Formal Verification Techniques for Behaviorally Synthesized Loop Pipelines

Disha Puri[1], Sandip Ray[2], Fei Xie[1], and Kecheng Hao[1]

[1] Department of Computer Science, Portland State University, Portland, OR 97207.
[2] NXP Semiconductors

Abstract. Behavioral Synthesis is the process of compiling an Electronic System Level (ESL) design in high-level languages such as C, C++ or SystemC into Register-Transfer Level (RTL) implementation in hardware description languages such as Verilog or VHDL. Loop pipelining is a critical transformation in this process, which improves the throughput and reduces the latency of the synthesized hardware. It is complex and error-prone, and a small bug can result in faulty hardware with expensive ramifications. Therefore, it is critical to certify the loop pipelining transformation so that designers can trust the behavioral synthesis process. Certifying a loop pipelining transformation is however, a major research challenge because there is a huge semantic gap between the input sequential design and the output pipelined implementation, making it infeasible to verify their equivalence with automated sequential equivalence checking (SEC) techniques.

Certification of a loop pipelining transformation is possible by a combination of theorem proving and SEC: (1) creating a certified pipelining algorithm which generates a reference pipeline model by exploiting pipeline generation information from the synthesis flow (*e.g.*, the iteration interval of a generated pipeline) and (2) conduct SEC between the synthesized pipeline and this reference model. A key and arguably, the most complex component of this approach is development of the certified loop pipelining algorithm. We propose a framework of certified pipelining primitives which we show are essential for designing pipelining algorithms. Using our framework, we build a certified loop pipelining algorithm. We also propose a key invariant in certifying this algorithm, which links sequential loops with their pipelined counterparts. This is unlike other invariants that have been used in pipeline proofs. This certified algorithm is essential in the overall approach to certify behaviorally synthesized pipelined designs. We test the scalability and robustness of our algorithm on industrial-strength ESL designs that result in tens of thousands of lines of RTL implementations.

1 Introduction

In recent years, the demand for hardware with smaller form factor and higher transistor density has been steadily increasing. Consequently, it has become difficult to create high quality hardware by hand-crafting RTL designs. A behavioral

synthesis tool takes a behavioral design description (in C, C++, or SystemC), often referred to as Electronic System Level (ESL) design, and automatically generates an optimized RTL design (in hardware description languages such as VHDL or Verilog). Studies have shown that ESL reduces the design effort by 50% or more while attaining excellent performance results [29]. It has recently received significant attention, as the steady increase in hardware complexity has made it increasingly difficult to design high-quality designs through hand-crafted RTL under aggressive time-to-market schedules. A recent example is VP9 G2 hardware decoder IP developed by Google [60], which has been implemented primarily in standard C++ and synthesized to RTL logic for different target technologies and performance points using Calypto's Calatpult High Level Synthesis tool [5]. Nevertheless, and in spite of availability of several commercial behavioral synthesis tools [4, 45, 15], the adoption of the approach in mainstream hardware development for microprocessor and SoC design companies is dependent on designers' confidence that the synthesized RTL indeed corresponds to the ESL specification.

Loop pipelining is one of the most complex transformations in behavioral synthesis. It is available in most commercial synthesis tools (*e.g.*, AutoESL [61] and Cynthesizer [45]) and is crucial to producing hardware with high throughput and low latency by allowing temporal overlap of successive loop iterations. Certifying the correspondence between a sequential design and its pipelined counterpart is challenging due to the huge semantic gap between the two designs. As a result, hardware designers are vary of using current behavioral synthesis tools as they are often deemed either (a) aggressively optimized but error-prone or (b) reliable but overly conservative, thus often producing circuits of poor quality or performance [16, 36]. Therefore, ensuring correctness of behaviorally synthesized pipeline designs is a critical issue in bringing behavioral synthesis into practice.

An approach for certifying loop pipelining transformations using a combination of SEC and theorem proving techniques has been proposed earlier [20]. The most critical and complex component of this approach is developing a loop pipelining algorithm with two key properties: (1) it generates a reference pipeline model by exploiting pipeline generation information from the synthesis flow (e.g., the iteration interval of a generated pipeline) and the reference model can be compared with a pipelined RTL implementation using SEC effectively, and (2) it can be mechanically verified to correctly preserve the semantics of sequential (non-pipelined) specification of loop execution. The viability of this approach was shown by comparing pipeline generated from their algorithm with RTL implementation using SEC. However, this algorithm was not certified as well as incomplete. Certification is a key component without which correctness of behavioral synthesis process for pipelined designs cannot be claimed. Our work on developing a certified loop pipelining algorithm using our framework of certified primitives is important to facilitating formal verification of behaviorally synthesized pipeline designs.

The key contributions of this paper are:

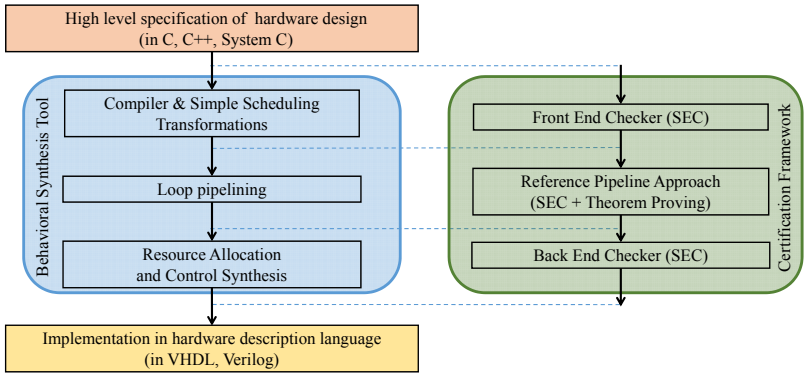

Fig. 1. Certification Model for Behaviorally Synthesized Pipelines

1. Identifying the key provable primitives essential in pipelining algorithms for behavioral synthesis and certifying these primitives in ACL2 theorem prover;
2. Formalizing an invariant to link the sequential loop before pipelining with the pipelined loop;
3. Developing our own executable loop pipelining algorithm in ACL2 using those primitives and certifying this algorithm using ACL2 theorem prover;
4. Testing our certified loop pipelining algorithm on industrial-strength designs

The remainder of this paper is organized as follows. Section 2 provides background on the overall project and explains the context of our theorem proving work. Section 3 discusses our formalization of the intermediate representations used in behavioral synthesis. We also discuss the correctness statement for loop pipelining algorithms. Section 4 discusses the research challenges. Section 5 discusses our framework and a certified loop pipelining algorithm we have developed using the framework. Section 8 provides a proof sketch for our algorithm. Section 9 provides evaluation of robustness and scalability of our algorithm on industrial-strength designs. The related work is discussed in Section **??**. We then conclude and discuss future work in Section 11.

2 Background and Context

The overall goal of the project is to provide a mechanized framework for certifying hardware designs synthesized from ESL specifications by commercial behavioral synthesis tools. One obvious approach is to apply standard verification techniques (SEC or theorem proving) on the synthesized RTL itself. Unfortunately, such a methodology is not practical. As mentioned earlier, the large gap in abstraction between the ESL and RTL descriptions means that there is little correspondence in internal variables between the two. Consequently, direct SEC between the two reduces to cost-prohibitive computation of input-output

equivalence. On the other side, applying theorem proving is also troublesome since extensive manual effort is necessary and this effort needs to be replicated for each different synthesized design. It is also infeasible to directly certify the implementation of the *synthesis tool* via theorem proving. In addition to being highly complex and thus potentially requiring prohibitive effort to formally verify with any theorem prover, the implementations are typically closed-source and closely guarded by EDA vendors and thus out of reach of external automated reasoning communities.

To address this problem, previous work developed two key SEC solutions, which we will refer to below as *Back-end* and *Front-end*. We then discuss the gap between them, which is being filled by theorem proving efforts in this dissertation. The certfication model is illustrated in Figure 1.

