Exploiting Area/Delay Tradeoffs in High-level Synthesis

Alex Kondratyev, Luciano Lavagno, Mike Meyer, Yosinori Watanabe

Cadence Design Systems
San Jose, CA, USA

***Abstract* – This paper proposes an enhanced scheduling approach for high-level synthesis, which relies on a *multi-cycle behavioral timing analysis* step that is performed *before and during scheduling*. The goal of this analysis is to accurately evaluate the criticality of operations and determine the most suitable candidate resources to implement them. The efficiency of the approach is confirmed by testing it on industrial examples, where it achieves, on average, 9% area savings after logic synthesis.**

I. INTRODUCTION

Historically, the task of high-level synthesis is divided into resource allocation, scheduling and resource binding. Allocation determines which resources will be used, then scheduling answers the question of when (at which state) every operation of the specification is executed, while binding specifies a particular resource (from the allocated set) to implement every operation.

The approaches known in the literature either solve these problems sequentially or take a naïve formulation of the combined problem, which is too expensive to solve for practical designs. In [1] it was argued that these problems must be solved together, in order to obtain a high-quality implementation, which (1) is competitive with manual design and (2) is guaranteed to be implementable within the given timing constraints. The difficulty is due to the tight relationship between scheduling and binding, because the choice of a resource to implement an operation and its schedule are mutually dependent. For example, if some addition operation is on the critical path, then the scheduler may either choose the fastest implementation (e.g., using carry lookahead) or schedule it in a later clock cycle using a slower but cheaper implementation (e.g., using a ripple-carry adder).

To break this mutual dependency problem in large industrial scale designs, it is not sufficient to simply formulate a combined scheduling and binding problem, e.g., using Integer Linear Programming, because the solution space is too large to find good solutions. Thus, it is necessary to restrict the solution space without sacrificing the quality of results. To address this issue, we present an approach that first performs a *detailed timing analysis spanning multiple clock cycles*, considering the actual pin-to-pin gate-level delays for the resources. This computes the true *sequential slack* of each operation *within a behavioral pre-schedule Data Flow Graph* (DFG) [2]. The goal of this timing analysis is to evaluate the criticality of DFG operations by using their sequential slack. This information is then exploited by a subsequent *heuristic* joint scheduling and binding step to choose the best state and resource for a given operation. Contrary to the traditional setting of static timing analysis used in logic synthesis, operations can be scheduled not just within *one state* but within a *set of states*. The problem is therefore formulated as finding the sequential slack on a DFG whose vertices are operations and whose edges are dependencies between operations. We implemented this approach in our commercial high-level synthesis tool, and here we discuss its experimental effectiveness.

II. MOTIVATION

A. Resource variations

The high-level synthesis task is typically done in three steps [2]:

1. Allocation chooses the type and number of resources to use;
2. Scheduling defines the control steps (states) at which every operation must be executed, and
3. Binding binds every operation to a particular resource from the multi-set chosen in step 1.

Choosing a proper set of resources during the allocation step is non-trivial for two reasons:

1. Some operations may be executed by several types of resources (for example addition can be executed by an adder or by an adder_subtractor), and
2. Operations may have different widths of operands, and decisions must be made on how to group them during allocation. For example, assume that two addition operations must be implemented: add(6,6) and add(3,8) (where numbers in brackets show the width of the operands). Then, one needs to decide whether to allocate an adder(6,8) for both of them or to allocate two different adders, adder(3,8) and adder(6,6).

Allocation becomes prohibitively complex if, in addition to these two aspects, the designer wants to consider delay/area variations of resources of the same type.

Table 1. Area and delay trade-offs for multiplier and adder

Mul 8*8bit	delay(ps)	430	470	510	540	570	610
	area	878	662	618	575	545	510
Add 16bit	delay(ps)	220	400	580	760	940	1220
	area	556	254	225	216	210	206

Table 1 shows area/delay trade-offs for resources implemented with the TSMC 90nm library. One can see that area/delay numbers for these resources vary widely: 2-3x area and 1.5-6x delay.

