Equivalence Checking for Behaviorally Synthesized Pipelines

Kecheng Hao
Dept. of Computer Science
Portland State University
kecheng@cs.pdx.edu

Sandip Ray
Dept. of Computer Sciences
University of Texas at Austin
sandip@cs.utexas.edu

Fei Xie
Dept. of Computer Science
Portland State University
xie@cs.pdx.edu

ABSTRACT

Loop pipelining is a critical transformation in behavioral synthesis. It is crucial to producing hardware designs with acceptable latency and throughput. However, it is a complex transformation involving aggressive scheduling strategies for high throughput and careful control generation to eliminate hazards. We present an equivalence checking approach for certifying synthesized hardware designs in the presence of pipelining transformations. Our approach works by (1) constructing a provably correct pipeline reference model from sequential specification, and (2) applying sequential equivalence checking between this reference model and synthesized RTL. We demonstrate the scalability of our approach on several synthesized designs from a commercial synthesis tool.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*automatic synthesis, optimization, verification*

General Terms

Algorithms, Performance, Reliability, Verification

Keywords

Equivalence checking, behavioral synthesis, pipeline

1. INTRODUCTION

As hardware complexity increases, it is getting increasingly difficult to develop a high-quality hardware system via hand-crafted RTL. Electronic System Level (ESL) designs provide a promising solution to this problem, by facilitating more abstract design description (*e.g.*, with SystemC). The adoption of this approach, however, is crucially dependent on the correctness of *behavioral synthesis* [13, 16, 11, 5, 3], *viz.*, the compilation of an ESL description to RTL.

A critical transformation in behavioral synthesis is loop pipelining, producing temporal overlap of successive loop iterations. It is available in most state-of-the-art tools (*e.g.*, AutoESL [17], CatapultC [12], and Cynthesizer [4]) and is crucial to the synthesis of high-throughput hardware. However, it induces retiming and out-of-order executions; furthermore, the mapping of internal operations is lost between the sequential description and the pipelined RTL. This rules out standard sequential equivalence checking (SEC) techniques for their comparison. In particular, some key optimizations (*e.g.*, cutpoints) become inapplicable.

We present an SEC framework for certifying synthesized designs with pipelined loops. We have applied our tool on industrial-size designs with thousands of lines of RTL, synthesized by AutoESL. This scalability is derived from tight integration with the synthesis flow. Instead of directly comparing the synthesized RTL with the sequential description, we develop an intermediate *pipeline reference model*. This model provably preserves the semantics of the sequential description. However, our model generation algorithm is parameterized by pipeline parameters, whose values are obtained from the synthesis tool; this ensures that the structure of the generated model is similar to that of the synthesized RTL, and enables internal operation mapping between the reference model and the RTL.

The rest of this paper is organized as follows. Section 2 provides relevant background. Sections 3 and 4 present our approach. We present experimental results in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2. BACKGROUND

2.1 Behavioral Synthesis

A behavioral synthesis tool applies a sequence of transformations to an ESL specification to transform it into RTL. Figure 1(a) illustrates an ESL description with a simple loop structure and Figure 1(b) shows the schematics of the pipelined RTL design synthesized by AutoESL. The following transformation phases are involved in this synthesis.

- First, *compiler transformations* are applied to the ESL description. For instance, constant propagation is used in the example to remove unnecessary variables.
- The second phase is *scheduling*, which optimizes the clock cycle for each operation. In this phase, operations are chained across conditional blocks and decomposed into smaller multi-cycle sub-operations. Loop pipelining is employed as part of this phase.
- The third phase is *resource binding and control synthesis*. This phase binds operations to hardware units,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2012, June 3-7, 2012, San Francisco, California, USA.
Copyright 2012 ACM 978-1-4503-1199-1/12/06 ...$10.00.