Back-end SEC: The key insight behind back-end SEC is that automated SEC techniques, while ineffective for directly comparing synthesized RTL with the top-level ESL description, are actually suitable to compare the RTL with the intermediate representation (IR) generated by the tools after the high-level (compiler and scheduling) transformations have been applied. In particular, operation-to-resource mappings generated by the synthesis tool provide the requisite correspondence between internal variables of the IR and RTL. Furthermore, a key insight is that while the implementations of transformations are unavailable for commercial EDA tools, most tools provide these IRs after each transformation application together with some other auxiliary information. To exploit these, an SEC algorithm was developed between the IR (extracted from synthesis tool flow after these transformations) and RTL [49, 21, 22, 66]. The approach scales to tens of thousands of lines of synthesized RTL.

Front-end SEC: Of course the back-end SEC above is only meaningful if we can certify that the input ESL indeed corresponds to the extracted IR produced after the compiler and scheduling transformations applied in the first two phases of synthesis. To address this, another SEC technique was developed to compare two IRs [63, 65, 64]. The idea then is to obtain the sequence of intermediate representations $\mathrm{IR}_0, \ldots, \mathrm{IR}_n$ generated by the compiler and scheduling transformations, and compare each pair of consecutive IRs with this new algorithm. Then back-end SEC can be used to compare IR_n with the synthesized RTL, completing the flow.

A Methodology Gap: Unfortunately, the front-end SEC algorithm can only compare two IRs that are structurally close. If a transformation significantly transforms the structure of an IR then the heuristics for detecting corresponding variables between the two IRs will not succeed, causing equivalence checking to fail. Unfortunately, loop pipelining falls in the category of transformations that significantly changes the structure of the IR. It is a quintessential transformation that changes the control/data flow and introduces additional control structures (to eliminate hazards). This makes front-end SEC infeasible for its certification. Furthermore, most commercial implementations are of course proprietary and consequently not available to us for review; applying theorem proving on those

implementations is not viable from a methodology perspective. Thus a specialized approach is warranted for handling its certification.

3 Formalization

3.1 Intermediate Representation

A behavioral synthesis tool performs a number of compiler and scheduling transformations (including pipelining, which is the focus of this paper). Certification of behavioral synthesis transformations thus requires a formalization of the design representation manipulated by these transformations. The formalization we use is *Clocked Control Data Flow Graph* (CCDFG) [49]. Structurally, a CCDFG is a control and data flow graph augmented with a schedule. The control flow is broken into basic blocks. The instructions are grouped into microsteps which can be executed concurrently. A scheduling step represents a group of microsteps which can be executed in a single clock cycle. State of a CCDFG at a particular microstep is a list of all the variables of a CCDFG with their corresponding values.

The semantics of CCDFG require a formalization of the underlying language used to represent the individual instructions in a scheduling step. The underlying language we use is the LLVM language [38]. LLVM is a popular compiler infrastructure for many behavioral synthesis tools (*e.g.*, AutoESL [61], xPilot [11], LegUp [6]). We provide semantics to these instructions through a standard, state-based operational formalization [44]. It includes assignment, load, store, bounded arithmetic, bit vectors, arrays, and pointer manipulation instructions. Executing a microstep in a CCDFG implies changing the current state of CCDFG based on meaning of the instructions in the microstep and producing a new state. Assigning meanings to most instructions is standard; one exception is the so-called "ϕ-construct". A ϕ-construct is a list of ϕ-statements. A ϕ-statement is $v := \phi[\sigma, X][\tau, Y]$, where v is a variable, σ and τ are expressions, and X and Y are scheduling steps: if it is reached from X then it is the same as the assignment statement $v := \sigma$; if reached from Y, it is the same as $v := \tau$; the meaning is undefined otherwise. ϕ-constructs are necessary due to the structure of the SSA (static single assignment) form of the LLVM code.

3.2 Loop Pipelining Transformation

A behavioral synthesis tool automatically generates a RTL design from an ESL design through a series of transformations. Pipelining a loop is a critical transformation in behavioral synthesis. For our purposes, a *pipelinable loop* is a loop with the following additional restrictions [20]:

1. no nested loop;
2. only one *Entry* and one *Exit* block; and
3. no branching between the scheduling steps.

These restrictions reflect the kind of loops that can be actually pipelined during behavioral synthesis. For instance, synthesis tools typically require inner loops to have been fully unrolled (perhaps by a compiler transformation) in order to pipeline the outer loop.

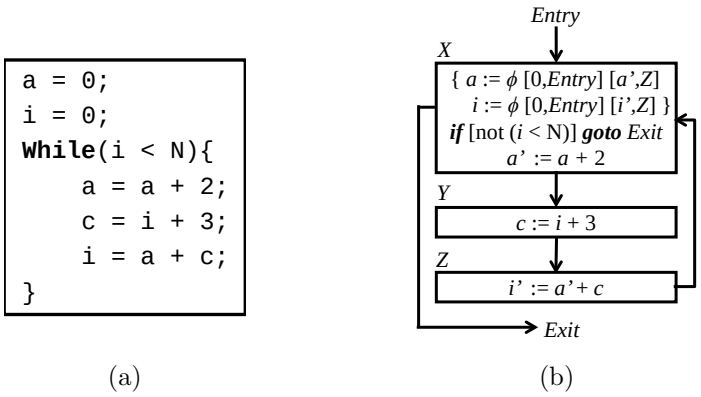

Fig. 2. (a) Loop in C (b) Loop CCDFG before pipelining.

Figure 2(a) illustrates the C code (ESL description) for a loop at the beginning of the synthesis process. The C code does not have a schedule or the concept of a clock cycle. Figure 2(b) shows CCDFG of the sequential loop just before loop pipelining. The loop has three scheduling steps: X, Y and Z. The scheduling step before the loop is $Entry$ and after the loop is $Exit$. Since, there are three scheduling steps in the loop, one iteration can be executed in three clock cycles.

Behavioral synthesis tools use complicated heuristics and aggressive scheduling strategies to find an optimized pipeline interval (clock cycles after which a new iteration can be started such that there are no data hazards). Figure 3 shows the pipelined CCDFG with a pipeline interval equal to one. The new scheduling steps in the pipelined CCDFG created by combining scheduling steps from different iterations of the sequential CCDFG are called scheduling supersteps. Observe that the three iterations of the pipelined loop take five clock cycles as opposed to nine clock cycles in the sequential loop. Loop pipelining reduces the number of clock cycles required to execute the loop, hence this transformation is used by synthesis tools to increase throughput and reduce latency.

3.3 Correctness of Pipelined CCDFG

Loop Pipelining is a critical and complex transformation. So, it is important to certify that the pipelined CCDFG is indeed correct. Correctness of loop pipelining can be informally stated as below.

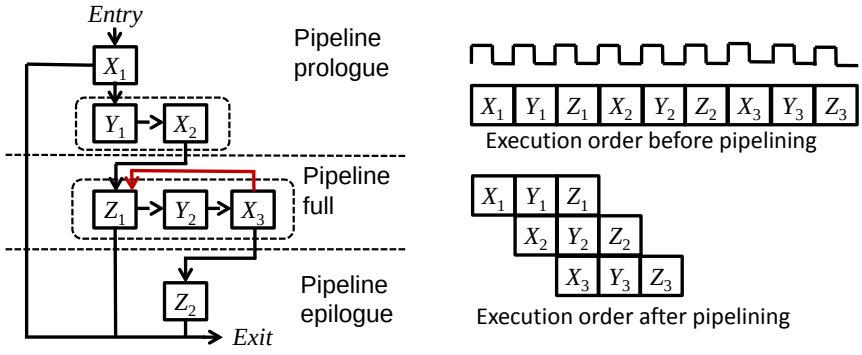

Fig. 3. Pipelined CCDFG. The horizontal arrows in the scheduling supersteps indicate data forwarding.

Let L be a loop in CCDFG C, and let L_α be the pipelined loop CCDFG. Let V be the set of variables mentioned in L, and U be the set of all variables in C. Suppose we execute L and L_α from CCDFG states s and s' respectively, such that for each variable $v \in V$, the value of v in s is the same as that in s', and suppose that the state on termination are f and f' respectively. Then (1) for any $v \in V$, the value of v in f is the same as that in f', and (2) for any $v \in (U \backslash V)$, the value of v in f' is the same as that in s'.

Remark: Condition (2) is the *frame rule* which ensures that variables in C that are not part of the loop are not affected by L_α.

