Resource Sharing in Pipelined CDFG Synthesis

Somsubhra Mondal and Seda Öğrenci Memik
Department of Electrical and Computer Engineering
Northwestern University, Evanston, IL 60208

Abstract

Efficient use of limited available resources on an FPGA remains a crucial problem for synthesizing pipelined designs. Resource sharing addresses this challenge. In this paper, we propose resource sharing techniques that can be incorporated into an automated synthesis flow to generate pipelined designs. Given a synthesized pipelined design, we create a direct relationship between available time slack on modules and the multiplexing overhead due to sharing. This flexibility is maximally exploited without violating any throughput constraints. We propose different techniques to address resource sharing problems of varying restrictions. Specifically, we propose an optimal algorithm for Constant-Slack Resource Sharing and a heuristic for the general Intra-Pipeline Stage Resource Sharing. On an average the demand on arithmetic functional units can be reduced by 39.5% for a set of benchmarks from the multimedia domain using our resource sharing technique.

1. Introduction

A major challenge faced by all synthesis efforts has been the issue of creating designs that comply with the resource capacity of the target programmable device. Synthesis tools primarily try to optimize latency and/or throughput objectives, and generally they push the utilization of the target device to full extent often exceeding the available capacity and causing infeasible solutions. Such high-performance implementations can benefit immensely from global management of resource re-use, i.e., resource sharing, if this is performed with an awareness of the performance constraints.

In this paper, we propose novel resource sharing techniques for pipelined designs. Our techniques enable resource sharing while considering the area and delay overhead of multiplexing. The delay penalty due to multiplexing is efficiently "hidden" by systematically exploiting the time slack on non-critical portions of the design while obeying the overall latency/throughput requirements. Our experiments show that substantial reduction in the number of resources can be achieved without compromising the performance goals. Our specific contributions in this paper are as follows:

- We formulate the intra-pipeline stage resource sharing problem,
- We formulate instances of resource sharing problem for which we propose optimal sharing mechanisms under a given pipeline length and throughput constraint, and
- Finally, we propose a heuristic for the general problem instance and present experimental results.

The rest of the paper is organized as follows. We overview related work in Section 2. In Section 3 we briefly discuss about the model of hardware compilation flow used. In Section 4, we formulate the intra-pipeline stage resource sharing problem and present techniques to solve two variations of this problem and provide proof of optimality for the constant slack case. We present our experimental results in Section 5 and conclude with a summary in Section 6.

2. Related Work

There are several proposed tools to create a direct path from high-level design descriptions to programmable hardware. Developers of some of these tools discuss the challenge of generating designs that comply with the resource constraints of the target hardware. Kaplan et al. report that several benchmarks synthesized with their automated flow were too large to finish the synthesis [1]. Similarly, Ziegler et al. [2] report that in some cases their automated pipelining technique for reconfigurable systems produces designs that exceed the available capacity of the target FPGA.

Memik et al. [3] proposed a heuristic for global resource sharing to complement hardware compilation flows for FPGAs. However, their technique has no direct applicability to pipelined designs. There are several other proposed resource sharing techniques for ASICs. Wakabayashi et al. [4] introduced condition vectors to detect mutually exclusive operations in a CDFG to enable resource sharing. Kim et al. [5] proposed a technique to transform a DFG with conditional branches into an equivalent representation without conditional branches. Raje and Bergamaschi [6] proposed a heuristic algorithm to perform resource sharing for both registers and functional units considering interconnect costs.

Our contribution distinguishes itself in three aspects. First, we incorporate multiplexing delay overhead into the problem formulation and derive a direct relationship between this overhead and the available time slack in the design. Second, we formulate special cases of the resource sharing problem and propose provably good algorithms in addition to a general heuristic. Finally, our technique targets CDFGs containing several hundreds of basic blocks, where a flat representation of the complete CDFG is not manageable.

