POLITECNICO DI TORINO
Repository ISTITUZIONALE

Realistic performance-constrained pipelining in high-level synthesis

Original
Realistic performance-constrained pipelining in high-level synthesis / Kondratyev, A.; Lavagno, Luciano; Meyer, M.; Watanabe, Y.. - (2011), pp. 1-6. (Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011).

Availability:
This version is available at: 11583/2501070 since:

Publisher:

Published
DOI:

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

31 May 2026

Realistic Performance-constrained Pipelining in High-level Synthesis

Alex Kondratyev, Luciano Lavagno, Mike Meyer, Yosinori Watanabe
Cadence Design Systems
San Jose, USA
{kalex, luciano, meyer, watanabe}@cadence.com

***Abstract*— This paper describes an approach to pipelining in high-level synthesis that modifies control/data flow graphs before and after scheduling. This enables the direct re-use of a pre-existing, timing- and area-aware non-pipelined simultaneous scheduler and binder. The approach ensures that the RTL output can be synthesized within the given timing and area constraints. Results from real industrial designs show the effectiveness of this approach in improving Pareto optimality with respect to area, delay and power.**

Keywords- pipelining, high-level synthesis, design exploration

I. INTRODUCTION

In spite of constantly improving CAD tools, the number of available transistors grows faster than the ability to effectively design circuits that use them. Moving beyond RTL calls for a new set of design tools that will provide the capability of design exploration and synthesis from high-level specifications, enabling the effective use of billions of transistors per chip.

We developed a commercial high-level synthesis (HLS) tool that has been adopted by many semiconductor and system companies in their production design flows, and their experience demonstrates the effectiveness of this technology for productivity gains.

This paper discusses a particular feature of the tool, namely the ability to automatically obtain a *high-performance pipelined implementation*, which has been used for *designs with up to 2GHz final clock speed*. The essential problem of pipelining consists of two parts. One is resource binding, i.e. how each operation involved in the pipeline should be implemented, while the other is scheduling, i.e. at which pipeline stage each of such operations should be executed. As presented in Section III, the approaches known in the literature either solve these two problems sequentially, or take a naïve formulation of the combined problem which is too expensive to solve for practical designs. The key distinction in our approach is to solve these problems together, while the resulting algorithm is still applicable to commercial designs of current high complexity. Our approach first applies certain transformations to the Control Data Flow Graph (CDFG) that represents the functionality of the design, and then a synthesis procedure is applied to the modified CDFG to solve the binding and scheduling together. This approach captures the effect of pipelining in the CDFG, rather than in the synthesis algorithms, and thus the same synthesis procedure is applicable whether the design is implemented with pipelining or without pipelining. This brings another advantage that a single synthesis engine can be used consistently to effectively explore both pipelined and non-pipelined architectures of the design.

II. DESIGN STEPS

Our tool takes as input a set of SystemC modules containing one or more threads and methods, as well as design requirements and constraints. The modules may be totally *untimed*, that is, have no clock statements; or *partially timed,* that is, have *some* or *all* clock statements inserted. A frequent use case for a partially timed specification is one in which the protocol is specified in a cycle-accurate manner (for example, a bus protocol), while the main computation is loosely timed (for example, it is annotated with a maximum latency and a target clock cycle). The design requirements and constraints consist of the clock frequency, communication protocols, etc.

A very simple example of input to our tool is shown in Figure 1~~Figure 1~~.

```
class example1: public sc_module {
    sc_in<bool> clk, rst;
    sc_in<int>  mask, chrome, scale, th;
    sc_out<int> pixel;
    …
};
void example1::thread() {
    wait();
    while (true) {
        int aver = 0;
        wait(); // s0
        do {
            int filt = mask;
            delta = mask * chrome;
            aver += delta;
            if (aver > th) {
                aver *= scale;
            }
            wait(); // s1
            pixel = aver * filt;
        } while (delta != 0);
    }
}
```

Figure 1. Example of SystemC specification

The design flow with our tool is shown in Figure 2~~Figure 2~~.

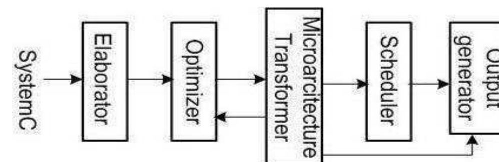

Figure 2. Tool design flow

At the end of elaboration, the input specification is represented by its control flow graph (CFG) and data flow graph (DFG) [14]. Nodes of the CFG either serve to fork/join control flow (conditionals and loops in SystemC) or correspond to "wait()" calls in SystemC. The DFG nodes are operations, and its edges are data dependencies between nodes. Every DFG operation is associated with a particular

edge of the CFG. Figure 3 shows the CFG and DFG for the body of the do_while loop in Figure 1.