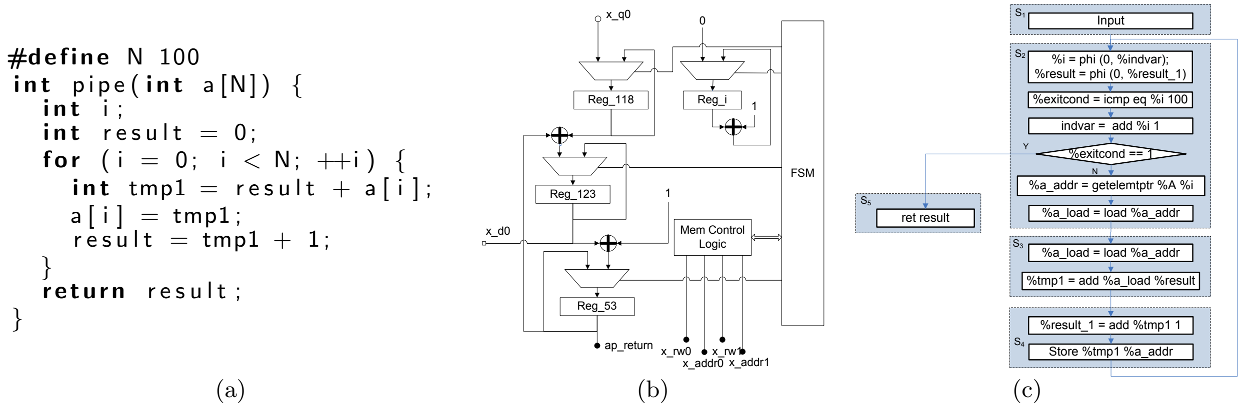

Figure 1: (a) C Code with Loop Design. (b) Schema of RTL Synthesized by AutoESL. (c) CCDFG

and allocates registers. For instance, the "+" operation is bound to a hardware adder. Furthermore, a control circuit is generated (typically as a finite-state machine module) to implement the schedule.

After the transformations, the design is expressed as RTL.

2.2 A Certification Framework

In [14], we proposed a verified/verifying framework for certifying behaviorally synthesized RTL. The key idea was to compare the RTL with the design representation after the high-level (compiler and scheduling) transformations have been applied to the ESL description. To achieve this, the framework introduced CCDFG, a formalization of design specification that augments the traditional Control/Data Flow Graph (CDFG) with a schedule. High-level transformations can be certified offline (by theorem proving) to preserve CCDFG semantics. The transformed CCDFG is compared with RTL through SEC via dual-rail symbolic simulation. In [6], we also developed three optimizations to optimize SEC performance. The resulting framework was scalable to industrial-size designs.

However, this previous work ignored pipelining. In particular, the optimizations were critically dependent on ready availability of mapping information for internal operations between the CCDFG and the RTL, which was determined from the resource bindings performed by the synthesis tool; pipelining destroys this ready correspondence, making direct mapping inapplicable.

2.3 CCDFG Formalization

Figure 1(c) illustrates the CCDFG corresponding to the C code shown in Figure 1(a). Details of the formal semantics of CCDFG are presented in previous paper [14]. Formally, 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. The formalization assumes that the underlying language provides the semantics for *primitive operations* (*e.g.*, arithmetic operations, comparison, etc.). The operations are partitioned into *microsteps* that stipulate the operations that can be executed concurrently. Finally, microsteps are grouped into a schedule which specifies the microsteps that are completed within a single clock cycle. Following standard conventions, the control flow is broken up into basic blocks; data dependencies follow the "read after write" paradigm: op_j is dependent on op_i if op_j occurs after op_i in a control path and computes an expression over some variable v that is assigned most recently by op_i in the path. A CCDFG execution is formalized by a state-based semantics. A *CCDFG state* (resp., *CCDFG input*) is a valuation of the state (resp., input) variables. Given a sequence of inputs, an *execution* of a CCDFG G with microstep partition M and schedule T is a sequence of CCDFG states that corresponds to an evaluation of the microsteps of M respecting T.