The problem of area/delay choices for resources is not unique to HLS. RTL synthesis usually implements a solution that starts from the fastest possible timing, followed by area recovery for gates with slack, after timing has been met. A similar approach was previously assumed to work equally well for HLS. In the next example, however, we will disprove this claim and show that in HLS the separation of timing convergence and area recovery may lead to highly non-optimal solutions.

B. Scheduling example

Consider the SystemC specification shown in Figure 1. Assume that the desired throughput for this example is 3 clock cycles to compute

978-3-9810801-8-6/DATE12/©2012 EDAA

an interpolation point (outer while loop iteration). To fit 4 iterations of the loop in 3 clock cycles, one must unroll the loop. This results in the following DFG (see Figure 2(a)), which requires the scheduling of 7 multiplications and 4 additions in 3 states, which requires at least 3 multipliers and 2 adders.

```
void interpolation::thread() {
    while (true) {
       …
       for (int i = 0; i < 3; i++) {
           x *= deltaX;
           deltaX *= scale;
           sum += x;
       }
       wait();
       fx.write(sum);
}  }
```

Figure 1. Example of SystemC specification

Assume a clock cycle of 1100ps and ignore the delays of multiplexors and registers (this simplification is done for the sake of illustration only; our actual implementation estimates them).

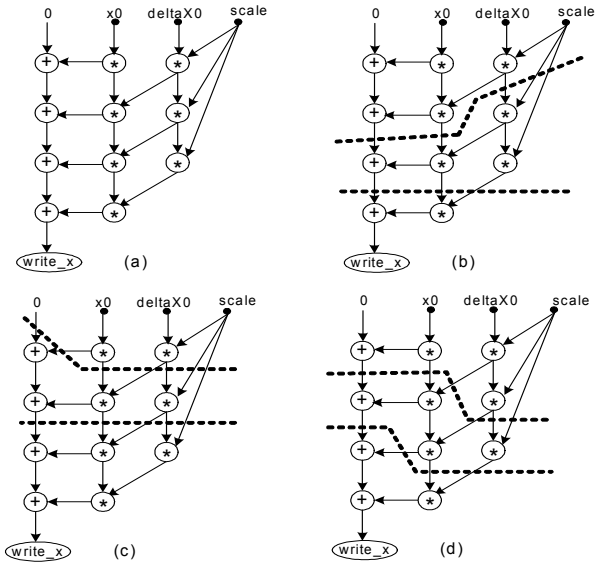

Figure 2. Different schedules for interpolation example

Case 1. Assuming the fastest resources, and using the *As Soon As Possible (ASAP)* policy, gives the schedule shown in Figure 2(b), where dotted lines show state boundaries. The critical path includes two multiplications and one addition, resulting in 2*430 + 220 = 1080ps. This satisfies the clock cycle, but gives almost no room for area recovery: all three multipliers and one adder are on critical paths. Area recovery for the second adder is possible and reduces its area to 221 units (see Table 2).

Case 2. The opposite strategy is to start scheduling by assuming the slowest resources, then reduce their delays on the fly when faced with negative slack. Its result is shown in Figure 2(c), and the critical path (see state 3) again consists of two multiplications and an addition with slightly larger area than in Case 1 (see Table 2).

The optimal scheduling solution is presented in Figure 2(d) and Table 2. It provides almost 50% area savings.

This scheduling example illustrates that:

1. Contrary to the RTL methodology, starting from the fastest resources followed by area recovery, may result in highly non-optimal implementations.
2. Starting from the slowest resources and upgrading them on the fly may also result in highly non-optimal implementations.
3. The scheduler needs guidance on the criticality of the operations to be scheduled. If the best delays for adders and multipliers (as shown in the last row of Table 1) had been known, the optimal scheduling from Figure 2(d) could have been easily found.

To address issues 1-3, we propose to perform ***multi-cycle timing analysis on the DFG to find the (heuristically) best resource for every operation before scheduling***. This provides the type of guidance to the scheduler that is missing in conventional algorithms.

