Equivalence Checking for Function Pipelining in Behavioral Synthesis

Kecheng Hao*, Sandip Ray† and Fei Xie*
* Dept. of Computer Science, Portland State University, Portland, OR 97207, USA
{kecheng, xie}@cs.pdx.edu
† Strategic CAD Labs, Intel Corporation, Hillsboro, OR 97124, USA
sandip.ray@intel.com

***Abstract*—Function pipelining is a key transformation in behavioral synthesis. However, synthesizing the complex pipeline logic is an error-prone process. Sequential equivalence checking (SEC) support is highly desired to provide confidence in the correctness of synthesized pipelines. However, SEC for function pipelining is challenging due to the significant difference between the behavioral specification and synthesized RTL. Furthermore, function pipelines include hardware logic for dynamically inserting "bubbles" (pipeline stalls), which bring additional difficulties in equivalence checking. We develop an SEC framework for behaviorally synthesized function pipelines by (1) building a reference pipeline model with a certified function pipelining transformation, which faithfully captures bubble insertion; and (2) checking the equivalence between the reference model and synthesized RTL. We demonstrate the scalability of our approach on industry-strength designs synthesized by a commercial tool.**

I. Introduction

With increasing hardware complexity and stringent time-to-market requirements, recent years have seen a gradual migration from register transfer level (RTL) hardware designs towards electronic system level (ESL) designs. Practicality of the ESL approach crucially depends on correctness of *behavioral synthesis*, *i.e.*, compilation from the ESL description to RTL [1]. However, many behavioral synthesis transformations are complex, depending on subtle design invariants.

Function pipelining (a.k.a. system-level pipelining) is an important transformation supported by most state-of-the-art behavioral synthesis tools. It aims to improve quality of synthesized RTL by overlapping executions of multiple successive invocations of the same function. However, it is a complex transformation and consequently error-prone. Thus, sequential equivalence checking (SEC) support for verifying correctness of the synthesized pipelines is critical for adoption of behavioral synthesis. However, function pipelining introduces overlapped execution, which leads to a significant difference between the behavioral specification and the RTL. Furthermore, typical bugs are not likely to be exposed by feeding the pipeline with one transaction: subtle corner cases typically involve the overlapped execution of multiple transactions at different pipeline stages; therefore, SEC must account for all possible input sequences as well as for arbitrary "bubbles" (pipeline stalls) between successive inputs. A brute-force SEC approach comparing input/output relations of the behavioral specification and the RTL does not scale.

We present an approach to certifying behaviorally synthesized function pipelines. We break the certification into two steps: (1) a *reference transformation*, which takes certain pipeline parameters from behavioral synthesis to generate a pipeline reference model; and (2) equivalence checking between this reference model and the RTL. The reference model accounts for arbitrary bubble insertion in the pipelines. The mapping between behavioral-level operations and RTL functional units is preserved during SEC, permitting some optimizations such as cutpoints [2]. We demonstrate our approach on industrial designs synthesized by a commercial tool.

While our work has parallels to the significant research on verification of microprocessor pipelines, there are fundamental differences. Our reference pipeline can be viewed as a generic, correct-by-construction abstraction of function pipeline; thus, instead of developing pipeline-specific reasoning techniques, our work emphasizes building an abstraction that makes standard SEC techniques practicable. We compare our approach with pipeline verification approaches in Section VI.

II. Background

A. Behavioral Synthesis

Behavioral synthesis is an automated process to transform an ESL specification to an RTL implementation. Behavioral synthesis tools generally accept synthesizable subsets of ANSI C, C++, or SystemC, and generate an RTL defined in Verilog or VHDL. Fig. 1(a) illustrates a simple ESL specification.

```
int pipe(int in1, int in2)
{
    static int a = 2;
    int b, c;
    b = a * in1;
    a = b + in2;
    c = a * b;
    return c;
}
```

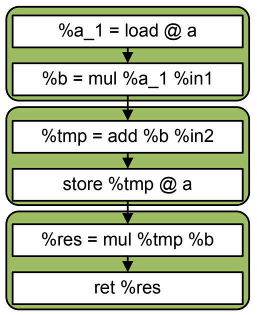

Fig. 1. (a) C Source Code of a Function. (b) Corresponding CCDFG

Behavioral synthesis flow can be divided into three phases: 1) *compiler transformation*; 2) *scheduling*; 3) *resource binding and RTL generation*. After the above three phases, the design is at a level of abstraction that can be expressed in RTL.