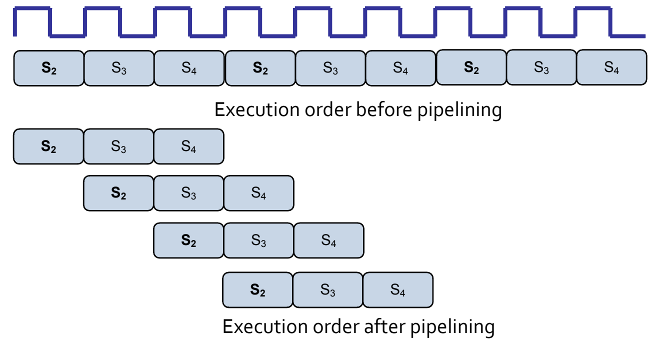

Figure 2: Execution Orders Before and After Pipelining. The rounded boxes S_2, S_3, and S_4 are scheduling steps in the sequential design.

3. CHALLENGES WITH LOOP PIPELINES

Loop pipelining allows multiple successive iterations of a loop to operate in parallel by executing a new iteration before the previous iteration completes. Consider pipelining the loop in Figure 1(a). Figure 2 shows the execution orders of the scheduling steps in the loop body before and after pipelining. In the sequential design, execution of iteration i involves reading the value of $a[i]$ from the memory in S_2, adding i and $a[i]$ in S_3, and storing new value to the memory and computation of *result* in S_4. However, with pipelining, iteration $i+1$ is initiated before iteration i completes.

The result of overlapping executions is a significant difference in the schedule of operations between the CCDFG of the sequential design and the RTL generated from the pipeline. Each scheduling step of the pipeline is composed of

a number of scheduling steps of the sequential design; there is no longer a direct operation mapping between the CCDFG and RTL. Furthermore, due to the difference in the execution order of the scheduling steps, the controlling finite-state machines are also different. A direct SEC between the two reduces to comparison of their input-output relations, which is prohibitively expensive for loops with many iterations.

4. SEC WITH REFERENCE MODEL

Our solution to the above problem is to develop a *reference pipelining transformation* on CCDFGs. Given a CCDFG G and certain pipeline parameters (see below), we generate a new CCDFG G' by pipelining the loops. Note that our transformation is different from that used by the synthesis tool to generate the pipelined RTL. The synthesis tool transformation includes algorithms and heuristics to *determine* how many iterations to pipeline, when to introduce stalls and bubbles, etc.; on the other hand, our algorithm merely *takes* such information as parameters to create G'. In fact, we obtain this information from the synthesis tool itself. Thus the output CCDFG G', if successfully generated by our algorithm,[1] is guaranteed to have close structural correspondence with the synthesized RTL. On the other hand, irrespective of the actual value of these parameters, G' is guaranteed to be semantically equivalent to G and can therefore be soundly used instead of G for SEC.

The following definition characterizes the loops handled by the algorithm.

REMARK 1 (CONVENTIONS). *For a given 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 mean 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 a 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}*.*

DEFINITION 1 (PIPELINABLE LOOP). *For a CCDFG* $G \triangleq \langle G_{CD}, M, T\rangle$ *and for* $T_0 \subseteq T$*, we say that* T_0 *induces a "pipelinable loop" if (1) the projection of* G *on* T_0 *is a cycle* C*, and (2) in the projection of* G *on* T *there is a unique node (called the "entry node") in* C *with a predecessor outside* C *and a unique node (called the "exit node") in* C *with a successor outside* C*.*

REMARK 2. *The notion of pipelinable loops is more restrictive than the common loop definition in programming languages. In particular, a pipelinable loop has a single exit and loop nesting is disallowed. Our definition is based on the kind of loops that can be pipelined during behavioral synthesis. For instance, if a design contains nested loops, then the inner loop can be unrolled completely (possibly by compiler transformations) before the outer loop can be pipelined.*

[1]Our algorithm does not use semantic invariants of the program being transformed. Thus we may fail to pipeline a loop for a given number of iterations (and report spurious hazard) when in fact such a pipeline is hazard-free. However, in practice we have not seen a case where the synthesis tool generates a pipeline with specific parameters and our algorithm reports a spurious hazard on those parameters.

