best optimizes his cost, which could be a function of area or latency or both.

In all the graphs below, the y-axis represents PS Delay, or effectively, the inverse of throughput, while the x-axis represents area in ND2 gates.

IDCT

The 2-D IDCT, used to reconstruct compressed images, is represented by the following equation:

$$bd(x,y) = \sum_{u=0}^{N-1}\sum_{v=0}^{N-1} BD(u,v)F(u,x)F(v,y) \tag{6}$$

Except for N, the order of the IDCT, all other terms in the equation are irrelevant to the understanding of our results. N is 8 in our examples, implying that approximately 64 additions and 64 $\times$ 2 multiplications need to be computed for every output $bd(x,y)$, where $x, y \in 0 \ldots 7$.

The algorithm we chose for evaluating equation (7), consists essentially of 3 $N \times N$ matrix multiplications. We considered 3 architectures, that compute 8×8 matrix multiplications with different "extents of parallelism". Design 1 consists of 1 basic block (BB), Design 2 of 8 BBs, and Design 3 of 32 BBs, where a BB is a block used to evaluate one term of the product matrix. Note that Design 3, consisting of about 500 nodes (15 nodes per BB), is the largest design we have considered, and our algorithm for pipelining and component selection took less than a second for this example.

Figure 11 (log-log scale) depicts all three topologies on the same graph (log-log scale) [2]. The y-axis represents the delay per sample (or the inverse of throughput), where a sample consists of the 64 terms of the BD input matrix. The latency of all three architectures is fixed, and equal to the PS Delay.

As can be seen, a large design space has been explored by varying the architecture and component selection alone, where the delay of the designs ranges from about 17,000 ns to less than 100 ns (or approximately 60 KHz. to 10 MHz.), while the cost varies from about 20,000 to 1,000,000 gates (though designs are not evenly spread in this range).

[2] Note that the design points are joined by a curve for purposes of graph readability; this does not imply a continuous design space.

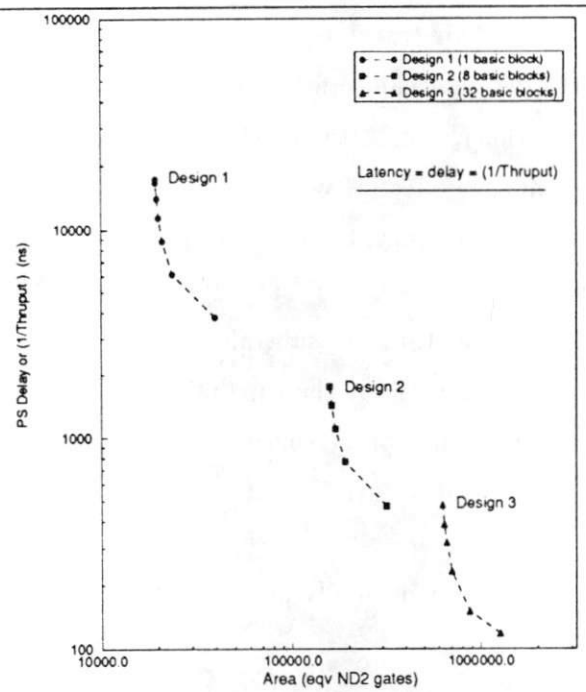


Figure 11: PS Delay vs. Area of 3 different architectures with fixed latency for an 8×8 IDCT.

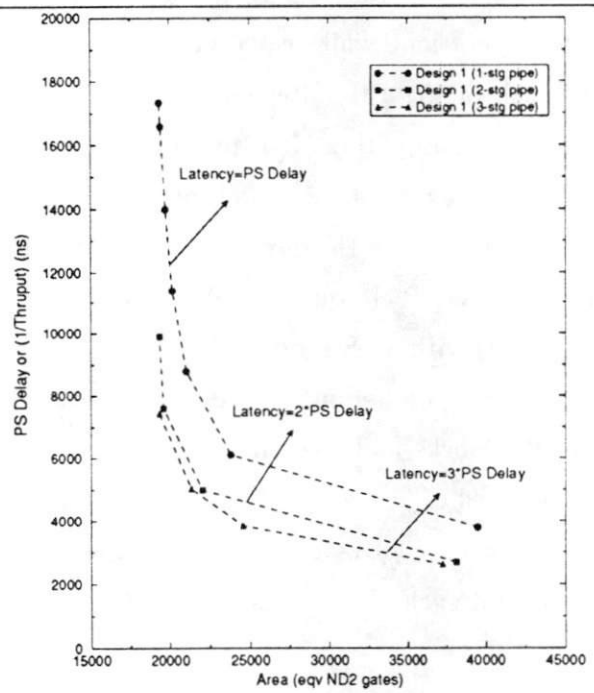


Figure 12: PS Delay vs. Area of Design 1 for 8×8 IDCT, with latencies of 1, 2 and 3 ($\times$ PS Delay).