B. A Certification Framework

In previous work [3], [2], we introduced an SEC framework for behavioral synthesis. A formalization of design specification, called CCDFG, was defined, which is the traditional Control/Data Flow Graph (CDFG) augmented with a schedule. High-level transformations are certified offline by the verified component to preserve CCDFG semantics. The transformed CCDFG is compared with RTL through SEC via dual-rail symbolic simulation with several integrated optimizations. The framework was shown to scale to industrial designs.

C. CCDFG Formalization

Fig. 1(b) illustrates the CCDFG for the C code in Fig. 1(a). Formal semantics of CCDFG are presented in [3]; they include (1) state-based semantics for individual operations, and (2) interpretation of control and data flows and scheduling constructs. A CCDFG $G \triangleq \langle G_{CD}, M, T\rangle$, where G_{CD} is the control/data flow graph, M is a *microstep partition*, and T is a schedule. Operations are partitioned into *microsteps*; each microstep includes operations that can be executed concurrently.

For CCDFG $G \triangleq \langle G_{CD}, M, T\rangle$ and a set $t \in T$, we use the term "projection of G on t" to denote the CCDFG $G_t \triangleq \langle G'_{CD}, M', \{t\}\rangle$ where G'_{CD} and M' contain only the operations in G_{CD} and M respectively, that are members of t. For set $T_0 \subseteq T$, we use "projection of G on T_0" to denote the following graph G'. The nodes of G' are given by the set $\mathcal{N} \triangleq \{G_t : t \in T_0\}$; given $g_0, g_1 \in \mathcal{N}$, there is an edge from g_0 to g_1 if there are operations o_1 and o_2 such that $o_1 \in g_0$, $o_2 \in g_1$ and there is an edge from o_1 to o_2 in G_{CD}.

Since a schedule is a partition of microsteps, T_0 induces a partition of G_{CD} such that if $t_0 \neq t_1$ the partition induced by t_0 is disjoint from that induced by t_1. We can describe CCDFG $G \triangleq \langle G_{CD}, M, T\rangle$ uniquely as the triple $\langle S, E, M\rangle$ where S and E denote the nodes and edges of the projection of G on T, and M is the set of microstep partitions refined by T. We use this view in the reminder of this paper.

III. Function Pipelining

Function pipelining allows overlapped execution of successive function invocations. Fig. 2 compares the executions of the non-pipelined and pipelined versions of the design shown in Fig. 1. There are three operations: two *muls* and one *add*. Without function pipelining, the circuit can accept new input every three clock cycles. However, with function pipelining, it can accept new input every clock cycle: thus, function pipelining can dramatically improve the circuit throughput.

Behavioral synthesis generates handshake signals to implement the synchronization between the synthesized pipeline and its surrounding circuits [4]: these signals include *start*, *done* and *allow*. The *start* signal indicates if there are valid inputs ready for execution in the pipeline and the *allow* signal indicates if the pipeline is ready to start a new transaction in the next clock cycle. The handshake happens when both *start* and *allow* are high. The *done* signal indicates that the pipeline produced some valid output data. However, when the pipeline is ready to accept a new input (*i.e.*, when *allow* is high), the upstream circuit may not be able to get the new input data ready (*i.e.*, *start* is low); in this case, a bubble is inserted into the pipeline as shown in Fig. 2 (b). When there are bubbles in the pipeline, the pipeline typically disables the corresponding functional units to save power. Correctly disabling the idle functional unit without affecting the rest of the pipeline is challenging and error-prone. Therefore, SEC must carefully take bubbles into account. In addition to bubbles, complexity of SEC in function pipelining comes from overlapping execution of multiple transactions. It leads to a significant difference in the schedule of operations between the CCDFG of the sequential design and the RTL implementing the function pipeline. Function pipelines may have fewer scheduling steps, but each step executes more operations. Thus, standard SEC techniques are not effective.

