Synthesis of Power-Optimized and Area-Optimized Circuits from Hierarchical Behavioral Descriptions [*]

Ganesh Lakshminarayana and Niraj K. Jha

Dept. of Electrical Engineering, Princeton University, NJ 08544

Abstract

We present a technique for synthesizing power- as well as area-optimized circuits from hierarchical data flow graphs under throughput constraints. We allow for the use of complex RTL modules, such as FFTs and filters, as building blocks for the RTL circuit, in addition to simple RTL modules such as adders and multipliers. Unlike past techniques in the area, we also customize the complex RTL modules to match the environment in which they find themselves. We present a fast and efficient algorithm for mapping multiple behaviors onto the same RTL module during the course of synthesis, thus allowing our synthesis system to explore previously unexplored regions of the design space. These techniques are at the core of an iterative improvement based approach which can accept temporary degradation in solution quality in its quest for a globally optimal solution. The moves in our iterative improvement procedure explore optimizations along different dimensions such as functional unit selection, resource allocation, resource sharing, resource splitting, and selection and resynthesis of complex RTL modules. These inter-related optimizations are dynamically traded off with each other during the course of synthesis, thus exploiting the benefits that arise from their interaction. The synthesis framework also tackles other related high-level synthesis tasks such as scheduling, clock selection, and V_{dd} selection. Experimental results demonstrate that our algorithm produces circuits whose area and power consumption are comparable to or better than those produced using flattened synthesis, within much shorter CPU times. The efficacy of our algorithm in the power-optimization mode is illustrated by the fact that it produces circuits that consume upto 6.7 times less power than area-optimized circuits working at 5 $Volts$ at area overheads not exceeding 50%.

1 Introduction

High-level synthesis is the process of deriving an optimized register-transfer level (RTL) architecture from a behavioral description, usually specified as a data flow graph (DFG) for data-dominated circuits. In this paper, we present a technique for synthesizing power- as well as area-optimized data-dominated circuits from hierarchical DFGs under throughput constraints. *Data-dominated* behaviors have a predominance of arithmetic operations, and an absence of control-flow constructs. They are commonly encountered in digital signal processing and image processing applications.

[*]This work was supported in part by NSF under Grant No. 9319269 and in part by Alternative System Concepts, Inc. under an SBIR contract from Air Force Rome Laboratories.

Permissions to make digital/hard copy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 98, San Francisco, California
(c) 1998 ACM 1-58113-049-x/98/06 ..$5.00

Hierarchical high-level synthesis can be divided into two sub-problems, (i) deriving hierarchical information from a flattened behavioral description, and (ii) synthesizing a circuit using a hierarchical behavioral description. The latter problem, which is addressed in this paper, was also considered in [1, 2, 3, 4]. In [1] and [3], the problem of synthesizing area-optimized circuits from hierarchical behaviors was considered. The technique presented in [4], which was geared towards applying power-optimizing transformations, supports two levels of hierarchy. Past research in flattened high-level synthesis for low power has considered allocation and assignment [5, 6], module selection [7], scheduling [6], and transformations [4]. A technique which solved the problems of allocation, assignment, module selection, scheduling, clock selection, and V_{dd} selection simultaneously to produce power-optimized architectures was presented in [10]. We compare experimental results obtained by our method with those presented in [10].

The following general observation can be made about existing hierarchical high-level synthesis techniques: they handle large designs much faster than flattened high-level synthesis techniques, however, the solutions produced are generally of inferior quality. This is because fine-grained optimizations, which are within the scope of flattened synthesis techniques, are not visible while performing hierarchical synthesis. Our work aims at combining the global perspective afforded by hierarchical synthesis with the flexibility of flattened synthesis to produce compact, power-optimized architectures, within short periods of time. Our algorithm accepts as input a hierarchical DFG (arbitrarily deep hierarchies are allowed), and a library of modules, which can be *simple* (adders, multipliers) or *complex* (RTL modules implementing specific DFGs). Other inputs include typical input traces to aid power estimation, and a throughput constraint for synthesis. Our algorithm works iteratively, *i.e.*, it begins with an initial solution, which is iteratively refined. This can happen in four ways: (A) simple or complex modules in the initial or intermediate solution are replaced by new ones, chosen from the library, (B) complex modules are resynthesized by descending the hierarchy, (C) a pair of simple (complex) modules is combined into one simple (complex) module which executes the functionality of both its constituents, and (D) a simple (complex) module is split into multiple simple (complex) modules. Move *A* allows us to take advantage of user-specified simple and complex modules, and move *B* allows us to tailor the complex modules to their environments. Move *C* helps create compact circuits, and move *D* helps create new optimization opportunities. Note that in case of move *B*, modules, whose internal descriptions are not available or cannot be altered, are not resynthesized. Furthermore, the algorithm can support chained, multi-cycled, and pipelined functional units, and deal with multi-function arithmetic-logic units (ALU's). It enables escape from local minima by considering a sequence of moves at a time, where individual moves can have negative gain, *i.e.*, lead to temporarily degraded RTL architectures.