Lets consider a typical design scenario, where the designer places a throughput constraint of, say, 0.2 MSamples/sec. This translates to about 5000 ns per sample. By looking at the graph in Figure 11, the designer can immediately narrow down to Design 1, since no other design has delays within this range. After selecting the design, he can then turn to the graph in Figure 12 that presents the effect of varying the latency of Design 1. Assuming that the designer can tolerate a large latency, he/she can then pick a 3-stage pipelined design so as to minimize the total area.

We would like to point out that the substantial exploration, from 100 ns to 17,000 ns, would not have been possible without the capability of using multiple implementations of operators in the design. Had our library consisted of just one implementation per operator, Figure 11 would have consisted of just 3 points, one for each architecture.

Beamformer

The beamforming problem is formally described by the following equations:

$$y_e^b(i) = \sum_{k=0}^{P-1} D_e(i-k)c_e^b(k),\ \forall b \in 1 \ldots M, \forall e \in 1 \ldots N \tag{7}$$

$$R^b(i) = \sum_{e=1}^{N} y_e^b(i)\omega_e^b,\ \forall b \in 1 \ldots N \tag{8}$$

We will not elaborate further on these equations. Suffice it to say, that equation (7) represents a FIR filter operation, while equation (8) involves a phase rotation, that is the multiplication of each of the FIR filter outputs with a constant (ω_e^b), and then the summation of these products. Equation (7) is repeated for all "elements" ($e \in 1 \ldots N$) and "beams"($b \in 1 \ldots M$), while equation (8) is repeated for all "beams". The "size" of the Beamformer is thus characterized by the number of elements, N, the number of beams, M, and the order of the FIR filter, P. In our analysis, we have considered a 4-element (N=4), 4-beam Beamformer (M=4) with an 8th-order (P=8) FIR filter.

Once again we considered 3 different architectures that differed in the number of FIR filters, the number of PR blocks and the number of adders in the subsequent summation operation. Design 1 consisted of just one FIR filter and PR block (all FIR filter blocks were implemented with 8 multipliers and 7 adders). Design 2 consisted of 2 FIR and PR blocks and Design 3 consisted of 4 such blocks. Each of these designs was also pipelined into 2 and 4 stages.

The results of varying the design topology for a fixed latency are given in Figure 13, and the effect of varying the latency of Design 2 is shown in Figure 14. Once again, a large design space ranging from a throughput of 4000 ns to 100 ns, and a cost of 25,000 to

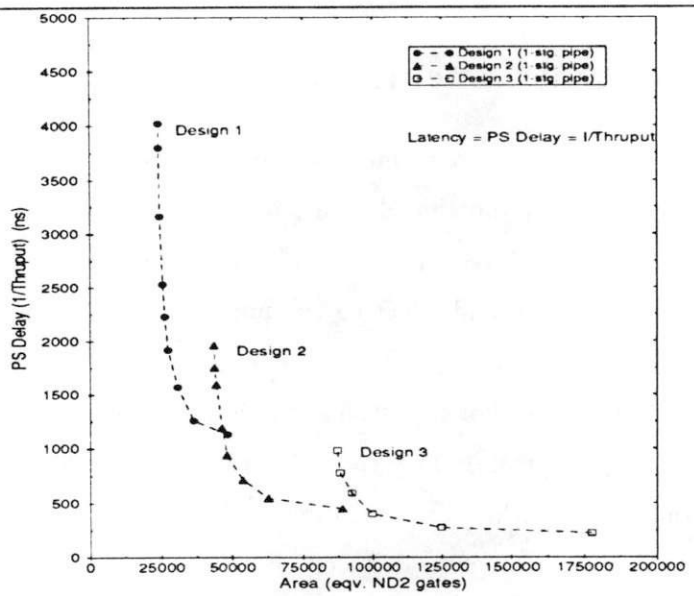


Figure 13: PS Delay vs. Area of 3 different architectures with fixed latency for a 4-element, 4-beam Beamformer system.