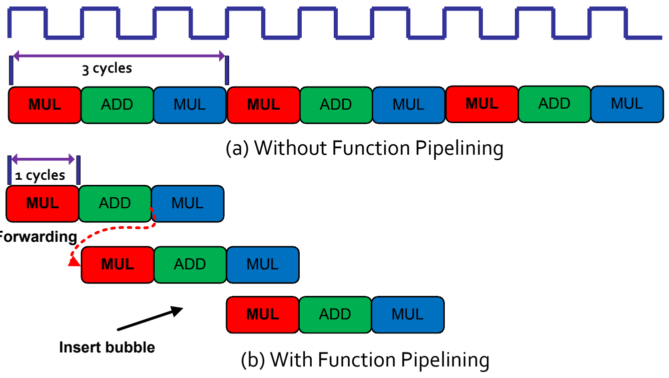

Fig. 2. Difference between Un-Pipelined Version and Pipelined Version

Comparison with loop pipelining: Our top-level approach for function pipeline verification has analogues to our previous SEC approach for loop pipelines [5], *viz.*, developing a reference model for the pipelined CCDFG that is semantically equivalent to the sequential design and can be used for SEC with the RTL. However, the reference model generation for function pipelines is inherently different from loops and involves subtle challenges not encountered in loop pipelines, leading to drastically different algorithms. A major difference between loop and function pipelines is in the logic used for handling bubbles. In loop pipelining, the pipeline interval (number of cycles between invocation of two successive loop iterations) is determined statically during synthesis time and an FSM is synthesized to control the invocation of a new iteration. However, for function pipelining, initiation of a new function transaction is governed by upstream circuit, and is therefore non-deterministic to the pipeline synthesis algorithm. Consequently, the pipelined circuit must permit arbitrary delays (and therefore, bubble) between two successive function invocations. Correspondingly, to certify an implementation of a functional pipeline, its correctness must be checked for all possible bubble insertion scenarios. A naïve approach is to build one pipelined CCDFG for each such scenario and apply SEC between all possible pipelined CCDFGs and the synthesized RTL. Unfortunately, the number of such pipelined CCDFGs is exponential to the number of scheduling steps of the CCDFG before pipelining, making this approach impractical.

IV. SEC for Function Pipelining

Our approach entails building a pipelined reference model, while still avoiding the exponential cost due to bubble insertion discussed above. Our function pipelining transformation takes a CCDFG before pipelining G and certain pipeline parameters to generate a pipelined CCDFG G'. Checking the equivalence between CCDFG G and RTL is equivalently translated to equivalence checking between pipelined CCDFG G' and RTL. CCDFG G' allows operations to execute concurrently, closely corresponding with the RTL through careful modeling of bubble insertion. Thus, we can leverage the existing SEC approach to check CCDFG G' and RTL.

In this paper, we focus on the pipelines which satisfy the following requirements: (1) all sub-functions have been fully inlined; (2) all loops have been fully unrolled; (3) there is no global variable (other than static variables). Our framework actually supports loops and sub-functions by extending the approach discussed here with compositional reasoning; we do not discuss that extension in the paper due to space limitation. Global variables can be avoided by explicitly rewriting them as static variables together with corresponding interfaces.

A. *Algorithm to Build Reference Model*

As a pedagogical simplification, assume there is no branch among scheduling steps, but allow branches inside scheduling steps. Note that if CCDFG G has branches, we can merge the destination scheduling steps into one single scheduling step; thus the branch between the scheduling steps is equivalently converted into a branch inside the merged scheduling step. With this simplification, we can view CCDFG G as a sequence of scheduling steps from the function entry to the exit.

Task Interval is an important metric to measure the performance of function pipelines: it is the number of clock cycles that must elapse between two transactions. We can partition CCDFG G into a sequence of sub-CCDFGs according to task interval I. Each sub-CCDFG is called a pipeline unit, which is defined in Definition 1. All scheduling steps within one pipeline unit execute sequentially, and different pipeline units can execute concurrently. In the example shown in Fig. 2 (b), because the pipeline can start a new transaction every clock cycle, each scheduling step is a pipeline unit.

Definition 1 (Pipeline Unit): Given a pipeline task interval I and a CCDFG $G \triangleq \langle G_{CD}, M, T\rangle$, T can be partitioned into a set of sub-schedule $\{T_0, T_1, \ldots, T_n\}$. Each T_i takes I clock cycles (except possibly the last partitioned schedule T_n which may be less than I). Therefore, G can be partitioned into a set of sub-CCDFGs $\{G_0, G_1, \ldots, G_n\}$, respectively. $G_i \triangleq \langle G_{CD}, M, T_i\rangle$ is called a pipeline unit.