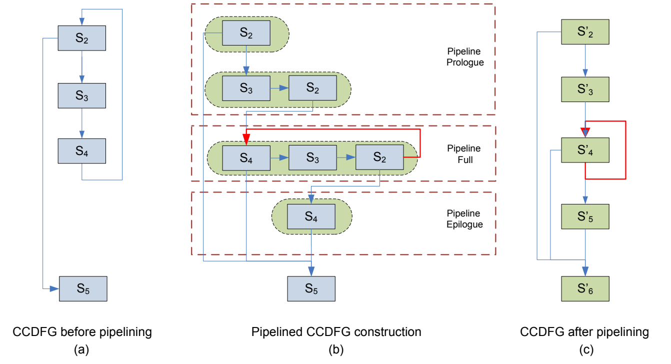

Figure 3: Input and Output CCDFGs of Loop Pipelining Transformation

REMARK 3. *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*. Given a set* T *of scheduling steps, one can describe the 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 rest of the paper for pipelinable loops.*

Given CCDFG G, our reference transformation replaces each loop L in G with the pipelined refinement of L as described in Algorithm 1. Here I is iteration interval, which indicates how many clock cycles later a new iteration is to be "fed" into the pipeline, and N is the number of scheduling steps in L. Values of these parameters are readily available from AutoESL.

Algorithm 1 PIPELINELOOP($L = \langle S, E, M\rangle$, I, N)

1: $S'_1 \leftarrow GenerateSchedulingSteps(S, I, N)$
2: $\langle S'_2, M'_1\rangle \leftarrow GeneratePipelineRegs(S'_1, M, E, I)$
3: $E'_1 \leftarrow GenerateEdges(S'_2, E, I, N)$
4: $\langle S'_3, M'_2\rangle \leftarrow GenerateForwarding(S'_2, M'_1, E'_1, I)$
5: **return** $\langle S'_3, E'_1, M'_2\rangle$

Figure 3 illustrates the use of the algorithm on our simple example. We now discuss the different steps of the algorithm in greater detail.

Algorithm 2 GenerateSchedulingSteps (S, I, N)

1: $S_G \leftarrow \emptyset$;
2: $iter \leftarrow 0$ /*loop iteration*/
3: **while** $iter * I < N$ **do**
4: $\quad S_G \leftarrow mergeIteration(S_G, S, I, iter)$
5: $\quad iter \leftarrow iter + 1$
6: **end while**
7:
8: /*build new edges within one single scheduling step */
9: **for** each step s' in S_G **do**
10: $\quad$ **for** each step pair $(s'[pos], s'[pos+1])$ in s' **do**
11: $\quad\quad e' \leftarrow buildEdge(s'[pos], s'[pos+1])$
12: $\quad\quad s' \leftarrow append(s', e')$
13: $\quad$ **end for**
14: **end for**
15: **return** S_G

