An Approach for Redesigning in Data Path Synthesis[†]

Christos Papachristou, Haidar Harmanani and Mehrdad Nourani

Department of Computer Engineering
Case Western Reserve University
Cleveland, Ohio 44106

Abstract — **A new method of redesign at the register-transfer level using a transformational process is proposed. The redesign approach takes into consideration ALU and interconnect cost in addition to layout area estimation. The idea is to start with a design, possibly generated by a synthesis system, and iteratively improve it by means of a reallocation process. The method is based on a set of reallocation transformations along with systematic strategies as to how to apply them together with a layout estimation model.**

I. Introduction

High Level Synthesis is the automatic generation of a register transfer level (RTL) design (datapath/controller) from a behavioral level description, subject to a set of constraints. The datapath is usually synthesized in two steps — *operation scheduling* and *resource allocation*. One of the main tasks of high-level synthesis is to find the structure that best meets the constraints while minimizing other costs [6]. The actual circuit layout can be later generated from the datapath using a silicon compiler.

The computational complexity of the scheduling and allocation problems forced researchers to develop several heuristic methods. Most of these methods can handle technology-independent constraints, i.e., optimize datapaths with respect to the number of ALUs, registers and multiplexers. However, these optimization techniques may not satisfy the constraints at the actual chip layout level. To address this deficiency, recent synthesis techniques have attempted to optimize designs with respect to the area rather than to the number of datapath components, using library-based component cost. Although these designs are more realistic, they are non-optimal because they are usually based on heuristics. More importantly, these techniques still do not consider placement and routing of components which significantly affect the overall chip area. Recently, there have been some attempts to alleviate this problem during the RTL design process. One approach was by combining high level synthesis with layout such as in [11]. Some stochastic layout estimation models were developed as well. The problem with these models is that their accuracy is not known [5, 10, 1].

A. Motivation and Related Works

Given an RTL level description of a datapath, the purpose of this work is to further improve its cost while looking at lower level issues, specifically layout. Based on reallocation transformations and strategies, the redesign process works as a post-synthesis process to iteratively improve the design the cost. The motivation for the redesign approach is twofold:

1. Due to the non-optimal nature of the synthesis process, there is still a lot of room for improvement of the component cost of a datapath design.
2. In order for design modifications to be effective, they should consider the placement and routing cost in addition to component cost.

[†]This work is supported in part by the Semiconductor Research Corporation (SRC) under contract 92-DJ-147

The cornerstone of our redesign method are two elements:

1. A set of transformations that are used as a tool to perform reallocation of the datapath design.
2. A layout estimator based on a realistic datapath layout model that can quickly estimate the layout cost during the course of design modifications.

The issue of redesign in high level synthesis was discussed in [3] were a manual approach was used to modify the schedule and to affect the result of the allocation by manually inserting and/or removing registers and other components. A later version of Fasolt [4] used a layout model to automatically drive the choice of design transformations. Finally, [2] proposed a redesign method to modify designs so that to be reused as new designs with different specifications.

This paper is organized as follows. In section 2 we discuss design transformations in general while section 3 describes the our redesign approach along with the transformations used. Sections 4 and 5 discusses our reallocation phases. Results are finally discussed in sections 6.

II. Design Transformations

The design of digital circuits is usually described at various levels of abstractions. Three known levels of the design hierarchy are the *behavioral*, the *structural* and the *layout* level. The design behavior can be described by a design language such as VHDL. RTL design structures are described using a netlist of RTL components such as registers, functional units and multiplexers. Finally, the layout of a design is described by means of the geometries of the individual primitive cells. Other design levels such as the gate level are of course widely used in the design hierarchy. Informally speaking, a ***transformation*** consists of a set of rules that, subject to certain preconditions, modify a given design A into another design A', in the same design level as A, with A' being functionally equivalent to A.

