Forward-Looking Objective Functions: Concept & Applications In High Level Synthesis

Jennifer L. Wong
Univ. of California, Los Angeles
3563F Boelter Hall
Los Angeles, CA 90095
(310) 825-1872
jwong@cs.ucla.edu

Seapahn Megerian
Univ. of California, Los Angeles
3750 Boelter Hall
Los Angeles, CA 90095
(310) 206-3864
seapahn@cs.ucla.edu

Miodrag Potkonjak
Univ. of California, Los Angeles
3532G Boelter Hall
Los Angeles, CA 90095
(310) 825-0790
miodrag@cs.ucla.edu

ABSTRACT
The effectiveness of traditional CAD optimization algorithms is proportional to the accuracy of the targeted objective functions. However, behavioral synthesis tools are not used in isolation; they form a strongly connected design flow where each tool optimizes its own objective function without considering the consequences on the optimization goals of the subsequently applied tools. Therefore, efforts to optimize one aspect of a design often have unforeseen negative impacts on other phases of the design process.

Our objective is to establish a systematic way of developing and validating new types of objective functions that consider the effects on subsequently applied synthesis steps. We demonstrate the generic forward-looking objective function (FLOF) strategy on three main steps in behavioral synthesis: (i) Transformation, (ii) Scheduling, and (iii) Register Assignment. We show how the FLOF can be used in the first two phases to reduce the total number of registers required in the third phase.

Categories and Subject Descriptors
B.6.3 [**Logic Design**]: Design Aids – Optimization.

General Terms
Algorithms, Design.

Keywords
Objective Functions, Behavioral Synthesis, Transformations, Scheduling, Register Assignment

1. INTRODUCTION
The effectiveness of each step in behavioral synthesis is dependent on two main components: (i) algorithmic optimization mechanisms and (ii) targeted objective functions. Tremendous efforts have been dedicated to the development of increasingly sophisticated optimization mechanisms, resulting in intricate and effective schemes that leverage on complex mathematical results. However, the effectiveness of these algorithms is limited by the accuracy of the targeted objective functions. Interestingly, while essentially all CAD tools use some type of objective functions, there has been very little effort to develop systematic theory and practice of designing and evaluating objective functions. Often, the objective functions used are too simplistic. Examples are the use of the number of literals to predict area in logic synthesis and the number of execution units to predict area in behavioral synthesis. Increasing complexity of deep-submicron is bound to make these objective functions more irrelevant. Also, the need for predicting more complex design metrics, such as power and testability, provide additional impetus to study objective functions during synthesis.

One of the key difficulties in developing high quality objective functions is that synthesis tools are mainly applied following the waterfall model and that optimization goals do not take into account the consequences of current choices on optimization in later steps. For example, often slight minimization in the number of execution units has as direct consequence an additional overhead during the variable to register assignment phase. A small saving, if any, during scheduling is compounded with significant additional area due to registers. Note that registers are relatively expensive in comparison with the majority of execution units in terms of both area and power.

Our goal is to develop a systematic approach for developing objective functions that span more than one design step. We want to demonstrate the effectiveness of this approach by utilizing the same optimization mechanisms with both traditional and the new forward-looking objective function in three behavioral synthesis steps: transformations, scheduling, and register assignment.

1.1 Motivational Example
In order to illustrate how an objective function can capture the impacts of optimization choices on future steps, we use a simple example from behavioral synthesis. Figure 1(a) shows the schedule of a computation involving four variables: A, B, C, and D. The results of the computation are the variables X and Y. The circles represent an arbitrary operation and the directed line segments represent the variables and precedence relationships in the computation. If we assume that each operation takes one clock cycle and that the available time is four clock cycles, the solution produced by the ALAP algorithm uses 2 execution units and is optimal. The subsequent application of register assignment, results in 6 registers. The numbers on the right side of each scheduling instance in Figure 1 indicates the size of the variable cut in each control step, i.e. the number of variables that are alive. Since simultaneously alive variables must be stored in separate registers, the maximum variable cut (shown in bold) is a lower bound on the number of required registers.