2 Background

In this section, we introduce the basic concepts used in our work. We first explain the notion of hierarchical behaviors and cir-

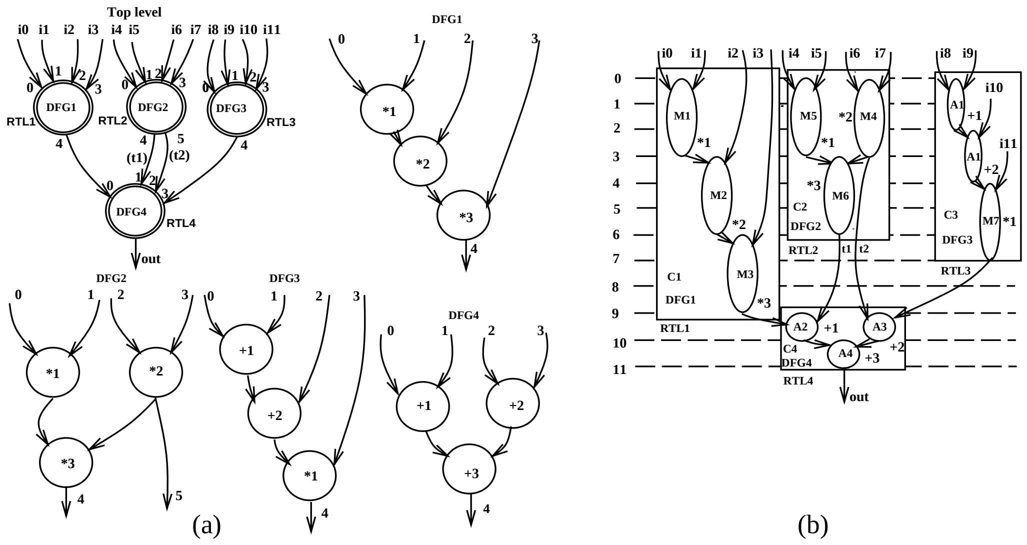

Figure 1: (a) A hierarchical DFG, and (b) a scheduled and assigned hierarchical DFG

cuits with an example.

Figure 1(a) shows a hierarchical DFG. The DFG labeled "Top level" is the *top-level DFG*. Nodes of the DFG can represent *operations*, like addition and multiplication, or *hierarchical nodes* like convolution and filtering. In general, if the input hierarchy is preserved, hierarchical nodes usually represent specific, well-characterized behaviors. However, if the hierarchy is the output of a partitioning process, hierarchical nodes could represent arbitrary behaviors. The edges in the top-level DFG entering and exiting hierarchical nodes are annotated with numbers which represent connectivity between the top-level edges and edges in the underlying DFG. For instance, the edge labeled 0 feeding node *DFG1* in the top-level DFG is connected to edge 0 in *DFG1*. The DFGs representing hierarchical nodes can in turn consist of simple and hierarchical nodes. Although the hierarchical DFG shown in Figure 1(a) is acyclic, the system is capable of handling loops at any hierarchical level.

A hierarchical node is implemented by an *RTL module*. Just as multiple operations can map to the same module library element, multiple hierarchical nodes can map to the same RTL module. An RTL module can itself be described hierarchically as an interconnection of RTL modules, functional units, multiplexers, and registers.

The *environment* of an RTL module for a hierarchical node mapped to it is an ordered set consisting of the arrival times of the inputs to the hierarchical node and the times when the outputs of the hierarchical node are *consumed*. The environment is defined only for RTL modules which are part of a scheduled, allocated, and assigned RTL circuit. The *profile* of an RTL module for a hierarchical node is defined as an ordered set consisting of the *expected* input arrival times and output arrival times. Unlike the environment, profile is defined for any module, irrespective of whether it is a part of an RTL circuit.

Example 1: Consider the hierarchical DFG shown in Figure 1(a) again. The hierarchical nodes are annotated with their names (placed inside the circle representing the node) and the names of instances of the RTL modules they are mapped to (placed outside the circle). For example, hierarchical node *DFG1* is mapped to instance *RTL1*. Figure 1(b) shows a schedule and assignment of the flowgraph in Figure 1(a). The environment of *RTL3* with respect to *DFG3* can be described by the set $Env(RTL3, DFG3) = \{0,\ 0,\ 0,\ 0,\ 9\}$ under the input/output ordering $\{0,\ 1,\ 2,\ 3,\ 4\}$. This environment assumes that the four inputs of *RTL3* arrive at time 0. Note that the output of *RTL3* is consumed by *RTL4* at time 9. The profile of *RTL3* with respect of *DFG3* can be represented as $Profile(RTL3, DFG3) = \{0,\ 0,\ 2,\ 4,\ 7\}$ under the same ordering. This is because the first two inputs of *RTL3* are expected at time 0, the third at time 2 and the fourth at time 4. When the inputs arrive in this fashion, the output is expected at time 7. Given the profile of a module and the input arrival times, the output arrival times can be computed as illustrated next. If the inputs at terminals 0, 1, 2, and 3 of *RTL3* arrive at times 2, 5, 3, and 7, respectively, then *RTL3* can begin operation at $max(2-0, 5-0, 3-2, 7-4) = 5$ cycles, which implies that the output is available after 12 cycles. In this example, each hierarchical node is mapped to a unique instance of an RTL module. In general, however, different hierarchical nodes could map to the same instance of an RTL module and functionally identical hierarchical nodes could map to different RTL modules or to different instances of the same RTL module. ■

