Component Selection in Resource Shared and Pipelined DSP Applications [†]

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

Abstract

In general, high-performance DSP designs are heavily pipelined and, in order to reduce the pipeline cost, these designs employ techniques such as component selection and resource sharing to select the appropriate number and type of components. In this paper, we present an algorithm to perform the three tasks of pipelining, resource sharing and component selection, so as to minimize design cost for a given throughput constraint. Experiments conducted on several examples demonstrate the superiority of performing all three tasks, rather than just a combination of any two of these tasks, as done in previously published algorithms.

1 Introduction

Designers of high-performance systems have focused their efforts on satisfying the conflicting goals of high-performance and low-cost. In general, high-performance constraints are met by **pipelining** the design into several concurrently executing stages, such that at any given time each pipe stage operates on a different sample from the stream of incoming samples. This concurrency or increased parallelism has two effects: firstly, it increases the throughput of the design, since a larger number of operations are now computed in parallel. Secondly, to handle these concurrent operations, it contains a larger number of components, thereby increasing the cost of the design. It is possible, however, to reduce this design cost by using techniques such as resource sharing and component selection that together select the number and type of resources that will satisfy constraints at minimal cost.

Resource sharing, also known as scheduling, refers to utilizing the same resource to perform several different operations over different time-steps. Since one resource can now potentially do the job of many, a fewer number of such resources are required, resulting in a lower cost. The design cost can be further reduced by combining resource sharing with **component selection**, which refers to the selection of operator implementations from a library containing multiple implementations per operator type. In general, by allowing multiple implementations within a design, fast components can be selected for critical operations, and at the same time, the slower and cheaper ones can be used for non-critical operations.

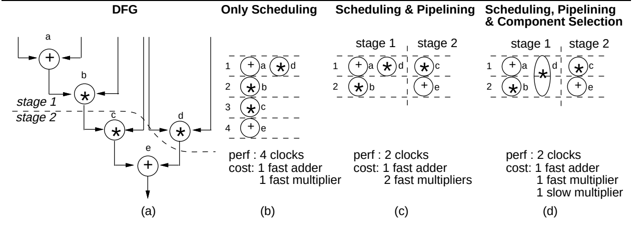


Figure 1: Effect of combining pipelining, scheduling, and component selection.

In this paper, we present an algorithm that combines all the three design tasks, pipelining, resource sharing and component selection, so as to satisfy high-performance requirements at low costs. If we only schedule the design we obtain a high resource utilization, but low concurrency, and consequently a low throughput. By introducing pipelining we increase the throughput, and by introducing component selection we further improve the resource utilization and hence reduce design cost.

In Figure 1, we demonstrate the benefit of performing all three tasks, rather than just a combination of two, for the data flow graph in Figure 1(a). The fastest design obtainable by only resource sharing or scheduling (Figure 1(b)) requires 4 clocks, assuming that all components have an execution delay of just 1 clock. If we now pipeline the design into 2-stages, where nodes a, b and d are executed in the first pipe stage and nodes c and e in the second stage, the fastest design obtainable will have a delay of only 2 clocks. However, this now requires 2, instead of 1, unit-delay multipliers (b can share the multiplier with either d or c; similarly, a and e can share the same adder). If we now allow component selection in the design (Figure 1(d)), we can select a slower multiplier for operation d, thus bringing down the total design cost, while maintaining the same delay.

The rest of the paper is organized as follows. In the next section, we discuss related research and also explain how we differ from it. Sections 3 and 4 present our definition of the problem and the algorithm we have used to perform pipelining, scheduling and component selection. Finally, we present results in Section 5, and conclude the paper in Section 6.

[†]This work was supported by the Semiconductor Research Corporation (grant #94-DJ-146) and by the National Science Foundation (grant CDA-9422095).

2 Previous work

Related research can be classified into three categories with respect to scheduling, pipelining and component selection. The first category consists of algorithms that pipeline and schedule a given DFG so as to optimize either the cost, the latency, or the throughput of the DFG, while at the same time, satisfying either performance or resource constraints. However, these algorithms do not perform component selection, that is, they use restricted libraries that contain only single implementations of components. The designs are thus forced to use the same component on non-critical and critical paths, resulting in designs that are inefficient and more costly. Examples that fall in this category are the algorithms in the Sehwa tools [1] and the tools from the GE Corporate R&D Laboratories [2].