3. Model of Hardware Compilation Flow

In our generic flow a datapath for each basic block[1] is synthesized individually. This approach has been adopted from existing flows in literature [1, 7]. For a given CDFG, data rate, and throughput constraint, pipeline stages are created where multiple basic blocks can be placed within the same pipeline stage. However, such basic blocks may reside in different pipeline stages and can process different data streams simultaneously. Therefore, our resource sharing techniques operate on a given pipelined CDFG with the assumption that only basic blocks within the same pipeline stage can share resources.

The input to the high-level synthesis stage is a CDFG. This computation model contains both data and control dependencies. CDFGs are commonly used by many state-of-the-art high-level synthesis and compilation flows targeting FPGAs [8-10]. In our CDFG representation, each node corresponds to a basic block. The edges represent the control precedence between the basic blocks and each basic block node has an internal DFG representation.

4. Resource Sharing for Pipelined CDFGs

In this section, we formulate the intra-pipeline stage resource sharing problem and present techniques to solve two variations of this problem and provide proof of optimality for the constant slack case.

4.1. Definitions

In this subsection we present definitions of some basic terms which we use in the remainder of our discussion.

Definition 1. If P is the length of a single pipeline stage, then for each basic block datapath, the difference between P and the length of the

[1] A basic block is the same entity as in the compiler terminology, which refers to a straight-line code segment with a single entry and a single exit point.

longest path passing through a module in the datapath is defined as the **module slack**. We will use module slack and slack interchangeably in this discussion.

Definition 2. Multiplexing is necessary in order to route different inputs to a shared resource. **Multiplexing cost** is defined as the amount of additional delay incurred at the input ports of a module due to multiplexing.

We assume that the multiplexing cost of a 2×1 multiplexer is one unit of latency. We increment the multiplexing cost by one for each additional multiplexer input. Hence, the multiplexing cost of a k×1 multiplexer will be k-1. Although such a direct relationship may appear as an overly simplified assumption, our experiments reveal that multiplexers with more than 4 inputs are rarely used. Hence, their delays can be expected to correlate within such a narrow spectrum, i.e. from a 2×1 multiplexer to a 4×1 multiplexer.

Definition 3. Within a basic block datapath two modules are defined to be **slack independent**, if no path exists from one module to the other in the datapath. In that case, any change in the slack or delay of one module would not affect the other module. They are defined to be **co-dependent** otherwise.

4.2. Intra-Pipeline Stage Resource Sharing

Given a pipelined CDFG with datapaths synthesized for individual basic blocks, our problem is to find the maximal resource sharing among functional units without exceeding the given latency of a pipeline stage. While pursuing this goal we restrict our resource sharing techniques only between basic blocks within the same pipeline stage. Also, no two modules from the same basic block can be shared because it might violate the schedule of operations within that basic block.

4.2.1. Problem Formulation

The intra-pipeline stage resource sharing problem is formulated as follows:

Given a set of basic block datapaths $S = \{bb_1, bb_2, ..., bb_m\}$ and a pipeline stage latency of P, find a set of modules that can be shared among multiple basic blocks within each stage to:

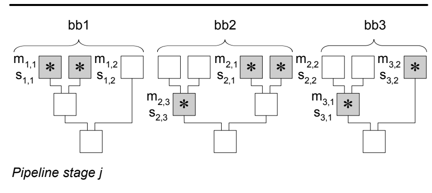
minimize *number of modules in each pipeline stage*

subject to *critical_path(bb_i) $\leq P$, for $1 \leq i \leq m$*

Resource sharing of different resource types can be performed separately because only those resources with same functionality can be shared. However, the sharing decisions for different resource types will have a relation, because within a datapath multiplexing cost will be introduced due to sharing of a certain type of resource. This additional delay will affect the availability and distribution of slack for those modules that are co-dependent with the module under consideration. Hence, this will restrict the possibility of sharing co-dependent modules of other types when it is their turn to be considered for sharing. However, having lesser number of resource types can alleviate the negative impact of co-dependency, based on the following observations. First, in certain application domains such as signal processing and multimedia there are only a few fundamental operations that occur most frequently. Second, as the operations become simpler in nature (basic logical operations like AND, OR, etc.), the gain through sharing is overshadowed by the area overhead of using multiplexers.