In general, we can distinguish among the following transformation classes:

- *Behavioral transformations* which produce functionally equivalent but behaviorally different data flow representations. For example, applying the distributive property on the expression $E = a * (b + c)$ changes the data flow behavior of E into that of $E' = ab + ac$, where E' is obviously functionally equivalent to E.
- *Structural transformations* which modify an RTL design, but do not change its behavior. For example, merging an adder and a subtractor of a datapath into an ALU capable of performing both operations is a structural transformation.

The structural transformations can be further classified into the following two categories:

- *Rescheduling transformations,* which change the timing of the datapath structure for example, inserting a register in a datapath may, under some conditions, prolong the schedule by one time step.
- *Reallocation transformations,* those which change the structure but not the datapath schedule.

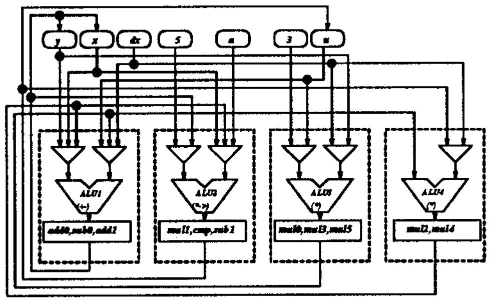
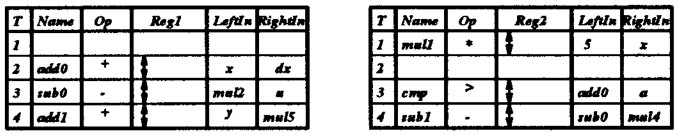


Fig. 1. Datapath for the HAL example



T	Name	Op	Reg1	LeftIn	RightIn
1					
2	add0	+	↕	x	dx
3	sub0	-	↕	mul2	u
4	add1	+	↕	y	mul5

ALU1

T	Name	Op	Reg2	LeftIn	RightIn
1	mul1	*	↕	5	x
2					
3	cmp	>	↕	add0	a
4	sub1	-	↕	sub0	mul4

ALU2

Fig. 2. A partial Grid representation for HAL datapath

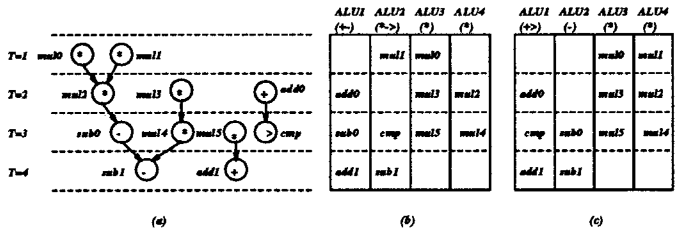


Fig. 3. The HAL example before and after reallocation

This paper concentrates on performing restricted exploration of the RTL space, i.e., the allocation space, by means of reallocation transformations. Thus, preserving not only the design behavior but also the design schedule. One of the advantage of the reallocation process is that the allocation space is much smaller and more manageable than the scheduling space. However, the disadvantage is that it is not consequently as flexible.

III. Reallocation of Datapath

A. Reallocation Approach

We base our reallocation method on a *Transformational Process* based on an iterative improvement approach which is an efficient way to solve computationally complex problems. The method starts with a design relatively good, derived by a synthesis method, and applies a set of reallocation transformations so that to improve the design cost. The process will stop once no further improvement is possible. The following observations motivated our approach:

1. A transformation may lead to an improvement in overall ALU area cost if it can change the functionality of the ALUs to which it is applied. This is true whether it leads to single function or to multifunctional ALUs.
2. The cost of some components such as multipliers dominate the area cost of a chip. Thus it is useful to give priority to more expensive components in reallocation considerations.

