

# Dynamic Template Generation for Resource Sharing in Control and Data Flow Graphs

David C. Zaretsky<sup>1</sup>, Gaurav Mittal<sup>1</sup>, Robert P. Dick<sup>1</sup>, and Prith Banerjee<sup>2</sup>

<sup>1</sup> *Department of Electrical Engg. & Computer Science  
Northwestern University  
2145 N. Sheridan Road, L324  
Evanston, IL 60208-3118  
{dcz, mittal, dickrp}@ece.northwestern.edu*

<sup>2</sup> *College of Engineering  
University of Illinois at Chicago  
851 South Morgan Street  
Chicago, IL 60607-7043  
prith@uic.edu*

## Abstract

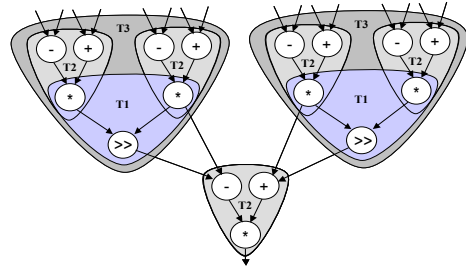
*High-level synthesis compilers often produce reoccurring patterns in intermediate CDFGs during translation. By identifying large reoccurring patterns, one may reduce area and communication overhead by efficiently reusing hardware for multiple operations. This paper presents an algorithm for dynamically generating templates of reoccurring patterns for resource sharing in CDFGs. Results show 40-80% resource reduction using small, incremental template growth, and variations within a 5% margin among varying look-ahead depths.*

## 1. Introduction

Traditionally, the high-level synthesis problem is one of transforming an abstract design into a detailed hardware specification. The abstract design is generally converted into an intermediate CDFG that is optimized to improve design power, frequency, timing, and area. It is interesting to observe that high-level synthesis compilers often produce reoccurring patterns in the resulting CDFG during translation. We surmise that it is possible to re-associate these reoccurring patterns of instruction sequences into collections of operation sets, or *templates*, that can be used to share hardware resources to reduce area and improve placement and routing. The concept of extracting reoccurring patterns in a CDFG is called *regularity extraction*.

Figure 1 illustrates a CDFG with reoccurring patterns of operations, allowing several possible resource sharing schemes. Sharing only individual functional units, such as adders and multipliers, would result in the removal of 13 operations. However, the cost of routing and multiplexing logic to share a single functional unit for an entire design is unacceptably high. One might assume it is preferable to use a small

number of large templates in sharing resources. However, growing template T3 only removes 7 operations. Consequently, implementing template T2 would produce the best coverage. This example illustrates two important aspects in template generation: (1) a cost function is essential in the decision factor for growing templates, and (2) template selection has a direct impact on future growth. For instance, selecting T1 will prevent the growth of T2 due to overlapping structures.



**Figure 1. Reoccurring patterns in a CDFG.**

The contribution of this work is an area optimization algorithm that uses regularity extraction to generate templates of reoccurring patterns for resource sharing. This work was motivated by our experience with the FREEDOM compiler [9][10], which translates software binaries into hardware descriptions for FPGAs. The proposed approach is related to the technology mapping and the graph covering problems. However, in the classical technology-mapping problem, a static library of modules is provided with the goal of covering the CDFG with a mapping of each operation to a gate in the technology library. For reconfigurable devices, or at high levels of abstraction, static libraries may not be readily available, requiring a more dynamic approach. We use a heuristic to dynamically grow templates based on a cost function and the frequency of

reoccurring patterns within the CDFG. We apply backtracking and varying look-ahead depth when growing templates to evaluate the trade off between solution quality and run time.

## 2. Related Work

Many approaches to resource sharing have been evaluated. The graph-covering problem may be solved optimally using a binate covering formulation. However, this problem is NP-Complete. Villa et al. [8] proposed acceleration techniques based on cost bounding. However, even at its best, binate covering can be slow for large CDFGs.

Many researchers have resorted to heuristics in order to obtain solutions quickly. Memik et al. [2] proposed a method of resource sharing for FPGAs using a combination of five heuristics to merge resources across basic blocks in a CDFG. They assume a technology library is provided for the target FPGA and the operation nodes have been annotated with area and delay measurements. Callahan et al. [1] described a method for mapping operations to FPGAs given a library of templates to match. Prior to performing graph covering, the DAGs are partitioned into trees. However, this can result in a significant area penalty. Chowdhary et al. [5] proposed a dynamic template generation algorithm that considers the complete set of templates for trees and single-output DAGs. The complexity of considering all possible templates can be high. Rao et al. [3] and Kastner et al. [11] proposed methods for extracting templates from DAGs based on incremental template matching. However, these greedy approaches may become trapped in local minima. Guo et al. [4] proposed a similar method for the MONTIUM architecture with a size limit for generating templates. However, the templates are grown based on frequency of occurrence; they do not consider a cost function or discuss the degree to which their solutions deviate from optimality.