4.2.2. Important Underlying Assumptions

Let us summarize our operating assumptions before we discuss our resource sharing techniques. We assume that for each pipeline stage we are given a set of modules representing each datapath in the stage. All sets contain the same resource type and within each set modules are independent.



$R = \{(s_{1,1}, s_{1,2}), (s_{2,1}, s_{2,2}, s_{2,3}), (s_{3,1}, s_{3,2})\}$

Figure 1: A sample input for the intra-pipeline stage resource sharing problem.

A snapshot of a possible input to our resource sharing problem is depicted in Figure 1. As shown in the figure, there are three basic blocks in the pipeline stage j. Modules shaded in gray in the datapaths are multipliers in this example. Each module $m_{i,k}$ in each basic block bb_i is associated with a slack value $s_{i,k}$. Then, the input to the resource sharing problem is a *set of sets R* containing the slack values of independent modules with positive slack. Minimizing the resource requirement for these modules will then be equivalent to minimizing the overall resource demand for a given resource type in that pipeline stage. In our technique we use these slacks effectively to perform resource sharing without affecting the pipeline stage latencies and hence the throughput. In the following subsections we will describe different cases for intra-pipeline stage resource sharing problem.

4.2.3. Case 1: A Pipeline Stage with Constant Amount of Slack for All Modules

Assume that we are given a pipeline stage with m basic blocks and the i^{th} block contains $|bb_i|$ independent modules of a given type. Let all modules across all basic blocks have the same slack value of s. Then the input set of sets R to our resource sharing problem is:

$R = \{(s_{1,1}, \ldots, s_{1,|bb1|}), \ldots, (s_{m,1}, \ldots, s_{m,|bbm|})\}$

Constant-Slack Resource Sharing (R, *s*)

```
Step 1    Sort the groups in R according to their number of elements
Step 2    row_count = number of elements in the largest group
Step 3    Starting from the first element of the first group place elements
          into a matrix M with row_count rows in column major order
Step 4    Divide M into sub-matrices of (row_count × s+1) dimension
Step 5    m = number of matrices after division
Step 6    if m == 1
Step 7        Minimum number of resources = row_count
Step 8        Replace modules in each row with one shared module
Step 9        return
Step 10   else
Step 11   if columns_in_last_matrix > ceiling ((s+1)/2)
Step 12       swap_shift_dense(first full matrix, partial matrix, s)
Step 13   else
Step 14       swap_shift_sparse(first full matrix, partial matrix, s)
```

Figure 2: Pseudo code for the Constant-Slack Resource Sharing Algorithm.

Although all slack values are equal, we use indexing to differentiate between the modules within and across basic blocks. We now describe our **Constant-Slack Resource Sharing** algorithm that *optimally* minimizes the number of modules required. We outline the Constant-Slack Resource Sharing algorithm in Figure 2 followed by the details of the post-processing procedures.

The sets within R are sorted in non-increasing order of the number of elements they contain. Each set in R is referred to as an *Old Group*. These groups are transformed into *New Groups*, where each set represents a sharing from multiple basic blocks. This transformation is the core of our Constant-Slack Resource Sharing algorithm.

We start with constructing a *module distribution matrix*. If K is the size of the largest set in R, then this matrix has K rows and $\lceil (total\_number\_of\_modules / K) \rceil$ columns. Let us denote this number of columns as C. We then place all nodes from each Old Group into this matrix in column major order so that no two modules from the same basic block are shared.

swap_shift_dense(first full matrix, partial matrix, s)