However, if we considered the impact of scheduling on the lifetimes of the variables, we will obtain the schedule shown in

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

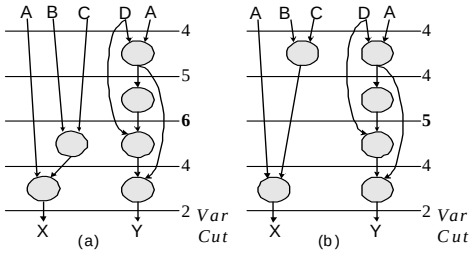

Figure 1. Simple scheduling example.

Figure 1b. For example, if the objective function used by scheduling has as component proportional to the sum of lifetimes of variable, we will schedule the operation with B and C inputs in the first clock cycle so that we have only one variable alive after that instead of two as in the ALAP schedule. If we follow this objective function, the resulting schedule (Figure 1b) will also require two execution units. However, the sum of the lifetimes of the variables will be reduced and more importantly the maximal cut and eventually the required number of register will be reduced. The key observation is that although the register assignment is done later using a completely different algorithm (graph coloring), the effectiveness of the step is directly impacted by targeted objective function during scheduling. Therefore, if one wants to optimize across boundaries of individual synthesis tasks, he has to use objective functions that captures the requirements of consecutive synthesis steps.

2. RELATED WORK

Objective functions and their use in optimization has been the focus of intense study for decades [6]. Scheduling is one of the mandatory tasks that must be performed in behavioral synthesis. A number of Scheduling algorithms have been proposed and include [1,5]. Register assignment, typically performed following Scheduling, has also attracted a great deal of attention. Due to its importance, a variety of approaches have also been proposed for Register Assignment [11]. Finally, transformations are considered by many as the behavioral synthesis task with the highest potential to optimize a design. In the last two decades, a number of approaches have been proposed [8,9]. A more comprehensive analysis of high level synthesis, Scheduling, Register Assignment, and Transformations can be found in [3].

3. TECHNICAL PRELIMINARIES

3.1 Global Flow: High-Level Synthesis

Traditionally, the global flow of high-level synthesis has consisted of two major steps: (i) Scheduling and (ii) Resource Allocation. Scheduling refers to the assignment of operations to control steps, while resource allocation refers to the mapping of objects to physical units. One of the main tasks performed in the resource allocation step is *register assignment* which refers to the assignment of variables to physical registers. In order to make the technical discussions in this paper self-contained, we briefly discuss Scheduling, Register Assignment, and Transformations.

3.2 Scheduling

The main goal of Scheduling is to create a plan for the execution of operations subject to a specified set of constraints. The constraints of this optimization process are two fold: (i) the inherent execution order of the operation imposed by the computation, and (ii) physical hardware limits. Having a limited number of adders, multipliers, and other physical functional units that the high-level operations must ultimately be mapped to, restricts the number of operations that can be scheduled in the same control step. Typically, the goal of Scheduling is to schedule a design using a given set of functional units, while reducing the number of control steps required to perform the computation.

The computational model behind our discussion is the synchronous data flow (SDF) model [7]. For the sake of simplicity, we focus our attention on the case where each node consumes two and produces one sample on every execution. The SDF model is well suited for specification of computations in numerous application domains such as DSP, communications, and multimedia. The syntax of a targeted computation is defined as a hierarchical control-data flow graph (CDFG) [10]. A CDFG represents the computation as a flow graph, with nodes, data edges, and control edges. Operation Scheduling is thus the process of partitioning the set of operations in a CDFG into groups such that the operations in the same group are executed concurrently in the same control step while all precedence constraints are satisfied. The three main types of scheduling in high-level synthesis are [5]:

1. *Time-Constrained*: scheduling a CDFG in a given number of control steps while minimizing the required number of functional units.
2. *Resource-Constrained*: scheduling a CDFG using a given set of functional units while minimizing the required number of control steps.
3. *Hybrid*: attempting to schedule a CDFG for a given maximum number of control steps and the number of available functional units.