3.4 An approach to verification of loop pipelining transformation

Hao *et al.* proposed a pipeline generation algorithm using feedback (like pipeline interval) from the synthesis tool [20]. They show that to verify the correspondence between sequential CCDFG and pipelined RTL, it is sufficient to perform the following three steps.

1. Check that the algorithm can generate a pipeline reference model for the parameters reported by synthesis.
2. Use SEC to compare the pipeline reference model with the synthesized RTL.
3. Prove the correctness of the algorithm.

The pipelining algorithm is simpler than that used by the synthesis tool because the synthesis tool uses advanced heuristics to determine the pipeline parameters (such as how many iterations to pipeline, when to introduce stalls etc.), while this pipelining algorithm only uses those parameters to generate a reference model. The algorithm is shown to be scalable but it is not certified.

4 Research Challenges

To understand the complexities involved in mechanical certification of an algorithm that was not designed originally with certification in mind, we need to re-visit the general approach to applying formal reasoning on software programs. The typical approach is to break the program into a number of pieces, prove key lemmas characterizing the role of each piece, and then chain these lemmas together into a proof of the correctness of the entire program. Crucial to this approach, however, is the requirement that each program piece can be characterized by a succinct invariant that can be easily verified. However, in a program not developed with reasoning in mind, optimizations typically destroy the structural disciplines and modularity of the individual program pieces. This makes it difficult to identify and isolate the components that actually maintain succinct, interesting invariants.

Our first approach was to certify their implementation as it is using theorem proving. But, our experience was that it is a difficult approach, one that we need not endure. In general, in order to certify such an arbitrary implementation, one has to either (1) restructure the implementation into one that is more disciplined, and prove the equivalence between the two, or (2) come up with very complex invariants that essentially comprehend how invariants from each individual piece are conflated together in the implementation. Both approaches require extensive human interaction, resulting in the proverbial euphemism of proofs of programs being orders of magnitude more complex than the programs themselves [41].

In our work, however, we can "get away" without verifying the specific implementation while still being able to certify the design generated by behavioral synthesis without loss of fidelity. The key observation, as above, is that it is sufficient to develop *any* certifiable algorithm that generates a pipelined CCDFG from a sequential implementation which can be effectively applied with SEC. In particular, any certifiable algorithm that has the same input-output characteristic as the proposed algorithm is sufficient. Thus, our work is on identifying certifiable primitives and invariants of a loop pipelining transformation and developing a pipeline generation algorithm using those primitives, achieving the dual goal of mechanical reasoning of the algorithm and amenability of the resulting reference model to SEC.

Note that our framework is independent of the inner workings of a specific tool, and can be applied to certify designs synthesized by different tools from a broad class of ESL descriptions. Also, the approach produces a certified reference flow, which makes explicit generic invariants that must be preserved by different transformations. Checking correctness using formal methods prompted us to address the issues lacking in the previous algorithm. To ensure that control flow is maintained, we had to deal with branches. The previous algorithm introduces the concept of Exit edges but does not explain/implement them. The previous authors checked the output of their algorithm with RTL under the assumption that the loop never exits, hence they did not face any issue while testing. However, removing a conditional branch in a loop and furthermore, adding the conditional

branch back in the middle of a pipelined loop requires complex reasoning which we manage using one of our primitives, explained in Chapter 5.

Also, the invariant that data flow is maintained at each step enabled us to find a bug in the previous algorithm. The previous algorithm moves a statement to make sure one particular data hazard is removed, but in doing so they move the statement across a conditional branch statement. Our primitves ensure that such a move is not possible. We have restructred the algorithm so that instead of going across a conditional branch in the same iteration, the movement of step is now to the previous iteration, explained in Chapter 5.

5 Our Approach

One of the most complex requirement of verifying behaviorally synthesized pipelined designs is a certified loop pipelining algorithm which can generate a pipeline reference model for industrial strength designs. This pipeline reference model must have a similar structure to the pipelined RTL generated by behavioral synthesis tools such that they can be compared using SEC.

Pipeline synthesis is based on the key observation that execution of successive iterations can be overlapped without affecting execution as long as data and control dependecies are correctly maintained. Thus, the three main activities of a pipeline synthesis algorithm are to (1) identify and remove possible hazards (2) overlap the successive iterations according to the pipeline interval, and (3) ensure proper placement of conditional and unconditional branches. In our case, the identification of data hazards is simplified since the synthesis tool provides a pipeline interval. If we can use this pipeline interval to build our design, then the pipeline reference model is comparable to RTL in abstraction. Thus, instead of *discovering* a pipeline interval ourselves by analyzing read and write variables of every design so that no hazard is introduced, we reuse the provided interval. We have developed a framework of five certified pipelining primitives which allows us, among other things, to prevent possible data hazards. Our framework also provides a primitive to overlap successive iterations and a provision to add and remove branches when required while still maintaining the control flow. We now discuss the framework in detail.

6 Framework of Provable Pipelining Primitives

We believe that the following primitives are necessary and essential in creating any pipelining algorithm in behavioral synthesis.

ϕ-elimination primitive – A ϕ-statement is "`v = phi [`σ `X] [`τ `Y]`", where `v` is a variable, σ and τ are expressions, and `X` and `Y` are basic blocks: while execution, if the ϕ-statement is reached from `X` then it is the same as the assignment statement `v =` σ; if reached from `Y`, it is the same as `v =` τ; the meaning is undefined otherwise. Reasoning about the ϕ-statement is complex since after its execution from a state, say s, the state reached depends not only on the state s but also on previous basic blocks in the execution history. However, we must

handle it since it is used extensively in loops to perform different actions depending on whether the loop body is executed the first time. One of the key steps in loop pipelining is, therefore, ϕ-elimination *i.e.*, replacing ϕ-statement with appropriate assignment statements when the previous basic block is known.

Shadow register primitive – We define a shadow register microstep as simply an assignment statement with symbol expression (x) assigned to a new value (x_reg). We call all the new introduced variables as shadow registers. Intuitively, it is correct that in a sequence of steps, if we assign a variable to a shadow register and replace all occurences of x with x_reg till the next write of x, we should not have made any difference in the execution. Also, since we are not changing the value of x itself, the state after end of execution for both CCDFGs as far as real variables are concerned (all variables excluding all shadow registers) is same.

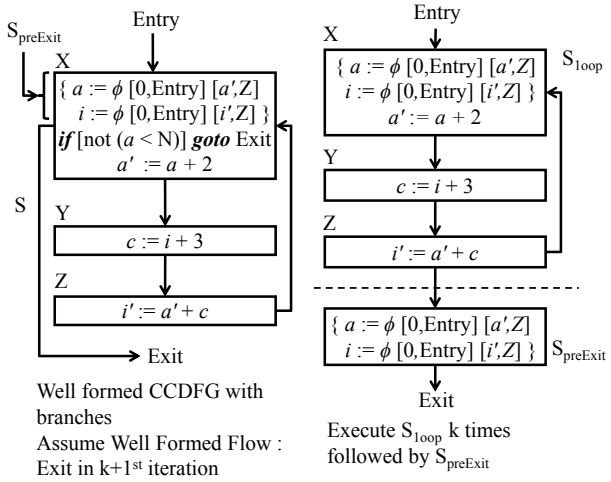

Fig. 4. Branch Primitive

Branch primitive – Branch instructions are required to determine the control flow. However, reasoning about execution of branch instructions in a loop everytime we apply a primitive can make proof very complex. We note that if we specifically assume that the exit condition becomes true after completing k iterations, then we can remove the conditional branch. To understand the branch primitive (c.f. Figure 4), let's assume there is a conditional branch in the sequential loop structure S, which points to either the next microstep in sequence or exits the loop by branching to the scheduling step $Exit$. Let $S_{preExit}$ be the collection of microsteps before this branch in S and let S_{loop} be the corresponding CCDFG loop without the conditional branch. The conditional branch primitive allows us to replace S with S_{loop} followed by $S_{preExit}$. Similarly, the primitive

also allows us to introduce an exit conditional branch by replacing S_{loop} followed by $S_{preExit}$ with S. Note that since k can take any value $k \geq 0$, we are not compromising on the correctness statement. It can be proved that executing S k times such that it exits in the $(k+1)$ st iteration is same as executing S_{loop} k times followed by $S_{preExit}$.