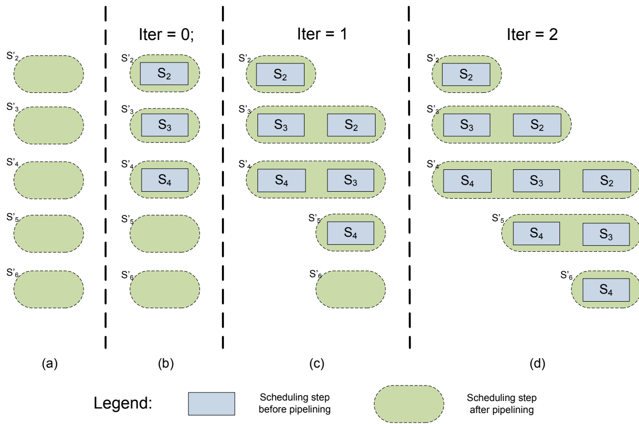

Figure 4: Construction of Scheduling Steps

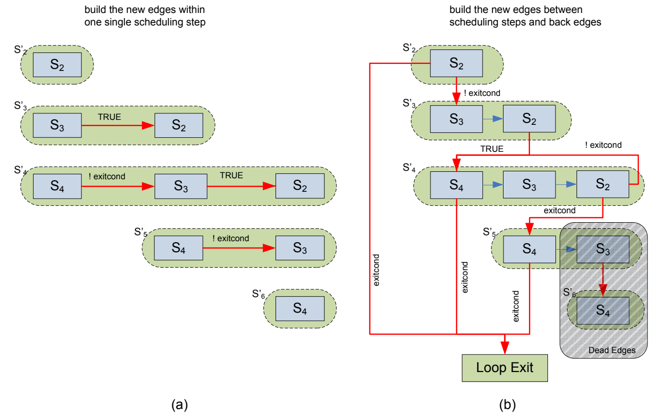

Figure 5: Construction of Edges

Algorithm 2 describes the construction of scheduling steps of the pipelined CCDFG. The algorithm simulates the process of "feeding" a new loop iteration into the pipeline until the pipeline is full. Consider the sequence of iterations shown in Figure 4. The output is an array (initially empty) of graphs. Each graph represents the projection of the reference pipeline CCDFG at a single scheduling step. We first build the nodes of each graph in the array (Lines 3-6); we then compute the edges within each graph (Lines 8-14). The set of nodes of each graph in S_G is determined by I and N. The algorithm updates S_G for every iteration. If the pipeline is not yet full, *i.e.*, can accept a new iteration but no iteration is completed yet (Line 3), then a new iteration is introduced and merged with the existing iterations in the pipeline by subroutine $mergeIteration$. Subroutine $mergeIteration$ merges each scheduling step in the new iteration with the corresponding steps already in pipeline, returns new scheduling steps as shown in Figure 4(b), (c), (d). To model the exit, the pipeline enters the "flushing" stage in which iterations are completed without new iteration being introduced. The pipeline full stage corresponds to the new loop body for the pipelined CCDFG while the prologue and epilogue correspond to the entry and exit.

We now build the edges for each graph in S_G. The goal is to ensure that the new control flow respects that of the input loop. The process is demonstrated in Figure 5 (a). Recall that a scheduling step of the pipeline involves a number of scheduling steps of the original CCDFG (across several iterations). To ensure that the original control flow is respected, a scheduling step s' of the pipeline is executed following the iteration order. This is achieved by adding edges enforcing the evaluation of microsteps from left to right. For instance, in S'_4 shown in Figure 5 (a), an edge is created to connect S_4 and S_3. Since S_4 is from an earlier iteration, the direction is from S_4 to S_3. The edge condition $!exitcond$ states that loop does not exit. If the loop exits at iteration i, all iterations from $(i+1)$ must be skipped: this is ensured by inserting the exit condition on all such edges. Subroutine $buildEdge$ creates the correct edge condition according to the control flow.

Algorithm 3 inserts "pipeline registers" between iterations to facilitate correct data flow and prevent variables from being overwritten before being consumed. In a CCDFG, the effect of pipeline registers is mimicked using temporary variables as follows. We first compute all program variables that may be overwritten before being consumed(Line 2); this constitutes the variables that potentially require pipeline registers. To find such variables, we compare the distance between the producer ms_p and the last consumer ms_c; if the distance is greater than I, v is assigned the new data value of the next iteration before current iteration's value has been fully consumed; this warrants insertion of pipeline variables in every scheduling step between ms_p and ms_c. The value is propagated every clock cycle following the CCDFG data flow. In Figure 6, variable $\%a_addr$ is computed in S_2 and the last use scheduling step is S_4. The distance is greater than $I = 1$, therefore, temporary variables a_addr_pipe1 and a_addr_pipe2 are inserted. Subroutine $addPipelineReg$ generates new microsteps for assignments of the pipeline variables, create new edges to integrate these microsteps into the data path, and updates the schedule.