Unless otherwise specified, we assume that the number of functional units of each type and the maximum allowed number of control steps are specified a-priori (i.e. Hybrid). The simplified problem where only one type of operation is used, which is computationally intractable [4], can formally be stated as:

Problem: ***Scheduling***
Instance : *A CDFG C, a time constraint T, parameter N.*
Question: *Does there exist a Scheduling of C into T control steps such that the number of operations from C in each control step is less than N and all precedence constraints in C are satisfied?*

Two well-known heuristics for Scheduling are the as-soon-as-possible (ASAP) and the as-late-as-possible (ALAP) techniques. In Section 4, we use these heuristics in building the Scheduling FLOF. These heuristics are very efficient in determining the range of valid control steps that each operation can be scheduled in, and approximately the lifetimes of variables. According to the ASAP scheduling heuristic, operations are scheduled in the earliest possible control steps such that no constraints are violated. This is achieved by a simple traversal of the computation CDFG in breadth-first order. Analogously, following the ALAP heuristic, operations are scheduled in the latest possible control steps such that no constraints are violated. Once the process is complete, the schedule is simply “translated” where the first operations in the scheduled CDFG are in the first control step. A non-deterministic choice exists in both ASAP and ALAP scheduling when the number of operations allowed in each control step is fixed (i.e. limited hardware). The heuristics don’t specify the order that independent operations should be scheduled. Different implementations address this issue in varying ways.

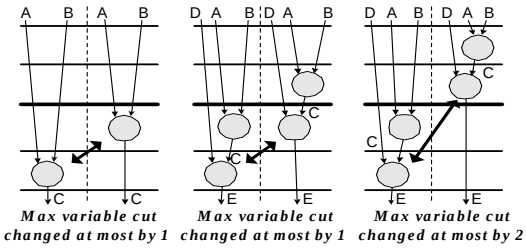

Figure 2. Basic blocks of scheduling and variable-cut trade-offs.

In our subsequent discussions, when we refer to a *cut*, we are referring to *variable cuts.* A ***variable cut*** refers to the variables that are alive in a scheduled CDFG between control steps. We assume that a variable is alive at the *end* of the control step where it's created until the *beginning* of the control step where it's consumed. Similarly, an ***operation cut*** in a scheduled CDFG refers to the operations scheduled in a particular control step. For example, the maximum operation cut refers to the cut in the control step with the largest number of operations of a particular type. Note that the maximum operation cut indicates the minimum number of required functional units (of each type).

3.3 Register Assignment

Register assignment is the process of mapping variables in a scheduled CDFG to physical storage units (registers). Clearly when two variables are simultaneously alive in a scheduled CDFG, they cannot be mapped to the same register. The register assignment problem can easily be modeled as a Graph Coloring problem where nodes in the graph correspond to variables in the CDFG and an edge connects any two nodes (variables) that are alive at the same time. Each color in a graph coloring solution corresponds to a unique register. The NP-complete Graph Coloring decision problem [4] can formally be states as:

Problem: ***Graph Coloring***
Instance : *A graph G(V,E) with vertex set V and edge set E, number of colors k.*
Question: *Does there exist a coloring of the graph G that assigns colors to each vertex $v \in V$ such that for any edge $e \in E$ the vertices connected by e have different colors, and that no more than k unique colors are used?*

Although graph coloring is a difficult problem, the *maximum variable cut* in a scheduled CDFG can provide a lower bound limit on the number of required colors. This is because all variables in a *cut* are alive by definition and thus must be mapped to separate registers and therefore separate colors. We use this lower bound on the required number of registers to build our Scheduling FLOF.

3.4 Transformations

Transformations can serve as powerful tools in optimizing a design for Scheduling. Some examples of transformations are associativity, distributivity, and retiming.