The goal of the optimizer is to simplify the DFG and CFG as much as possible, by applying standard compiler optimizations, such as constant propagation, operand width reduction, operation strength reduction, etc. The branch predication transformation is essential, since it replaces fork-join structures in the CFG by a straight-line segment with predicates enabling operations, as shown in Figure 4.

This transformation increases the mobility of operations in conditional branches, to enable the computation of *a+b, c+d,* and *func()*, before evaluating the branch condition *cond*, which can help to implement the design under tight timing constraints when the values necessary to evaluate *cond* are arriving late.

The micro-architecture transformer changes the control structure significantly. Examples of transformations include function inlining, loop unrolling and loop pipelining. It is difficult to accurately determine which ones of these transforms will improve the final implementation. Hence, we provide means for designers *to manually and easily* specify their intent.

When optimizations and micro-architectural choices are complete, the CFG and DFG are scheduled (Figure 2~~Figure 2~~). The purpose of this step, described in detail in Section IV, is to bind each DFG operation to a time step (edge) in the CFG and to a particular resource from the given set of resources. Moves of DFG operations within the CFG are guided by a cost function that takes into account the timing, area and latency constraints

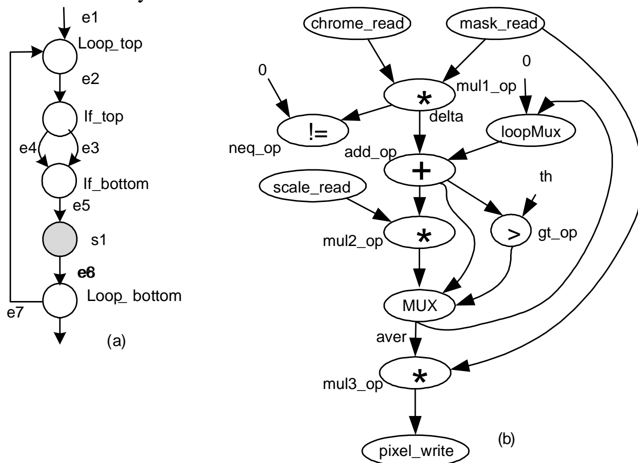

Figure 3. CFG (a) and DFG (b) for the do_while loop body

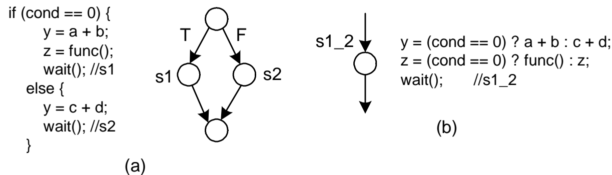

Figure 4. Predicate conversion.

When the scheduler succeeds, the output generator (Figure 2~~Figure 2~~) produces a set of models at different abstraction levels: starting from higher level models (which are used for simulation, for example, a virtual prototype of an SOC) to RTL (which is used for synthesis).

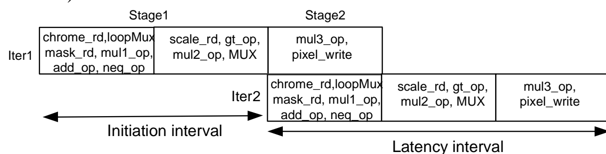

Figure 5. Pipelining the example of Figure 1~~Figure 1~~ with LI=3 and II=2

Loop pipelining overlaps operations from different iterations to create a more compact schedule [1]. Its two key parameters are: *Initiation Interval* (II) to specify the number of cycles between the execution of consecutive iterations, and *Latency Interval* (LI) to define the number of cycles to execute a single iteration, as illustrated in Figure 5~~Figure 5~~. The iteration is partitioned into $\lceil LI/II \rceil$ pipeline stages, where each stage contains II states.

III. PRIOR WORK

In hardware synthesis (contrary to the software domain), resources are not fixed and delays of operations are often not multiples of clock cycles. A common situation is time-constrained pipelining, where resources are minimized, given upper bounds on execution delay. Known algorithms often appeal to heavy machinery (such as ILP) to address scheduling and pipelining in a uniform setting [3]. They are slow and not scalable, which calls for development of efficient heuristics for automatic pipelining in HLS. In the Sehwa system [4] pipeline synthesis is done using list scheduling, which takes into account inter-loop dependencies by defining "resynchronization events" that prevent pipelines from running at the maximum speed. The resynchronization mechanism is decoupled from the scheduling procedure, thus leading to sub-optimal binding of operations to control steps. Hence [5] improves the Sehwa approach by using interleaved "ASAP" (as soon as possible) and "ALAP" (as late as possible) scheduling. Inter-iteration dependencies are taken into account during the folding step in a constructive way, i.e. if folding succeeds, then all dependencies are satisfied. ~~When~~ If folding fails, some pipelining constraints (e.g. loop latency) need to be relaxed. Separation of scheduling and constraint checking is a significant source of inefficiency of this method.