Algorithm 3 GeneratePipelineRegs (S, M, E, I)

1: $S' \leftarrow S; M' \leftarrow M$
2: $V_{pr} \leftarrow getPipelineRegisterVars(S, M, E, I)$
3: **for** each variable v in V_{pr} **do**
4: $\quad ms_p \leftarrow getProducer(v)$
5: $\quad ms_c \leftarrow getLastComsumer(v)$
6: $\quad \langle S', M' \rangle \leftarrow addPipelineReg(S', M', E, ms_p, ms_c)$
7: **end for**
8: **return** $\langle S', M' \rangle$

Algorithm 4 shows the construction of edges governing the control flow of the pipelined CCDFG. Figure 5 (b) shows how to build edges between new scheduling steps (Lines 3-6). One example is the edge from S_2 in S'_2 to S_4 in S'_3. Because the pipeline is still in prologue stage, the edge condition is that loop does not exit.

The back edge of the new loop connects the last scheduling step of the pipeline full stage to the first one. $S'[N-1]$ is the last one and $S'[N-I]$ is the first step in the pipeline full stage. Finally, for an unbounded loop, exit can occur in any iteration. Thus, we must allow the pipeline to start flushing in any iteration, even when the pipeline is not full (Lines 12-17). In the example shown in Figure 5(b), the exit point of the loop is in S_2, therefore in pipeline epilogue, the

edge from S_4 to S_3 will never be valid. This is because the loop would have already exited and the S_3 and S_4 of the new iteration will not execute. The dead edges will be removed to simplify the final CCDFG.

Algorithm 4 GenerateEdges (S, E, I, N)

1: $E' \leftarrow \emptyset$
2: /*build the edges between new scheduling steps*/
3: **for** each step pair$(S[i], S[i+1])$ in S **do**
4: $\quad e' \leftarrow buildEdge(S[i], S[i+1])$
5: $\quad E' \leftarrow append(E', e')$
6: **end for**
7: /*build the back edge*/
8: $s_{src} \leftarrow S[N-1]$; $s_{dst} \leftarrow S[N-I]$
9: $e_{backedge} \leftarrow buildEdge(s_{src}, s_{dst})$
10: $E' \leftarrow append(E', e_{backedge})$
11: /*build the early exit edge*/
12: $i \leftarrow N-1$
13: **while** $i < sizeof(S) - 1$ **do**
14: $\quad e' \leftarrow buildEdge(S[i], s_{loopexit})$
15: $\quad E' \leftarrow append(E', e')$
16: $\quad i \leftarrow i + I$
17: **end while**
18: **return** $\langle E' \rangle$

Algorithm 5 GenerateForwarding (S, M, E, I)

1: /*find all loop carried dependencies*/
2: $D_{lc} \leftarrow getLoopCarriedDependencies(S, M, E)$
3: $S' \leftarrow S$; $M' \leftarrow M$
4: **for** each pair (o_w, o_r) in D_{lc} **do**
5: $\quad$**if** $checkForwarding(o_r, I, S')$ **then**
6: $\quad\quad \langle S', M' \rangle \leftarrow moveOp(o_w, o_r, S', E, M')$
7: $\quad$**else**
8: $\quad\quad$**return** $ERROR$
9: $\quad$**end if**
10: **end for**
11: **return** $\langle S', M' \rangle$

A critical puzzle is computation of data forwarding paths along pipeline iterations (Algorithm 5). Data forwarding is critical to achieving aggressive pipelining and eliminate the data hazards. The first key observation is that forwarding is only necessary for loop carried dependencies, which extend back to the previous iteration. D_{lc} denotes a list of dependencies and Subroutine $getLoopCarriedDependencies$ finds all loop carried dependencies. Each dependency is pair of operations (o_w, o_r), o_w is the last write operation in the loop body and o_r is the first read operation. Subroutine $checkForwarding$ checks if the data forwarding is possible (*i.e.*, whether the value is computed before use) for these variables in the scheduling steps of the pipeline. We then implement forwarding using so-called "Φ nodes". Φ nodes are special operators in compiler transformations and are widely used in resolving conditional branches in a number of compilers, and used to postpone computation of control flow to run time. In particular, a Φ node is introduced in a basic block which has multiple predecessors; the values of variables in a Φ node for a specific execution are given by the specific block which actually precedes the node in that execution. To understand its utility for data forwarding,