At this stage, we formalize the notion of throughput constraint. For the class of applications we consider, the circuit should be able to process inputs at a constant rate, and it does not pay to be able to process inputs any faster. The sampling period is the number of time units allowed for completing one iteration of the design. The throughput is the inverse of the sampling period, *i.e.*, it is the number of samples processed per unit of time (clock cycles).

3 Motivational examples

In this section, we motivate key algorithmic features. The main constituents of our algorithm are the moves we use to refine the initial solution. As mentioned in Section 1, moves are of four types. The first two types of moves involve replacement or resynthesis of an existing module, and the next two types involve merging and splitting of modules. Note that modules can be simple or complex. Example 2 illustrates the benefits of the first two move types and Example 3 illustrates the benefits of the third.

Example 2: In this example, we illustrate the following: combining *coarse-grain* knowledge available at the higher levels of the hierarchy with *fine-grain* techniques traditionally associated with flattened high-level synthesis can significantly enhance solution quality. To this end, we perform moves of type *A* (which leverage off coarse-grain knowledge by replacing an instance of an RTL module by another one, better suited to the environment), and type *B* (which perform fine-grain resynthesis), on an example RTL circuit to demonstrate the power savings obtainable.

Consider the problem of synthesizing a power-optimized RTL circuit for the hierarchical DFG shown in Figure 1(a), under a sampling period constraint of 12 cycles. The simple modules present in the module library, their areas and delays (in number of clock cycles) are summarized in Table 1. These library modules also have

Table 1: Summary of functional unit and register properties

	add1	*add2*	*chained_add2*	*chained_add3*	*mult1*	*mult2*	*reg1*
Area	30	20	60	90	150	100	10
Delay	1	2	1	1	3	5	-

power models, which we omit for brevity [1]. The main fact to be noted is that, to perform the same sequence of operations, *mult2* consumes much less power than *mult1*. Figure 2 shows the complex modules available in the library. For each complex module, the DFG implemented, and the names of the instances of modules implementing individual operations are shown. For example, *RTL* module *C1* consists of three multipliers, *M1*, *M2*, and *M3*, which are of type *mult1*. Chains of functional units are shown enclosed within boundaries. For example, RTL module *C5* is a chain of three functional units of type *add1*. Note that a chain of adders can complete execution almost as fast as an individual adder, enabling a chain of three adders of type *add1* to complete in the same time, in cycles, as one adder of type *add1*. Also, we observe that the complex modules do not use all available resource sharing opportunities, *e.g.*, operations $+1$ and $+3$ in module *C4*, which can map to the same functional unit instance, are mapped to different instances. This is because these modules are power-optimized. Power optimization often (but not always) requires that operations be bound to different functional unit instances, even when the schedule permits the operations to share resources. A detailed discussion of the effect of resource sharing on power consumption can be found in [9]. The solution which is input to the procedure for the selection of moves of type *A* and *B* is shown in Figure 1(b). As we can see, DFGs *DFG1*, *DFG2*, *DFG3*, and *DFG4*, are mapped to complex modules of types *C1*, *C2*, *C3*, and *C4*, respectively.

