Certifying Loop Pipelining Transformations in Behavioral Synthesis

by

Disha Puri

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Fei Xie, Chair
Jingke Li
Suresh Singh
Sandip Ray
Fu Li

Portland State University
2016

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.

The result of this dissertation is a framework for creating certified pipelining algorithms utilizing a mechanical theorem prover. Using this framework, we have developed a certified loop pipelining algorithm. 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.

DEDICATION

To,

To

ACKNOWLEDGMENTS

I acknowledge ...

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 DISSERTATION SUMMARY

Developing a certified loop pipelining algorithm is a complex problem. We have developed a certified loop pipelining algorithm for behavioral synthesis by proper application of theorem proving techniques. The result of this dissertation is a framework of certified pipelining primitives which we believe are essential in developing any such pipelining algorithm. We systematically build a loop pipelining algorithm from ground up using these primitives and certify this algorithm.

1.2 IMPORTANCE OF THIS RESEARCH

Behavioral synthesis is the process of synthesizing an Electronic System-level (ESL) specification of a hardware design into a Register-Transfer Level (RTL) implementation. The idea of ESL is to raise the design abstraction by specifying the high-level, functional behavior of the hardware design. Designs are typically specified in a language such as C, C++, or SystemC. The approach is promising since the user is relieved of developing and optimizing low-level implementations. 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. Studies have shown that ESL

reduces the design effort by 50% or more while attaining excellent performance results [8]. Nevertheless, and in spite of availability of several commercial behavioral synthesis tools [2,4,19], the adoption of the approach in main-stream hardware development for microprocessor and SoC design companies has been tentative. One key reason is the lack of designers' confidence that the synthesized RTL indeed corresponds to the ESL specification. To satisfy the power and performance demands of modern applications, a behavioral synthesis tool applies hundreds of transformations, many involving complex and aggressive optimizations that require subtle and delicate invariants. Therefore, it is not surprising that these transformations contain bugs, which can result in errors in synthesized hardware. Thus it is critical to develop mechanized support for certifying the semantic equivalence between ESL and RTL designs. However, the large difference in abstraction between the two representations makes such certification non-trivial.

Loop pipelining is a critical transformation in behavioral synthesis. The goal of this transformation is to increase throughput and reduce latency of the synthesized hardware by allowing temporal overlap of successive loop iterations. It is performed by most state-of-the-art commercial synthesis tools [4, 17, 19]. Unfortunately, it is also one of the most complex transformations [16], and posts challenges for developing viable automated certification techniques. In particular, efficient overlapping of loop iterations creates a design that differs significantly from its sequential counterpart. Thus there is little correspondence in internal signals between the two designs, making it infeasible to compare them through automated sequential equivalence checking (SEC). 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. 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 by Hao et al. [6]. The most critical and complex component of their approach (c.f. Section 2.3) 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. Hao et al. showed the viability of their approach by proposing such a loop pipelining algorithm. However, their algorithm was not certified and is in fact, as we discuss later in Chapter 4, not easily certifiable. Therefore, this dissertation on developing a certified loop pipelining algorithm using our framework of certified primitives is important to facilitating formal verification of behaviorally synthesized pipeline designs.

1.3 PROBLEM STATEMENT

Certifying an algorithm especially as complex as loop pipelining is not easy by any known conventional methods. To develop a certified loop pipelining algorithm in behavioral synthesis, we need to address the following key challenges brought about by the semantic gap between the sequential and pipeline designs.

– *Formalizing an invariant that links loop in a sequential design with loop in the corresponding pipelined design.* A sequential loop executes its iterations in sequence. The previous loop iteration is fully complete before the next iteration.

A pipelined design, however, overlaps the consecutive iterations of a given design based on pipeline interval. As a result, a loop in the pipelined design executes statements from different iterations of the corresponding sequential loop design. Identifying a provable inductive invariant that links the sequential loop with the pipelined loop is, therefore, a major challenge.

– *Identifying and certifying underlying primitives in a loop pipelining algorithm.* Certifying a complete loop pipelining algorithm at once, directly on the basis of input and expected output, would require a complex invariant proof and is also unnecessary in our case. We have taken an alternate approach of decomposing the algorithm into certifiable primitives. We propose that if each primitive maintains an invariant that the semantic run of the intermediate representation before and after application of the primitive is same, we can prove that the algorithm also maintains the invariant. This approach, however, requires a crisp understanding of the essential steps involved in developing a pipeline loop from a sequential loop. We need to succinctly identify primitives which maintain the given invariant and are also certifiable by theorem proving. Each primitive would require a systematic approach for its proof.

– *Certifying the complete loop pipelining algorithm based on certified primitives.* Although, the primitives act as backbone for our algorithm, their certification alone does not automatically certify the entire algorithm. We need to identify the conditions under which a primitive is correct and make sure that every application of the primitive in the algorithm has the required assumptions.

1.4 OVERVIEW OF OUR APPROACH

We believe that a certified loop pipelining algorithm can be developed by systematic application of theorem proving techniques. Our approach is based on a realization that in order to generate a pipeline loop design from a sequential loop design, there are two broad steps: (1) identification and removal of data hazards, and (2) overlapping the executions of subsequent iterations after the removal of data hazards. To remove data hazards and to overlap iterations, we have built a framework of succinct provable pipelining primitives. Certification of each primitive requires separate careful reasoning in a mechanical theorem prover which we describe later in Chapter 6. Certifying an application of a primitive in the context of the algorithm further involves ensuring that addition of any primitive does not alter the underlying assumptions in the syntax, for example, if we assume there are no return statements in a given representation, applying any primitive should also maintain that assumption. We use these primitives as a backbone to build our loop pipelining algorithm with distinct decomposable components one step at a time. Each component satisfies the invariant that the semantic runs of intermediate representations before and after the component are the same. We elaborate on our approach later in Chapter 5.