Retiming is a special transformation action applied to looping CDFGs (repeating) and can be accomplished by doing either of the following: (i) remove one delay element from each input of an operation and add a delay at each of its outputs; (ii) remove one delay element from each output of an operation and introduce a new delay element at each of its inputs.

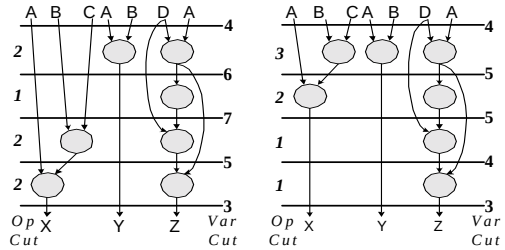

Figure 3. Example of how adding functional units can reduce the variable cut (& potentially the number of registers).

A delay element is a special type of operation that "delays" the variable at its input until the next iteration of the scheduled CDFG. We discuss examples of retiming and its impact on register minimization in Section 4.2.

4. FORWARD-LOOKING OBJECTIVE FUNCTIONS

The term *objective function* refers to a function that quantifies the quality of a solution (or a partial solution) in an optimization problem. Objective functions essentially form the backbone of any optimization algorithm and serve as the guide in obtaining desired solutions. In a sorting problem for example, the objective function $OF_{Sort}(\ )$ can be defined as the percentage of items that are correctly sorted in a solution instance. Thus, the objective of a sorting algorithm would be to find a solution such that the objective function $OF_{Sort}(\ )$ is maximized.

The success of an optimization algorithm often depends heavily on carefully designed and tuned objective functions. However, modern layered optimization flows, that are necessary to approach and solve complex problems, often hinder or severely limit the usefulness of objective functions. Layering complex problems into logical steps serves as a tool to abstract away details that can make a problem very difficult to approach otherwise. Although each individual level may work well separately, final results may not be as good as possible due to the interaction between the levels. This is mostly due to the fact that the effects of actions in one layer will impact subsequent layers. For example, in system-on-chip design, optimizing the high-level communication among different modules may have adverse effects that will not be apparent until the floorplanning or place-&-route stages when new constraints (such as physical placement and mapping) take effect.

Here, we propose *forward-looking objective functions* (FLOF) as a generic means to overcome the restrictions imposed by layered optimization flows. To build a successful FLOF, one must carefully analyze the problem specific components in each step that may impact future steps, formulate quantitative expressions, and establish a method for combining the components into the final objective function format. Hence, the outline for designing an FLOF is:

1) Identify problem specific components and their effect on future steps (interaction between layers);
2) Quantify the components;
3) Combine components into a final objective function with appropriate normalization.

We demonstrate this process in detail in the following subsections to design a Scheduling FLOF with the goal of minimizing the required number of registers subject to the

traditional scheduling constraints (total number of available functional units and control steps).

4.1 Scheduling

As stated above, the key factor in formulating an objective function in one level that will be successful in achieving good quality results in the overall flow is by the inclusion of properties and insights that can predict the impact of decisions from one optimization level onto future optimization levels. In order to make this clear, here we detail the steps required to build a FLOF for our example problem, namely Scheduling and its impact on Register Assignment in high-level synthesis.

In order to build the Scheduling FLOF, we must first itemize the ways in which specific schedules will potentially impact the register assignment process. As discussed in Section 3, register assignment is a hard optimization problem so we do not expect to be able to predict exact outcomes. Our goal here is to combine careful analysis and problem specific insights to formulate our objective function such that we favor scheduling solutions that are promising in terms of their expected required number of registers.

Consider the three scheduling instances in Figure 2. The shaded circles represent an arbitrary operation that consumes two inputs and produces one output. Suppose that the bold horizontal line is where the maximum variable cut lies. Let us assume that the location of the maximum cut does not change as we explore the different options (i.e. we assume that the maximum cut does not move). We use the term "move" to signify a scheduling option. The term "move" can mean re-scheduling as in iterative improvement-type algorithms or simply the decision process of where to schedule an operation in a constructive algorithm.