Table 2. Comparison of different scheduling solutions

Impl.	Mults	Adds	Area
Case1	Del:d1=d2=d3 = 430 Area: a1=a2=a3= 877	Del: d1=221, d2=621 Area: a1=556, a2=221	3408
Case2	Del: d1=d2=d3=430 Area: a1=a2=a3=877	Del: d1=221, d2=550 Area: a1=556, a2=232	3419
Opt.	Del:d1=d2=d3=550 Area: a1=a2=a3= 572	Del: d1=d2=550 Area: a1=a2=232	2180

III. PRIOR WORK

Timing analysis is typically used in HLS to check whether an operation can be executed in the current control step or should be postponed to a later one [3, 4, 5, 6, 7]. For this purpose, the computation of the combinational slack of the operations within a given control step would suffice. Our setting is different because (a) we want to evaluate the timing mobility of each operation within its lifespan (which can cover several control steps) and (b) this analysis should be performed ***before*** scheduling the given DFG.

[8] was the first work to provide a framework for timing analysis before scheduling. It considers pairwise timing constraints (minimal and maximal) between operations of the DFG and suggests a constructive way to build a constraint graph where minimal constraints are represented by forward edges, while maximal constraints are represented by backward edges. The authors proposed a quadratic procedure for checking consistency of the constraint graph, which is too time-consuming for a timing analysis algorithm that must be repeated before scheduling every operation.

[9] proposed to translate the control flow graph (CFG) [2] and DFG into a netlist whose nodes correspond to operations. Connections between operations are mediated by special "connection timing modules" that are reconfigurable to represent both a wire and a register. The derived network of modules was used during path-based scheduling [5] to evaluate which part of a given path fits within the current control step. This model could capture moving operations across control steps, but was limited to considering only the combinational slack of operations.

Finally, in [10], a hierarchical timing model was suggested for modules used in HLS. This model captures both combinational and sequential aspects of module behaviors. It reduced timing analysis to applying the Bellman-Ford algorithm to the timing constraint graph, which is more efficient than the method in [8], but is still costly for practical applications (see experimental results). Another limitation of the approach in [10] is that it cannot model the mobility of operations within several clock cycles.

On the other hand, the idea of performing sequential timing analysis is well established in the domain of digital circuit design. [11] suggested an approach for simultaneous retiming and clock skew scheduling to improve the clock cycle of a circuit. The kernel

of this approach is a timing evaluation of criticality of gates in a circuit considering sequential constraints. The proposed method minimized the clock cycle by first applying clock skew scheduling and then retiming to the optimized schedule. The process is iterated until a fixed point. This method was enhanced in [12] by developing an efficient timing analysis algorithm that uses a retiming formulation with linear complexity for practical cases (although the worst case could be quadratic). This work also proposed a definition of sequential slack for gates, expressed in terms of sequential arrival and required times, that we will use later for operations in the DFG.

The novelty of our work is as follows:

1. We propose the notion of a timed DFG that explicitly represents the lifetimes of operations using a weighting mechanism and considering only forward edges.
2. We show that timing analysis on this DFG is reducible to the computation of sequential slack [12] of DFG operations, with a worst case linear complexity.
3. We show that area/delay tradeoffs in resource allocation for DFG operations are reducible to the sequential slack budgeting problem.
4. We propose a new scheduling framework that is tightly integrated with the timing analysis procedure.
5. All of our algorithms are suitable for arbitrary control structures, not just the acyclic DFG considered by most past work.

IV. BASIC DEFINITIONS

The main definitions are introduced by using the example in Figure 3. Following a standard compilation flow, the input specification is elaborated into a control flow graph (CFG) and data flow graph (DFG) [13]. The nodes of the CFG either serve to fork/join control flow (conditionals and loops in SystemC) or correspond to "wait()" calls in SystemC (state nodes).

```
void resizer::filter() {
...
    while (true) {
       for (int i=0; i < 1024; i++) {
           int x = a.read() + offset;
           if (x > th) {
               wait(); // s0
               y = x / scale - offset;
            } else {
                wait(); // s1
                y = x * b.read();
            }
            wait(); // s2
            out.write(y);
}    }  }
```

Figure 3. Example of SystemC specification

The DFG nodes, on the other hand, represent operations, while the DFG edges are data dependencies between them. Every DFG operation is associated with a particular edge of the CFG. Figure 4 shows the CFG and DFG for the body of the *for* loop in Figure 3 (state nodes are represented by shaded circles).

The CFG abstracts the computation and shows only the control paths and their latency.