The second category of algorithms perform scheduling using multiple implementation libraries; however, they do not pipeline the DFG, and hence are unable to obtain high throughput designs. Pipelining can give orders of magnitude higher throughput, which these algorithms are unable to achieve. Examples of such algorithms are TBS [3] and the algorithm by Timmer et al. [4].

The third category of algorithms perform pipelining and component selection without any resource sharing or scheduling. Thus, designs obtained from these algorithms have a one-to-one mapping between components and nodes in the DFG. Though these designs can satisfy high-throughput constraints, they may not be cost-optimal since they do not support resource sharing. Examples of such algorithms are the ILP approach presented in [5] and a heuristic-based selection technique presented in [6].

The algorithm presented in this paper differs from all the algorithms mentioned above since it performs resource sharing (or scheduling), pipelining *and* component selection, and hence it results in designs that not only satisfy high-throughput constraints, but do so cost-efficiently by sharing resources amongst operators and by using fast components on critical paths and the slower components wherever possible on non-critical paths.

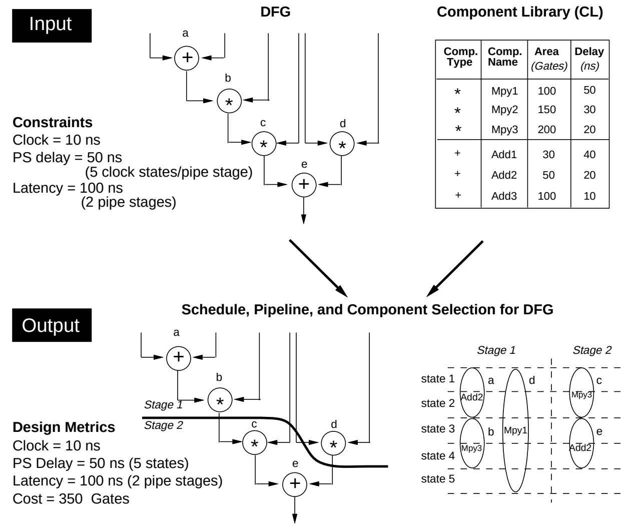
3 Problem statement and definitions

In this section, we first define the terms *Clock, PS delay* and *Latency* and then follow it with the definition of our problem.

Definition 1: *Clock* is the maximum delay of any scheduled state.

Definition 2: *PS delay* is the maximum delay of any pipe stage that also represents the sample inter-arrival delay, that is the delay between the arrival of two consecutive input samples. Furthermore, *PS delay* is a multiple of the clock delay ($m\times$clock, for an m-state per stage pipeline), since a pipe stage contains an integer number of scheduled states, each of which has a delay equal to the clock. *Throughput*, which is often the prime constraint on DSP systems, is the inverse of the *PS delay.*

Definition 3: *Latency* is the total execution time ($n\times PS$ *delay*, for an n-stage pipeline), that is, the time between the



Comp. Type	Comp. Name	Area *(Gates)*	Delay *(ns)*
*	Mpy1	100	50
*	Mpy2	150	30
*	Mpy3	200	20
+	Add1	30	40
+	Add2	50	20
+	Add3	100	10

Figure 2: An example illustrating the inputs and outputs of the component selection and pipelining algorithms.

arrival of an input sample and the availability of the corresponding output. The latency is thus a multiple number of *PS delays.*

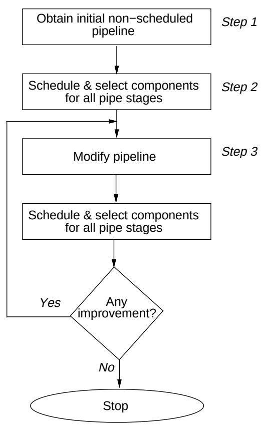
Our problem may be defined as follows:

> 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, $\mathcal{CL}$, comprising a set of three tuples $\langle ComponentType$, $Area$ and $Delay\rangle$, and constraints on clock, PS (Pipe Stage) *delay*, and latency, **pipeline** the DFG into $\lfloor$latency$/PS\ delay\rfloor$ stages of delay $\leq PS\ delay$, **schedule** each pipe stage into $\lfloor PS\ delay/$clock$\rfloor$ states of delay $\leq$ clock, and **select components** so as to minimize cost (given by the sum of the area of datapath components).