We separate our reallocation method into two phases. *The first phase* uses transformations and aims at exploring modifications in ALUs cost by altering their functionalities. During this phase, MUX and register costs are considered only in order to break ties among similar cost transformations. *The second phase* attempts to reduce the multiplexer cost but uses the layout estimator as a driving engine. The rationale is: 1) not to destroy whatever improvement phase I has accomplished; 2) not to reduce the number of registers or multiplexers on the account of the layout cost of the data path. We measure the RTL design cost by the total area of the RTL components (ALUs, Registers, MUXes) obtained from a component library. Although this metric maybe adequate for some designs, we use a layout estimator to provide a much more realistic cost evaluation of our designs.

B. Reallocation Transformations

The *reallocation transformations* correspond to moves of operation and store variables of the datapath structure, while keeping the schedule fixed. The transformations are characterized by: a) *a Transformation Rule*, b) *a Prerequisite Condition* and c) *a Transformation Effect.*

As a result of the allocation process, a DFG variable V_a is mapped into two distinct variables, namely an *operation* variable OV_a and a *store* variable SV_a, where OV_a and SV_A are produced and stored, respectively, in a distinct ALU and register pair of the datapath. The basic reallocation transformations are described informally below and followed by a relatively small example for illustration:

1. *Operation Variable Move:* Move operation OV_a in time step $T(OV_a)$ from ALU_i to ALU_j. We indicate this transformation by $[MOVE : OV_a \in ALU_i \rightarrow ALU_j]$. The prerequisite is that there is no resource conflict for this move, meaning that ALU_j is not performing any operation during $T(OV_a)$. The effect is redistribution of functions between ALU_i and ALU_j along with a connection effect.
2. *Store Variable Move:* Move variable SV_a whose life span is $L(SV_a)$ from register REG_i to Register REG_j. We show this transformation by $[MOVE : SV_a \in REG_i \rightarrow REG_j]$. The prerequisite is that there is no variable in REG_j whose life span overlaps $L(SV_a)$. There is again a connection effect of this move.
3. *ALU Insert:* Insert an ALU in the data path. This can be used when an operation or store variable move (or swap), respectively, can not be applied. This transformation may occur when there is a time conflict on $T(OV_a)$ with some operation in ALU_j.
4. *Register Insert:* Insert a register in the data path. This transformation may occur when $L(SV_a)$ overlaps with some variable life-spans in REG_j. This transformation makes the move out of ALU_i or REG_i valid (satisfying transformation prerequisites) although the destination is changed.
5. *Eliminate ALU or Register:* Reduce an ALU or register if they carry no variables as a result of prior move or swap transformations. This is basically a cleanup transformation.

By definition, a transformation is *valid* if it can be applied without violating its prerequisites. It should be noted that more complex reallocation transformations can be defined as sequences of the above basic transformations assuming that their prerequisites are valid. For example, one can split a given ALU or a register into one or more ALUs or registers by applying respectively ALU or register insert followed by a move transformation. In fact, we can define the SWAP transformation in terms of the *move transformations.* Note that it is easy to see that each of the above transformations 1 through 5 has an *inverse* transformation, for example, the inverse of *Insert* is *Eliminate.* Although it is not difficult to show that the above transformations are correct, no formal proof of correctness will be given here.

C. Illustrating the transformation with an example

We shall describe the above transformations by reference to the HAL differential equation example. The datapath for this example is shown in Fig. 1 and has been generated using a high-level synthesis system [8]. The underlying scheduled DFG is shown in Fig. 3(a). For illustration purposes, we will use a *grid-like datapath* representation, shown in Fig. 2. The first column indicates the time step at which the operation was scheduled. The second shows the operation name while the type is shown in the third column. The fourth column is the register in which the operation value is stored along with its life span. The last two columns are the inputs for the operation which will be used later for interconnect calculation.

For simplicity, we compact the four ALU grids into a single representation, the *datapath chart* shown in Fig. 3(b). In this datapath chart, the entry $(T = 2, ALU_1)$ indicates that in time step 2, ALU_1 produces operation variable *add*0. Also, entry $(T = 2, Reg_1)$ indicates that *add*0 is stored, with life-span from $T = 2$ to $T = 3$, in Reg_1, connected to ALU_1.