Definition 1. [CFG] A CFG is a directed graph $G = (V, E, v0, S)$, where V is a set of nodes, and E is a set of directed edges $e=(v1, v2)$. $v0$ is the unique "start" node, while $S \subset V$ is a set of state nodes.

Definition 2. [DFG] A DFG is a directed graph $D = (O, C)$, where O is a set of vertices (operations) and C (connections) is a set of directed edges $c= (o1, o2)$, where c exists when operation $o2$ depends on results produced by $o1$.

Edges E of G are distinguished into forward and backward edges, where backward edges go from ancestors to predecessors when doing a depth-first traversal of the CFG from its origin [13].

The DFG and CFG are related through the use of two main mappings between DFG operations and CFG edges.

Definition 3. [DFG-CFG mappings]. Given CFG $G = (V, E, o, S)$ and DFG $D = (O, C)$, mapping $birth: O \rightarrow E$ defines the birthday edge for every DFG operation (which is the edge defined by the location of the operation in the source code). Mapping $sched: O \rightarrow E$ defines the scheduled edge for every DFG operation (which is the edge assigned to an operation as a result of its scheduling).

For example: for statements *x=a.read()+offset* and *y=x*b.read()* *birth(add) = e1* and *birth(mul) = e4*.

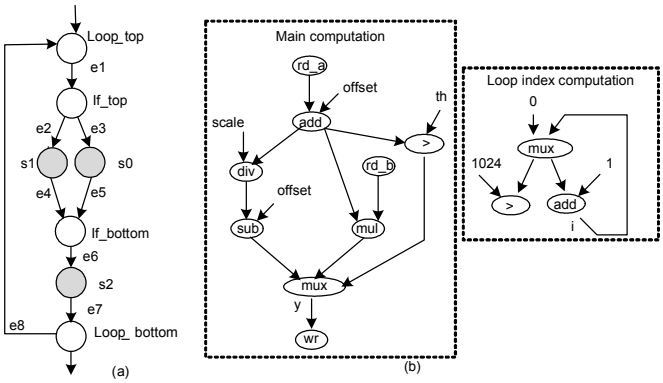

Figure 4. CFG (a) and DFG (b) for the for loop body

Some operations have no flexibility in scheduling, hence their birthday edges are the only ones where they can be legally scheduled. Examples of these operations are I/O operations (read/write), because they implement the protocol of communication between the SystemC model and its environment.

To capture the flexibility of scheduling DFG operations on CFG edges, let us introduce the notion of *operation span* (*opSpan*), which generalizes the notion of an ASAP/ALAP interval to the case of an arbitrarily complex CFG.

Definition 4. [OpSpan] The opSpan of operation o is a topologically ordered set of CFG edges $span(o) = \{e_1,...,e_k\}$ *where:*

- e_1 (called the early edge of o and denoted by *early(o)*) is the "first" edge that is forward reachable from every early edge of any direct predecessor of o, and
- e_k (called the late edge of o and denoted by *late(o)*) is the "last" edge from which every late edge of any direct successor of o is reachable.

Below are examples of opSpans for some operations in the DFG in Figure 4(b): *span(wr) = {e7}*, because *wr = out.write(y)* is a fixed operation, *span(div) = {e1,e2,e4}*.

V. TIMING ANALYSIS ON THE DFG

Definition 1. [Latency] 1. Given a pair of CFG edges *(e1,e2),* the latency between *e1* and *e2 (latency(e1,e2))* is defined as the minimum number of state nodes in all forward paths between *e1* and *e2*. Latency is undefined if *e2* is not forward reachable from *e1*.

2. Given DFG $D = (O, C)$, with two mappings *early(o)* and *late(o)* that define early and late edges for every op o, the latency of DFG

edge *(o1,o2)* is defined as *latency(early(o1),early(o2))* in the corresponding CFG.

For the CFG in Figure 4(a), the following are examples of edge latencies: *latency(e4,e6) = 0*, *latency(e1,e7) = 2* and *latency(e3,e4)* is undefined. For the DFG in Figure 4(b), the latency of *(add,div) = 0* because they have the same early edge *e1*, while the latency of *(add,mul) =1* because *mul* cannot start earlier than *e5*.