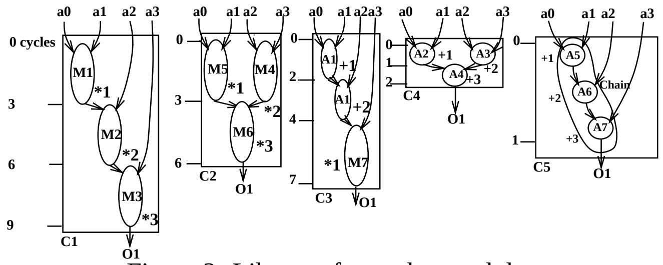

Figure 2: Library of complex modules

We now illustrate the application of moves of types *A* and *B* to the given solution. The first step is the identification of a set of modules to target. *RTL1*, *RTL2*, and *RTL3* are selected by this process. We then try to evaluate the *slack* available to the selected modules, *i.e.*, we ascertain the earliest input arrival times and the latest output arrival times whose satisfaction by the selected modules would ensure the schedulability of the implementation. For example, we can see that the outputs of *RTL2* can arrive nine cycles later than the inputs. In other words, *RTL2* can have a profile of $\{0,0,0,0,9,9\}$, though the profile for *RTL2* in the initial solution is $\{0,0,0,0,6,3\}$. We now apply a move of type *A*, followed by a move of type *B*. A move of type *A* would identify the "best" (lowest power solution if the objective is power optimization, and most compact solution if objective is area optimization) library module which can replace *RTL2*. In this example, the current implementation is indeed the best available in the library, therefore, no changes are made. We then apply a move of type *B*, *i.e.*, we resynthesize *RTL2* with the new, relaxed constraints, to see if we can obtain a better, lower power match for the required profile. The resynthesis procedure for *RTL2* would suggest the replacement of modules *M5* and *M4*, currently of type *mult1*, by *mult2*, which would significantly reduce power consumption. This illustrates the *utilization of high-level knowledge*, which results from a global view of the hierarchy, to *drive synthesis at lower hierarchical levels*, a key algorithmic feature.

The application of a move of type *A* can *change the DFG representing a hierarchical node*. For example, for the case of *RTL1*, module *C2* is recognized as a better candidate than module *C1* (note that *C1* and *C2* implement functionally equivalent behavior), and the substitution is performed. Recognizing such opportunities requires *knowledge provided by the user* regarding the functional equivalence of different DFGs. Many hierarchical DFGs are constructed out of several, commonly-used "building blocks" like *dot-product*, *butterfly*, *etc.*. These building blocks are well studied and have carefully analyzed behaviors, and a number of DFGs describing individual building blocks are available, each with its distinct advantages. A move of type *A* tries to select the "best" DFG which describes a hierarchical node, *i.e.*, the DFG which is best suited to the environmental constraints. ■

A move of type *A* is followed by a move of type *B*, which attempts to improve the RTL module further. This is because the library element selected, though a good match, might not have been designed with the current environmental constraints in mind.

The example given ahead illustrates the benefits of moves of type *C*, which merge two modules into one. Candidate modules for merging can be either simple or complex.

Current hierarchical synthesis systems do not allow different (anisomorphic) DFGs to be implemented by the same RTL module. This often results in implementations with a large area. The main hurdles in performing this merging are as follows:

- It is difficult to determine the best candidate DFGs for merging prior to synthesis.
- Current research tackles the problem of designing an RTL circuit, which can support multiple behaviors, by just considering it as a multi-behavior high-level synthesis problem. Thus, a traditional technique would need to perform scheduling, allocation, and assignment afresh for every pair of modules which need to be merged. Clearly, the complexity of this approach would render it unusable in practice.

Our algorithm overcomes the first limitation because it is iterative, *i.e.*, during the course of the algorithm, different merging configurations are considered, and the best one is selected. Since we need to assess several sharing configurations during synthesis, our procedure to derive an RTL module which can execute multiple DFGs needs to be fast. We solve this problem by just *embedding the RTL modules, which implement the DFGs, into a new RTL module*. The schedule, assignment, *etc.*, for individual DFGs is unaltered, and the merged RTL module cannot execute the DFGs in parallel. The procedure for merging tries to find an embedding which honors clock cycle constraints, while minimizing the area of the merged RTL module.

Example 3 : This example illustrates the area saving obtainable by implementing different DFGs on the same RTL module. In Figure 3, DFGs *DFG1* and *DFG2* are mapped to RTL modules *RTL1* and *RTL2*, respectively. Operations in the DFGs are shown annotated with the names of the modules they are mapped to, and variables are shown annotated with their names (unparenthesized) and the names of the registers they are mapped to (parenthesized). RTL module *NewRTL* can execute both the DFGs, and preserve the original schedules and assignments of the individual DFGs. All three RTL modules presented in this example were placed and routed using tools from the OCTTOOLS suite. Puppy was used

[1] We use the power models presented in [10]. Details of the power estimation methodology can be found in [8]