Interchange primitive – Let m and n be two adjacent scheduling steps (or in general, any collection of microsteps) in a CCDFG where both m and n do not have any microsteps containing branch statements. Also, there are no read write hazards between m and n. By read write hazards, we mean that m does not read or write any variable which is written in n and vice versa. Then, the interchange primitive allows us to interchange the order of m and n in the given CCDFG. Note that under the given assumptions, if initial state is the same, then the state reached after executing m followed by n is same as the state reached after executing n followed by m.

Superstep construction primitive – This operation entails combining the scheduling steps of the successive iterations, forming scheduling "supersteps" that act as scheduling steps for the pipelined implementation. Supersteps must account for read-after-write hazards, i.e, if a variable is written in a scheduling step X and read subsequently in Z then Z cannot be in a superstep that precedes X in the control/data flow. Note that we implement data forwarding (forward value of data within a single clock cycle); thus X and Z can be in a single superstep.

7 Our Loop Pipelining Algorithm

Given a sequential loop S in CCDFG C and pipeline interval I, we can create a pipelined loop P using Algorithm 1. Note that every step of the algorithm is build from ground up using our framework of provable primitives such that the algorithm can be certified by theorem proving. A quick overview of primitives used in the algorithm at each step are shown in Figure 5.

Algorithm 1 Pipelining Algorithm

```
1: procedure PipelineLoop(S, I)
2:     S_1 ← RemoveBranches(S)
3:     S_2 ← UnrollLoopOnce(S_1)
4:     S_3 ← φ − Elimination(S_2).
5:     S_4 ← DataPropagation(S_3, I).
6:     S_5 ← GenerateShadowRegisters(S_4, I).
7:     S_6 ← SuperstepConstruction(S_5, I).
8:     P ← AddBranches(S_6)
9:     return (P).
10: end procedure
```

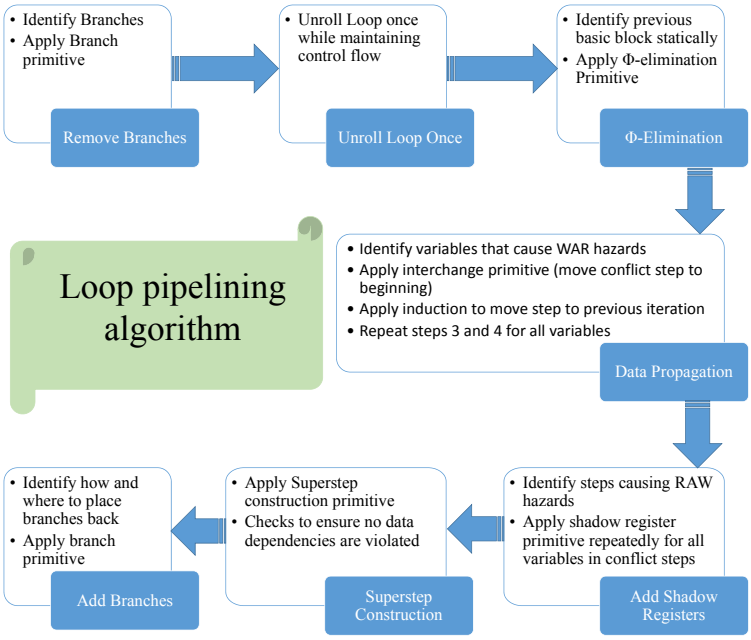

Fig. 5. Our Loop Pipelining Algorithm (built using primitives)

Now, we describe the steps to convert a sequential loop CCDFG (c.f. Figure 4) to a pipelined loop CCDFG in detail:

Remove Branches: We apply the branch primitive on S (c.f. Figure 4) to remove branches by explicitly defining the control flow in S.

Unroll Loop Once: We have already established that the first iteration behaves differently than the rest of the iterations due to ϕ-construct. So, we simply unroll the loop S_{loop} once and call the first iteration S_{pre}.

ϕ-elimination: We apply the ϕ-elimination primitive on S_{pre}, S_{loop} and $S_{preExit}$ to return a CCDFG in which all the ϕ-statements have been replaced with their corresponding assignment statements. Note that ϕ-construct is only in the first scheduling step of any iteration, so the remaining scheduling steps are the same in all the iterations.

Data propagation: Algorithm 2 describes how to compute candidates for data propagation across pipeline iterations. It is a critical step in removing data hazards. We want to make sure that when we pipeline a loop, we do not read a variable which has not yet been written. A critical observation is that data propagation is required only for loop carried dependencies. $GetLoopCarriedDependencies$ identifies the microsteps where loop carried dependencies are being read. Then, $CheckConflict$ checks whether there would be a conflict when we pipeline the loop. Conflict occurs when the value being read in a microstep is not yet written in the pipelined loop execution. If so, $RelocateMSteps$ relocates the microstep

Algorithm 2 Data propagation

```
1: procedure DataPropogration(L)
2:     msteps ← GetLoopCarriedDependencies(L)
3:     for each mstep in msteps do
4:         if CheckConflict(L, mstep, N, I) ≠ 0 then
5:             L ← RelocateMStep(L, mstep)
6:         end if
7:     end for
8:     return (L)
9: end procedure
```

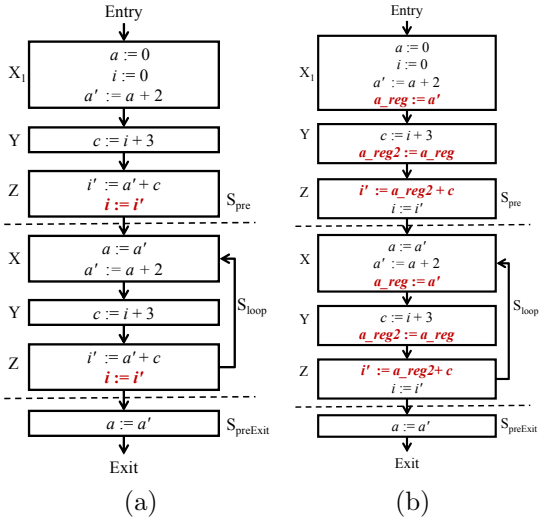

Fig. 6. (a) After Data Propagation (b) After adding Shadow Register

which is causing the data hazard to the previous iteraton. Note that this step requires multiple applications of interchange primitive and very complex induction. This step ensures that any variable which is being read has already been written. Note that in order to maintain the invariant, only those microsteps can be propagated which exist in $S_{preExit}$, which means only those steps which occur before the conditional branch in original CCDFG can be relocated. This ensures that our algorithm does not have the bug which the previously proposed algorithm had. In Figure **??**(b) we found that the loop carried dependency i' in X would create a conflict when we would move X before Z while pipelining. So, we relocate the microstep $i := i'$ as shown in Figure 6(a). This step needs to be repeated for every variable found using *GetLoopCarriedDependencies*.

Generate shadow registers: Algorithm 3 inserts shadow registers to prevent variables from being overwritten before being read.

Algorithm 3 Generate shadow registers

```
1: procedure GenerateShadowRegisters(L, I)
2:     V ← GetAllVariables(L).
3:     for each v in V do
4:         w_v ← WriteVariable(v, L).
5:         r_v ← LastReadVariable(v, L).
6:         if RequireShadowRegister(r_v, w_v, I) ≠ 0 then
7:             L ← AddShadowRegister(w_v, L).
8:         end if
9:     end for
10:    return (L).
11: end procedure
```

We first compute all program variables that may be overwritten before being read, which means these are the variables that require shadow registers. To find such variables, *GetAllVariables* first gets a set of all variables. Then, for each variable, we compare the distance (the number of scheduling steps) between the write of the variable w_v (*WriteVariable*) and the last read of the variable r_v (*LastReadVariable*) in an iteration; if the distance is greater than I (pipeline interval), the variable is assigned the new data value of the next iteration before the current iteration's value has been fully consumed; this warrants insertion of shadow registers in every scheduling step between the r_v and w_v. The value is propagated every clock cycle following the CCDFG data flow. We apply the shadow register primitive on the microstep which writes the variable (*AddShadowRegister*). We assign that variable to a new temporary variable called shadow register in every new scheduling step and replace all subsequent reads of that variable with the shadow register till its next write. In Figure 6(b), we introduce a shadow register a_reg in X and a_reg2 in Y. This step is also repeated for all the variables found using *GetAllVariables*.