The notion of sequential slack for a node in a netlist is known [12]. A netlist is defined by nodes that are combinational gates and edges that are connections between gates. Edges have weights equal to the number of flip-flops contained by this connection. To reuse the definition of sequential slack for operations, we must convert the DFG to a representation similar to a netlist. One difficulty is that DFG operations are not fixed on particular edges, but can be scheduled anywhere inside their span. To represent this flexibility, we introduce the notion of a *timed DFG*.

Definition 2. [Timed DFG]. Given DFG $D = (O, C)$, with two mappings *early(o)* and *late(o)* that define early and late edges for every op *o,* the timed DFG $D^t = (O^t, C^t)$ is defined as a directed graph obtained from *D* by the following steps:

1. Make DFG *D* acyclic by excluding backward edges.
2. Remove constant inputs from all operations (constants do not affect timing).
3. For every operation *o* introduce a sink node *s(o)*, with an edge $o \rightarrow s(o)$, which means that *early(s(o)) = late(o).*
4. Set the weight of every edge in D^t to its latency.

Figure 5(a) shows a DFG for the "main computation" from the example in Figure 4 with opSpans of the DFG operations shown in brackets. Figure 5(b) shows the timed DFG for this piece of code.

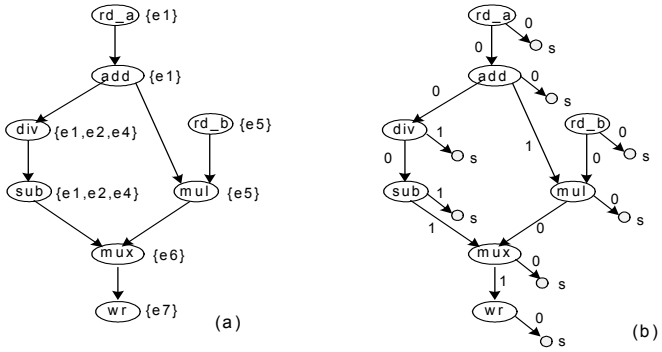

Figure 5. Construction of timed DFG

Definition 3. [Arrival/required times of DFG operation] Given a timed DFG $D = (O,C)$, clock period *T* and mapping $del: O \rightarrow R$ that for every $o \in O$ returns its delay, the arrival and required times for operations are defined as follows:

$Arr(o)=max(Arr(o_i)+del(o_i) - T * latency(o_i, o)), o_i \in Predecessors(o)$

$Arr(o) = 0$, if *o* is a source operation (i.e. $Predecessors(o) = \varnothing$)

$Req(o) = min (Req(o_i) - del(o) + T* latency(o,o_i)), o_i \in Successors(o)$

$Req(o) = T$, if *o* is a sink operation (i.e. $Successors(o) = \varnothing$)

Definition 4. [Sequential slack of DFG operation]. Given arrival and required times for operation *o,* its sequential slack is computed as $slack(o) = Req(o) - Arr(o)$.

The proposed algorithm for sequential slack computation is presented in Figure 6. Its complexity is linear in the number of connections in the DFG.

Sequential Slack computation:

1. Given DFG D(O,C) and CFG G(V,E) compute spans of operations from D
2. Construct timed DFG $D^t(O^t,E^t)$ by D
3. Denote by O_{sorted} the result of sorting O^t topologically
4. Compute operation arrival times in the order of O_{sorted}
5. Compute required time in the reverse order of O_{sorted}
6. Compute sequential slack of DFG operations

Figure 6. Algorithm for sequential slack computation

Let us illustrate the computation of sequential slack for the timed DFG, shown in Figure 5(b), under the following assumptions: the delay of I/O operations is *d*, the delay of all other operations is *D*, and the clock cycle is $T: D+d < T < 2*D$. The results are shown in Table 3, where arrival times are computed downward starting from the first row, while required times are computed upward.

Table 3. Sequential slack computation for example 2.