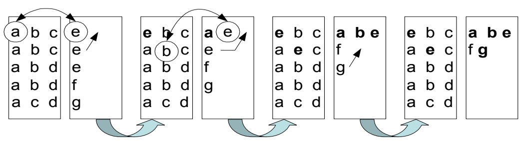
```
Step 1   for each non-full row of the partial matrix starting from bottommost
Step 2       for each element E in the non-full row
Step 3           if there is no element from the same group as E in the topmost
                 non-full row
Step 4               Shift E into the topmost non-full row
Step 5           else
Step 6               E* = element in the topmost non-full row of the same
                     type as E        // this element is in conflict with E
Step 7               Swap E* from the topmost non-full row with the element
                     in the corresponding location in the first full matrix
Step 8               Shifts E up into the topmost non-full row
Step 9               if topmost non-full row reaches s+1 elements
                         break
```

swap_shift_sparse(first full matrix, partial matrix, s)

```
Step 1   diagonal_pos = number_of_columns in partial matrix
Step 2   for each non-full row of the partial matrix
Step 3       for all elements in the row immediately below
Step 4           if there is no element from the same group in upper row
Step 5               Shift elements into the upper row
Step 6           else
Step 7               Swap conflicting elements of the upper row with the
                     elements in the corresponding location (shifted to
                     right by diagonal_pos) in the first full matrix
Step 8               Shift E from lower row up into the non-full row
Step 9               Shift remaining elements in the lower row to the left
Step 10              Shift all remaining rows one position up if needed
Step 12              if non-full row reaches s+1 elements
Step 13                  break
```

Figure 3: *swap_shift_dense* and *swap_shift_sparse* procedures.

We divide these columns into groups of (s+1) starting from the leftmost column. Note that s is the constant slack amount possessed by all modules, so that for the (s+1) modules we have a multiplexing cost of s, thereby effectively using the slack amount. We will obtain $\lceil C / (s+1) \rceil$ matrices. All except the last one of these matrices will be of dimension ($K \times (s+1)$). We refer to those as *full matrices*. The last matrix will contain K rows, but possibly less than $s+1$ columns. We refer to this matrix as the *partial matrix*. Now, each row of these full and partial matrices is called a *New Group*.



before sharing: 20 modules
after sharing: 7 modules

Figure 4: Execution of swap_shift_sparse procedure.

Next we post-process the partial matrix to further maximize sharing. We refer to this step as the *swap and shift* step. The main idea behind the swap and shift procedure is to push as many elements from lower rows of the partial matrix to the upper rows thereby creating rows of full (s+1) element count. This in turn, reduces the total number of *New Groups*, which is the number of resources needed after sharing (in addition to zero-slack modules, which cannot be shared). There are two different cases we need to handle at this stage. First, when the number of columns in the partial matrix is less than (s+1)/2, we refer to it as the *sparse* case. Second, when the number of columns in the partial matrix is larger than (s+1)/2 it is called the *dense* case.

We present two procedures: *swap_shift_dense* and *swap_shift_sparse* for these two cases (shown in Figure 3). Our algorithm runs in O($n$lg$n$ + Ks) time, where n is the total number of *Old Groups*. Figure 4 illustrates a sample execution of the *swap_shift_sparse* mechanism.

Theorem 1. The Constant-Slack Resource Sharing algorithm produces a sharing with minimum number of resources. Hence, Constant-Slack Resource Sharing algorithm is optimal.

Proof: If after the initial division all m matrices are full, all modules are maximally shared. Therefore, we have obtained the optimal solution with $K \times m$ (for K rows and m matrices) modules.

If the m^{th} matrix is not full there are two possible cases, either the partial matrix is *sparse* or *dense*, as defined above. First, let us assume that the partial matrix is sparse. The *swap_shift_sparse* procedure eliminates one row at a time from the partial matrix by shifting elements to the uppermost non-full row. In case of a conflict, i.e., if there is an element in the non-full row belonging to the same *Old Group* as the element we are trying to shift up, we only need to swap it with an element of a full matrix. Every shift operation is guaranteed to find an element that will not create a conflict with any element in the non-full row under consideration as well as with any element that we are trying to push up into the non-full row. The only possible conflict can occur between the current elements of the non-full row and the elements to be pushed. As we use one row of the full matrix to resolve conflicts we also alter the positions (along the diagonal of the full matrix, with a stride of the number of elements swapped at a time) from which we take the particular elements for swapping operation, as we move one row at a time through the full matrix. This ensures that as we push multiple rows upwards to fill one and the same non-full row, we will not be using the same type of elements from the full matrix to resolve conflicts. Because this would create conflicts with the elements that we transferred from the full matrix in the previous swap operations.