Fig. 3 shows examples of these transformations. Fig. 3(b) and 3(c) show an example before and after applying the reallocation process. Fig. 3(c) shows the result of moving *mult*1 from ALU_2 into ALU_4, time step $T = 1$. The same figure also shows the result of the swap transformation of operations *sub*0 and *cmp*, time step $T = 3$, between ALU_1 and ALU_2, respectively. Note that as a result of these reallocation transformations we have reduced the appearance of multiplications from three ALUs into two ALUs.

D. Reallocation Strategies

As noted in the previous section, the transformations are associated with two design modifications. First, a local redistribution of operations and/or register variables. Second, a limited connection change in the original datapath. We remark further that the change in component cost, including the MUX cost, due to the transformations can be easily computed from the library. The connection cost can be more accurately accounted for by invoking the layout estimator.

The question that arises is how to apply the transformations in a systematic way. A straightforward method to attack this problem is to apply these transformations in a hill-climbing fashion leading to an incremental cost reduction transformation by transformation. An annealing method based on individual transformations is another alternative. The disadvantage of these methods is that they are not well guided and thus time consuming. We notice that an improvement in data path cost may require applying a set of transformations. Thus, in our approach we will take a more global view by selecting not one-by-one, but sequences of transformations. Specifically, a *strategy* consists of alternative sequences of transformations that have common characteristics and perform a reallocation task. The evaluation of the strategies is based on well guided cost estimation choices.

IV. Phase I Strategy

A. Reduce

An RTL datapath design consists of a number of single and/or multifunctional ALUs among other components. We note that in a multifunctional ALU the most costly operation dominates the overall ALU cost. Then it is useful to develop a strategy with the aim of reducing the number of such costly operations from the datapath. In the example of Fig. 3, multiplication dominates the cost of the datapath. Note that the same strategy will apply to reduce cheaper operations from the ALUs if there is no room to improve the cost by transformations of more expensive operations.

Consider a datapath consisting of $ALU_1, ..., ALU_n$. Let O_k denote an operation type with $k = 1, ..., m$ where m is the number of operation types in the data path. Let $f(O_k)$ denote the *frequency* of the operation O_k, i.e., the number of times O_k appears in the ALUs of the datapath.

The basic property that characterizes the constituent transformations of *Reduce* is that they all have the same *source* ALU. Informally, the strategy is as follows:

- *Reduce Strategy:* Removing one operation of type O_k from an ALU. We show this by the notation: $Reduce(ALU_i, O_k)$. To achieve this strategy we have to move all operation variables of type O_k from ALU_i to other ALUs. For $Reduce(ALU_i, O_k)$ to be possible there must exist at least one sequence of valid moves that make up this strategy.

The motivation of $Reduce(ALU_i, O_k)$ is to reduce the frequency of O_k, $f(O_k)$, by one or more. For our example of Fig. 3, $f(*) = 3$, $f(+) = 1$, $f(-) = 2$. Clearly, after $Reduce(ALU_2, *)$ in Fig. 3(b), we have $f(*) = 2$. In this example, we used $Reduce(ALU_2, *)$ since $*$ is the most expensive operator and dominates the cost of the datapath components. To achieve this strategy, we applied $[MOVE : mul1 \in ALU_2 \rightarrow ALU_4]$. Let us assume that in this example multiplication is three times more costly than an addition, subtraction, or comparison. Then the result of this strategy is a 20% cost reduction on component (ALU) cost.

Thus, our reallocation approach aims at reducing the frequency of the most costly operation in the datapath, by applying the Reduce strategy for this operation repeatedly, as much as possible. This approach can continue and apply to other operations, in decremental order of their costs. This works with $*$ and $+$ operations because $Cost(*) > Cost(+)$. However, there is a question about choosing between $+$ and $-$ for the Reduce strategy because their costs are about the same. We handle these questions by introducing the *Move Table* and procedures with respect to an operation type and which select a given strategy.