Another group of heuristic methods relies on semantics-preserving transformations that restructure the CDFG and rewrite the loop to select a repetitive pattern to be used as the body of a pipelined loop (this pattern is often called the *pipeline kernel*). In percolation scheduling [6] the loop is incrementally unrolled and operations migrate upwards until the pipeline kernel emerges. Lately this approach was enhanced in [7] by adding circular dependency analysis during scheduling, covering inter-iteration and resource dependencies and improving results when using memories.

Modulo scheduling [8] is one more popular technique in which the pipeline kernel is automatically detected when performing scheduling with backtracking. In modulo scheduling assigning of an operation to a timing slot is modeled by explicitly placing several instances of this operation II slots apart. If this causes a conflict with previously placed operations the schedule chooses a candidate for unscheduling and backtracks. The drawback of the modulo scheduling is that its formulation is significantly more involved than that of traditional scheduling and requires a specialized engine to address pipelining.

In the last decade basic techniques for pipelined scheduling did not change much and are used directly in high-level synthesis tools such as Spark [9] (percolation scheduling), Streamroller[10], Pico [11] (modulo scheduling). Most of the enhancements were targeted to lift the limitation of applying pipelining only to the innermost loop [11,12,13]. These enhancements are orthogonal to our proposed approach, which focuses on pipelining loops as specified by the user, who may want to unroll or merge loops beforehand in order to satisfy cost, performance and power requirements.

We claim (based on a broad user experience) that separation between scheduling and kernel selection, as well as between scheduling and binding, is the main obstacle that prevents obtaining high quality of results in automated hardware pipelining. *The main contribution of this paper is an approach that makes pipelining decisions on the fly during the process of unified scheduling and*

binding. This improves the efficiency and has a negligible impact on the complexity of the scheduling.

IV. SCHEDULING APPROACH

A major drawback of most past research in high-level synthesis is that the *splitting between scheduling and binding does not consider detailed timing when defining the schedule of operation*s. Since meeting a precise clock period is an essential requirement, we perform iterative *simultaneous scheduling and binding passes* [14]. At every pass, a latency-, clock cycle- and resource-constrained scheduling problem is solved, using a limited set of resources. If a scheduling pass fails, an internal expert system is called to choose an action to relax some of the constraints. Its portfolio includes adding states (where permitted by the designer), adding resources, performing speculation, etc.

A. Creating an initial set of resources

In order to create a lower bound to the set of resources for the given CDFG, we define a mapping that relates every operation to compatible resource types, where a resource type is represented as a combination of the operation type (addition, subtraction, multiplication, etc.) with operand and result widths. E.g. A1[7:0] + B1[4:0] and A2[5:0] + B2[6:0] could be implemented by an 8x6 bit adder. We do not merge resources of very different bit_widths, to avoid bad impact e.g. on power consumption. Then we create a set of intervals through intersection and union of ASAP/ALAP ranges of operations of compatible types. Finally, we estimate the resource demand for every interval and choose the lower bound to be the maximal among the demands for all intervals.

This approach improves over [15] in two ways. First, life spans of operations are timing aware. In other words, ASAP and ALAP are determined by performing approximate timing analysis on the DFG, initially ignoring the sharing multiplexers. Second, we take into account mutual exclusivity of operations coming from the predicate transform (see Section II).

B. Pass scheduler

The salient characteristics of the pass scheduler are listed below:

1. *High accuracy of timing estimation.*

Scheduling an operation binds it not only to a CFG edge, but also to a resource, taking into account the delay of multiplexers and the *potential false combinational cycles created by them*.

The scheduler is tightly integrated with logic synthesis. It builds a netlist for the part of the CDFG that has been scheduled so far, and performs timing queries (whose results are cached appropriately) on the netlist.

2. *Chaining and use of multi-cycle operations.*

Analyzing scheduler results on a set of customer designs, we found that the combinational depth of logic in a single cycle often exceeds 5 operations. This clearly shows the importance of supporting combinational chaining within a cycle. The support for (possibly pipelined) multi-cycle operations is also required to permit binding of operations to predesigned IP blocks.