For the *dense* case, we perform one element swap at a time to push one element to the non-full row. For each row that we try to eliminate by pushing its elements upwards, we are guaranteed to find an element from the full matrix to resolve a conflict. To eliminate one row, we can again use the elements in a single row of the full matrix. In this case, because the non-full rows in the partial matrix contain more than (s+1)/2 elements to begin with, we will never need to push more than one row into a non-full row before it is completed. Therefore, we will never need to swap elements of same *Old Group* more than once into the same row of the full matrix. Hence, we do not need to transfer elements along the diagonal of the full matrix.□

4.2.4. Case 2: A Pipeline Stage with Different Slack for Different Modules

We propose an efficient heuristic to perform resource sharing for the problem instance with arbitrary number of basic blocks in a stage, where each module can possess a different amount of slack. The input to our algorithm is R and we try to minimize the number of *New Groups* as before. Figure 5 summarizes our heuristic algorithm.

Intra_Stage_Resource_Sharing(R)

```
Step 1   List_OG ← Ø// contains all nodes in non-increasing order
Step 2   List_NG ← Ø// contains groups of nodes
Step 3   for i = 1 to n
Step 4       for each node in i
Step 5           insert in List_OG in non-increasing order
Step 6           for each node in List_OG
Step 7               Search for a suitable group for the node starting from the last
                     group in List_NG
Step 8               if group found
Step 9                   Insert node at the end in the group found
Step 10                  Update List_NG with this updated group
Step 11              else
Step 12                  Make a new group
Step 13                  Insert node in group
Step 14                  Insert group at the end in List_NG
```

Figure 5: Pseudo code for our Intra-Pipeline Resource Sharing heuristic.

The nodes are arranged in non-increasing order of their slack values, and redistributed to *New Groups*. Starting from the first node of List_OG, it is checked to see if it can be added to any existing group in List_NG, starting from the last group. To check if the node can be inserted in any existing group, the slack value is compared to the last element of that group. If the difference between the size of the current group and the slack value is less than 1, it means that the node can be inserted in this group, provided no two elements are from the same initial group. If a node cannot be inserted in a group, the previous groups are checked until the first group is reached. If a suitable group is not found a new group is created and the node is placed there. This is done iteratively until List_OG is empty.

4.2.5. Resource Sharing with Co-dependent Modules and Multiple Resource Types

So far we have discussed techniques to perform sharing for a single resource type. The problem formulation also assumed that modules in each problem instance *R* are independent. We perform a multi-stage optimization in order to handle the most general case. We first identify the module type, which consumes the highest percentage of total area in terms of number of Lookup Tables (LUTs), which is a good measure of estimating the area requirement for each module.

If multipliers are dominating in terms of area, we place our heuristic in the inner loop of a procedure, where we identify the maximum number of independent multipliers in each datapath and perform sharing for those. Then, we distribute the remaining slack among the rest of the multipliers, if there are any, and choose the next set of independent multipliers. After finishing the multipliers we move on to adders and repeat this procedure.

5. Experimental Results

We have used a subset of the MediaBench multimedia benchmark suite [11] for our experiments. CDFGs were automatically extracted from them by the SUIF and Machine-SUIF compiler. We implemented a simple automated synthesis stage, where each basic block is synthesized through direct mapping, which is commonly used by hardware compilers that map designs onto programmable hardware.

Our resource sharing problem assumes to be given a pipelined CDFG. Development of an automated pipelining scheme is out of the scope of this paper. We levelized each CDFG and assigned all basic blocks in the same level to one pipeline stage. We applied a simple heuristic to balance the number of basic blocks in each pipeline stage. Among all pipeline stages, we identified the one with longest critical path delay. We fixed our pipeline stage latency to this value. Slack values for individual modules are then computed based on this latency and the path delays in individual basic block datapaths. For multipliers and adders, we assumed a latency of 4 cycles and 2 cycles respectively. For all other operations we assumed single cycle latency. We synthesized the VHDL descriptions using Synplify Pro 7.2 from Synplicity and implemented in the Xilinx ISE 4.2i design environment targeting a Xilinx Virtex-E xcv2000e-7bg560 chip.