## 3. Linear DAG Isomorphism

The key to regularity extraction is identifying common patterns in a CDFG. Unlike trees, for which dynamic programming may be used to quickly find optimal solutions for template matching, heuristics are commonly used to identify isomorphic patterns in DAGs. Gemini [6] was developed as a means of validating circuit layout through graph isomorphism using a graph coloring technique. Rao et al. [3] proposed a string matching technique for comparing graphs using very simple string formulas, called *K-formulas*. Zibin et al. [7] presented efficient methods of solving the linear isomorphism problem by

utilizing commutative and associative properties of operations.

We have implemented a similar linear DAG isomorphism technique using string matching that encapsulates additional properties of each operation, including predicates, sign, precision, and bit-range select. The algorithm takes as input a *nodeset* consisting of a single-rooted DAG, sorted topologically in order of leaves to root. Beginning with the leaves, the algorithm calls a hash function on each node in the *nodeset* to generate a string representation of the operation as a function of its properties and the hash expressions of its input nodes. The hash function uses the associative and commutative properties of operations by sorting the input hash values alphanumerically. Two DAGs are isomorphic if the hash values of their *root* nodes are equivalent.

## 4. Growing Templates

Templates are grown by first building *nodesets* and then matching DAGs. A *nodeset* is grown by performing a depth-first traversal of a DAG beginning at a root node, and incrementally adding the nodes at each stage of the hierarchy until a maximum *look-ahead* level has been reached, which is bounded by the height of the data flow graph. We consider all permitted levels of look-ahead simultaneously, e.g., a look-ahead value of 2 produces nodesets for look-ahead of 0, 1, and 2. When growing subsequent levels, two nodesets may only be combined via their *root* node. Figure 2 shows the nodesets and corresponding hash expressions generated from a DAG with a *look-ahead* depth of 2.

The order in which templates are grown is important. If a secondary path exists between a nodeset and a joining node in which an intermediate node lies, the joining node may not be added to the set unless the node on the intermediate path is first included. Otherwise, a cycle is produced. In Figure 2, the subtract node has a *reconverging path* through the multiply operation to the root. Had the subtract node been added first, the multiply node would become both an input and output to the nodeset, resulting in a cycle. Consequently, the multiply node is added in the second stage, and the subtract node is added in the third stage.

In each stage of template growth, we check for *reconverging paths* by performing a depth-first traversal of all paths from a node before adding it to a nodeset. A path containing a node that does not belong to the nodeset is said to have diverged. The presence of any subsequent nodes along the same path that belong to the nodeset indicates a *reconverging path*.

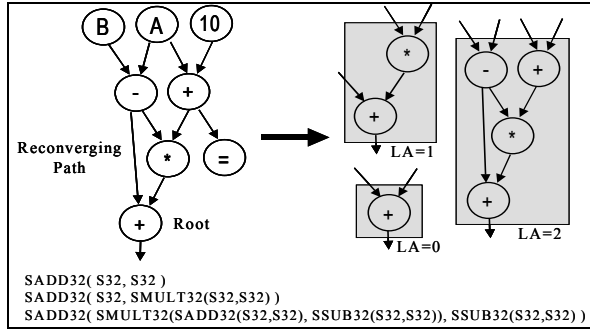


Figure 2. Generated nodesets and expressions.

## 5. Template Matching and Selection

Each nodeset is added to a *table of expressions* under its respective hash key value, resulting in a complete mapping of expressions to a list of equivalent nodesets. During this procedure it is necessary to remove any overlapping nodesets that arise locally within each set of template matches. Villa et al. [8] describe a method of obtaining a close approximation of the maximal independent set using an adjacency matrix, in which the number of nodesets is maximized by first selecting those that conflict with the least number of nodesets, and then eliminating any rows/columns that intersect with a '1'. Expressions with less than two matches are pruned, since there is no benefit to instantiating single-match templates.