3. *Handling combinational cycles.*

The occurrence of combinational cycles during scheduling is illustrated in Figure 6~~Figure 6~~. This cycle is never sensitized in any reachable control state. The false paths could be reported to the downstream logic synthesis tool; however, this reduces greatly the room for optimization because logic synthesis must preserve the pins specifying the path. Instead, we *avoid bindings resulting in combinational cycles*. This may require extra resources, but in practice improves the final post-synthesis quality of results and satisfies established RTL coding rules.

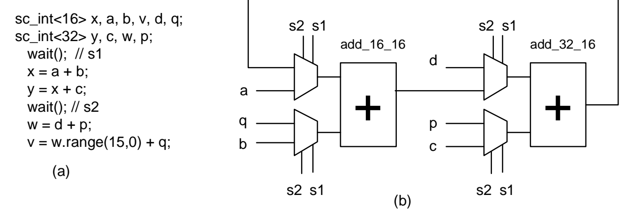

Figure 6. Combinational cycles in scheduling

4. *Support of user-defined constraints.*

The tool provides a wide set of constraints to express user intentions about the final implementation. These constraints range from specifying the edge and/or resource for a particular operation (to be respected during scheduling) to providing latency bounds for loops or blocks of code (for example, between I/O operations).

The list scheduler implementing the preceding features is presented in Figure 7~~Figure 7~~. The priority function takes into account the mobility of the operations defined by timing-aware ASAP/ALAP intervals (similar to Force-Directed Scheduling), the complexity of operations (more complex ones are scheduled first), the size of the fanout cone of an operation, etc.

When the pass scheduler fails, the set of scheduling constraints must be relaxed. The history of the scheduling pass is recorded in a set of restraints, which are issued every time a binding of an operation to an edge and/or a resource fails. Restraint analysis is done for the fanin cones (in the DFG) of the failed operations and of those whose scheduling suggests some room for improvement. Restraints are assigned weights based on their proximity to failed operations and the number of failures they help solve. Each restraint suggests a set of actions that can be applied to improve the scheduling. Timing restraints could be fixed by adding states to the CFG, by adding resources or by speculating operations. Restraints stemming from combinational cycles forbid the use of a resource for an operation, etc. Every action has an estimated cost, which is combined with the number of restraints solved by this action and the restraint weight. The action with the best estimated gain wins and is used to relax the constraints for the next scheduling pass.

```
SCHEDULE_PASS(CFG C, DFG D, clock period Tclk,
Library L, User Constraints U)
Paths ← Set of combinational paths in CFG;
forall  p in Paths {
 forall edges in p {
   Ready ← operations ready to schedule;
   Compute_op_priorities(Ready);
   op_best ← highest priority op;
   Op_res ← resources compatible with op_best;
      forall r in Op_res {
        if (bind(op_best,r) == success) break;
      }
   if(op_best failed and e is last in lifespan){
      Failed_ops ← op_best;
   } else {Update(op_best, Ready);}
  }
  if (Failed_ops != ∅) {return failure;}
}
```

Figure 7. Performing a single scheduling pass

Example 1. Sequential Microarchitecture. Let us illustrate the scheduling process on the example from Figure 1~~Figure 1~~, with: $1 \leq latency \leq 3$ for the do-while loop. The scheduler starts from latency 1. For this example we will use Tclk=1600 and the artisan_90nm_typical library. Finding the initial set of resources is trivial: multiplication is the only repeated operation and 3 multiplies

are to be scheduled in at most 3 states, which suggests that a single multiplier suffices (it is a lower bound that might be reconsidered during scheduling).

TABLE 1. INITIAL SET OF RESOURCES WITH DELAYS

resource	mul	add	gt	neq	ff	mux2	mux3
delay (ps)	930	350	220	60	40/70	110	115

Table 1 shows the fastest logic implementation for the resources of Example 1. Note that the table includes registers (*ff*) and 2 and 3-input multiplexers for resource and register sharing.

I/O operations are scheduled at the very same states where they are specified in the input code. Hence *chrome_read* and *mask_read* are scheduled in *s1* and *mul1_op* is enabled for scheduling. To check the feasibility of binding *mul1_op* to resource *mul* in *s1,* the scheduler builds the datapath shown in Figure 8~~Figure 8~~(a). Note that resource *mul* is instantiated with muxes at its inputs. This improves timing estimation when resources are shared between several operations. Using a very simple timing model *for the sake of illustration*, the delay of the netlist of Figure 8~~Figure 8~~(a) is:

del_ = FF_~~hold~~ launch + del_mux + del_mul + del_mux + FF_setup = 40 + 110 + 930 + 110 + 40 = 1230

The delay satisfies the clock cycle constraint, and thus the scheduler moves to the next ready operation (*add_op*). Figure 8~~Figure 8~~(b) shows the datapath model when binding *add_op* to resource *add* in state *s1* (the DFG has a single addition operation, thus there are no input muxes). The delay of this netlist is 1580, which satisfies the clock cycle and the the binding of *add_op* in state *s1* is accepted.