B. Move Tables and Procedures

The move table is an effective data structure that captures all move/swap transformations concerning O_k in an efficient way to form the Reduce strategies. For our example data path, the Move Tables with respect to operations $*$ and $+$ are shown in Fig. 4(a) and 4(b), respectively. The rows correspond to the time steps T_i and the columns to ALU_j. Every entry (T_i, ALU_j) of a Move Table consists of either a set of integers $a, b, c, ...$ or a blank. The integers denote destination ALUs for moving operation type O_k, such as a $*$, from ALU_j, and these moves are valid in time step T_i. The blank entries mean no moves are valid in that time step.

Thus, in Fig. 4(a), entry (T_2, ALU_3) is 2 meaning that the destination ALU is ALU_2, i.e., the following move is valid: $[MOVE : mul3 \in ALU3 \rightarrow ALU_2]$. Some entries in the Move Table are accented. The reason is that those destinations are associated with a Swap rather than a Move. For example, entry (T_3, ALU_3) is $2'$, meaning the following transformation is valid: $[SWAP : mul5 \in ALU_3 \leftrightarrow cmp \in ALU_2]$. In Fig. 4(b), the entry (T_4, ALU_1) is $2', 3, 4$. This means either of the three transformations with respect to $+$ from ALU_1 are valid. In effect, every integer entry in the Move Table represents a move or swap transformation in symbolic form.

Given this structure of the Move Table for O_k, we can compose the strategies $Reduce(ALU_i)$ by the following simple procedure:

- For every integer entry in a time step under ALU_i, and for every step, combine a single integer entry in each time step with a single integer entry in every other time step. The result is all possible move/swap sequences that compose $Reduce(ALU_i)$.

Thus in Fig. 4(b), the strategy $Reduce(ALU_1, +)$ is achieved by the following three move/swap sequences written symbolically in AND/OR form:

$2\&(2' \text{ OR } 3 \text{ OR } 4) = (2\&2') \text{ OR } (2\&3) \text{ OR } (2\&4)$

Also, in Fig. 4(a), the strategy $Reduce(ALU_3, *)$ is achieved by the sequence of moves $4\&2\&2'$.

Although there are very few sequences in the above examples that compose each $Reduce(ALU_i)$ strategy, it may appear that in the worst case there may be very many such sequences. In general, one may have a growth of the order of $(n-t)^t$ where t is the number of time steps and n the number of ALUs. However, this will happen only when dealing with *sparse* datapath charts, and this is not the case with designs that have gone through an allocation process.

An interesting property of the Move Table is that in every row the non blank entry sets are identical to each other. For example, entries under ALU_2 and ALU_3 in time step 2 or 3. The reason is these entry sets indicate the "empty slots" of the datapath chart (Fig. 2) that an operation, say $*$, can move. Since ALU_2 and ALU_3 both do $*$ in time step 2, then $*$ can move into the same slots. This property significantly reduces the time for building the Move Table. We have the following comments.

- *Comment 1:* We note that in the Move Table of Fig. 4(a) the column under ALU_1 is all blank. The reason is that we do not want to move $*$ from the other ALUs into ALU_1 because such a transformation will increase the data path cost by creating a new multiplier.
- *Comment 2:* Reducing the frequency of an operator say $*$ not only reduces the number of ALUs containing $*$, but it may even reduce the actual number of ALUs.
- *Comment 3:* The notion of Reduce can be extended to apply not only to a single operator but also to an operation set, for example, if $O_k = \{+, -\}$ then $Reduce(ALU_i, O_k)$ refers to removing $\{+, -\}$ from ALU_i. We remark that when this applies to two single function ALUs, ALU_i and ALU_j, containing only $+$ and $-$ respectively, then $Reduce(ALU_i, O_k)$, if possible, will modify ALU_j into a multifunctional, $+, -$, whereas ALU_i will in effect be eliminated. In this case, $Reduce(ALU_i, O_k)$, has the effect of *merging* two single function ALUs into one multifunctional ALU, which could reduce the cost. The point here is that by properly choosing the set O_k in the Reduce strategy, we can have several different effects on the datapath while reducing the cost.