The example in Figure 2 illustrates the problem. Given are a $\mathcal{DFG}$, a $\mathcal{CL}$, and constraints on the clock (10 ns), *PS delay* (50 ns) and latency (100 ns). These constraints imply that there can be no more than two pipe stages, where each pipe stage has at most 5 states of delay at most 10 ns each. The output consists of a scheduled, pipelined and "mapped" $\mathcal{DFG}$, where each node is assigned to a state within a pipe stage and is also assigned a component of the corresponding type from the component library. For instance, in the example, nodes a, b and d belong to the first pipe stage, while nodes c and e belong to the second pipe stage. Each pipe stage is scheduled into 5 states allowing resource sharing between operators that may or may not be in the same pipe stage. Thus, nodes a and e share the same adder, and nodes b and c share the same multiplier. The final component selection consists of one component each of *Mpy1, Mpy3,* and *Add2* bringing the total cost of the design to 350 gates.

4 Algorithm

For a given $\mathcal{DFG}$, $\mathcal{CL}$, and set of constraints, our algorithm performs pipelining, scheduling, and component



Obtain initial non−scheduled pipeline
Step 1
Schedule & select components for all pipe stages
Step 2
Modify pipeline
Step 3
Schedule & select components for all pipe stages
Any improvement?
Yes
No
Stop

Figure 3: Pipelining, resource sharing, component selection: algorithm overview.

selection, in the sequence presented in Figure 3.

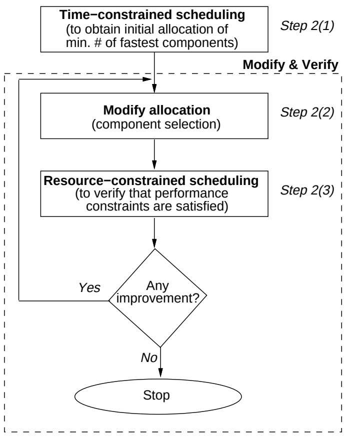
We first obtain an initial non-scheduled pipeline, in which each pipe stage is of delay *PS delay*. In the next step we schedule each pipe stage into $\lfloor PS\ delay/\mathrm{clock} \rfloor$ states, so as to allow resource sharing between operators within and across pipe stages. Component selection is also performed at this step. The pipeline is then modified with the aim of reducing its cost, and the scheduling and selection is re-done for the new pipeline. The pipeline modification and re-scheduling is repeated till there is no further cost reduction.

In the following sections, we explain the three steps of the algorithm, namely, obtaining the initial pipeline, scheduling each pipe stage, and modifying the pipeline.

4.1 Obtaining the initial pipeline

The algorithm used to obtain the initial pipeline has been presented, in detail, in [6]. Due to space limitations we will not elaborate on it, except to mention that the input to the algorithm consists of the $\mathcal{DFG}$, the $\mathcal{CL}$ and the *PS delay* constraint, and the output consists of a mapped and partitioned $\mathcal{DFG}$ where each node is mapped to a component of the corresponding type and the $\mathcal{DFG}$ is partitioned into $\lfloor \mathrm{latency}/PS\ delay \rfloor$ concurrent pipe stages, each of delay no larger than *PS delay*. The aim of this step is to satisfy the latency and *PS delay* constraints with the lowest cost, obtained without resource sharing, but with component selection. By going through the component selection phase, we attempt to obtain the best possible non-scheduled pipeline for the given $\mathcal{DFG}$ and component library. This represents a good starting point for the scheduling step, since it contains equal delay stages and since it has already undergone some cost optimization.

In order to further reduce the cost of this initial pipeline, in the next step, we schedule each stage of the initial pipeline so that resources may be shared by multiple operations. In other words, we replace several slow components in the non-scheduled pipeline by a smaller number of faster components, that have a lower overall cost.