Superstep construction: Now that we have removed the data hazards, we can successfully pipeline the loop using the pipeline interval I. We combine the scheduling steps of the successive iterations, forming scheduling "supersteps" that act as scheduling steps for the pipelined implementation. Supersteps must account for read-after-write hazards, i.e., if a variable is written in a scheduling step s and read subsequently in s' then s' cannot be in a superstep that precedes s in the control/data flow. A scheduling step is allowed to move up another scheduling step only if there are no intermediate read and write conflicts. Note that we implement data forwarding; thus s and s' can be in a single scheduling superstep. Superstep construction on S_{pre} and S_{loop} creates a CCDFG with three parts: prologue P_{pre}, P_{loop} which is the full pipeline stage and epilogue P_{post} as shown in Figure 7. We will later prove using our invariant that executing P_{pre} followed by k iterations of P_{loop} followed by P_{post} is equivalent to executing S_{pre} followed by x iterations of S_{loop}, where value of x is determined based on value of k, pipeline interval I and number of scheduling steps in S.

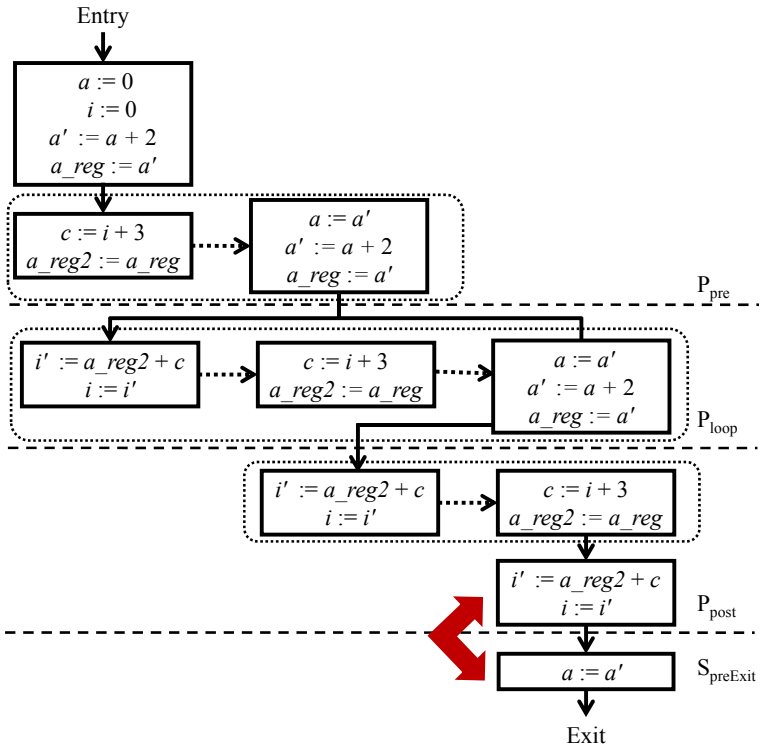

Fig. 7. After superstep construction

Add Branches: To add the branches back, we use the a combination of interchange primitive and reverse of Branch primitive. Note in Figure 7, if there are no read write hazards in between the last scheduling step Z of P_{post} and $S_{preExit}$, we can interchange them using interchange primtive. Now recall from the branch primitive that if there is a loop structure S_{loop} with a conditional branch, then executing S_{loop} such that it exits in the (k+1)st iteration is same as executing S_{loop} without the conditional branch followed by only those steps from S_{loop} which occur before the branch $S_{preExit}$. Now, we apply the reverse of branch primitive here. P_{loop} in Figure 7 is a loop structure without a conditional branch, followed by a collection of microsteps $P_{preExit}$ (here, a collection of Z, Y and $S_{preExit}$). Then, we can add an exit conditional branch in P_{loop} after the microsteps $P_{preExit}$. This branch points to the next scheduling step after the loop P_{post} if the exit condition is true. We can add the conditional and the unconditional branch as shown in Figure 8.

We now have the final pipelined loop structure. We describe a proof sketch for the primitives and the algorithm in the next section.

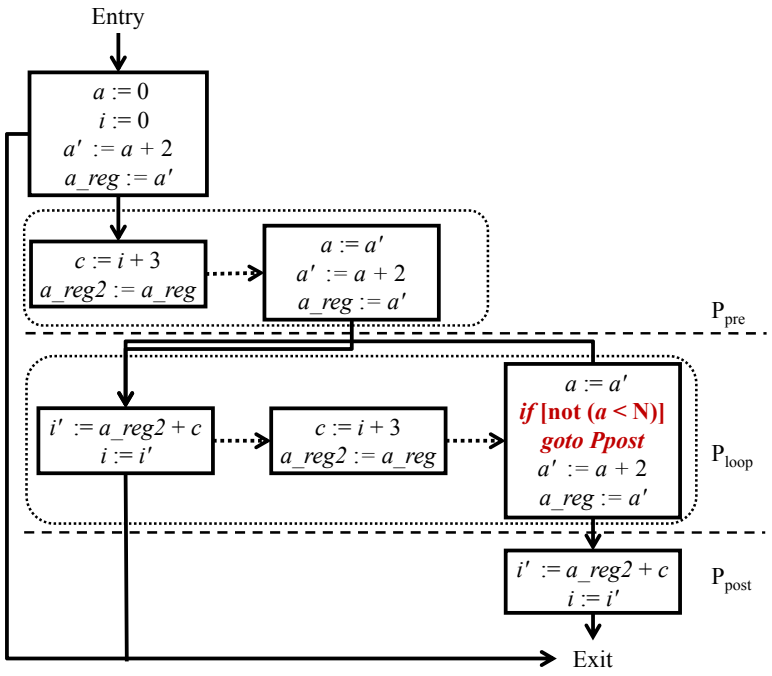

Fig. 8. Final Pipelined CCDFG

8 Proof Sketch

Certification of our loop pipelining algorithm naturally requires a certification of each of our primitives. In addition, we need to ensure that every time a primitive needs to be applied, the conditions under which the primitive can be applied are maintained. We discuss both aspects below.

8.1 Correctness of Primitives

We must prove that applying a particular primitive is correct, *i.e.*, maintaining a certain invariant. This is proven without considering how it is applied in the context of a pipeline synthesis algorithm. We give an outline of the proof to justify that the primitives are correct.

ϕ-elimination primitive: We prove that the algorithm correctly resolves the ϕ to create multiple assignment statements. We induct along the length of each sub-microstep of ϕ-construct and relate it to one corresponding assignment statement.

Shadow register primitive: We prove that adding a shadow register microstep, $a\_reg = a'$ does not change the value of any variable in the state except the shadow variable. In essence, we prove that if a variable is not written in a

microstep, then its value in the state before and after executing that microstep is same. Also, we prove that after executing the shadow register microstep, value of a_reg in the state is equal to value of a'. Furthermore, since now the value of a_reg is equal to value of a', we prove that executing a statement which reads a' has the same effect on the state as executing a statement which reads a_reg till the next write of a'. This needs to be done for all types of statements *e.g.,* assignment statements (with different types of operations like load, add, mul, getelementptr *e.t.c.*), store statement, branch statement etc. We determine the variables read and written in a statement by analyzing the statements. Note that a_reg is a new variable which is neither written nor read in the given statements.

Interchange primitive: We prove that we can interchange any two adjacent microsteps (excluding branch microsteps) which do not have read-write conflict. We prove that given an initial state, the state after executing microsteps m and n is the same as the state after executing n then m if m and n have no read-write conflict. Suppose, the state after executing m and n is s_1 and that after executing n and m is s_2. We prove that for any variable x, its value remains same in s_1 and s_2. After normalizing the states, we can prove that s_1 is equal to s_2, i.e., the states are the same after executing the two microsteps in a sequence or in an interchanged order. Again, reasoning about read and write of statements involves reasoning about execution semantics of all types of microsteps present in the language which is not trivial.

Branch primitive: We prove that executing S k times such that it exits in the $(k+1)$ st iteration is same as executing S_{loop} k times followed by $S_{preExit}$. We need to define a notion of a well-formed-flow to ensure that we can show that the branch does not exit in the first k iterations. We also need a way to track the backedge along the unconditional branch and ensure that it points back to the beginning of loop S.