Procedure 1: The following reallocation procedure is used to reduce the ALU cost of a given datapath based on the Reduce strategy for operation set type O_k.

1. Form the Move Table of *current-datapath.*
2. Generate $Reduce(ALU_i, O_k)$ for $i = 1, ...n$.
3. Compute the cost of *current-datapath* for the strategy in question.
4. For $i = 1, ..., n$ select a strategy with lowest cost.
5. Perform the selected strategy generating *new-datapath.*
6. Go to step 1.

The above procedure terminates when in some *new-datapath* no Reduce strategy is possible (step 2). If there are n ALUs, then clearly there could be $n-1$ iterations, at worst.

We illustrate this procedure for our running example of Fig. 3(a). Using the Move Table of Fig. 4(a), the $Reduce(ALU_i, *)$ strategy, $i = 2, 3, 4$ results in three datapath charts, respectively shown in Fig. 5 a,b,c. Of the three strategies the one with the least cost is $Reduce(ALU_2, *)$ with new $f(*) = 2$. However, the Reduce strategy on $*$ is no longer possible. Thus we use the datapath chart of Fig. 5(a) for further applications of Reduce using other operations. We will now consider the earlier question, how would we choose among the operations $\{+, -, cmp\}$ assuming, as it is usually the case, that they have almost the same cost.

To handle this problem, it is reasonable to choose the operation with the largest frequency for the one to apply the next Reduce. In the example of Fig. 5(a), we have $f(+) = 1, f(-) = 2, f(cmp) = 1$. Thus the $-$ is the most frequent operation to apply Reduce, i.e., to reduce $f(-)$ by one. So we apply $Reduce(ALU_i, -)$ with $i = 1, 2, 3, 4$ using the Move Table for $-$, Fig. 6(a), based on the new datapath chart. Thus $Reduce(ALU_1, -)$ and $Reduce(ALU_2, -)$, respectively, can be effected by the symbolically represented swaps, $2'$ and $1'$, respectively. Their corresponding new datapath charts are shown in Fig. 6(b) and (c) respectively.

T	ALU1	ALU2	ALU3	ALU4
1		4	4	
2			2	2
3			2'	2'
4				

(a)

T	ALU1	ALU2	ALU3	ALU4
1				
2	2			
3				
4	2',3,4			

(b)

Fig. 4. Move tables for the runing example

T	ALU1	ALU2	ALU3	ALU4
1			mul0	mul1
2	add0		mul3	mul2
3	sub0	cmp	mul5	mul4
4	add1	sub1		

(a)

T	ALU1	ALU2	ALU3	ALU4
1			mul0	mul1
2	add0	mul3		mul2
3	sub0	mul5	cmp	mul4
4	add1	sub1		

(b)

T	ALU1	ALU2	ALU3	ALU4
1		mul1	mul0	
2	add0	mul2	mul3	
3	sub0	mul4	mul5	cmp
4	add1	sub1		

(c)

Fig. 5. Data Path charts for our runing example

Note that the Reduce strategy should reduce the frequency of the operation it applies and this disqualifies several other possible destination moves of $-$. Of the two possible Reduce above, both making $f(-) = 1$, the one which does not increase $f(+)$ is chosen, namely $Reduce(ALU_1, -)$.

The above example leads to the following general procedure for reallocation a datapath, outlined below.

Procedure 2:

1. Begin with the most costly operator, O_k.
2. $Reduce(ALU_i, O_k)$, $i = 1, ..., n$ by applying Procedure 1.
3. Choose, on the basis of cost, one $Reduce_i$
4. Form new datapath and Move Tables
5. If the other operators do not have similar cost (i.e., $+, -, >$), then goto 1.
6. Compute frequencies for the rest of operations.
7. Choose most frequent operator, say O_{k+1}
8. $Reduce(ALU_i, O_{k+1})$,
9. Go to step 3

This procedure terminates when all the Reduce strategies have applied to all operations in the data path, and each Reduce has terminated by Procedure 1.