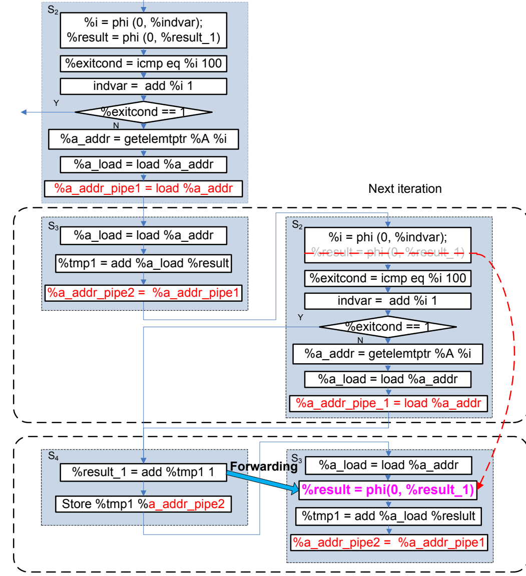

Figure 6: Pipeline Registers and Forwarding

consider Figure 6. In the non-pipelined design Φ operators can occur only in scheduling step S_2. The valid value of variable *%result* is computed by the Φ node in scheduling step S_2. Since we desire to execute scheduling steps S_2 and S_3 within a single scheduling step, we move the Φ from S_2 to S_3 and forward the value directly from the producer to the consumer. In general, to implement pipeline forwarding, we need to relocate the position of the Φ operator for a variable to immediately before its first consumer, also update the assignment of Φ node according to the new control flow. The "move" is implemented in *moveOp*, which will generate new scheduling S' and new microstep partition M'.

5. EXPERIMENTAL RESULTS

We implemented the loop pipelining algorithm on top of our verified/verifying certification framework for behavioral synthesis [14]. SEC is implemented by cycle-by-cycle dual-rail, word-level symbolic simulation between CCDFG and RTL that utilizes CVC3 SMT engine, as well as several optimizations including cutpoints, cut-loop, and modular analysis [6]. We ran our tool on a collection of pipelined designs synthesized by AutoESL. All experiments were conducted on a workstation with 3GHz Intel Xeon processor with 2GB memory.

Table 1 illustrates the results. Our framework could successfully handle SEC for synthesized designs with pipelined loops involving several thousand lines of RTL within reasonable time and memory bounds. Note that this success on pipelines depends on the applicability of other optimizations during SEC. The reason is that because of the presence of non-trivial loops, SEC without cut-loop optimization requires expensive fixed-point computation which runs out of memory and time. For all designs, brute-force SEC between the unpipelined CCDFG and the RTL times out. SEC between the pipelined CCDFG and the RTL can mostly finish. With the optimizations applied, SEC finishes with reduced memory and time usages. The results thus support

Table 1: Loop Pipelining Experimental Results

Design	RTL #line	App. Domain	Loop Info.			Pipeline Info.		Without Opt.		With Opt.	
			Interval	Depth	Operations	Forwarding	Pipeline Register	Mem. (MB)	Time (Sec)	Mem. (MB)	Time (Sec)
MemoryOp	291	Memory operation	1	4	18	2	2	24	38	4	0.3
TEA	383	Cryptography	1	4	28	4	2	-	-	40	6.2
XTEA	483	Cryptography	1	3	37	4	1	-	-	52	7.8
CORDIC	485	Data processing	1	3	31	4	0	38	7.9	5	0.9
SmithWater	517	Data processing	2	3	73	3	0	-	-	134	50.2
FIR	610	Signal processing	3	5	27	3	1	763	127.4	63	10.8
YUVToRGB	756	Image processing	2	6	77	1	5	-	-	335	128.9
MotionComp	1248	Image processing	1	3	53	3	0	434	132.2	50	11.4
DES	3292	Cryptography	1	3	17	2	2	468	364.7	257	163.3

our preference to compare the RTL with a closely resembling pipelined CCDFG that facilitates the optimizations, rather than develop a specialized SEC algorithm for pipelines.