The first instance (left) in Figure 2 illustrates the trivial observation that moving a single operation can only impact the maximum cut by at most one variable. The middle and rightmost cases shows that a group of two operations must "move" by at least two control steps in order to increase the effect on the max cut to two variables. The main observations here are two fold: (i) since operations consume more variables than they produce, the variable cut can be reduced by scheduling more operations *before* the cut and (ii) the size of the maximum variable cut can be reduced by the scheduling such that the variable lifetimes are decreased. Note that in Figure 2, moving operations "up" effectively reduces the lifetime of the input variables while potentially increasing the lifetime of the output variables. Since there are more input variables than output variables for each operation, the total lifetime of variables is reduced. In the following subsection we analyze how variable lifetimes can play an important role in determining the required number of registers for scheduling.

Throughout our discussions we assume that the number of available functional units in each control step is fixed. Let us briefly consider how the addition of new functional units can serve to reduce the maximum variable cut using an example. The two diagrams in Figure 3 show the same scheduled CDFG. The schedule on the left requires only 2 functional units while the schedule on the right requires 3. However, the maximum variable cut in the left is 7 compared to 5 on the right. The main observation here is that variables A, B, and C live for relatively long periods of time. The scheduling flexibility for the consumption of these variables can be leveraged to greatly reduce their lifetimes. Although in the process, one extra functional unit is used, the variable cut is reduced by 2. This alone, may potentially translate to a savings of 2 registers in the register assignment step.

As the example in Figure 3 suggests, in certain instances, there is clearly a trade-off in the maximum variable cut and the maximum operation cut. For larger cases (most real designs) the savings in register requirements can be substantial relative to the extra overhead required for the additional functional units. Furthermore, since registers are much more costly in terms of power and area than adders for example, the benefits of trading functional units for register savings can be substantial. However, as stated earlier, for the purpose of brevity and simplicity, we assume the number of functional units is fixed when developing the Scheduling FLOF with register minimization as the goal.

We have identified two main attributes of variables in scheduling that can ultimately impact the minimum number of required registers: variable *lifetimes, and* scheduling *difficulty*. Generally, variables that are alive for shorter time periods are better for minimizing the number of registers since the likelihood that they will overlap with other variables is reduced. We refer to this property as the "*Small-Potato Principle*". It's a well known fact in the agricultural community that relatively smaller potatoes can be packed more densely in a given volume than larger ones. Analogously, it's not unreasonable to expect that shorter lived variables, *in general*, will be easier to schedule and "pack" into fewer registers than variables that live for relatively long periods, potentially overlapping with many other variables. Throughout the rest of this discussion, we will focus on properties and aspects of the scheduling problem that can be leveraged to build a FLOF to minimize the number of required registers.

Due to the nature of the register assignment problem, in addition to the lifetime of a specific variable, we must also consider the interplay among all variables. Although the lifetime of a variable x may be short, it could potentially have adverse effects on the required number of registers if many other variables must also be alive in the same time frame as x. In order to address this, we propose a *weighted lifetime* to characterize individual variable lifetimes. The goal here is to introduce a scaling factor to model the observation that variables that are alive while many others are also alive are more likely to adversely impact the ultimate number of required registers than otherwise.

However, variable lifetimes are not known prior to actually having a schedule in place. Therefore, the objective function we propose here relies on the aforementioned ASAP and ALAP scheduling heuristics to compute the initial approximate variable lifetimes. For a given variable x we define its lifetime:

$$L(x) = \mathrm{ALAP}(x) - \mathrm{ASAP}(x)$$

where ASAP( ) indicates the earliest and ALAP( ) indicates the latest control step at which a variable can be alive. We also define the set Vc(x) for a variable x:

$$V_C(x) = \{V_1, V_2, \ldots, V_T\}$$

where V_i is the cardinality of the variable-cut in control step i, if variable x is alive in control step i and 0 otherwise. Note that here cut cardinalities must again be approximated using the ASAP/ALAP scheduling heuristics.