Algorithm 1 BuildPipeline($G = \langle S, E, M\rangle$, I, N)

```
1: $\langle S'_1, M'_1\rangle \leftarrow GeneratePipelineRegs(S, E, M, I)$
2: $S'_2 \leftarrow GenerateSchedulingSteps(S'_1, I, N)$
3: $E'_1 \leftarrow GenerateEdges(S'_2, I)$
4: $\langle S'_3, M'_2\rangle \leftarrow GenerateGuards(S'_2, E'_1, M'_1)$
5: $\langle S'_4, M'_3\rangle \leftarrow GenerateDataForwarding(S'_3, E'_1, M'_2, I)$
6: return $G' = \langle S'_4, E'_1, M'_3\rangle$
```

Algorithm 1 shows the sequence of high-level steps involved in generating a pipelining reference model. It takes CCDFG G, task interval I, and the number of scheduling steps N. It involves five steps, *viz.*, (1) inserting pipeline registers, (2) constructing new scheduling steps, (3) generating

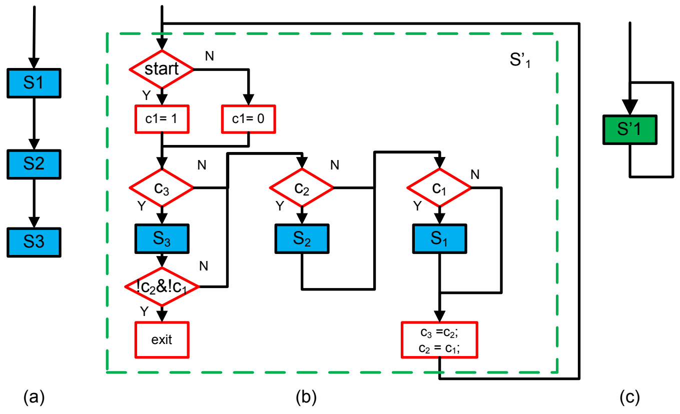

Fig. 3. Input and Output CCDFGs of Function Pipelining Transformation. The sequential CCDFG is on the left, and the pipelined CCDFG is on the right, which has a single scheduling step $S1'$. The middle figure shows how $S1'$ relates to the scheduling steps of the sequential CCDFG.

new control edges, (4) restricting control and data flow through guard variables, and (5) implementing data forwarding. We describe these steps in detail below. Fig. 3 illustrates the result of applying these steps to our simple example in Figure 1. The CCDFG on the left is the one before pipelining, the CCDFG on the right is the generated pipelined reference model, and the figure in the middle shows how the scheduling steps of the pipeline correspond to those in the original CCDFG.

1) Inserting Pipeline Registers: Since the pipeline can overlap transactions of more than one function invocation, it may need extra registers to store the intermediate values to prevent variables from being overwritten; this is achieved by pipeline registers. The basic idea of this step is to insert variables to mimic pipeline registers. For each variable v, we check the necessity by comparing the life time l_v with I. The life time of a variable is the distance between its producer and the last consumer. If l_v is greater than I, pipeline registers for v is required, otherwise, not necessary. The number of pipeline registers required is determined by l_v and I.

2) Constructing Scheduling Steps: In the pipelined CCDFG G', a scheduling step s' consists of multiple scheduling steps of CCDFG G. All steps in s' can complete within one clock cycle. A key step for constructing s' is to correctly group scheduling steps from G. The grouping result, according to the pipeline parameters provided by behavioral synthesis, must match the behavior of the synthesized pipeline. To achieve this, for the ith scheduling step s' in G', we collect the ith scheduling step from all pipeline units of G. Then s' maintains the following invariants. (1) Let α and β be any two scheduling steps in G collected to execute in s'; then α and β must belong to different pipeline units. (2) Every pipeline unit (except possibly the last) must have some scheduling step in s'.