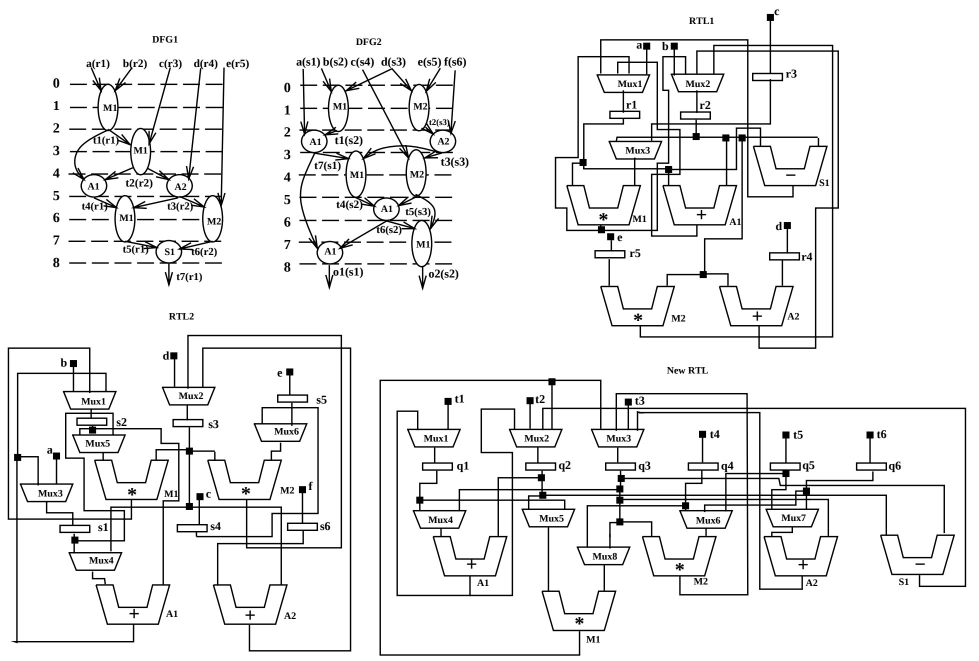

Figure 3: Mapping of two distinct DFGs to the same RTL module

Table 2: Labeling the new RTL module to implement DFG1 and DFG2

New RTL	*q1*	*q2*	*q3*	*q4*	*q5*	*q6*	*A1*	*A2*	*M1*	*M2*	*S1*
RTL1	-	*r1*	*r2*	*r3*	*r4*	*r5*	*A1*	*A2*	*M1*	*M2*	*S1*
RTL2	*s1*	*s2*	*s3*	*s4*	*s5*	*s6*	*A1*	*A2*	*M1*	*M2*	-
Library	*reg*	*reg*	*reg*	*reg*	*reg*	*reg*	*Add1*	*Add1*	*Mult1*	*Mult1*	*Sub1*
Area	*5*	*5*	*5*	*5*	*5*	*5*	*20*	*20*	*50*	*50*	*20*

for placement and Mosaico was used for routing. *RTL1* had an area of 57.94 units, *RTL2*, 53.89 units, and *NewRTL*, 61.67 units. This significant area saving is possible, without significant computational effort, due to the technique of *RTL embedding*, which we have devised. Our technique simply constructs a new RTL module, *NewRTL*, in which the original RTL modules can be embedded. The goal of our procedure is to find the minimum area embedding (including a measure of interconnect) which satisfies clock cycle constraints. The correspondence between the components of *NewRTL* and their counterparts in *RTL1* and *RTL2* is given in Table 2. The module library used and the area of the simple modules are also shown. ■

4 The algorithm

In this section, we present a brief description of the basic components of our synthesis algorithm and their interactions. Details of individual components can be found in [8].

Figure 4 describes the top level of our synthesis algorithm. The algorithm accepts as input a DFG, a library of simple and complex RTL modules, an objective function to optimize (area or power), and a constraint that specifies when the primary inputs arrive and when the primary outputs are expected. The core iterative improvement procedure is enclosed in loops which iterate through the set of available V_{dd}'s and allowed clock periods. [2] The best solution for a given V_{dd} and clock period is obtained through variable-depth *iterative improvement*, a general optimization technique that starts with an initial solution and improves its quality through the application of a sequence of *moves* [11]. This algorithm derives its power from the ability to perform moves which worsen the quality of the solution, as an intermediate step, in striving for a globally optimal solution.

First, the library is searched to find the best possible implementation of the input behavior that satisfies the constraints imposed. If such an implementation cannot be found, one is synthesized using the routine INITIAL_SOLUTION. This routine maps each simple node in the DFG to the fastest implementation available in the library. DFGs which represent hierarchical nodes are handled in the same manner. Each operation is mapped to a separate functional unit, and each variable to a separate register, resulting in a completely parallel architecture. The initial solution is improved through the application of moves of types *A*, *B*, *C*, and *D*. When a move is performed, its validity is checked by scheduling to make sure that the throughput constraints are still met.

The scheduler is called whenever we need to check the validity of an assignment. Before scheduling, we derive an ordering for the operations that need to execute on the same functional unit or RTL module, and variables that need to be stored in the register. This ordering imposes extra dependencies in the DFG, which are modeled as edges. Under this scenario, scheduling of a node reduces to the problem of finding the longest path from a primary input to the node [10]. Scheduling of DFGs is a well-studied problem [12], and we do not discuss it further in this paper.