Superstep construction primitive: This primitive is for overlapping iterations while maintaining data and control dependencies. It is built on interchange primitive but while interchange primitive handles only two adjacent microsteps, superstep construction moves around scheduling steps with multiple microsteps. The interchange primitive is extended by non-trivial induction along the length of the scheduling steps to achieve the desired result. Superstep construction primitive is proved using the interchange primitive and our key invariant described in detail below.

8.2 Key Invariant on Correspondence Between Back-edges of Sequential and Pipelined Loops

Our key invariant defines a "correspondence relation" between the back-edges of the sequential and pipelined CCDFGs explained in an earlier paper in detail [47].

8.3 Correctness of Our Algorithm

Following is an English paraphrase of the final theorem.

If the pipeline generation succeeds without error, executing the pipelined CCDFG (a combination of *pre*, *loop* and *post*) for *no_pp_steps* generates the same state of the relevant variables as executing the sequential CCDFG c for *no_seq_steps*.

The algorithm is essentially built from ground-up using primitives as shown in Section 5. However, apart from proving correctness of each primitive and our key invariant, we also need to ensure that the primitive is applied by our algorithm properly, *i.e.*, the environment assumptions on which the **correctness of primitive** depends are maintained appropriately by the algorithm at the point where the primitive is applied.

We can take each stage one by one to understand the complexity involved in verifying the algorithm as a whole, over and above the verification of individual primitives.

In the *RemoveBranches* stage, which is the first stage of pipelining algorithm, we have to create a correspondence between randomly executing a CCDFG with branches using basic-block, sub-basic-block and location with executing a CCDFG in sequence without a conditional and unconditional branch. Since we have a non-streamlined run on one side and a steamlined sequential run on the other side, there are theorems involved with finding the next microstep randomly and proving that it is same as the next microstep in streamlined order. We also need to prove that the applicaton of the branch primitive is correct. After this step and for all the subsequent steps, we need to show that there are no relevant branches in CCDFG. In the ϕ-to-assign stage, we replace one microstep of C with more than one microsteps in C'. In addition to inductively reasoning about application of a primitive in entire CCDFG, we also have to ensure that addition of new microsteps does not affect the basic structure of the CCDFG. These well-formed-conditions need to be maintained at each step to ensure that the primitives can be applied. The upshot is that an inductive theorem relating C and C' must be strong enough to comprehend the global effects. For instance, an inductive statement showing the correctness of ϕ-elimination must account for the fact that the number of microsteps of C is different from that of C'. Thus an execution of C for n microsteps must correspond to an execution of C' for a different number m of microsteps, where the number m is a function of n and the structures of C and C'; the statement of the correctness of ϕ-elimination must characterize the value of m precisely, perhaps defining functions that statically and symbolically execute C and C', in order to be provable by induction. Furthermore the functions so introduced for static symbolic execution must themselves be proven correct. Data propogation involves identifying the appropriate statements that cause conflict and applying interchange primitive multiple times to move the microstep to the beginning of the loop. We need to make sure that the conditions under which interchange primitive can be further applied are maintained after each application. The movement of steps requires non-trivial induction These stages need to be repeated for as many variables as are in conflict. Recall that shadow register step adds many more new statements to assign temporary values to new shadow variables. The addition of

new microsteps means that in addition to inductively reasoning about application of a primitive in entire CCDFG, we also have to ensure that basic structure of the CCDFG is maintained. Moreover, we need to reason about read and write of variables across a number of microsteps. The proof is analogous to the proof of shadow-register primitive. However, the primitive is applied multiple times based on the variabes which are causing conflict. This gets tricky as after application of one primitive, there are new variables introduced and we can only claim that the relevant variables have same value.This step requies proof of invariant and multiple applications of interchange primitive as explained earlier. The proof required is the reverse of branch-primitive. However, a key requirement is that branch-primitive can be applied only when we have a `well-formed-ccdfg`, so we need to ensure that the structure of the *loop* before adding branches is such that the final *loop* in the pipelined CCDFG is indeed a `well-formed-ccdfg`.

9 Viability of our Approach

As mentioned earlier, the viability of this approach was tested in [22]. They used a pipelining algorithm to generate a pipeline reference model and compared their pipelined implementation with pipelined RTL using SEC to justify their approach. If we replace their algorithm with our certified algorithm and still produce the same pipelined implementation with same shadow registers and data propagations, we can claim that our algorithm is also suited for certifying behaviorally synthesized designs. We have successfully tested the pipeline reference model generated by our certified algorithm with the pipelined reference model generated by the previous algorithm across several industrial strength designs in different domains(c.f Figure 9). The test designs are non-trivial to pipeline with varied pipeline intervals and depths and require data forwarding and use of temporary shadow registers to remove data hazards.

Design	RTL Lines #	App. Domain	Loop Interval	Loop Depth	No. of operations	Pipeline Register
MemoryOp	291	Memory Operation	1	4	18	2
TEA	383	Cryptography	1	4	28	2
XTEA	483	Cryptography	1	4	37	1
SmithWater	517	Data Processing	2	3	73	0

Fig. 9. Behaviorally synthesized pipelined designs tested using our algorithm

10 Related Work and Novelty of Our Approach

Besides behaviorally synthesized pipelines, there are mainly two other kinds of pipelines, hardware pipelines and software pipelines. There has been a significant amount of work in formal or semi-formal verification of processor (hardware) pipelines [46, 12]. Sawada and Hunt [53] presented a technique that models the trace of executed instructions using a table-based representation called a MAETT. These approaches require involvement of user to a great degree, especially in control dominated designs. Burch and Dill [3] presented a technique to verify the correctness of the implementation model of a pipelined processor against its instruction-set architecture (ISA) model based on quantifier-free logic of equality with uninterpreted functions. The technique has been extended to handle more complex pipelined architectures by several researchers [58, 59, 42, 57, 55]. All the above techniques attempt to formally verify the implementation of pipelined processors by comparing the pipelined implementation with its sequential (ISA) specification model, or by deriving the sequential model from the implementation. There are significant differences in goals and techniques between these efforts and ours. Microprocessor pipelines include optimized (hand-crafted) control and forwarding logics, but have a static set of operations based on the instruction set. Behaviorally synthesized loop pipelines tend to be deep with a high complexity at each stage, but control and forwarding logics are more standardized since they are automatically synthesized. Furthermore, microprocessor pipeline verification is focused on one (hand-crafted) pipeline implementation, while our work focuses on verifying an *algorithm that generates pipelines*. As explained earlier in section 6 that our invariant is very different from a typical invariant used in the verification of pipelined machines (*e.g.*, for microprocessor pipelines). We make explicit the correspondence with the sequential execution. The key requirement from a pipeline invariant, *viz.*, hazard freedom, is left implicit and arises indirectly as a proof obligation for invariance of this predicate.

Software pipelining [39, 37]is a form of out of order execution. It is performed by compiler rather than a processor. Behaviorally synthesized loop pipelines are similar in reasoning to software loop pipelines except that since behavioral synthesis is automatic, it is much more streamlined than software pipelines. Our understanding of hazards and reasoning behind pipelining algorithm is very closely related to recent work on verification of software pipelines. In particular, Tristan and Leroy [56] present a verified translation validator for software loop pipelines. The loop pipelines in behavioral synthesis considered in this paper are close in structure to software loop pipelines, although our formalization (*e.g.*, CCDFG) has different semantics from the Control Flow Graphs they use, reflecting the difference between eventual targets of compilation (*viz.*, hardware vs. software). However, the fundamental difference is in the approach taken to actually certify the pipelines. Tristan and Leroy's approach decomposes the certification problem into two parts, a "dynamic" part that is certified on a case-by-case basis and a "static" part that is certified in the Coq theorem prover [2] once and for all. The theorem proven by Coq is informally paraphrased as follows:

> Suppose the pipelining algorithm generates a pipeline $\mathcal{P}$ from a sequential design $\mathcal{S}$. Suppose symbolic simulation of $\mathcal{S}$ and $\mathcal{P}$ verifies certain "dynamic" verification conditions (VCs). Then $\mathcal{S}$ and $\mathcal{P}$ are indeed semantically equivalent.