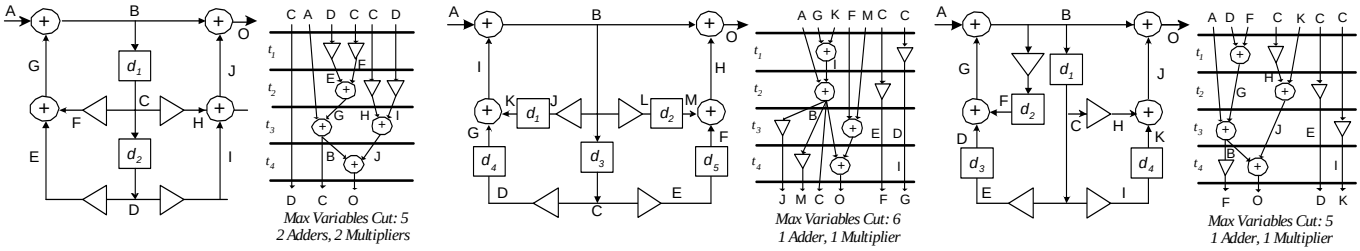

Figure 4. Figure 4. Retiming Transformation and its effect on the maximum variable cut after scheduling a simple bi-quad.

As mentioned above, the cardinality of variable cuts play an important role in determining how many registers are required for a given scheduled computation. In general, if a variable belongs to several large cuts, then it's more likely to remain in large cuts. There are also many other factors such as complex variable dependencies that one can consider. However, ultimately, the size of the largest variable cut will determine the minimum number of required registers for a given schedule. We thus formulate the *weighted lifetime* for the variable *x*:

$$WL(x) = \mathbf{max}(\ V_C(x)\ ) \cdot L(x)$$

In the definitions above, we compute the longest possible variable lifetimes and worst-case cut cardinalities using the ASAP/ALAP heuristics. Another alternative is to use the mid-point of ASAP/ALAP ranges for each operation as the scheduling heuristic and compute lifetimes and variable cut-cardinalities. As each operation is scheduled during the scheduling algorithm, we replace the heuristic based approximates in our above formulations with the appropriate data (variable lifetimes and variable cut cardinalities) and re-compute the *WL* for each variable.

To quantify the scheduling *difficulty* of a variable, we first note that each variable has one producer and at least one consumer. The only exceptions are input variables that have no producers and output variables that have no consumers. Generally, the higher the number of the consumers a variable has, the more difficult it's to manipulate the schedule in order to reduce its lifetime.

Suppose the set $C(x)$ denotes the set of operations that consume variable x. We define the set C_T(x):

$$C_T\,(x) = \{C_{T1}, C_{T2}, \ldots, C_{T|C(x)|}\}$$

where C_{Ti} is the control step where the *i*-th consumer of variable *x* is scheduled. If the operation is not scheduled yet, then we use the ALAP heuristic here. We represent the scheduling (or re-scheduling) *difficulty* $D(x)$ of a variable *x* as:

$$D(x) = \sum_{i=i}^{|C_T(x)|} \left(C_{Ti}(x) - \min\left(C_T(x)\right)\right)$$

The *difficulty* $D(x)$, as defined, increases with the number of consumers of *x* as well as relative difference between each consumption time and the earliest consumption time of variable *x*. Now that we have a quantitative expression for each of the identified components of scheduling for register minimization FLOF, we are faced with the difficult task of combining. Combining refers to the process of putting together each component to form the objective function. Since components often represent varying and unrelated aspects of a problem, proper normalization is a critical and often difficult task to tackle.