Table 1: Modules used (adders, multipliers, and multiplexers).

	Jdmerge	adpcm	convolve	getblk	Jctrans
+ before	174	31	59	32	168
+ after	70	18	24	18	53
* before	46	6	8	39	22
* after	24	6	4	19	13
2×1 MUX	43	15	9	22	20
3×1 MUX	17	0	3	3	12
4×1 MUX	6	0	5	2	3
5×1 MUX	1	0	0	0	5
6×1 MUX	3	0	1	0	3
7×1 MUX	2	0	0	0	6

Table 1 summarizes the module distribution before and after sharing. Table 2 presents the area requirements in number of LUTs for each benchmark before and after sharing. We also report the percentage reduction in area. An average reduction by 39.5% is achievable. In the case of adpcm, in the second stage there were very few multipliers, which could not be shared mainly due to the fact that multiplier resources were in basic blocks located in different pipeline stages, which cannot share resources.

Table 2: Reduction in area after synthesis.

	Jdmerge	adpcm	convolve	getblk	Jctrans
No Sharing	11800	1672	2512	8156	2688
Sharing	6252	1524	1280	4140	1552
% reduction	47.0	8.9	49.0	49.2	43.2

Note that this area gain is achieved without increasing the pipeline latency. We have only manipulated the unused slack to share modules so that the timing constraints are never violated. From Table 1, we observe that in all the benchmarks 2×1 multiplexers dominate for which the area requirement is very small.

6. Conclusions

In this work, we performed theoretical analysis of the resource sharing problem for pipelined CDFG synthesis. We also proposed provably good resource sharing techniques for some special cases as well as an efficient general purpose heuristic to solve the Intra-Pipeline Stage Resource Sharing problem. Our techniques exploit the already available slack in a design to account for multiplexing overhead thereby enabling effective resource sharing. Our techniques are complementary to any hardware compilation flow to synthesize pipelined designs from high-level descriptions.

7. References

1. Kaplan, A., P. Brisk, and R. Kastner. *Data Communication Estimation and Reduction for Reconfigurable Systems*. 2003. IEEE/ACM Design Automation Conference.
2. Ziegler, H., et al. *Coarse-Grain Pipelining on Multiple FPGA Architectures*. in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2002.
3. Memik, S.O., et al. *Global Resource Sharing for Synthesis of Control Data Flow Graphs on FPGAs*. in *IEEE/ACM Design Automation Conference*. 2003.
4. Wakabayashi, K. and T. Yoshimura. *A Resource Sharing and Control Synthesis Method for Conditional Branches*. in *IEEE International Conference on Computer Aided Design*. 1989.
5. Kim, T., et al., *A Scheduling Algorithm for Conditional Resource Sharing - A Hierarchical Reduction Approach.* IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, 1994. **13**(4): p. 425--438.
6. Raje, S. and R.A. Bergamaschi. *Generalized Resource Sharing*. in *IEEE/ACM International Conference on Computer Aided Design*. 1997.
7. Cong, J., et al. *Architectural Synthesis Integrated with Global Placement for Multi-Cycle Communication*. in *IEEE/ACM International Conference on Computer Aided Design*. 2003.
8. Chen, D., J. Cong, and Y. Fan. *Low Power High-Level Synthesis for FPGA Architectures*. in *International Symposium on Low Power Electronic Design*. 2003.
9. Gupta, S., et al. *SPARK : A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations*. in *VLSI Design*. 2003.
10. Mei, B., et al. *DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures*. in *International Symposium on Field Programmable Logic*. 2002.
11. Lee, C., M. Potkonjak, and W.H. Mangione-Smith. *MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems*. in *International Symposium on Microarchitecture*. 1997.