6. RELATED WORK

Koelbl *et al.* [8] provide a tutorial introduction on methods of comparing high-level designs with RTL. Chauhan *et al.* [2] propose a technique for SEC between non-cycle-accurate designs by constructing a pair of normalized cycle-accurate designs from the original designs. Kundu *et al.* [9] propose the use of bisimulation correspondence to validate designs generated by behavioral synthesis. However, neither approach provides pipelining-specific equivalence checking strategies that effectively integrate with behavioral synthesis flows.

There is a significant literature on verifying pipelined microprocessors [1, 7, 10, 15], which has parallels with our work. However, there has been very little published work on formal verification of pipelines generated by behavioral synthesis. Nevertheless, any viable SEC framework for behavioral synthesis (*e.g.*, Synopsys Hector tool) must handle loop pipelining. To our knowledge current implementations either involve cost-prohibitive input-output comparison or require the user to provide the requisite mappings.

7. CONCLUSIONS AND FUTURE WORK

We have presented an approach to equivalence checking of pipelined designs generated by behavioral synthesis. Its efficiency and scalability has been attested by application to industrial-size case studies. The key insight is that a parameterized, synthesis-guided reference transformation on CCDFG permits comparison with RTL even after mappings with the original sequential specification has been destroyed by an aggressive transformation such as pipelining. Furthermore, the approach permits smooth integration with pipeline-oblivious transformations such as cut-loop.

In future work, we plan to handle more industrial examples. We also plan to handle more diverse pipelines, including function pipelines and pipelines for nested loops.

Acknowledgment

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

8. REFERENCES

[1] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *Proc. of CAV*, 1994.

[2] P. Chauhan, D. Goyal, G. Hasteer, A. Mathur, and N. Sharma. Non-cycle-accurate sequential equivalence checking. In *Proc. of DAC*, 2009.

[3] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. Behavioral and Communication Co-Optimizations for Systems with Sequential Communication Media. In *Proc. of DAC*, 2006.

[4] Forte Design Systems. *Cynthesizer Manual*, 2011.

[5] D. Gajski, N. D. Dutt, A. Wu, and S. Lin. *High Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1993.

[6] K. Hao, F. Xie, S. Ray, and J. Yang. Optimizing equivalence checking for behavioral synthesis. In *Proc. of DATE*, 2010.

[7] R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *Proc. of ICCAD*, 1995.

[8] A. Koelbl, Y. Lu, and A. Mathur. Formal Equivalence Checking between System-level Models and RTL. In *Proc. of ICCAD*, 2005.

[9] S. Kundu, S. Lerner, and R. Gupta. Validating High-Level Synthesis. In *Proc. of CAV*, 2008.

[10] J. Levitt and K. Olukotun. A scalable formal verification methodology for pipelined microprocessors. In *Proc. of DAC*, 1996.

[11] Y.-L. Lin. Recent developments in high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 2(1), 1997.

[12] Mentor Graphics. *Catapult C Reference Manual*, 2011.

[13] A. Pnueli, M. Siegel, and E. Singerman. *A Survey of High-Level Synthesis Systems.* Kluwer Academic Publishers, 1991.

[14] S. Ray, K. Hao, F. Xie, and J. Yang. Formal verification for high-assurance behavioral synthesis. In *Proc. of ATVA*, 2009.

[15] M. N. Velev and R. E. Bryant. Verification of pipelined microprocessors by correspondence checking in symbolic ternary simulation. In *Proc. of ACSD*, 1998.

[16] R. Walker and R. Camposano. *A Survey of High-Level Synthesis Systems.* Kluwer Academic PublishersBoston, MA, USA, 1991.

[17] Xilinx. *AutoESL Reference Manual*, 2011.