Time−constrained scheduling (to obtain initial allocation of min. # of fastest components)
Step 2(1)
Modify & Verify
Modify allocation (component selection)
Step 2(2)
Resource−constrained scheduling (to verify that performance constraints are satisfied)
Step 2(3)
Any improvement?
Yes
No
Stop

Figure 4: Approach for resource sharing & component selection.

4.2 Resource sharing & component selection

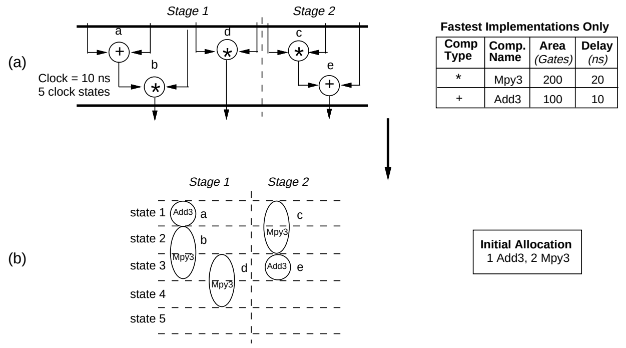
Our approach for resource sharing starts with a time-constrained scheduling step (Figure 4) from which we obtain an allocation in terms of the smallest number of fastest components required to satisfy constraints. We then modify this allocation (Step2(2)) by replacing a component with one or more slower ones, such that the cost of the modified allocation is less than the cost of the initial allocation. The modified allocation is then used as an input to a resource-constrained scheduling approach to ensure that the modified allocation still satisfies constraints (Step2(3)). The steps of modifying the allocation towards lower costs and ensuring the satisfaction of constraints is repeated till no further cost reduction is possible.

4.2.1 Time-constrained scheduling

The aim of this step is to obtain the smallest number of fastest components required to schedule the pipelined $\mathcal{DFG}$ into $\lfloor PS\ delay/\mathrm{clock} \rfloor$ states. In order to do this, we use the force-directed scheduling algorithm [7] with a reduced library containing only the fastest components of each type, instead of the complete library which contains multiple implementations per component type. This is demonstrated in Figure 5 for the example from the previous section. All pipe stages of the $\mathcal{DFG}$ are to be scheduled concurrently as shown in Figure 5(a). The component library is reduced to contain only *Mpy3* and *Add3*, the fastest multiplier and adder respectively. This $\mathcal{DFG}$, reduced component library, and performance constraint of five 10 *ns* states is given as an input to the force-directed scheduling algorithm. The output of the algorithm, shown in part (b) of the figure, consists of a schedule that gives the minimum number of components to satisfy the performance constraints, which, in this case, is 2 multipliers and 1 adder.

4.2.2 Modify & Verify

Having obtained the minimum number of fastest components, the next step is to try and replace each component



Fastest Implementations Only

Comp Type	Comp. Name	Area (Gates)	Delay (ns)
*	Mpy3	200	20
+	Add3	100	10

Figure 5: Demonstrating the input and output of the time-constrained scheduling step.

with one or more slower (and hence cheaper) components that still satisfy constraints and, at the same time, reduce the overall cost. The algorithm for modifying this initial allocation is outlined in Figure 6. The inputs to this algorithm consist of the pipelined $\mathcal{DFG}$, the complete component library $\mathcal{CL}$, the initial allocation obtained from the force-directed scheduling step, and constraints on the clock and the number of states.

As the first step, for each component type, t, we determine the floor of the cost ratio between the most and the least expensive component. This is denoted as R_t. Thus, for the example in Figure 5, $R_\star$ is 2 ($\lfloor 200/100 \rfloor$) and R_+ is 3 ($\lfloor 100/30 \rfloor$). Next, for each component type, t, we determine a list, L_t of all component subsets of length at most R_t, and of smaller cost than that of the most expensive component of that type. Lists $L_\star$ and L_+ are shown in Figure 7 for the given component library. The lists, called the *component substitution lists*, are sorted in increasing order of cost as shown in the figure.