Once the table of possible template matches is constructed, template matches are selected for implementation from the table of expressions using a cost function based on the number of operations covered, less the cost of implementing the template with replicated hardware (see Section 6). Different techniques may be considered when choosing a cost function. If precise area measurements are available for operations, it may be feasible to obtain nearly minimal area solutions. For our purposes, we chose a wiring cost for several reasons. First, it is often difficult to approximate the area cost at an abstract level, since we must predict how the resulting CDFG will be translated to hardware. Second, we wish to give preference to large, complex networks in order to reduce interconnect and simplify routing. Finally, reducing the nets effectively reduces the area as well.

After selecting a template from the table of expressions, all remaining nodesets in the table that conflict with these matches are pruned. This process continues until the table is either empty or no suitable matches are found. The next level of nodesets is then grown and added to the table of expressions, followed by selection of new template matches. During this process, some template matches may be combined into larger ones, possibly resulting in single template

matches. These matches are pruned and its nodes are added back to the pool for future template matching, since they produce no cost benefit

## 6. Building Template Structures

To ensure that a template performs the same function for all isomorphic DAG structures, it must have a tree structure. The disadvantage of this approach is that it may result in an increase in area. However, this increase is bounded because it is possible to prune a template if it is deemed too costly to implement for the number of available matches. The template tree structure can be constructed from any of the matched nodesets by decomposing the DAG into a tree using a post-order traversal, beginning at the root node. For each template match, we identify all fanin and fanout nodes with edges to external operations, adding them as input and output ports to the template.

## 7. Dynamic Resource Sharing

Figure 3 presents a *Resource Sharing* algorithm, comprised of the techniques described in this paper. The procedure takes as input a graph  $G$ , a *cost function* for template selection, a *look-ahead* value for generating the table of template expressions, and a *backtracking* value. As described earlier, each template selection impacts future template growth. Thus, the question arises: if the second, third, or fourth best template expression were chosen first, how would this affect the end results?

Using backtracking, we can evaluate how well the solution compares with optimality by selecting the  $i^{th}$  best template expression first, and then selecting the implementation of templates with the best cost at the end of each iteration. The backtracking depth is bounded by the size of the table of expressions.

```

ResourceSharing(G, costfunc, look_ahead, backtrack)
1  best_T = NULL
2  best_cost = 0
3  changes = true
4  while ( changes ) do
5    changes = false
6    E maps exprs to nodeset lists
7    T maps templates to nodeset lists
8    GenerateTableOfExprs(G, E, look_ahead)
9    for i = 0 to min(backtrack+1, E.size) do
10     SelectTemplateMatches(E, T, i, costfunc)
11     for each template t in T do
12       template_cost += GetCost(t, costfunc)
13     if template_cost > best_cost then
14       best_cost = template_cost
15       best_T = T
16       changes = true
17   BuildTemplates(G, best_T)

```

Figure 3. Pseudo-code for resource sharing.

## 8. Experimental Results

This section reports results on ten benchmarks using the proposed resource sharing algorithm. The CDFGs were generated from assembly code using the FREEDOM compiler [9][10], and were unrolled several times, thereby increasing design complexity.

Table 1 shows results for varying look-ahead and backtracking depths in terms of the number of templates generated, the maximum template size, the percentage resources reduced, and the total run time for the resource sharing. The percentage resource reduction is the cost of the operations removed, less that required for implementing the templates. Variations in template sizes and resource reduction are due to the template growth order. Note that the number of templates generated decreased as the look-ahead depth increased. This is expected, since larger templates are discovered initially. However, increasing the look-ahead depth caused an exponential growth in the size of the hash values, requiring  $O(2^d)$  time for string matching. This explains the exponential increase in execution times with increased look-ahead depth. The results show quality gains saturate at look-ahead of approximately 3, with reduction in resource usage ranging from 40-80%. In the last set, a backtracking depth of 10 was used to quantify the optimality of the algorithm. The results show that reduction in resource usage only varied within a 5% margin, indicating that our resource sharing algorithm produces efficient results, even using small, incremental template growth.

## 9. Conclusion

This paper presents a regularity extraction algorithm for resource sharing in CDFGs. We use a heuristic to dynamically grow templates based on a cost function and the frequency of reoccurring patterns, while applying backtracking and varying look-ahead depth to evaluate the trade off between solution quality and run time. Experimental results on ten benchmarks indicate that small, incremental template growth is preferable for resource sharing to obtain near-optimal results in reduced CPU time.