Since we are interested in reducing *lifetime* (*Small-Potato Principle*) and *difficulty* of all variables while satisfying the given hardware limitations we specify the combined lifetime *cWL* and combined difficulty *cD*:

$$cWL = \sum_{i=1}^{n} WL(x_i)\,,\ cD = \sum_{i=1}^{n} D(x_i)$$

where x_i represents the *i*-th variable in the given computation CDFG, and *n* is the total number of variables. The combination scheme for *cWL* and *cD* are not restricted to the ones specified above. Many other reasonable alternatives exist that can perform just as well, such as median, root-mean-square (L_2 *norm*), and simply taking the largest variable weighted lifetime (L_∞ *norm*).

A popular method that has proven very effective in forming optimization objective functions is the linear combination of components. Following this well established and proven school of thought we formulate the ***Scheduling For Register Minimization FLOF$_{S\text{-}RM}$*** as: $FLOF_{S\text{-}RM} = \alpha \cdot cWL + \beta \cdot cD$ where α and β are experimentally determined and tuned parameters. We set α equal to the largest weighted lifetime $WL(x_i)$, and β equal to the largest difficulty $D(x_j)$ as computed by the above heuristics. This choice seemed to perform best among other selections.

4.2 Transformations – Register Assignment

We now consider how the insights learned in the Scheduling domain and the corresponding FLOF can be applied to the Transformations domain. Consider the three retimed "bi-quads" in Figure 4. The "bi-quad" computation structure is widely used as a basic building block for designing infinite-impulse-response (IIR) filters. A circle represents an addition operation, a triangle represents a constant multiplication, and a square represents a delay.

Although all three cases in Figure 4 perform the same computation, the use of retiming as the transformation has resulted in three different schedules as shown to the right of each figure. The number of required adders and multipliers as well as the maximum variable cut size are also listed. Although during the Transformations phase of the design process a schedule does not exist, the scheduling FLOF developed in the previous subsection can still guide the optimization process in making choices such that the required number of registers in the future may be minimized. The use of the FLOF above is possible since initially we only rely on the ASAP/ALAP scheduling heuristics that can be computed very efficiently. In the cases presented, the scheduling FLOF can indicate the solution that is the most promising as a candidate for minimizing the required number of registers. The same insights from the scheduling step can be applied to this step using the approximate schedules.

5. Experimental Results

We used a variety of CDFG structures adopted from [2] to evaluate the performance of our proposed scheduling FLOF. The selected

set of designs includes the elementary sine function, a linear controller (LinearCntrl13), two eighth-order Avenhaus IIR filters with different structures (cascade8, parallel8), a volterra filter, a differentiator, several transforms (Hilbert, Wavelet, Winogradfft11), a seventh-order IIR filter, and three FIR filters (Dsfir51, Dskais55, fir00). We use 0.25 micron technology.

To obtain the results, we use HYPER which is a resource utilization driven complex heuristic synthesis package [10]. For each case, we synthesized and compared the results with and without the $FLOF_{S\text{-}RM}$ as the objective function in HYPER. The experimental results are in Table 1. The first column contains the design name, followed by the number of nodes, the number of different types of operations in the graph, and the active area used without the default HYPER heuristics. The results of area and percentage improvement in area are shown in the second half of the table. The fifth column presents the active area obtained by using the scheduling $FLOF_{S\text{-}RM}$. The final column shows the percentage improvement.

On average, the approach for Scheduling and Register Assignment using the $FLOF_{S\text{-}RM}$ reduced the area requirement by 11.24% with a median of 9.05%. For the smallest design, the improvement was 5.2%, while for the largest design the improvement was approximately 20%. For Transformations (by using the corresponding FLOF in HYPER), Scheduling, and Register Assignment (results shown in the final two columns) there is an average improvement of 23.37% and a median improvement of 16.15%. Although some of the active area improvement was a result of a reduction in execution units, more that 80% of the improvement is due to register minimization.