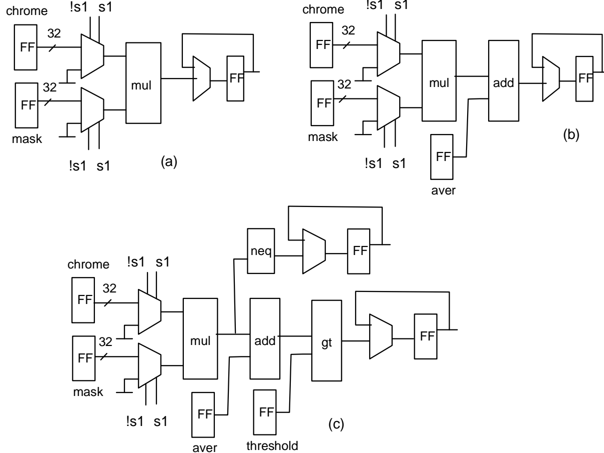

Figure 8. Datapath modeling during scheduling

Finally Figure 8~~Figure 8~~(c) shows the datapath when the scheduler binds operations *neq_op* and *gt_op* to the corresponding resources in the same state *s1*. The critical path going through resource *gt* has a delay of 1800ps and results in a negative slack of -200ps for this resource. The scheduler rejects the binding of *gt_op* in state *s1* and fails because *s1* is the only state of the loop. This indicates that the specification is overconstrained under the current set of resources and latency 1.

To relax constraints the scheduler analyzes the set of restraints for failed operations: lack of resources (for operation *mul2_op*) and a negative slack (for operation *gt_op*). The corrective actions are to add state to the loop body or to add one more *mul* resource. Adding a state can resolve both resource contention and negative slack; hence the scheduler adds state *s2* and restarts the pass scheduler.

The second pass will fail again when attempting to bind *mul3_op* in state *s2* because resource *mul* is busy implementing operation *mul2_op*. The corrective action is to add one more state (*s3*) to the loop (adding one more multiplier does not help because two multiplications cannot fit in the given clock cycle). Using 3 states in the loop, the scheduler succeeds with the schedule of Table2. It uses the minimum set of resources and has a throughput of 3 cycles per iteration.

TABLE 2. SCHEDULE FOR EXAMPLE 1.

Res / state	*mul*	*add*	*gt*	*neq*	*mux*
s1	*mul1_op*	*add_op*		*neq_op*	
s2	*mul2_op*		*gt_op*		*mux_op*
s3	*mul3_op*				

With higher throughput requirements, the user may explore the pipelining capabilities provided by the scheduler, as an alternative to selecting faster, and hence much more expensive, resources.

V. PIPELINING APPROACH

Based on extensive discussions with designers, we formulated the following conditions for effective pipelining:

1. *The II parameter must be specified by the designer.*

It is known that finding the smallest II for a given loop is an NP-complete problem [1], and many efforts have been documented to find good heuristics for it. We believe that the practical importance of this problem *in the context of high-level synthesis* is limited, because in hardware the II is a fundamental parameter defining the system throughput, and designers do not have much flexibility in changing it. At most they can change it in conjunction with the clock cycle (for example, double the II and halve the clock period in order to reduce area at the same overall throughput).

2. *The LI parameter should be chosen (within designer-specified bounds) by the tool.*

When the pipelined loop performs many (often an infinite number of) iterations before exiting, the LI impact on throughput is negligible, since it affects only the epilogue of the loop. Exploration often starts from LI = II+1 (the minimum for pipelined execution).

3. *The choice of the pipeline kernels must be timing-driven.*

This suggests that finding the kernel should be incorporated into the scheduling process, instead of the traditional "schedule-then-move" approaches like modulo scheduling [2, 8].

The following procedure describes a new approach for automatic pipelining of a loop that: a) requires minor extensions of the constrained scheduling algorithm described in Section IV, and thus shares all its advantages, b) is effective, as will be shown in the final section, and c) takes into account timing when choosing the pipeline kernel, thus resulting in a pipeline that can be implemented with the required clock cycle.