Op	Arr(op)	Req(op)	slack(op)
rd_a	0	Req(add) – del(rd_a) = 2T – 4D –d	2T – 4D –d
add	Arr(rd_a) + del(rd_a) = d	min(Req(div) – del(add), Req(s) – del(add), Req(mul) – del(add) + T) = 2T – 4D	2T – 4D –d
div	Arr(add) + del(add) = d + D	Req(sub) – del(div) = 2T – 3D	2T – 4D –d
sub	Arr(div) + del(div) = d + 2D	Req(mux) – del(sub) + T= 2T – 2D	2T – 4D –d
rd_b	0	Req(mul) – del(rd_b) = T – 2D –d	T – 2D –d
mul	max(Arr(rd_b)+ del(rd_b), Arr(add) –T + del(add))=d	Req(mux) – del(mul) = T – 2D	T – 2D – d
mux	max(Arr(sub)–T+del(sub), Arr(mul)+ del(mul)) = d + 3D –T	min(Req(wr) – del(mux) + T, Req(s) – del(mux)) = T – D	2T – 4D –d
wr	Arr(mux) – 1T + del(mux) = d + 4D – 2T	T – del(wr) = T - d	3T – 4D – 2d

Observing Table 3, one can deduce that the critical path in this DFG is: *rd_a* → *add* → *div* → *sub* → *mux* because these 5 operations have the same minimal value of slack. Hence, the important property of combinational slack, namely that all gates on the critical path have the same minimal slack, is preserved.

Definition 4 does not consider clock boundaries when computing sequential slack. It can be generalized to modify the slack computation to respect clock boundaries, by preventing operations from being started too close to the clock edge, i.e. when their arrival time plus delay would exceed the clock period. The generalization is straightforward and is omitted for the sake of space. The sequential slack that respects clock boundaries when computing required and arrival time is called *aligned slack***.**

When the sequential slack for every operation is known, one can use it to perform area recovery in the same fashion as is done in logic synthesis with the zero-slack algorithm [14], but without the limitation to a single state.

The zero-slack algorithm identifies a path segment with minimum non-zero slack, and then distributes excess delay among gates on that path segment, updating slack for all affected gates. This process

is repeated until all gates have zero slack. Excess delays can also be distributed unevenly, taking into account sensitivities of the gate area to delay increase, topology of the network (gates with smaller fanin would affect fewer paths when their delays change), etc.

To speed up the budgeting process, one can use *slack binning*, i.e., consider the values of slack within some margin to be the same. Our experiments showed that imposing a margin of 5% of the clock cycle has negligible effect on the results of the budgeting, but significantly speeds up convergence.

The algorithm for slack budgeting is proposed in Figure 7.

Budgeting Sequential Slack:

1. Given DFG D(O,C) and CFG G(V,E) compute minimal and maximal delays of operations and construct timed DFG $D0^t$.
2. Compute sequential aligned slack in $D0^t$ assuming maximal operation delays.
3. Perform budgeting of negative aligned slack in $D0^t$ by decreasing delays in [min, max] range. The result is $D1^t$ and a new distribution of operation delays del1:O →R.
4. Perform budgeting of positive aligned slack in $D1^t$ by increasing delays in [min, max] range. The result is $D2^t$ and a final distribution of operation delays del2:O →R.

Figure 7. Algorithm for slack budgeting

In a single budgeting step, the delay of operation *o* can be updated at most $N = slack(o)/margin$ times, where *margin* is the size of the predefined bin in which slack values are considered to be the same. From this follows that the complexity of budgeting is $O(C*N)$, which is linear for practical examples, in which fanout is bounded.

VI. SLACK-BASED SCHEDULING

Performing slack budgeting by using the DFG, as argued above, achieves higher quality of results during high-level synthesis (as will be shown in section VII). The ultimate goal of scheduling is to relate every operation to a clock cycle and its implementation resource. This is done using two mappings: *bind: O →Res* (operations to resources) and *sched: O→E* (operations to edges). Slack information could be used as a quick check for design feasibility, before full-fledged scheduling and binding.

Proposition 1. Given CFG *G* and DFG $D = (O,C)$ with clock period *T,* one-to-one mapping *bind: O →Res* that binds every $o \in O$ to a dedicated resource *r* with delay *del(o)*. If for $\forall$ *o* aligned $slack(o) > 0$, then there exists a schedule *S* such that in the netlist defined by schedule *S* every resource has a positive combinational slack.