Algorithm 2 constructs the pipelined scheduling steps. Subroutine *getPipeUnits* returns the pipeline units in the design. Lines 5-9 collect the scheduling steps from the pipeline units. We then generate the control/data edges between those steps for the new scheduling step s' as shown in line 13-18. The edge is from left to right, because the scheduling steps in left are running the transactions entered the pipeline early. Subroutine *buildEdge* creates the edge between two

Algorithm 2 GenerateSchedulingSteps (S, I, N)

```
1: $P \leftarrow getPipeUnits();\quad S' \leftarrow \emptyset$
2: /*collect scheduling steps from pipeline units*/
3: for each $i$ in $I$ do
4:     $s'_i \leftarrow \emptyset$
5:     for each $p$ in $P$ do
6:         if $length(p) \geq i$ then
7:             $s'_i \leftarrow s'_i \cup p[i]$
8:         end if
9:     end for
10:    $S' \leftarrow S' \cup s'_i$
11: end for
12: /*build new edges within one single scheduling step */
13: for each step $s'$ in $S'$ do
14:     for each consecutive step pair $(s'[k], s'[k+1])$ in $s'$ do
15:         $e' \leftarrow buildEdge(s'[k], s'[k+1])$
16:         $s' \leftarrow appendEdge(s', e')$
17:     end for
18: end for
19: return $S'$
```

scheduling steps and subroutine $appendEdge$ appends the edge to them in s'. Fig. 4 show the pipelined scheduling steps for our running example. Here I equals to one; therefore there is only one scheduling step in G' and this scheduling step consists of all the scheduling steps before pipelining.

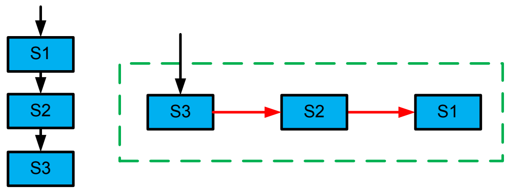

Fig. 4. Construction of Scheduling Steps and Edges

3) Building Edges: Algorithm 3 shows the construction of edges governing the control flow of the pipelined CCDFG. Lines 3-6 show the construction of edges between scheduling steps of the pipelined CCDFG G'. Besides, a back edge is generated from the last scheduling step to the first scheduling step. The pipelined CCDFG G' is formed as a loop. Fig. 3(b) shows the edges between scheduling steps in CCDFG G'.

Algorithm 3 GenerateEdges (S, I)

```
1: $E' \leftarrow \emptyset$
2: /*build the edges between new scheduling steps*/
3: for each consecutive step pair$(S[i], S[i+1])$ in $S$ do
4:     $e' \leftarrow buildEdge(S[i], S[i+1])$
5:     $E' \leftarrow E' \cup e'$
6: end for
7: /*build the back edge*/
8: $s_{src} \leftarrow S[I-1];\ s_{dst} \leftarrow S[0]$
9: $e_{backedge} \leftarrow buildEdge(s_{src}, s_{dst})$
10: $E' \leftarrow E' \cup e_{backedge}$
11: return $E'$
```

4) Generating Guard Variables: G' must be implemented as a loop since it has to initiate an arbitrary number of function invocations as determined by the upstream logic. Guard variables guarantee that the execution of this loop corresponding to each function invocation follows the control flow of the original CCDFG G and terminates properly. Algorithm 4 describes details of guard variable insertion, and Fig. 3(b) illustrates it with our simple example. First, subroutine $createGuardVariable$ creates guard variables $c_1, c_2, \ldots, c_n$ for all pipeline units. For each scheduling step s in CCDFG G, $insertGuard$ inserts a branch operation before entering it: if the guard variable is $true$, this scheduling step is enabled; otherwise it is skipped. After executing one pipeline unit, we propagate the value of the guard variable to its successor in the sequence. Recall that the pipelined CCDFG G' is a loop; one loop iteration executes all pipeline units. The assignment of the first guard variable c_1 depends on the *start* signal. Values of guard variables are propagated immediately before the back edge. The assignment and propagation of the guard variables are implemented by $genVarAssign$. We need to specially handle the exit of the pipelined CCDFG. It can only exit when c_n is $true$ and $c_1, \ldots, c_{n-1}$ are all $false$, which indicates the pipeline only has one last transaction running and this transaction is going to exit. In the pipelined CCDFG, we refine the semantics of the ret operation (function return). Operation ret defines the end of one transaction instead of the whole CCDFG, and generates an output (if not, returning void). We introduce a new operation $exit$ to denote the termination of the pipelined CCDFG, which is gated by guarded variables. Subroutine $insertExitOp$ inserts this gated $exit$ operation.