As mentioned in Example 2, moves of types *A* and *B* alter the implementation of a selected set of RTL modules and functional units with a view to minimizing the power consumption (or area) of the circuit. The application of these moves can be described by the algorithm shown in Figure 5. This algorithm corresponds to statement **7** of Figure 4. As is evident from the pseudocode shown in Figure 5, this process can be divided into three distinct parts: module group formation, constraint derivation, and resynthesis. Mod-

[2] The set of V_{dd}'s and clock periods can be drastically pruned by using a procedure from [10] to obtain the subset that needs to be considered.

```
SYNTHESIZE(DFG $G$, LIBRARY $L$, CONSTRAINT_SET $C$,
                OBJECTIVE $Obj$){
0  foreach supply voltage $V_{dd}$ in the pruned supply voltage set {
1    foreach clock period $clk$ in the pruned clock period set {
2      $Cur\_DP \leftarrow$ INITIAL_SOLUTION($G, L$);
       //If the library cannot supply an implementation of
       //the behavior, generate an initial implementation
3      $Pass\_gain \leftarrow \infty$; $Best\_DP \leftarrow Cur\_DP$;
4      while($Pass\_Gain > 0$){
6        for($i = 0; i < MAX\_MOVES; i++$){
7          MOVE $m1$ = GET_BEST_TYPE-
             _A_AND_B_MOVE($G$, $L$, $Cur\_DP$, $C$, $Obj$);
8          MOVE $m3$ = GET_BEST_RESOURCE-
             _SHARING_MOVE($G$, $L$, $Cur\_DP$, $C$, $Obj$);
9          if ($Gain(m3, Obj) < 0$)
10           MOVE $m3$ = GET_BEST_RESOURCE_-
             SPLITTING_MOVE($G$, $L$, $Cur\_DP$, $C$, $Obj$);
11         if ($Gain(m1, Obj) > Gain(m3, Obj)$)
12           IMPLEMENT_M1_AND_M2($Cur\_DP$, $m1$);
13          else IMPLEMENT_M3($Cur\_DP$, $m3$);
         }
14       $Pass\_Gain$ = Get sequence of moves
         with best cumulative gain;
15       if ($Pass\_Gain \leq 0$)
           break;
16       apply best move sequence to $Cur\_DP$;
       }
17     $Best\_DP$ = best solution seen so far;
}}}
```

Figure 4: Iterative improvement

ule group formation divides the modules in the current RTL implementation into groups for the steps that follow. This is followed by constraint derivation where each operation which is performed by a module belonging to the group under consideration is assigned a new constraint for synthesis. For instance, an addition operation which initially took two cycles to complete could be set a constraint of five cycles, or, a filter which accepted inputs in cycle 0 and produced an output in cycle 5 could be constrained to produce an output in cycle 7 due to this process. The new constraints must preserve schedulability of the implementation, *i.e.*, the new architecture, when rescheduled, should meet throughput constraints. Modules are then resynthesized to meet the new constraints. Note that the term “resynthesis” refers to alterations performed on RTL modules as well as choosing new functional units or RTL modules from the library. The relaxation in constraints might enable replacement of the library element performing the function by a slower, potentially lower power library element.

5 Experimental results

We have implemented the behavioral synthesis framework presented in the previous sections as the program *H-SYN* in 44,000 lines of C++. We have performed experiments to evaluate our techniques using several behavioral descriptions of digital signal and image processing applications. *H-SYN* reads in a textual description of the hierarchical DFG, and performs V_{dd} selection, clock selection, module selection, scheduling, allocation, and assignment to result in a highly power/area-efficient RTL circuit that consists of a datapath netlist, and a finite-state machine description of the controller. The controller and datapath netlists are merged and mapped to the MSU standard cell library using the SIS logic synthesis system. Placement and routing are performed using tools from the OCTTOOLS suite. A switch-level circuit extracted from the layout is simulated by a switch-level simulator, IRSIM, and the capacitance switched is recorded and used to compute the power.

We have synthesized flattened and hierarchical examples of several existing benchmarks to optimize for area and power.

```
GET_BEST_TYPE_A_AND_B_MOVE (DFG $G$, LIBRARY $L$,
 DATAPATH $Cur\_DP$, CONSTRAINT_SET $C$,
OBJECTIVE $Obj$){
//$Obj$ can be area or power
  $Module\_groups = \Phi$;
  $Best\_gain = -\infty$;
  $Module\_groups$ = FORM_MODULE_GROUPS($Cur\_DP$);
  //$Module\_groups$ is a set whose elements are sets of
  // modules which will be considered together for synthesis
  $mgroup = \Phi$ ;
  foreach($mgroup$, $Module\_groups$){
    $Gain = 0$;
    DERIVE_CONSTRAINTS($mgroup$, $Cur\_DP$);
    //derive constraints for resynthesis of each
    //node mapped to module group $mgroup$.
    foreach($M$, $mgroup$){
      Resynthesize module $M$ based on the constraints
      derived in the DERIVE_CONSTRAINTS routine
      $Gain$ = $Gain$ − COST($M$, $Obj$);
    }
    if ($Gain > Best\_gain$){
      store the current solution as the best solution;
      $Best\_gain = Gain$;
}}}
```