Thus for any pipeline instance $\mathcal{P}$ generated by their algorithm, symbolic simulation is executed between $\mathcal{P}$ and $\mathcal{S}$ to certify that $\mathcal{P}$ is indeed a correct pipelined implementation of $\mathcal{S}$. The dynamic VCs checked by symbolic simulation essentially certify that the pipeline generation did not overlook any hazards.

This is where our work differs from theirs. Our work is expected to provide a single theorem certifying the correctness of the reference pipelined implementation, without requiring further runtime hazard check. Furthermore, their correspondence theorem relates the pipelined implementation with a sequential design with a (bounded) unrolled loop, while our approach certifies the correspondence between the actual Control Flow Graph (CFG) and the pipelined implementation. Indeed, Tristan and Leroy remark that the mechanization of the correspondence between the CFG and unrolled loop is "infuriatingly difficult". We speculate this is so because they focus on verifying the correspondence between the unrolled loop and the pipeline. In our experience, attempting the formal correspondence between the unrolled sequential loop and pipelined design is indeed difficult since there is no formal way to connect to back edge of the loop with any of the edges in the pipeline. We believe that reconciling this problem and developing a fully certified pipeline generation algorithm would require backtracking from the correspondence with an unrolled loop (and hence translation validation) to a more complex invariant like ours. Of course we must note that we can "afford" to develop a fully certified algorithm in our approach since the pipelines are simpler (cf. Chapter 3); achieving this for arbitrary software pipeline may require further more subtle invariants.

Thorem provers are widely used for hardware verification. HOL theorem prover [17] has been used in several well-documented projects [10, 19]. ACL2 is also used a lot in hardware verification [13, 34, 35, 23, 48, 51, 50]. Our project is however somewhat different from the traditional applications of theorem provers. First, since an over-arching goal is to exploit automatic decision procedures, we use theorem proving primarily to complement automated tools. Second, we eschew theorem proving on inherently complex or low-level implementations. Third, interactive theorem proving is acceptable for one-time use, in certification of a transformation, but not as part of a methodology that requires ongoing use in certification of each design. The constraints are imposed by the the environment in which we envision our framework being deployed: it may not be possible to have a dedicated team of experts doing theorem proving as full-time jobs. Finally, the loop pipelining transformation we verify are proprietary to the synthesis tools. Therefore, our approach is targeting verification of transformations which are closed-source (and exceedingly complex), thus making traditional program verification techniques unusable. Our approach shows a novel way in which theorem proving can be applied even under those constraints, in concert with automated SEC.

In addition to technical contributions, we see our work as providing an important methodological contribution enabling use of theorem proving in situations where one needs to certify the result of an implementation on which theorem proving cannot be directly applied either because it is closed-source or because it is highly complex: (1) create a reference implementation, perhaps using as much information as available from the actual implementation, in our case information about pipeline intervals, (2) certify this simpler reference implementation with theorem proving, and (3) develop an SEC framework to compare the result of the reference implementation with that of the actual implementation. In addition to making theorem proving applicable on industrial flows without requiring us to certify industial implementations with their full complexity, this approach permits adjusting the algorithm (within limits) to suit mechanical reasoning while still affording comparison with actual synthesized artifacts. We have made liberal use of this "luxury", *e.g.*, we have been continually redefining our superstep construction function to facilitate proof of key structural lemmas of the invariant before settling on the final version. We believe similar approach is applicable in other contexts and may provide effective use of theorem proving within industrial verification flows.

11 Conslusion and Future Work

11.1 Summary

With our work, we have made the following major contributions:

1. *Developed a framework of succinct certified primitives essential to build pipelining algorithms* : Our primitives are essential for developing certified loop pipelining algorithm in behavioral synthesis. This framework can also be extended to certify other pipelining algorithms such as function pipelines.
2. *Designed and certified a reference loop pipelining algorithm* : We utilize our framework of certified primitives as backbone to build our certified loop pipelining algorithm. Since a primitive can only be applied under certain conditions, when certifying the algorithm, we prove that every application of our primitive is under correct conditions and certain assumptions are maintained after the application of a primitive. We also formalize and certify a key invariant for the correspondence between the sequential and pipelined CCDFGs and propose an algorithm for handling branch conditions in pipelines.
3. *Evaluated our algorithm on industrial-strength designs* : We test our algorithm on a variety of designs across different application domains. If our algorithm can generate a pipeline reference model for a design, we can compare it to the pipelined RTL generated by behavioral synthesis tools using SEC. If the SEC passes, we certify the application of loop pipelining transformation is correct. We show that our algorithm can discharge industrial-strength designs.

Our current ACL2 script has 296 definitions and 1012 lemmas, including many lemmas about structural properties of CCDFGs (but not counting those from the false starts).

Since, we have a certified loop pipelining algorithm, we can confidently say that there are no data hazards and executing a sequential loop is same as executing a pipelined loop created using our algorithm. We have tested the pipeline reference model created using our algorithm on a variety of designs across different application domains. This shows that our algorithm is practical and can be used for industrial strength designs with tens of thousands of RTL.

11.2 Future Work

Our work shows that it is possible to develop and certify an industrial-strength loop pipelining algorithm if we can decompose it into succint certifiable primitives. We have already identified and certified these primitives. Our algorithm has components which can identify data hazards based on the given pipeline interval. Then we use our certified primitives to remove those data hazards and create a pipelined implementation.

Function pipelining algorithms also have the same type of data hazards as we have mentioned in loop pipelining algorithms. However, while loop pipelines have a fixed pipeline interval which is known at compile time, function pipelines have a variable pipeline interval for every iteration. So, instead of identifying data hazards at once for every iteration, we would have to call those functions for each iteration. After we have identified the data hazards, we can use our certified primitives to remove those data hazards. We believe that if we can modify the algorithm to identify data hazards, then we can conveniently reuse our certified primitives to certify behaviorally synthesized function pipelines as well.

Bibliography

[1] Mark D. Aagaard, Byron Cook, Nancy A. Day, and Robert B. Jones. *A Framework for Microprocessor Correctness Statements*, pages 433–448. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[2] Yves Bertot, Pierre Castran, Grard (informaticien) Huet, and Christine Paulin-Mohring. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Texts in theoretical computer science. Springer, Berlin, New York, 2004. Donnes complmentaires http://coq.inria.fr.

[3] J. R. Burch and D. L. Dill. Automatic Verification of Pipelined Microprocessor Control. In D. L. Dill, editor, *Proceedings of the 6th International Conference on Computer-Aided Verification (CAV 1994)*, volume 818 of *LNCS*, pages 68–80. Springer-Verlag, 1994.

[4] Cadence. *C-to-Silicon Reference Manual*, 2011.

[5] Calypto. *Catapult Reference Manual*, 2014.

[6] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.*, 13(2):24:1–24:27, September 2013.

[7] R.O Chapman. *Verified high-level synthesis*. PhD thesis, Portland State Univeristy, 1994.

[8] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Satabs: Sat-based predicate abstraction for ansi-c. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 570–574, Berlin, Heidelberg, 2005. Springer-Verlag.

[9] Edmund M. Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *DAC*, pages 368–371. ACM, 2003.

[10] Avra Cohn. *The Notion of Proof in Hardware Verification*, pages 359–374. Springer Netherlands, Dordrecht, 1993.

[11] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. Platform-based behavior-level and system-level synthesis. In *2006 IEEE International SoC Conference*, pages 199–202. IEEE, 2006.

[12] David Cyrluk. Microprocessor verification in pvs - a methodology and simple example. Technical report, 1994.

[13] DavidRussinoff and Matt Kaufmann and Eric Smith and Robert Sumners. Formal Verification of Floating-Point RTL at AMD Using the ACL2 Theorem Prover, 2014.

[14] Gautam Doshi. Understanding the IA-64 architecture. Technical report, August 1999.

[15] Tom Feist. White paper: Vivado design suite. Technical report, Xilinx, June 2012.

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

[17] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, New York, NY, USA, 1993.

[18] Mike Gordon, Juliano Iyoda, Scott Owens, and Konrad Slind. Automatic formal synthesis of hardware from higher order logic. *Electron. Notes Theor. Comput. Sci.*, 145:27–43, January 2006.