The proof follows from the observation that arrival times of operations in a timed DFG define mapping *sched: O→E* (operations to edges), which together with mapping *bind* gives the schedule *S*.

One can also deduce that if slack budgeting for DFG *D* results in some operations having negative aligned slack, then there is no schedule that produces a netlist with positive combinational slack. This immediately follows from the observation that sharing of resources has only negative impact on timing.

In addition to providing these easy-to-check conditions for design feasibility, ***slack budgeting can be used to improve the quality of scheduling even in the presence of resource sharing***. Consider a typical scheduling framework [1], whose simplified description is presented in Figure 8 (initially ignoring steps in **bold**).

In step 1, a minimal set of resources (typically the fastest ones) to implement all operations in *D* is created. Then the resource- and timing-constrained scheduling problem is solved. If the current set of constraints (timing and resources) is infeasible, an expert system analyzes the problem and suggests how to relax the scheduling problem. Relaxation may result in adding a new resource, adding a state (if allowed by the designer), etc. After this, scheduling is repeated. Either this iterative process succeeds in producing a feasible schedule, or the expert system concludes that no relaxation exists to help scheduling because the design is overconstrained. In the latter case, one would need to change the specification or some constraints. As mentioned in Section 2, if the expert system uses the fastest resources, then area recovery during logic synthesis will be sub-optimal because it is applied only within a single state.

Scheduling algorithm:

Input: DFG D, CFG G, clock period T, Library L, User constraints U

0. Find optimal delays for operations by slack budgeting of DFG D.
1. Create a set of initial resources
2. Call Schedule_pass
3. If successful, do area recovery, return success
4. Else, relax constraints (add resource, add state, etc) and goto step3
5. If there is no relaxation contributing to schedule progress, return failure

Schedule_pass

1. E_{sort} = topologically sorted set of CFG G edges
2. forall *e* in E_{sort}
   a. schedule ready operations sorted by priorities
   b. if *e* is the last edge in *span(o)* and *o* is not scheduled, return failure
   c. recompute opspan of not-scheduled operations
   d. redo slack budgeting and if needed update resource delays

Figure 8. Scheduling with area/delay tradeoffs

The enhanced algorithm (now considering steps in **bold**) provides a different starting point for scheduling, by first computing the best set of possible resources for each operation from the ***globally budgeted*** delay/area standpoint. As a result, for critical operations the fastest resources are created, while for non-critical operations a slower but more area-efficient version is proposed.

In addition to providing a different starting point with a suitably better set of initial resources, changes must be made to the *Schedule_pass* algorithm itself. Sharing of a resource among operations *o1* and *o2* effectively results in merging the set of critical paths for *o1* and *o2* and introduces deviations from the timing analysis made on the original DFG. To take these changes into account, slack budgeting is redone after scheduling every edge. This requires the recomputation of the opSpan for all operations not yet scheduled. Since timing of operations when sharing a resource may only worsen, new slack violations may appear and must be fixed by decreasing the delays of operations.

VII. EXPERIMENTAL RESULTS

To quantitatively evaluate the effectiveness of the proposed approach, we selected an IDCT algorithm used in video decoding and performed an extensive design space exploration for it, using both pipelined and non-pipelined implementations, with latencies ranging from 32 to 8 clock cycles. We performed 15 HLS and logic synthesis runs, for the IDCT exploring a 20X power range, a 7X throughput range and a 1.5X area range. In all runs, we made sure that timing was met for the specified clock period after logic

synthesis. We first performed high-level synthesis using the conventional approach, i.e., using the fastest resources and then using area recovery. We then compared it with the results of our proposed approach (see Figure 8). The cell area results (pre-placement, using a TSMC 90nm library) are shown in Table 4, where A_conv and A_slack stand for area numbers for the conventional and slack-based approaches, respectively. They suggest the following observations:

Table 4. Area savings for timing-based approach

Des	A_conv	A_slack	Save %	Des	A_conv	A_slack	Save %
D1	90085	89287	0.1	D9	98506	84932	16.0
D2	65441	63974	2.3	D10	103026	88481	16.4
D3	67365	57440	17.3	D11	106247	93156	14.2
D4	68716	58651	17.2	D12	105657	103305	2.3
D5	76888	81566	-5.5	D13	79871	63232	26.2
D6	80848	83433	-3.0	D14	76963	71290	8.0
D7	83826	88017	-4.7	D15	86099	74238	16.0
D8	69210	62524	10.7	Average savings			**8.9**