For the sake of simplicity, assume that both II and LI are known for a loop (LI is handled by the scheduler by adding states within bounds, as in the non-pipelined case). The pipelining of a loop is implemented in two steps:

I. Scheduling of operations for a single loop iteration within the *latency interval* of a loop.

II. Folding the scheduled loop iteration into a pipeline with $PS = \lceil LI/II \rceil$ stages.

Step I is performed as follows:

1. Converting the loop into a straight-line sequence of nodes in the CFG. This is obtained by first balancing the latency of all fork/join regions of the loop body (coming from conditional statements) so that they all have LI states, and then applying *full predicate conversion.* Nested loops must either be unrolled or correspond to the "stalling" of the pipeline (waiting for an external condition). The stalling loops are ignored during the scheduling passes and inserted back in the CFG during the "fold back" step.

2. Identifying edges in the CFG of the loop body (also called "control steps" in the literature) that are "equivalent" from the scheduling point of view. These are the edges that are II states apart in the loop with LI states. They are folded onto a single edge in the final pipeline. For example, as shown in Figure 5~~Figure 5~~, operations from the first loop iteration scheduled in the first clock cycle of *Stage2* overlap with operations from the next iteration scheduled in the first clock cycle of *Stage1*. Operations scheduled on equivalent edges *cannot share a resource* (unless, of course, they depend on orthogonal predicates).

3. Scheduling a single iteration within *LI* state nodes.

Two additional requirements must be satisfied within the scheduler in order to address pipelining:

a) Preserving data causality between loop iterations.

Different loop iterations may be causally related. The next loop iteration may need to wait until previous iterations compute the necessary data. Iteration dependencies are represented by cycles that form strongly connected components (SCC) in the DFG of a loop. It is easy to see that preserving causality requires all operations from each strongly connected component of the DFG to be scheduled within *II* states. The DFG in Figure 3~~Figure 3~~(b) has a single strongly connected component: that computes *aver* (involving *loopMux, add_op, mul2_op* and *MUX*). Operations from this SCC must be scheduled in two adjacent states (since *II = 2* for this example). Note however that there is flexibility in choosing the stage in which to schedule an SCC, which might be exploited to achieve better timing. Hence the set of actions of the scheduler includes moving an SCC from one stage to another when facing negative slack.

b) Constrained resource sharing.

A resource used for operation *op* scheduled at edge *ej* is considered busy for all edges *ek* equivalent to *ej*.

Once the loop is successfully scheduled in LI states, it needs to be folded to reduce the number of states in the body to II. This is done by folding equivalent edges onto a single edge, whose scheduled set of operations is the union of the operations from the folded edges. Additional control is added to represent the pipeline stage that is being executed. In most loop iterations, all stages of the loop are executed simultaneously. The exceptions are: the prologue and epilogue of the loop, when only the first or the last stages must be active, and the stalling loops, in which no stage must be active while the stalling condition is true. To ensure this, all loop operations are predicated by the corresponding stage signals, generated from the appropriate FSM state registers (if the stage is not active, the operation is not executed).

Note that with the minor exception of requirements a) and b) from item 3 in Step I, the scheduling procedure for the pipelined loop remains identical to the non-pipelined case. This is a major advantage of the suggested approach.

Example 2. Pipelined Microarchitecture (II=2). Due to edge equivalence, resources should not be shared in states *s1* and *s3*, hence two *mul* resources must be created. Then scheduling proceeds exactly as for the sequential microarchitecture, except that it starts from 3 states (pipelining requires LI>II). The only pipeline constraint –is ensuring that *SCC={loopMux, add_op, mul2_op, MUX}* is scheduled in a single pipeline stage. This is satisfied for the schedule shown in Table 2, which is- applicable to the pipelined case as well (changing only bindings: *mul1_op→mul1, mul2_op→mul1 mul3_op→ mul2)*. This illustrates the– uniformity of the approach between the sequential and pipelined cases.

Example 3. Pipelined Microarchitecture (II=1). Pushing the throughput requirements to the limit results in implementing a single loop iteration per clock cycle. Scheduling with LI=2 fails because two chained multiplications in a single state exceed the clock period. The scheduler relaxes constraints and increases– LI to 3. The difference from the non-pipelined case is that no resource is shareable between states (II=1 makes all the edges equivalent), hence 3 multipliers are created in the initial set of resources, and *SCC={loopMux, add_op, mul2_op, MUX}* must be scheduled in one state.

Scheduling uses the same bindings as in Table 2 until *mul2_op*. This operation is a part of the SCC and must be scheduled in the same state as *add_op*. This however violates timing, and the scheduling pass fails. A novel feature of the suggested pipelining approach is that in the relaxation phase this failure is distinguished from an ordinary negative slack failure and the corrective action of moving the whole SCC to state *s2* is suggested. With this corrective action, the scheduler succeeds.