[19] Brian T. Graham. *The SECD Microprocessor: A Verification Case Study*. Kluwer Academic Publishers, Boston, MA, 2012.

[20] K. Hao, S. Ray, and F. Xie. Equivalence Checking for Behaviorally Synthesized Pipelines. In P. Groeneveld, D. Sciuto, and S. Hassoun, editors, 49*th International ACM/EDAC/IEEE Design Automation Conference (DAC 2012)*, pages 344–349. ACM, 2012.

[21] K. Hao, F. Xie, S. Ray, and J. Yang. Optimizing equivalence checking for behavioral synthesis. In *Design, Automation and Test in Europe (DATE 2010)*, pages 1500–1505. IEEE, 2010.

[22] Kecheng Hao. *Equivalence Checking for High-Assurance Behavioral Synthesis*. PhD thesis, Portland State Univeristy, 2013.

[23] David S. Hardin. *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[24] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2003.

[25] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[26] Ravi Mohan Hosabettu. *Systematic Verification of Pipelined Microprocessors*. PhD thesis, The University of Utah, 2000.

[27] Alan Hu. High-level vs. rtl combinational equivalence: An introduction. In *ICCD*, pages 274–279. IEEE, 2006.

[28] James K. Huggins and David Van Campenhout. Specification and verification of pipelining in the arm2 risc microprocessor. *ACM Trans. Des.*

Autom. Electron. Syst., 3(4):563–580, October 1998.

[29] I. Moussa and Z. Sugar and R. Suescun and A. A. Jerraya and M. Diaz-Nava and M. Pavesi and S. Crudo and L. Gazzi. Comparing RTL and Behavioral Design Methodologies in the Case of a 2M Transistors ATM Shaper, 1999.

[30] Christian Jacobi. Formal verification of complex out-of-order pipelines by combining model-checking and theorem-proving. 2404:309, 2002.

[31] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *In Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 78–89, 1993.

[32] Roel Jordans and Henk Corporaal. High-level software-pipelining in llvm. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '15, pages 97–100, New York, NY, USA, 2015. ACM.

[33] Daher Kaiss, Silvian Goldenberg, and Zurab Khasidashvili. Seqver : A sequential equivalence verifier for hardware designs. In *24th International Conference on Computer Design (ICCD 2006), 1-4 October 2006, San Jose, CA, USA*, pages 267–273, 2006.

[34] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, Boston, MA, June 2000.

[35] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Boston, MA, June 2000.

[36] Sudipta Kundu, Sorin Lerner, and Rajesh Gupta. *Validating High-Level Synthesis*, pages 459–472. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[37] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. *SIGPLAN Not.*, 44(6):327–337, June 2009.

[38] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO 2004)*, pages 75–84, March 2004.

[39] Raya Leviathan and Amir Pnueli. Validating software pipelining optimizations. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '02, pages 280–287, New York, NY, USA, 2002. ACM.

[40] Jeremy Levitt and Kunle Olukotun. Verifying correct pipeline implementation for microprocessors. In *Proceedings of the 1997 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '97, pages 162–169, Washington, DC, USA, 1997. IEEE Computer Society.

[41] Hanbing Liu and J. Strother Moore. Executable jvm model for analytical reasoning: a study. *Sci. Comput. Program.*, 57(3):253–274, September 2005.

[42] P. Manolios. Correctness of Pipelined Machines. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000)*, volume 1954 of *LNCS*, pages 161–178, Austin, TX, 2000. Springer-Verlag.

[43] Takeshi Matsumoto, Hiroshi Saito, and Masahiro Fujita. Equivalence checking of c programs by locally performing symbolic simulation on dependence graphs. In *Proceedings of the 7th International Symposium on Quality Electronic Design*, ISQED '06, pages 370–375, Washington, DC, USA, 2006. IEEE Computer Society.

[44] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960.

[45] Michael Meredith. *High-Level SystemC Synthesis with Forte's Cynthesizer*, pages 75–97. Springer Netherlands, Dordrecht, 2008.

[46] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[47] D. Puri, S. Ray, K. Hao, and F. Xie. Mechanical certification of loop pipelining transformations: A preview. In G. Klein and R. Gamboa, editors, *4th International Conference on Interactive Theorem Proving (ITP 2014)*, volume 7998 of *LNCS*. Springer, 2014.

[48] S. Ray and J. Bhadra. Abstracting and Verifying Flash Memories. In K. Campbell, editor, *Proceedings of the 9th Non-Volatile Memory Technology Symposium (NVMTS 2008)*, pages 100–104, Pacific Grove, CA, November 2008. IEEE.

[49] S. Ray, K. Hao, F. Xie, and J. Yang. Formal Verification for High-Assurance Behavioral Synthesis. In Z. Liu and A. P. Ravn, editors, *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis (ATVA 2009)*, volume 5799 of *LNCS*, pages 337–351, Macao SAR, China, October 2009. Springer.

[50] S. Ray and W. A. Hunt, Jr. Mechanized Certification of Secure Hardware Designs. In M. S. Abadir, L. Wang, and J. Bhadra, editors, *Proceedings of the 8th International Workshop on Microprocessor Test and Verification, Common Challenges and Solutions (MTV 2007)*, pages 25–32, Austin, TX, December 2007. IEEE Computer Society.

[51] S. Ray and W. A. Hunt, Jr. Connecting Pre-Silicon and Post-silicon Verification. In A. Biere and C. Pixley, editors, *Proceedings of the 9th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2009)*, pages 160–163, Austin, TX, November 2009. IEEE Computer Society.

[52] Hamid Savoj, David Berthelot, Alan Mishchenko, and Robert Brayton. Combinational techniques for sequential equivalence checking. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*,

FMCAD '10, pages 145–150, Austin, TX, 2010. FMCAD Inc.

[53] J. Sawada and W. A. Hunt, Jr. Verification of FM9801: An Out-of-Order Microprocessor Model with Speculative Execution, Exceptions, and Program-Modifying Capability. *Formal Methods in Systems Design (FMSD)*, 20(2):187–222, 2002.

[54] Klaus Schneider. *A Verified Hardware Synthesis of Esterel Programs*, pages 205–214. Springer US, Boston, MA, 2001.

[55] Sudarshan K. Srinivasan. Automatic refinement checking of pipelines with out-of-order execution. *IEEE Transactions on Computers*, 59(undefined):1138–1144, 2010.

[56] J. Tristan and X. Leroy. A Simple, Verified Validator for Software Pipelining. In M. V. Hemenegildo and J. Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*, pages 83–92, January 2010.

[57] 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. *International Journal on Embedded Systems*, pages 134–149, 2005.

[58] Miroslav N. Velev. *Formal Verification of VLIW Microprocessors with Speculative Execution*, pages 296–311. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[59] Miroslav N. Velev and Randal E. Bryant. Formal verification of superscale microprocessors with multicycle functional units, exception, and branch prediction. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 112–117, New York, NY, USA, 2000. ACM.

[60] VP9 Video Hardware RTL. WebM. VP9 Video Hardware RTL. `http://www.webmproject.org/hardware/vp9/`. Accessed: September 11, 2016.

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

[62] Jin Yang and Carl-Johan H. Seger. Introduction to generalized symbolic trajectory evaluation. In *ICCD*, pages 360–367. IEEE Computer Society, 2001.

[63] Z. Yang, K. Hao, K. Cong, F. Xie, and S. Ray. Equivalence Checking for Compiler Transformations in Behavioral Synthesis. In 31*st International Conference on Computer Design (ICCD 2013)*, pages 491–494, 2013.

[64] Zhenkun Yang, Kecheng Hao, Kai Cong, Li Lei, Sandip Ray, and Fei Xie. Scalable certification framework for behavioral synthesis front-end. In *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014*, pages 149:1–149:6, 2014.

[65] Zhenkun Yang, Kecheng Hao, Kai Cong, Li Lei, Sandip Ray, and Fei Xie. Validating scheduling transformation for behavioral synthesis. In *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016,*

Dresden, Germany, March 14-18, 2016, pages 1652–1657, 2016.

[66] Zhenkun Yang, Sandip Ray, Kecheng Hao, and Fei Xie. Handling design and implementation optimizations in equivalence checking for behavioral synthesis. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 117:1–117:6, New York, NY, USA, 2013. ACM.