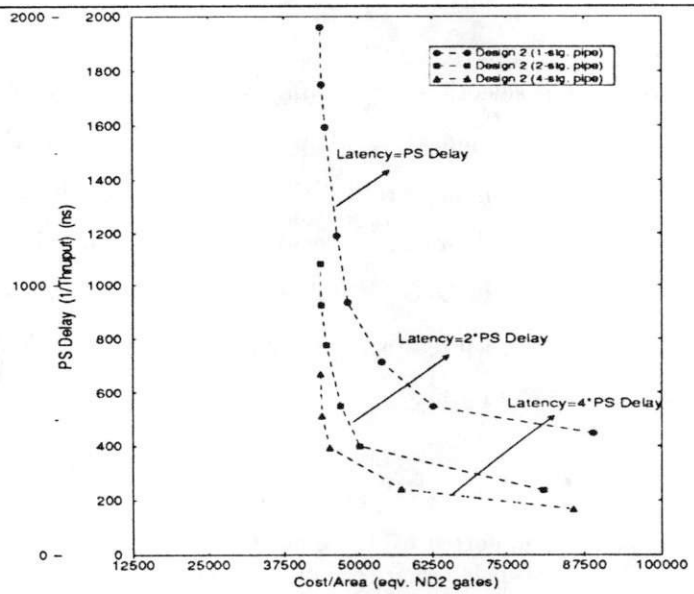


Figure 14: PS Delay vs. Area of Design 2 for a 4-element, 4-beam Beamformer system, with latencies of 1, 2 and 4 ($\times$ PS Delay).

175,000 gates has been explored. This would not have been possible without the ability to vary all three parameters - architecture, component selection and pipelining.

7 Summary and Conclusion

To summarize, we have presented a method of exploring the design space of high-performance pipelines by varying three important design parameters: *architecture*, *pipelining*, and *component selection*. This is achieved by manually writing different descriptions, and then using our algorithms for pipelining and selecting components. We demonstrated the effectiveness of our exploration strategy by applying it on two industrial-strength DSP systems, the Beamformer and 2-D IDCT. For both the examples, we obtained a large spread of designs, ranging from a throughput of 100 ns to 4000 ns for the Beamformer and from 100 ns to 17,000 ns for the IDCT, within a matter of seconds.

From our experiments, we also deduced that component selection adds an important dimension to the design exploration of high-performance pipelines; had we used a limited or single implementation library, we would have obtained less than one-tenth the number of designs we obtained with a realistic library that had several different implementations per operator. The use of a realistic library also leads to more efficient designs (in terms of throughput per unit cost), since slow components can be then be utilized on non-critical paths, while faster (and more expensive) components can be used on critical paths only when necessary.

To test our component selection algorithm, we compared its results with optimal results produced by exhaustively enumerating all possible designs. For the examples considered (the HAL benchmark, and an 8th-order FIR filter) 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.

Acknowledgements

This work was partially supported by the Semiconducter Research Corporation grant #93-DJ-146, and we gratefully acknowledge their support. We also extend our gratitude to Mike Butler at MITRE Corporation for providing us with the Beamformer example and for his insightful comments on our exploration strategy. Finally, we would like to thank Vijay Nagasamy at LSI Logic for the IDCT example, and for our useful discussions on the important design parameters for DSP systems.

References

[1] Smita Bakshi and Daniel D. Gajski. A component selection algorithm for high-performance pipelines. In *Proceedings of EURO-DAC*, 1994.

[2] R.J. Byrne. Large array beamformers: Summary report. Technical report, The MITRE Corporation, Bedford, MA, 1991.

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

[4] Paul M. Embree and Bruce Kimble. *C Language Algorithms for Digital Signal Processing.* Prentice Hall, Inc, Englewood Cliffs, New Jersey 07632, 1991.

[5] C.-T Hwang, Y.-C. Hsu, and Y.-L. Lin. PLS: A scheduler for pipeline synthesis. *IEEE Transactions on Computer Aided Design*, 12(9):1279–1286, September 1993.

[6] Ki Soo Hwang, Albert E. Casavant, Ching-Tang Chang, and Manuel A. d'Abreu. Scheduling and hardware sharing in pipelined data paths. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 24–27, 1989.

[7] Rajiv Jain, Alice Parker, and Nobhyung Park. Module selection for pipelined synthesis. In *Proceedings of the 25th Design Automation Conference*, pages 542–547, 1988.

[8] Rajiv Jain, Alice Parker, and Nobhyung Park. MOSP: Module selection for pipelined designs with multi-cycle operations. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 212–215, 1990.

[9] Jer-Min Jou and Shiann-Rong Kuang. Library-adaptively integrated data path synthesis for DSP systems. In *Proceedings of the IEEE International Conference on Computer Design*, pages 379–382, 1993.

[10] Nohbyung Park and Alice C. Parker. Sehwa: A software package for synthesis of pipelines from behavioral specifications. *IEEE Transactions on Computer Aided Design*, 7(3):356–370, March 1988.

[11] Loganath Ramachandran and Daniel D. Gajski. An algorithm for component selection in performance optimized scheduling. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 92–95, 1991.

[12] Adwin H. Timmer, Marc J. M. Heijligers, Leon Stok, and Jochen A. G.Jess. Module selection and scheduling using unrestricted libraries. In *Proceedings of the European Design Automation Conference*, pages 547–551, 1993.