After generating the component substitution lists, *Current_allocation* is initialized to the initial allocation obtained from the force-directed scheduling algorithm. This is the minimum number of fastest components that are required to schedule the $\mathcal{DFG}$ within the specified number of states. Next, for each component in *Current_allocation*, we find the lowest cost component set from the substitution list with which the component could potentially be replaced. From these component and replacement component pairs we select the pair $<c$, $R_c>$ that has the highest cost differential. As an example, consider the initial allocation shown in Figure 5(b) and the substitution lists in Figure 7. The lowest cost replacement for *Mpy3* is *Mpy1* with a cost differential of Cost(*Mpy3*) - Cost(*Mpy1*) $= 200 - 100 = 100$ and for *Add3* the lowest cost replacement is *Add1* with a cost differential of 70. The pair $<c$, $R_c>$, for this example, is then <*Mpy3*, *Mpy1*>.

In the next step, we obtain a *New_allocation* by replacing the component c with the component set R_c. We then check to see if performance constraints (clock and number of states) are satisfied with the *New_allocation*. For this we use a list-scheduling algorithm, explained in more detail in [8]. If constraints are satisfied, we keep the *New_allocation*, else we go back to the old allocation, which is maintained in

```
1.  For each component type, $t$, determine $R_t$ and $L_t$.
2.  If ($\forall t$, $L_t$ is empty)
3.      exit the program.
4.  Else
5.      Current_allocation = initial allocation obtained from
        force-directed scheduling algorithm.
6.      Loop
7.          For all components in Current_allocation
            determine the pair <$c$, $R_c$> that has the
            highest cost differential.
8.          New_allocation = Current_allocation
            after replacing $c$ with $R_c$.
9.          If (New_allocation satisfies constraints)
10.             Current_allocation = New_allocation
11.         else
12.             "Mark" component $c$ with $R_c$.
13.         End if
14.     Until (no component can be replaced with a
        cheaper component without a violation of constraints)
15. End if
```

Figure 6: Algorithm for modifying the initial allocation and verifying that constraints are satisfied.

the variable *Current_allocation*. We also mark the component c with the component set R_c to ensure that in future iterations c is not replaced with R_c or with any component that appears above R_c in the component substitution lists. Steps 6 to 14 are repeated till no component in *Current_allocation* can be replaced, either due to a violation of constraints or because slower components are unavailable in the library.

Component Library

Comp. Type	Comp. Name	Area (Gates)	Delay (ns)
*	Mpy1	100	50
*	Mpy2	150	30
*	Mpy3	200	20
+	Add1	30	40
+	Add2	50	20
+	Add3	100	10

$L_\star$

Components	Area
Mpy1	100
Mpy2	150

L_+

Components	Area
Add1	30
Add2	50
Add1, Add1	60
Add1, Add2	80
Add1, Add1, Add1	90

Figure 7: Creating the component substitution lists.

4.3 Modifying the pipeline

We now come to the third step of the algorithm (Figure 3) in which the pipeline obtained from Step 1 is modified and the scheduling step is repeated for the modified pipeline. Step 1 produces a "good" pipeline in that it tries to equalize the delay of all pipe stages; however, it does not consider the effects of scheduling, and hence it is possible that the pipeline is not optimal with respect to scheduling.

The algorithm for modifying the pipeline essentially moves nodes lying on the border of a pipe stage into the adjoining pipe stage (which may be either one pipe stage higher or lower), one at a time, evaluating each move. The move with the lowest "cost" is kept and the scheduling algorithm (Step 2) is repeated for this new pipeline. The cost of a move is evaluated as the "probable cost" of the new pipeline that is obtained after the move. The probable cost of a pipeline is given by the sum of the probable cost of all operators (we consider the cost of the fastest implementation at this point), a cost function similar to the one used in the force-directed scheduling algorithm. For more details please refer to [8].

5 Experiments

We have implemented the algorithm using "C" on a SUN SPARC 2 station. The complexity of the algorithm is $O(N^2 SlogNC^k)$, where N is the total number of nodes in the $\mathcal{DFG}$, C is the maximum number of implementations of any component type in the library, S is the number of states per pipe stage, and k is the maximum of the cost ratios between the most and the least expensive component of any component type.

We conducted two types of experiments: in the first experiment, we compare our algorithm with previously published ones and, in the second experiment, we study the impact of resource sharing and of component selection on the cost of a design.