Different implementation architectures for Example 1 are compared by throughput and area. All architectures provide meaningful trade-offs: higher throughput implies larger area. Another perspective could be provided by fixing the throughput and synthesizing P1, P2 and S with clock cycles T, T/2 and T/3 (to keep throughput constant). This experiment is illustrated, with a more realistic design, in Section VI. It may result in smaller area because *more non-timing critical* (hence smaller) *resources may require less total area than fewer critical* (hence larger) *ones.*

TABLE 3. COMPARING MICROARCHITECTURES FOR EXAMPLE 1.

	Sequential(S)	Pipe, II=2 (P2)	Pipe, II=1(P1)
#cycles/iteration	3	2	1
Area	16094	24010	30491

VI. EXPERIMENTAL RESULTS

Our experiments illustrate three key points of this approach.

1. The suggested pipelining method is effective and practical.

Figure 9~~Figure 9~~ shows a plot for about 40 industrial designs, obtained from companies that use our tool for their design flows and that use the pipelining method previously described. These designs have different complexities, with the number of operations ranging from 100 to over 6000 (the average is 1400). They include filters, FFTs, image processing algorithms, etc. The scheduler time, including using logic synthesis for area and performance estimates, never exceeds one hour (on average, it is 7 minutes). Execution time does not correlate with input CDFG size, but depends on the number of pass scheduler calls, which in turn depends on how tightly constrained the design is, by how many conflicts or cycles are discovered and must be avoided by restraints, by how many times resources need to be added to the initial estimate, and so on.

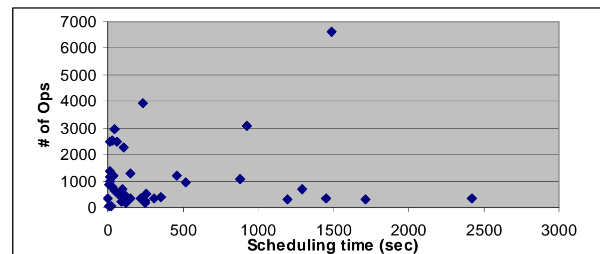

Figure 9. Profiling designs and scheduling times

2. Selecting the pipeline kernel by timing is vital.

We investigated the seven most timing-critical designs among those reported above, and disabled the action of moving SCCs to later pipeline stages when a negative slack is encountered. This resulted in

a significant increase of negative slack after synthesis, which had to be compensated by larger area during subsequent logic synthesis. The resulting ~~percentual~~ percentage area penalty for these designs is shown in Table 4. *This experiment demonstrates that timing awareness of the pipelining approach is a key to delivering a predictable design flow. It allowed us to achieve a frequency of 2GHz for one of the pipelined designs.*

TABLE 4. IMPACT OF TIME-DRIVEN HEURISTICS

	D1	D2	D3	D4	D5	D6	D7	Avg
% Area Penalty	14.7	2.7	33.0	21.5	3.7	6.4	12.9	13.5

3. Pipelining extends the area-delay-power trade-offs.

We selected an IDCT algorithm used in video decoding, and tried both pipelined and non-pipelined implementations, with latencies ranging from 32 to 8 clock cycles. We performed 25 HLS and logic synthesis runs, exploring a 20X power range, a 7X throughput range, and a 2X area range.

Figure 10~~Figure 10~~ shows the area/performance curves for various micro-architectural solutions, to explore the impact of pipelining. Each curve corresponds to a different microarchitecture (loop latency). The *delay* is actually the *inverse of the throughput* and is obtained by multiplying the Initiation Interval (which is the same as the latency for the non-pipelined cases, and half of the latency for the pipelined cases) by the clock cycle.