Algorithm 4 GenerateGuards (S, E, M)

```
1: $S' \leftarrow S;\quad M' \leftarrow M$
2: $P = getPipeUnits()$
3: $C = createGuardVariable(P)$
4: /*generate guard condition for each scheduling step*/
5: for each pipeline unit $p$ in $P$ do
6:     for each scheduling step $s$ in $p$ do
7:         $c \leftarrow getGuardVar(p)$
8:         $\langle S', M' \rangle = insertGuard(s, p, C, S', E, M')$
9:     end for
10: end for
11: /*generate assignments for guard variables*/
12: $\langle S', M' \rangle \leftarrow genVarAssign(C, S', E, M')$
13: /*insert exit operation*/
14: $S' \leftarrow insertExitOp(C, S', E)$
15: return $\langle S', M' \rangle$
```

We now discuss how the guard variables maintain separation of function invocations across different iterations. If *start* is present, the first guard variable c_1 is assigned $true$ when entering the loop in G'. During the execution of the first iteration of the loop body, only scheduling steps guarded by c_1 are enabled, and other steps are skipped. In the example shown in Fig. 3(b) s_1 is enabled and s_2 and s_3 are disabled in the first iteration. At the end of the first iteration, guard variable c_2 receives the enable token propagated from c_1, c_3 remain $false$. In the second iteration, c_1 still remains $true$, because there is a second start request. Thus s_1 and s_2 are enabled in the second iteration. The process continues until all guard variables $c_1, \ldots, c_3$ are $true$, the pipeline enters the *full* stage. In the full stage, when a new transaction is started, one early transaction

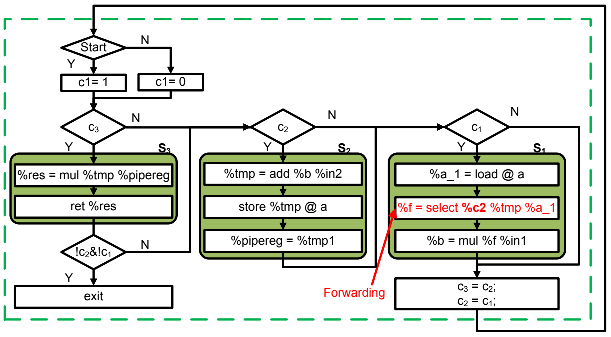

Fig. 5. Final Pipelined CCDFG

finishes at the same time. When the pipeline is in full stage and there is no *start* signal any more, the pipeline starts to *flush*. Guard variable c_1 is assigned to $false$ (since there is no start). Thus, s_2 and s_3 are enabled and s_1 is disabled. The disable token is propagated between guard variables at every loop iteration. If all guard variables are $false$, except the last one c_3, the pipelined CCDFG finishes the execution by executing $exit$ operation. We can easily insert bubbles into pipelines by toggling the *start* signal.

5) Implementing Pipeline Forwarding: In function pipelines, dependencies between transactions are introduced by static variables and forwarding can be implemented by mapping static variables to hardware registers which form feedback paths. In CCDFG, operations to fetch or store data to registers are represented by *load* and *store*, respectively; thus all pairs of operations requiring forwarding can be found by inspecting *load* and *store* pairs. However, for performance, behavioral synthesis may generate a combinational path to forward data directly. In Fig. 2(b), the output of *adder* is forwarded to the next transaction's multiplier without passing through the register; otherwise, the RTL cannot accept new transaction at every clock cycle. To mimic this combinational path, we check that there is a valid datapath from the forwarding source to the destination in the pipelined CCDFG. Absence of such a path is an indication of hazard.

However, the forwarding path may vary depending on bubbles. If the *adder* is disabled due to bubbles, the data forwarded to the *multiplier* by the combinational path is invalid. To handle this, we determine the forwarding path by checking whether the source operation is enabled. This check can be done through guard variables, and we insert a *select* operation to implement the check, as shown in Fig. 5 in the forwarding between the $adder$ and $multiplier$.

B. SEC between CCDFGs and RTL

Recall from Section III that handling bubbles is a major hurdle for certifying function pipelines. Bubbles in function pipelines affect their behaviors: (1) the idle operations are disabled; (2) the pipeline forwarding has different paths. These two are modeled in the pipelined CCDFGs by introducing guard variables. Recall however, that in order to fully check the behaviors of the pipelines with bubbles, we need to run SEC on all input sequence combinations.