5.1 Experiment 1: comparison

As discussed in Section 2, algorithms presented in the past either combine scheduling and pipelining, or pipelining and component selection, or scheduling and component selection; however no algorithm performs all three tasks. Our comparison with related research is thus limited, since we can only compare the quality of any two tasks while keeping the third constant. This comparison thus serves as more of a sanity check rather than as a good means of verifying our algorithm's quality.

TABLE 1
COMPARISON WITH TBS

Delay	FU Area *(gates)*		% difference
(ns)	TBS	Our Alg.	$\frac{our-TBS}{TBS}$
1800	20600	20800	+ 1.0
1900	17000	19300	+ 13.5
2000	14500	14000	- 3.5
2100	16000	11000	- 10.6
2200	10500	9300	+ 4.8
2300	9000	9000	+ 3.3
2400	9000	8000	0.0
2500	8500	8000	- 5.9
2600	8000	8000	0.0
2700	8000	8000	0.0
2800	8000	8000	0.0
2900	8300	6000	- 27.7
3000	7800	6000	- 23.1
3100	7800	5000	- 35.9
3200	8500	5000	- 41.1
3300	8300	4800	- 42.2
3400	7500	4800	- 36.0
3500	7800	4800	- 38.0
3600	4800	4800	0.0
3800	4000	4300	+ 7.5
3900	4000	4300	+ 7.5

We compare our algorithm with the TBS algorithm [3] which performs only scheduling and component selection. For this purpose we use the component library given in Figure 3 (page 94) of their paper. Since TBS does not pipeline the $\mathcal{DFG}$, we set the number of pipe stages in our designs to one.

The comparison is conducted for the elliptic filter benchmark. Results produced by both algorithms, given in Table 1, have a fixed clock of 100 ns and a total delay requirement as shown in the first column. On an average, our algorithm performs approximately 10% better than the TBS algorithm - this is largely due to the fact that for delays between 2900 and 3500 our algorithm selects one fewer multiplier than the TBS algorithm hence bringing down the design area significantly. We would like to iterate that this comparison should be considered as more of a sanity check rather than a measure of our algorithm's quality since it has been conducted for only one example and since we have not pipelined the $\mathcal{DFG}$.

5.2 Experiment 2: design quality

This section addresses the impact of resource sharing and of component selection on the design cost of four examples: the elliptic wave filter (EWF) benchmark [9], the differential heat release computation (DHRC) example [10], the differential equation solver (DES, also known as the HAL benchmark) [9], and the blurring (BLUR) benchmark [11]. For all experiments we have used the component library shown in Table 2 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 component delay is in ns.

TABLE 2
COMPONENT LIBRARY

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

Impact of resource sharing

In the first set of experiments we compare designs obtained with and without resource sharing, that is designs in which only the pipelining and component selection tasks are performed versus designs in which all three tasks are performed. Figure 8 presents results for the EWF and DHRC examples. In both graphs, points along the bold lines indicate designs without any resource sharing (that is with only 1 state per pipe stage) while the points on the dashed lines indicate those obtained with resource sharing.

For the 1-stage designs of the EWF example, resource sharing results in about a 50% area reduction and for the 2-stage designs about a 30% area reduction. For the 2-stage pipelined designs of the DHRC example the area reduction obtained by resource sharing is as high as 200%. Similar conclusions can be reached for the differential equation

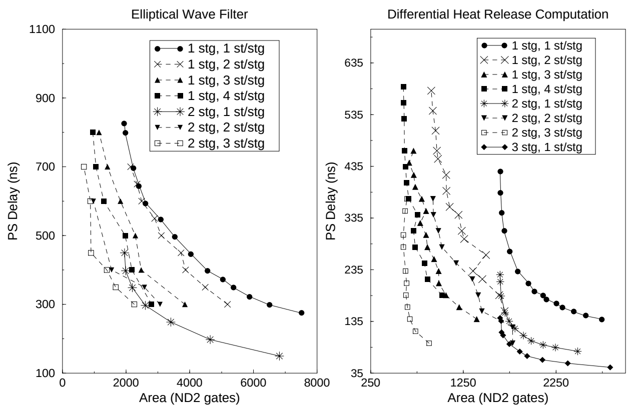


Figure 8: The effect of resource sharing on the cost of a pipeline for the EWF and DHRC examples.

solver and the blurring benchmark where the area reduction with resource sharing is between 10 and 50% for different *PS delay* and latency values.