Figure 5: Module selection moves

hier_paulin is a hierarchical DFG obtained by unrolling the well-known benchmark, *Paulin*. *dct* implements discrete cosine transform. *avenhaus_cascade*, *dct*, *iir*, and *lat* are hierarchical DFGs which represent filters and are a part of the HYPER package [4]. *test1* is the DFG in Figure 1(a). In all cases, the input hierarchy was preserved. To assess the efficacy of our technique, we compared our hierarchical algorithm to the flattened version of the same algorithm [10]. The flattened technique also takes advantage of reosurce sharing, module selection, V_{dd} selection, clock selection, and scheduling for low power, and is one of the most comprehensive techniques available today.

The results obtained are tabulated in Table 3. Column *L.F.* represents the *laxity factor* used for synthesis. Laxity factor is defined as the ratio of the given sampling period (inverse of specified throughput) to the minimum sampling period that can be attained for the input DFG using the library of modules provided. Columns *Flat* and *Hier* represent flattened and hierarchical versions of the same behavioral description. The circuits in column A were synthesized to optimize for area at a supply voltage of 5 $Volts$ and subsequently voltage-scaled for low power operation. The circuits in column P were synthesized to optimize for power. Rows A and P represent the area and power consumption, respectively, of the synthesized circuits. The area (power) is normalized with respect to the area (power) of an area-optimized, non-V_{dd}-scaled architecture synthesized from a flattened input description at the same laxity factor. For example, if row P of column P, under major column *Hier* is x, then the power consumed by a circuit obtained from a hierarchical behavioral description, optimized for power, is x times the power consumed by a circuit obtained from a flattened behavioral description, optimized for area, and operating at 5 $Volts$. All entries in row A of column A under major column *Flat* are 1. This is because this column represents the area of an area-optimized circuit, synthesized from a flattened behavioral description, and V_{dd}-scaled for low power, normalized with respect to the area of an area-optimized circuit, synthesized from a flattened behavioral description, which operates at 5 $Volts$. Since V_{dd}-scaling makes no difference to area, these entries are 1.

The area-power-synthesis time trade-offs for different laxity factors for flattened (*Fl*) and hierarchical (*Hi*) behavioral descriptions are summarized in Table 4. In this table, the area and power ratios are with respect to flattened area-optimized architectures, and

Table 3: Area (normalized) and power (normalized) results

Circuit	A/P	**L.F. = 1.2**				**L.F. = 2.2**				**L.F. = 3.2**			
		Flat		**Hier**		**Flat**		**Hier**		**Flat**		**Hier**	
		A	**P**	**A**	**P**	**A**	**P**	**A**	**P**	**A**	**P**	**A**	**P**
avenhaus_cascade	A	1	1.26	1.08	1.34	1	1.39	1.13	1.47	1	1.50	1.08	1.54
	P	1	0.66	0.87	0.58	1	0.54	0.85	0.47	1	0.55	0.85	0.47
lat	A	1	1.36	0.91	1.34	1	1.27	1	1.39	1	1.07	0.96	1
	P	1	0.55	0.99	0.54	1	0.30	0.93	0.30	1	0.52	0.83	0.39
dct	A	1	1.33	0.83	1.0	1	1.18	0.84	0.91	1	1.04	0.74	0.96
	P	1	0.62	0.90	0.55	1	0.50	0.92	0.44	1	0.58	0.92	0.50
iir	A	1	1.21	1.07	1.28	1	1.13	1.01	1.23	1	1.13	0.99	1.23
	P	1	0.61	0.93	0.61	1	0.46	0.86	0.39	1	0.42	0.81	0.36
hier_paulin	A	1	1.20	1.05	1.32	1	1.23	1.13	1.34	1	1.17	1.16	1.33
	P	1	0.59	0.97	0.55	1	0.57	0.97	0.50	1	0.43	0.87	0.37
test1	A	1	1.34	1.23	1.85	1	1.37	1.38	1.70	1	1.52	1.42	1.65
	P	1	0.54	0.68	0.44	1	0.39	0.68	0.30	1	0.37	0.75	0.29

are averaged over all examples for the corresponding laxity factor. For example, if the entry in column *Hi*, major column *Area ratio* is x, then the average area of power-optimized circuits, synthesized from hierarchical behavioral descriptions is x times the average area of area-optimized circuits, synthesized from flattened behavioral descriptions, for that laxity factor. Under the major column *Power ratio*, comparisons of power-optimized architectures are made against area-optimized architectures at $5V$ (labeled $5V$) and those that have been V_{dd}-scaled to just meet the sampling period constraint (labeled V_{dd}-sc). Major column *Synth. time* gives the synthesis time, in seconds, for area- and power-optimized circuits, averaged over all examples for a given laxity factor. The experiments were conducted on an SGI Challenge workstation with 256 MB RAM.