To address this problem, we utilize guard variables to encode all possible input combinations; thus the SEC needs to run only once. A new transaction can start at an arbitrary state, with the pipeline full, empty, or containing bubbles. We model the pipeline at different states by toggling guard variables. For instance, the execution shown in Figure 2(b) can be modeled as assigning $c_1 = true$, $c_2 = true$, and $c_3 = false$.

Our SEC then has the following three steps:

- Set the pipelined CCDFG to a symbolic state which starts a new transaction. Setting the pipeline state is done by encoding the guard variables. We set c_1 to $true$ and assign symbols to $c_2, \ldots, c_n$.
- Set the FSM in the RTL implementation to the same symbolic state as the CCDFG. The structure of the F-SM can be obtained from behavioral synthesis reports; we analyze the reports to determine the corresponding symbolic states for the CCDFG and RTL.
- Feed the same input symbolic data set to the CCD-FG and RTL, then run dual-rail symbolic simulation between them for one single transaction. The proof obligation is that the output of pipelined CCDFG and that of the RTL implementation are equivalent.

SEC checks whether the equivalence is preserved after executing one transaction by the CCDFG and RTL, when initiated from equivalent states. No special-purpose checking is required during SEC. Furthermore, since the mapping between the CCDFG operations and the RTL functional units is maintained, we can apply the *cutpoint* optimization to improve the scalability.

V. Experimental Results

We have implemented this approach on top of our existing certification framework [2]. SEC is implemented by a cycle-by-cycle dual-rail word-level symbolic simulation. The resulting tool was applied to a suite of designs synthesized by a commercial behavioral synthesis tool. The designs were selected from several different application domains, *e.g.*, *TEA* and *XTEA* are cryptographic algorithms with complex bitwise operations, and *FIR* is a signal processing application with an internal feedback path that is optimized by synthesis via data forwarding. Design sizes reflect typical industrial targets for pipeline synthesis; many involve several thousand lines of RTL. We used a workstation with 3GHz Intel Xeon processor and 2GB memory, and a running time bound of two hours.

Table I shows our experimental results. We first conducted brute-force SEC between the non-pipelined CCDFG and the RTL; none of the runs terminated within the time bound. We then conducted brute-force SEC between the pipelined CCDFG and the RTL; only the run on *FIR* finished. With *cutpoint* optimization, SEC succeeded on all designs with modest time and memory usages. Column *Cuts* shows the number of cutpoints identified for each design. The experiments demonstrate our approach of generating reference pipeline is viable since it preserves the internal mapping between CCDFG and RTL, enabling cutpoints. Note that the examples include pipelines with more than 40 stages; thus, a naïve implementation requiring exponential number of SECs as discussed in Section III would be impractical.

TABLE I. EXPERIMENTAL RESULTS ON EQUIVALENCE CHECKING FOR FUNCTION PIPELINING

Design	RTL #line	App. Domain	Func Info.			Pipeline Info.		Without Opt.		With Opt.		
			Interval	Depth	Operations	Forwarding	Pipeline Register	Mem. (MB)	Time (Sec)	Mem. (MB)	Time (Sec)	#Cuts
FIR	430	Signal processing	1	5	21	1	5	43	34.8	31	11.5	13
DCT	941	Signal processing	1	4	48	0	1	-	-	135	26.37	32
CORDIC	1450	Data processing	1	12	170	0	10	-	-	221	38.83	73
XTEA	1777	Cryptography	1	32	192	0	147	-	-	114	30.57	32
TEA	2325	Cryptography	1	43	192	0	211	-	-	100	40.39	85
YUVTORGB	2412	Image processing	1	5	96	0	4	-	-	333	251.62	48
MemoryOp	4106	Memory operation	2	39	96	1	75	-	-	43	89.53	75

VI. RELATED WORK

There has been extensive research on verifying pipelined microprocessors [6], [7], [8] with strong parallels to our work. However, there are significant differences in goals and techniques. Microprocessor pipelines include optimized (handcrafted) control and forwarding logics, but a static set of operations based on the instruction set. Function pipelines are for dynamically overlapping executions of arbitrary transaction sequences generated by compiling ESL functions; they tend to be deep with a high complexity at each stage, but control and forwarding logics are more standardized since they are automatically synthesized. It is hard to adopt SEC techniques for microprocessor pipelines directly for function pipelining, *e.g.*, lack of a standardized instruction set makes it difficult to identify targets for uninterpreted functions. Finally a critical difference between our approach and microprocessor verification is in the decomposition strategy. Microprocessor verification techniques directly compare a pipelined implementation with a sequential (ISA) abstraction; our approach targets correct-by-construction, generic abstraction *of the pipeline* (*viz.*, reference model); thus the SEC step comparing the reference model with RTL can be oblivious to pipelines, and pipeline-independent optimizations (*e.g.*, cutpoints) are smoothly reused which are critical to the scalability of our tool.