6. CONCLUSIONS

The success of an optimization algorithm often depends heavily on carefully designed and tuned objective functions. However, modern layered optimization flows, often hinder the usefulness of objective functions since the effects of actions in one layer on subsequent optimization steps are not considered. We proposed *forward-looking objective functions* as a generic means to overcome the restrictions imposed by layered optimization flows. FLOFs are built by carefully analysis of problem specific components in each step that may impact future steps, formulating quantitative expressions, and establishing a method for combining the components into the final objective function format. We demonstrated the details involved in FLOF design using the Scheduling, Register Assignment, and Transformations examples from High Level synthesis. Experimental results indicate that through the use of the proposed Scheduling FLOF for register minimization, significant area savings can be achieved, due to the fewer required registers in the final design.

7. ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation Career award under Grant No. ANI-9734166.

8. REFERENCES

[1] R. Camposano, W. Wolf, *High Level VLSI Synthesis*, Kluwer Academic, Norwell, MA, 1991.

[2] M. R. Corazao, et. al. "Performance Optimization using Template Mapping for Datapath-Intensive High-Level Synthesis", IEEE Trans. on CAD, Vol.15, pp. 877-888, 1996.

[3] G. De Micheli, *Synthesis and optimization of Digital Circuits*. McGraw-Hill, 1994.

[4] M. R. Garey, D. S. Johnson. Computers and Intractability: A Guide To The Theory Of NP-Completeness. W.H. Freeman, 1979.

[5] C.-T. Hwang, J.-H. Lee, Y.-C. Hsu. "A formal approach to the scheduling problem in high level synthesis." IEEE Trans. on CAD, Vol.10, (no.4), pp. 464–475, 1991.

[6] P. K. Jha, et. al. "An empirical study on the effects of physical design in high-level synthesis." Procs of the Seventh International Conference on VLSI Design, p. 11-16, 1994.

[7] E.A. Lee, D.G. Messerschmitt. "Synchronous Dataflow." Procs. of the IEEE, Vol.75, (no.9), pp. 1235-45, 1987.

[8] A. Orailogulu, D. D. Gajski. "Flow graph representation." ACM/IEEE Design Automation Conference (DAC), pp. 503–509, 1986.

[9] M. Potkonjak., J. Rabaey. "Optimizing resource utilization using transformation." IEEE Trans. on CAD, Vol.13, (no.3), pp. 277–292, 1994.

[10] J. Rabaey, et. al. "Fast Prototyping Of Data Path Intensive Architectures." IEEE Design and Test of Computers, Vol.8, pp.40-51, 1991.

[11] C. J. Tseng, D. P. Sieworek. "Automatic synthesis of data path on digital systems." IEEE Trans. on CAD, Vol.5, (no.3), pp. 379–395, 1986.

Table 1. Experimental results: Area savings obtained for Scheduling and Transformations (for minimizing number of registers).

Design Name	# Nodes	# Op. Types	Active Area (mm^2)	Activ Area S/R	% Area Improved S/R	Active Area T/S/R	% Area Improved T/S/R
Volterra (mult)	30	4	2.13	2.02	5.2	1.92	9.9
iit7	33	5	5.46	5.11	6.4	4.88	10.7
cascade8	34	4	2.41	2.25	6.7	2.14	11.0
parallel8	39	4	2.85	2.65	7.1	2.53	11.4
sine	49	6	1.45	1.32	8.9	1.25	13.6
Hilbert	49	5	3.81	3.48	8.6	3.32	12.9
Wavelet	53	5	4.33	3.93	9.2	3.66	15.4
LinearCntrl3	56	3	13.03	11.90	8.7	10.83	16.9
Differentiator	80	5	4.56	3.99	12.5	3.61	20.8
Winogradfft11	104	4	8.85	7.57	14.5	5.72	35.4
Dsfir51	127	4	10.88	9.12	16.2	6.73	38.1
fir1333	130	3	11.96	10.08	15.7	7.15	40.2
Dskais55	137	4	13.12	10.82	17.5	7.52	42.7
fir100	302	3	12.59	10.06	20.1	6.52	48.2