V. Phase II Strategy

A. Strategy

Once the component cost has been improved, phase II improves the interconnect area using a layout estimator. Our strategy in this phase is based on the following considerations:

- *Fixed schedule*: This forces to move or swap operations in the same control steps.
- *Fixed functionality*: We may only move or swap operations without altering the data path functionality since we do not want to degrade the result obtained in the first phase of reallocation.

T	ALU1	ALU2	ALU3	ALU4
1				
2				
3	2'			
4		1'		

(a)

T	ALU1	ALU2	ALU3	ALU4
1			mul0	mul1
2	add0		mul3	mul2
3	cmp	sub0	mul5	mul4
4	add1	sub1		

(b)

T	ALU1	ALU2	ALU3	ALU4
1			mul0	mul1
2	add0		mul3	mul2
3	sub0	cmp	mul5	mul4
4	sub1	add1		

(c)

Fig. 6. (a) Move table for $-$ using Fig. 5(a), (b) Data path after reduce ALU1, (c) After reduce ALU2.

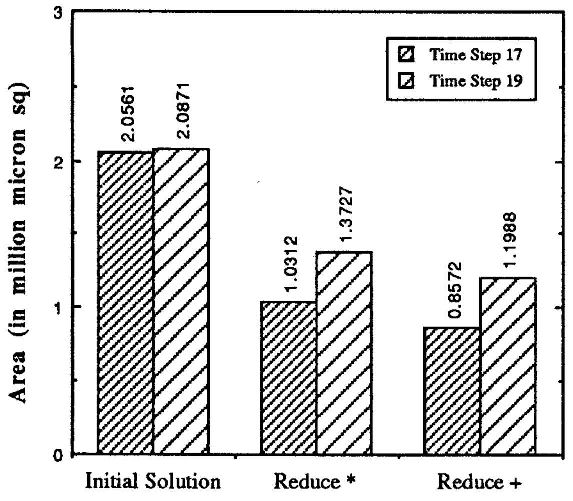


Fig. 7. Results from wave filter after applying Reduce strategies

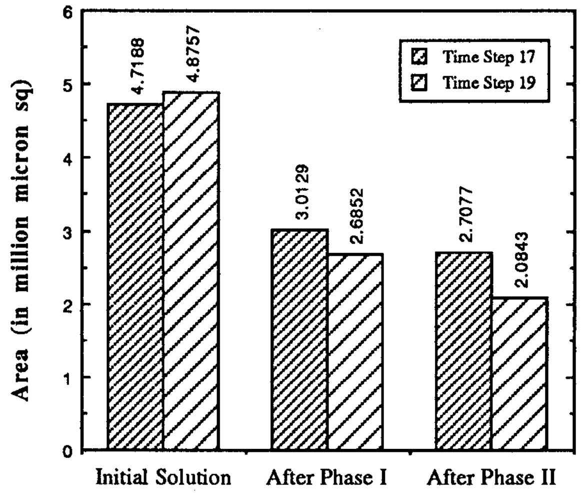


Fig. 8. Results from wave filter after Phase I and Phase II

- *Layout estimation critique*: At each iteration, the layout estimator can provide the interconnect cost which may account till up of 75% of the chip area. The final decision will be made based on the largests cost gain.