Koelbl *et al.* [9] provide a tutorial introduction on methods of formal equivalence checking between system-level models and RTL. Kundu *et al.* [10] presents an approach to validate the result of behavioral synthesis using insights from translation validation. Chauhan *et al.* [11] propose a technique for SEC between non-cycle-accurate designs by constructing a pair of normalized cycle-accurate designs from the original designs. However, neither approach provides SEC for function pipelining that effectively integrates with behavioral synthesis flows. Commercial SEC frameworks [11], [12] handle function pipelining. However, to our knowledge, current implementations either involve cost-prohibitive input-output comparison or require the user to provide the requisite mappings.

VII. CONCLUSIONS AND FUTURE WORK

We have presented an approach to equivalence checking of function pipelines generated by behavioral synthesis. Central to this approach is the construction of a reference CCDFG for a function pipeline. The reference CCDFG preserves the operation mapping to the RTL implementation, enabling SEC optimizations such as cutpoint, and it carefully models bubble insertion, avoiding construction of CCDFGs for different bubble insertion scenarios. Experimental results show the viability of our approach in practice.

Of course, a problem with our dependence on reference model is that the synthesis tool can generate a pipeline through advanced heuristics, but our reference implementation does not; in such case our approach will report a spurious inequivalence. While this is possible in theory, our experience suggests that our model is adequate in practice.

A planned future work is a mechanized correctness proof of reference pipeline generation, which is necessary for the completeness of certification. We have an informal correctness proof and our goal is to mechanize it in a theorem prover.

VIII. ACKNOWLEDGMENTS

This research was partially supported by National Science Foundation Grants #CCF-0916772 and #CCF-0917188 and by a research grant from Intel Corporation. We thank Yatin Hoskote, Naren Narasimhan, and Jin Yang for their advice and help.

REFERENCES

[1] Y.-L. Lin, "Recent developments in high-level synthesis," *ACM Transactions on Design Automation of Electronic Systems*, vol. 2, no. 1, 1997.

[2] K. Hao, F. Xie, S. Ray, and J. Yang, "Optimizing equivalence checking for behavioral synthesis," in *Proc. of DATE*, 2010, pp. 1500–1505.

[3] S. Ray, K. Hao, Y. Chen, F. Xie, and J. Yang, "Formal verification for high-assurance behavioral synthesis," in *Proceedings of ATVA*, 2009.

[4] P. R. Schaumont, *A Practical Introduction to Hardware/Software Codesign*. Springer, 2010.

[5] K. Hao, S. Ray, and F. Xie, "Equivalence checking for behaviorally synthesized pipelines," in *Proc. of DAC*, 2012, pp. 344–349.

[6] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *Proc. of CAV*, 1994, pp. 68–80.

[7] J. Levitt and K. Olukotun, "A scalable formal verification methodology for pipelined microprocessors," in *Proc. of DAC*, 1996, pp. 558–563.

[8] M. N. Velev and R. E. Bryant, "TLSim and EVC: a term-level symbolic simulator and an efficient decision procedure for the logic of equality with uninterpreted functions and memories," *IJES*, pp. 134–149, 2005.

[9] A. Koelbl, Y. Lu, and A. Mathur, "Formal Equivalence Checking between System-level Models and RTL," in *Proc. of ICCAD*, 2005, pp. 965–971.

[10] S. Kundu, S. Lerner, and R. Gupta, "Validating high-level synthesis," in *Proc. of CAV*, 2008, pp. 459–472.

[11] P. Chauhan, D. Goyal, G. Hasteer, A. Mathur, and N. Sharma, "Non-cycle-accurate sequential equivalence checking," in *Proc. of DAC*, 2009, pp. 460–465.

[12] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley, "Solver technology for system-level to rtl equivalence checking," in *Proc. of DATE*, 2009, pp. 196–201.