The results obtained indicate that our hierarchical synthesis algorithm produces circuits whose area and power consumption are comparable to those of circuits produced from flattened descriptions. This is significant because most previous hierarchical high-level synthesis systems synthesize circuits which are significantly less compact than than their flattened equivalents [3]. The synthesis time, which is measured as the time to compile the behavioral description into an RTL implementation, is significantly less for hierarchical circuits.

For application-specific integrated circuits, which typically have a short design cycle (order of weeks), reduction in synthesis time can be extremely significant. For larger hierarchical behavioral descriptions, we expect the ratio of synthesis times for flattened and hierarchical synthesis to be even greater. Note that flattened synthesis can also take advantage of module selection, scheduling, resource sharing, clock and V_{dd} selection techniques available for hierarchical synthesis. The efficacy of our power optimization techniques is illustrated by the improvement in power obtained at a relatively low area overhead. Under hierarchical synthesis, our method results in upto 6.7-fold reduction in power at area overheads not exceeding 50% over area-optimized circuits working at 5 $Volts$. On an average, hierarchical power-optimized designs consumed 13.3% less power than flattened designs optimized for power, and hierarchical area-optimized designs had an area overhead of 5.6% over flattened, area-optimized designs.

6 Conclusions

In this paper, we presented a methodology for the synthesis of power- and area-optimized circuits from hierarchical behavioral descriptions. Our algorithm can uniformly handle arbitrarily deep hierarchies, and aims at using the high-level knowledge available at higher hierarchical levels to drive fine-grain optimization techniques at lower levels. Our approach allows for the use of libraries of specialized RTL modules, such as filters and FFT's, in an efficient manner. Furthermore, it allows for customization of these libraries if needed. Unlike previous hierarchical synthesis methods, our algorithm allows multiple DFGs to be executed on the same RTL module. This is made possible by the technique of RTL embedding which we have developed for the purpose. We implemented our ideas in a program named *H-SYN* written in C++. Experimental results performed on commonly available benchmarks, including some from the industry, demonstrated the efficacy of our techniques in synthesizing power-optimized circuits with relatively low area overheads within short run times.

Table 4: Summary of area (ratio), power(ratio) , and synthesis time (seconds) results

L.F	**Area ratio**		**Power ratio**				**Synth. time (sec.)**	
			5V		V_{dd}**-sc**			
	Fl	**Hi**	**Fl**	**Hi**	**Fl**	**Hi**	**Fl**	**Hi**
1.2	1.28	1.36	.51	.47	.60	.55	844	261
2.2	1.26	1.34	.31	.27	.46	.40	854	322
3.2	1.24	1.29	.19	.17	.48	.40	1029	357

References

[1] M. Potkonjak and J. Rabaey, "A scheduling and resource allocation algorithm for hierarchical signal flow graphs," in *Proc. Design Automation Conf.*, pp. 7–12, June 1989.

[2] D. S. Rao and F. J. Kurdahi, "Controller and datapath tradeoffs in hierarchical RT-level synthesis," in *Proc. Int. Symp. High Level Synthesis*, pp. 152–157, May 1994.

[3] D. S. Rao and F. J. Kurdahi, "Hierarchical design space exploration for a class of digital systems," *IEEE Trans. Computer-Aided Design*, vol. 1, pp. 282–294, Sept. 1993.

[4] A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R. W. Brodersen, "Optimizing power using transformations," *IEEE Trans. Computer-Aided Design*, vol. 14, pp. 12–51, Jan. 1995.

[5] A. Raghunathan and N. K. Jha, "An ILP formulation for low power based on minimizing switched capacitance in datapath allocation," in *Proc. Int. Symp. Circuits & Systems*, pp. 1069–1073, May 1995.

[6] A. Dasgupta and R. Karri, "Simultaneous scheduling and binding for power minimization during microarchitecture synthesis," in *Proc. Int. Symp. Low Power Design*, pp. 69–74, Apr. 1995.

[7] L. Goodby, A. Orailoglu, and P. M. Chau, "Microarchitectural synthesis of performance constrained, low power VLSI designs," in *Proc. Int. Conf. Computer Design*, pp. 323–326, Oct. 1994.

[8] G. Lakshminarayana and N. K. Jha, "Hierarchical synthesis for low power," Tech. Rep., CE-J98-003, Princeton University, 1998.

[9] A. Raghunathan and N. K. Jha, "Behavioral synthesis for low power," in *Proc. Int. Conf. Computer Design*, pp. 255–270, Oct. 1994.

[10] A. Raghunathan and N. K. Jha, "An iterative improvement algorithm for low power datapath synthesis," in *Proc. Int. Conf. Computer-Aided Design*, pp. 597–602, Nov. 1995.

[11] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, Englewood Cliffs, NJ, 1982.

[12] D. D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Boston, 1992.