## 10. References

- [1] T. Callahan, P. Chong, A. DeHon, J. Wawrzynek, "Fast Module Mapping and Placement for Datapaths in FPGAs," in *Proceedings of the International Symposium on FPGAs*, pp 123-132, Monterey, CA, 1998.
- [2] S. O. Memik, G. Memik, R. Jafari, E. Kursun, "Global Resource Sharing for Synthesis of Control Data Flow Graphs on FPGAs," in *Proceedings of the Conference on Design Automation*, pp 604-609, Anaheim, CA, 2003.
- [3] D. Rao and F. Kurdahi, "Partitioning by Regularity Extraction," in *Proceedings of the Design and Automation Conference*, pp 235-238, Anaheim, CA, 1992.
- [4] Y. Guo, G. Smit, H. Broersma, and P. Heysters, "A Graph Covering Algorithm for a Coarse Grain Reconfigurable System," in *Proceedings of the Conference on Language, Compiler, and Tool for Embedded Systems*, pp 199-208, San Diego, CA, 2003.
- [5] A. Chowdhary, S. Kale, P. Saripella, N. Sehgal, and R. Gupta, "A General Approach for Regularity Extraction in Datapath Circuits," in *Proceedings of the International Conference on Computer-Aided Design*, pp 332-339, San Jose, CA, 1998.
- [6] C. Ebeling and O. Zajicek, "Validating VLSI Circuit Layout by Wirelist Comparison," in *Proceedings of the IEEE International Conference on Computer Aided Design*, pp 172-173, 1983.
- [7] Y. Zibin, J. Gil, and J. Considine, "Efficient Algorithms for Isomorphisms of Simple Types," in *Proceedings of the Symposium on Principles of Programming Languages*, pp 160-171, New Orleans, LA, 2003.
- [8] T. Villa, T. Kam, R. Brayton, and A. Sangiovanni-Vincentelli, "Explicit and Implicit Algorithms for Binate Covering Problems," in *IEEE Transactions on Computer-Aided Design*, vol. 16, pp 677-691, 1997.
- [9] D. Zaretsky, G. Mittal, X. Tang, and P. Banerjee, "Overview of the FREEDOM Compiler for Mapping DSP Software to FPGAs," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp 37-46, Napa, CA, 2004.
- [10] G. Mittal, D. Zaretsky, X. Tang, and P. Banerjee, "Automatic Translation of Software Binaries onto FPGAs," in *Proceedings of the Conference on Design Automation*, pp 389-394, San Diego, CA, 2004.
- [11] R. Kastner, S. Memik, E. Bozorgzadeh, M. Sarrafzadeh, "Instruction Generation for Hybrid Reconfigurable Systems," in *ACM Transactions on Design Automation of Electronic Systems*, vol. 7, issue 4, pp 605-627, 2002.

**Table 1. Benchmark results for resource sharing with varying look-ahead and backtracking depth.**

Benchmark	Lookahead = 1				Lookahead = 3				Lookahead = 7				Lookahead = INF				Lookahead = INF, BT = 10			
	#	Max	Perc.	T(s)	#	Max	Perc.	T(s)	#	Max	Perc.	T(s)	#	Max	Perc.	T(s)	#	Max	Perc.	T(s)
dot_prod	5	24	69.2	3.6	4	18	75.3	4.8	4	18	75.3	9.0	4	18	75.3	23.7	4	18	75.3	31.5
iir	9	18	78.4	19.5	6	48	76.8	29.8	6	48	76.8	47.6	6	48	76.8	70.8	6	48	76.8	120.6
matmul_32	10	24	61.3	4.2	9	18	64.7	7.1	6	18	59.9	11.3	6	18	59.9	20.7	9	20	61.6	36.9
gcd	7	5	43.6	0.8	6	5	43.6	0.8	5	5	41.0	0.8	5	5	41.0	0.8	5	5	41.0	2.1
diffeq	18	9	55.5	4.4	8	93	45.9	7.3	9	10	50.3	17.3	9	10	50.3	23.6	9	10	50.3	31.4
ellip	4	3	64.6	1.6	2	3	62.7	1.6	2	3	62.7	1.9	2	3	62.7	1.9	2	3	62.7	3.2
laplace	9	5	67.4	6.5	7	5	63.2	8.4	7	5	63.2	15.2	7	5	63.2	20.8	7	5	63.2	31.3
fir16tap	9	24	61.8	4.9	7	18	67.4	6.3	5	18	65.2	9.7	5	18	65.2	17.5	5	18	65.2	29.3
fir_cmplx	15	21	71.3	25.1	12	37	65.7	36.8	8	37	69.3	65.7	8	37	69.3	156.6	12	37	71.8	231.2
sobel	13	8	79.2	38.3	11	8	77.1	49.4	11	13	76.9	54.9	10	13	74.9	48.0	11	13	74.8	164.9