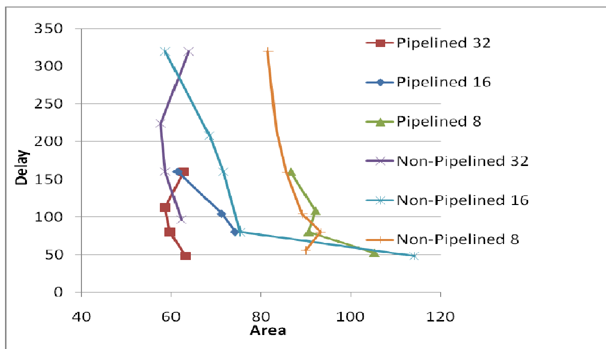

Figure 10. Area/delay for different micro-architectures

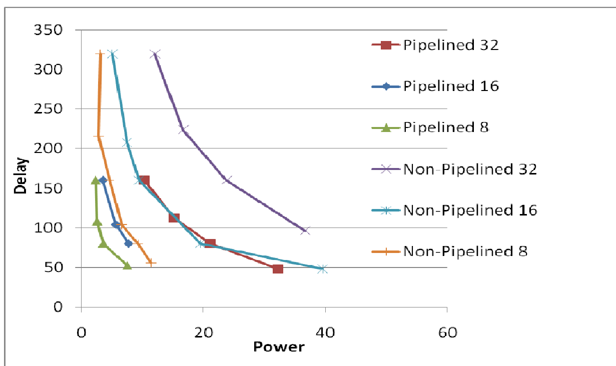

Figure 11. Power/delay for different micro-architectures

Pipelining in this case improves area at equal throughput, because it doubles the clock cycle, with a significant advantage in terms of area. Note also that the best Pareto point (bottom left) can be achieved only by pipelining (with Initiation Interval 16 and Latency Interval 32, called "Pipelined 32"). The same performance level could not be achieved without pipelining, except with a latency of 8 or 16 clock cycles ("Non-Pipelined 8" and "16"), but with larger area due to the faster clock.

Of course, that same low area, high performance point has a cost in terms of power, as shown in Figure 11 (it is the bottom point of the "Pipelined 32" curve).

VII. CONCLUSIONS

This paper describes a very practical and effective approach to pipelining for high-level synthesis. It improves over past approaches by using a highly accurate area and timing model while scheduling the pipeline. It automatically finds the best pipeline kernel and lets designers explore better area/delay/power solutions than with non-pipelined implementations. Finally, it re-uses essentially the same infrastructure of the non-pipelined scheduler, resulting in smooth and pragmatic trade-offs between pipelined and non-pipelined implementations.

REFERENCES

[1] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *Proc. ACM SIGPLAN '88*, 1988, pages 318-327.

[2] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc. Fourteenth Annual Workshop on Microprogramming*, 1981, pages 183-198.

[3] F. Sánchez and J. Cortadella. Time Constrained Loop Pipelining. In *Proc. ICCAD*, November 1995, pages 592-596.

[4] N. Park and A.C.Parker, "Sehwa: a software package for synthesis of pipelines from behavioral specifications", IEEE Trans. Computer-Aided Design, vol. 7, 1988, pp. 356-370.

[5] C-T Hwang, Y-C Hsu, and Y-L Lin, Scheduling for functional pipelining and loop winding, Proc ACM/IEEE 28th Design Automation Conference, pp 764-769, 1991

[6] R. Potasman, J. Lis, A. Nicolau, and D. Gajski. Percolation based synthesis. In *Design Automation Conference, 1990, Proceedings, 27th ACM/IEEE*, pages 444-449.

[7] L. Gao, D. Zaretsky, G. Mittal, D. Schonfeld, P. Banerjee. A software pipelining algorithm in high-level synthesis for FPGA architectures. In *ISQED 2009*, pages 297-302.

[8] B. Ramakrishna Rau: Iterative modulo scheduling: an algorithm for software pipelining loops. MICRO 1994: 63-74

[9] S. Gupta, N.Dutt, R. Gupta, Al. Nicolau: SPARK: A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations. VLSI Design 2003: 461-466.

[10] R. Schreiber, S. Aditya, R. Rau, V. Kathail, S. Mahlke, S. Abraham, G. Snider: High-Level Synthesis of Nonprogrammable Hardware Accelerators. ASAP 2000: 113-124.

[11] M.Kudlur, K.Fan, S.Mahlke: Streamroller: automatic synthesis of prescribed throughput accelerator pipelines. CODES 2006: 270-275.

[12] K. Turkington, A. Constantinides, K. Masselos, P. Cheung: Outer Loop Pipelining for Application Specific Datapaths in FPGAs. IEEE Trans. VLSI Syst. 16(10): 1268-1280 (2008)

[13] J. Cong, W. Jiang, B.Liu, Y. Zou: Automatic memory partitioning and scheduling for throughput and power optimization. ICCAD 2009: 697-704

[14] D. Knapp and M. Winslett. A Prescriptive Formal Model for Data-Path Hardware. In *IEEE Trans. Computer-Aided Design*, Vol. 11, No. 2, Feb. 1992, pages 158-184.

[15] Sharma A, Jain R. Estimating architectural resources and performance for high-level synthesis applications. In. Proc. 30th Design Automation Conference, 1993, pp 355- 360.