The algorithm for phase II has been summarized bellow:

1. *Step 0*: Use layout estimator to compute layout cost of the initial design (*dpcost*). Sort operation types based on the cost of their operators. Sort the ALUs based on their connectivity factor. Set $i = 1$ and $j = 1$.
2. *Step 1*: Consider ALU_i and find the operations of type j and their corresponding control steps e.g. O_k and t.
3. *Step 2*: Check control step t in $ALU_{i+1}, \cdots, ALU_n$. If it contains an operation of type j, call the layout estimator to compute *dpcost* after applying the *exchange* transformation.
4. *Step 3*: Evaluate the new data path cost. If there is improvement, accept that transformation, otherwise stop.
5. *Step 4*: Compute $j = j + 1$. If $j > k$ stop, otherwise go to step 1.
6. *Step 5*: Compute $i = i + 1$. If $i > n$ stop, otherwise go to step 1.

The algorithm will terminate in $n * k$ iterations, where n and k are the total number of ALUs and operation types, respectively.

B. Layout Estimation Process

The layout estimator that we have developed is fast enough to be effectively used within the datapath redesign process. In the following, we present the outline of the estimation algorithm. More details can be found in the proceeding of the current DAC [7].

The algorithm consists of two major steps:

- *Step I*: In the initial placement step, the estimator constructs an one-row design by finding the best permutation for modules while minimizing the interconnect lengths.
- *Step II*: In the second step, the estimator folds the one row layout into a number of smaller rows so that the height and width of the resulting rectangular shape layout satisfies the aspect ratio specified by the user.

VI. Results

The described reallocation scheme was implemented along with the layout estimator as a post-synthesis tool to our synthesis system [8]. We used the wave filter benchmark example [9] to further illustrate our method. We generated two schedules using time steps 17 and 19. The initial solution was allocated using a heuristic method [8]. Thus, the reallocation approach makes sense to derive improvements. The results for the wave filter are shown in Fig. 7 and 8. The first graph (Fig. 7) shows the improvement in the solution after executing the *Reduce* strategies. The first column indicates the design initial area generated by the synthesis system. The second column shows the solution after applying the strategy $Reduce(ALU_i), *)$, while the last column shows the area for the final solution after applying the strategy $Reduce(ALUi, +)$.

The layout area was computed and taken into consideration in Phase II, and the results are shown in Fig. 8 for time step 17 and 19 respectively. The First column indicates the initial solution with interconnect consideration generated by the synthesis system, while the second column indicates the area after applying phase I. The third column shows the improvement after applying phase II.

References

[1] X. Chen and M. Bushnell, " A Module Area Estimator for VLSI Layout," *Proc. 25th DAC*, 1988, pp. 54-59.

[2] M. Fujita, T. Kakuda, " A Redesign Approach to High-Level Synthesis," High-Level Synthesis Workshop, 1991.

[3] D. Knapp, "Manual Rescheduling and Incremental Repair of Register-Level Datapaths," *Proc. of the ICCAD*, 1989.

[4] D. Knapp, "Datapath Optimization Using Feedback," *Proc. of the EDAC*, pp. 129-134, 1991.

[5] F. J. Kurdahi and A. C. Parker, " Techniques for Area Estimation of VLSI Layouts," *IEEE Trans. on CAD*, January 1989.

[6] M. McFarland, A. Parker, and R. Composano, "The High Level Synthesis of Digital Systems," *IEEE Trans. on CAD*, February 1990, pp. 301-318.

[7] M. Nourani and C. A. Papachristou, " A Layout Estimation Algorithm for RTL Datapaths," *Proc. 30th DAC*, 1993.

[8] H. Harmanani, C. Papachristou, S. Chiu and M. Nourani, "SYNTEST: An Environment for System-Level Design for Test," *Proc. First Euro-DAC*, pp. 402-407, 1992.

[9] P. Paulin, J.P. Knight, "Forced-directed scheduling in automatic data path synthesis," *Proc. 24th DAC*, pp. 195-202, 1987.

[10] K. Ueda, H. Kitazawa et al., " Chip Floor Plan for Hierarchical VLSI Layout Design," *IEEE Trans. on CAD*, January 1985.

[11] J. Weng and A. C. Parker, "3D Scheduling: High-Level Synthesis with Floorplanning," *Proc. of the DAC*, 1992, pp. 668-673.