1. Exploiting area/delay tradeoffs has a positive impact on implementation quality, providing on average a 9% area saving after logic synthesis.
2. For three designs (D5, D6 and D7), the quality of the slack-based implementation deteriorates with respect to the conventional approach. The analysis shows that in these designs most resources end up being timing critical, which does not provide much room for improvement using a slack-based approach. The degradation comes from the fact that the scheduler was unable to recover from starting with slower resources and had to restrict sharing to meet timing. A similar phenomenon could be observed in the example shown in Section 2 (see Table 2) where starting from slower resources and upsizing them on the fly results in a worse area than starting from the fastest resources.

Computing timing introduces performance penalties during scheduling. This was evaluated (Table 5) by profiling the scheduling routines applied on design D1 from Table 4. The first column corresponds to conventional scheduling, the second column represents the scheduling time for the proposed slack-based approach, while the third column provides the scheduling time for the proposed approach when timing analysis is done using the Bellman-Ford algorithm as in [10]. Based on our experience with customers using our technology, we observe that 20% performance degradation is acceptable for a user, while the formulation based on the Bellman-Ford algorithm is impractical (see comments in Section III).

Table 5. Relative scheduling execution times

Conventional	Sequential slack based	Bellman-Ford based
1	1.18	10.2

VIII. CONCLUSIONS

This paper describes an enhanced framework for scheduling and binding in high-level synthesis. It improves over past approaches by using a behavioral timing analysis that quantitatively estimates the criticality of operations by computing their sequential slack. This information is used to choose the most area/time efficient set of resources during scheduling. The approach is implemented in a commercial high-level synthesis tool. Its application results in area savings of 9% over a conventional approach.

REFERENCES

1. A. Kondratyev, M. Meyer, L. Lavagno, Y. Watanabe, "Realistic Performance-constrained Pipelining in High-level Synthesis," Design, Automation & Test in Conference, pp. 1382-1387, 2011.

2. Giovanni De Micheli, *Synthesis and Optimization of Digital Circuits,* McGraw-Hill, 1994.

3. D. Gajski, N. Dutt, B. Pangrle, "Silicon compilation (tutorial)," in Proc. IEEE Custom Integrated Conference, pp.102-110, 1986.

4. Pierre G. Paulin, John P. Knight, "Force-Directed Scheduling in Automatic Data Path Synthesis," in Proc. DAC, pp. 195-202, 1987.

5. R. Camposano, "Path-based Scheduling for Synthesis," IEEE Trans on CAD of Integrated Circuits and Systems, vol.10, no. 1, pp. 85-93, 1991.

6. S. Gupta, N. Dutt, R. Gupta, and Al. Nicolau, "SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations," *VLSI Design*, pp. 461-466, 2003.

7. R. Schreiber et al, "High-level synthesis of nonprogrammable hardware accelerators," in Proc. IEEE International Conference on Application-Specific Systems, Architectures, and Processors, pp. 113-124, 2000.

8. D. Ku, G. Micheli, "Relative scheduling under timing constraints," in Proc. DAC, pp.59-64, 1990.

9. A. Kuehlmann, R. Bergamaschi, "Timing analysis in high-level synthesis," Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD '92), pp. 349-354, 1992.

10. N. Chandrachoodan, S. Bhattacharyya, L. Ray Liu, "The hierarchical timing pair model," in Proc. of IEEE Symposium on Circuits and Systems, pp 367-370, 2001.

11. X. Liu, M. Papaefthymiou, E. Friedman, "Maximizing performance by retiming and clock skew scheduling," in Proc. DAC, pp. 231-236, 1999.

12. J. Cong, S. Lim, "Physical planning with retiming," in Proc. of ICCAD, pp. 2-7, 2000.

13. Steven S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.

14. P. Hauge, R. Nair, E. Yoffa, "Circuit placement for predictable performance," in Proc. of ICCAD, pp. 88-91, 1987.