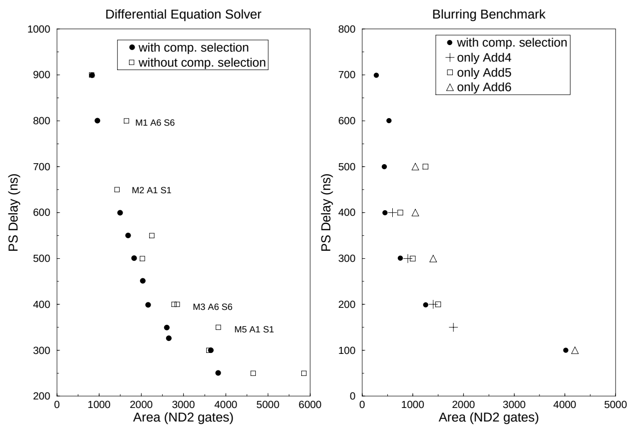


Figure 9: The effect of component selection on the cost of a pipeline for the DES and BLUR examples.

Impact of component selection

Next, we briefly demonstrate the impact of component selection on the design cost of the differential equation solver and the blurring benchmark in Figure 9. We compare designs obtained with and without component selection, by keeping constant all design parameters other than the component library. For the differential equation solver, we set the design parameters to 2 stages and 2 states per stage and then run our algorithm with the complete library shown in Table 2 and also with a number of reduced libraries some of which are annotated in the graph. These reduced libraries contain just one implementation per component type which were selected from the predominant ones in the designs obtained with component selection. For the entire range of *PS delay* values from 200 to 900 *ns*, the lowest-area designs are those obtained with component selection (shown as circles). The designs without component selection were as much as 50% higher in area and no reduced library could consistently give low area designs over the entire range of *PS delay* values. Similar experiments were conducted for the blurring benchmark, and, like in the previous example, the designs with component selection formed a lower bound on design area and none of the reduced libraries could produce low area designs over the entire range of *PS delay* values.

6 Conclusions

In summary, we have presented a design strategy that attempts to achieve high-throughputs at low costs by combining pipelining with scheduling and component selection. Though a combination of two features, pipelining and component selection, or pipelining and scheduling, can be used to achieve high-throughput values at low costs, it is the use of all three features that further minimizes the cost for a given throughput.

We conducted several experiments to demonstrate the impact of component selection and of scheduling on the cost of a pipelined design. From these experiments, we note that, for all the examples, the cheapest designs obtained for a given throughput are those that combine pipelining with resource sharing and component selection.

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] 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.

[4] 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.

[5] S. Note, F. Catthoor, G. Goossens, and H. D. Man, "Combined hardware selection and pipelining in high-performance data-path design," *IEEE Transactions on Computer Aided Design*, vol. II, pp. 413–423, Apr. 1992.

[6] S. Bakshi and D. D. Gajski, "A component selection algorithm for high-performance pipelines," in *Proceedings of EURO-DAC*, pp. 400–405, 1994.

[7] P. Paulin and J. Knight, "Force-directed scheduling for the behavioral synthesis of asics," *IEEE Transactions on Computer Aided Design*, vol. 8, pp. 661–679, June 1989.

[8] S. Bakshi, D. D. Gajski, and H.-P. Juan, "Component selection in resource shared and pipelined dsp applications," Tech. Rep. 95-15, Dept. of Information and Computer Science, University of California, Irvine, 1995.

[9] N. D. Dutt and C. Ramachandran, "Benchmarks for the 1992 High-Level Synthesis workshop," Tech. Rep. 92-107, Dept. of Information and Computer Science, University of California, Irvine, 1992.

[10] F. Catthoor and L. Svensson, *Application-Driven Architecture Synthesis*. P.O. Box 17, 3300 AA Dordrech, The Netherlands: Kluwer Academic Publishers, 1993.

[11] J. S. Lim, *Two-dimensional image and signal processing*. Prentice Hall Signal Processign Series, 1990.