1.5 VIABILITY OF OUR APPROACH

We have defined the syntax and semantics of intermediate design representation in ACL2 [13]. We have developed a framework of pipelining primitives essential for all pipelining algorithms. We have formalized and certified all of our primitives in ACL2 theorem prover. We have developed our loop pipelining algorithm from ground up using this framework.

We have also identified a unique invariant which proves that executing overlapped iterations is equivalent to executing sequential iterations. It differs from a typical invariant used for correctness of pipelined systems in that it explicitly specifies the correspondence between the sequential and pipelined programs at each transition. We elaborate on our invariant in Chapter 6. We have proved that our algorithm satisfies this invariant.

– Think more about this paragraph – We have certified the algorithm end-to-end which means that we have proved that running a sequential loop is same as ... pipeline.. given certain restrictions .. We elaborate on the proof in our proof sketch in Chapter 6. Our proof sketch shows that our primitives are sufficient and essential and that we can build a certified loop pipelining algorithm from ground up using our framework.

The major contributions of our dissertation are:

– Identifying the key provable primitives essential in pipelining algorithms for behavioral synthesis and certifying these primitives in ACL2 theorem prover;

– Formalizing an invariant to link the sequential loop before pipelining with the pipelined loop;

– Developing our own executable loop pipelining algorithm in ACL2 using those primitives and certifying this algorithm using ACL2 theorem prover;

– Testing our certified loop pipelining algorithm on industrial-strength designs

1.6   NOVELTY OF OUR APPROACH

There has been a significant amount of work on pipeline verification [1, 11, 15, 18]. However, most of pipeline verification research has focused on architectural

pipelines, in particular pipelined microprocessors. 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. 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*. Abstraction techniques such as MAETT [15] does not apply to our case and we have come up with a very different invariant.

Our work has analogues with recent work on software pipelines. In particular, Tristan and Leroy [16] present a verified translation validator for software loop pipelines. Their algorithm *checks* that a pipelined loop is equivalent to its sequential counterpart. However, their correctness statement is contingent upon the equivalence of symbolic simulation of the two designs; consequently they do not statically identify data hazards.

Our project is also somewhat different from the traditional applications of ACL2 in hardware verification. 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. We believe our approach shows a novel way in which theorem proving can be applied even under those constraints, in concert with automated SEC.

1.7 OUTLINE

The remainder of this dissertation is organized as follows. Chapter 2 provides background on the overall project and explains the context of our theorem proving work. Chapter 3 discusses our formalization of the intermediate representations used in behavioral synthesis. We also discuss the correctness statement for loop pipelining algorithms. Chapter 4 discusses an earlier proposed algorithm and the challenges in certifying an algorithm not written with certification in mind. Chapter 5 discusses our framework and a certified loop pipelining algorithm we have developed using the framework. Chapter 6 provides a proof sketch for our algorithm. Chapter 7 provides evaluation of robustness and scalability of our algorithm on industrial-strength designs. We then conclude with the major contributions of this dissertation and future work in Chapter 8.

Chapter 2

BACKGROUND AND CONTEXT

In this Chapter, we discuss the overall project of verifying behaviorally synthesized designs, and how the certification of loop pipelining fits into this project. The reader interested in a thorough understanding of other components of the project is welcome to review the prior publications [7, 14].

2.1 BEHAVIORAL SYNTHESIS

Behavioral synthesis is an automated compilation process from ESL description to RTL. Analogous to a regular compiler, a behavioral synthesis tool [3, 5, 17] performs the standard lexical and syntax analysis to generate an intermediate representation (IR). The IR is then subjected to a number of transformations, which can be categorized into three phases.

– **Compiler Transformations:** These include typical compiler operations, *e.g.,* dead-code elimination, constant propagation, loop unrolling, etc. A design may undergo hundreds of compiler transformations.

– **Scheduling Transformations:** Scheduling entails computing for each operation the clock cycle of its execution, accounting for hardware resource constraints and control/data flow. Loop pipelining, the focus of our dissertation thesis, is a component of this phase.

– **Resource Allocation and Control Synthesis:** This phase involves mapping a hardware resource to each operation (the "**+**" operation may be mapped to a hardware adder), allocating registers to variables, and generating a controlling finite-state machine to implement the schedule.

After the three phases above, the design can be expressed in RTL. The synthesized RTL may be subjected to further manual tweaks to optimize for area, power, etc.

2.2 CERTIFICATION FRAMEWORK

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.

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 [7,14]. 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 [20]. The idea then is to obtain the sequence of intermediate representations $IR_0, \ldots, IR_n$ 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.

2.3 A REFERENCE PIPELINE APPROACH

To develop a specialized approach for pipelines, a key observation is that while the transformation *implementation* is inaccessible to us, commercial synthesis tools typically generates a report specifying pipeline parameters (pipeline interval, number of loop iterations pipelined, etc.). The approach then is to develop an algorithm that takes as inputs these parameters and an IR $\mathcal{C}$ for the design before pipelining, and generates a *reference pipelined IR $\mathcal{P}$*. Note that this algorithm would be much simpler than that employed during synthesis; while the former includes advanced heuristics to *compute* pipeline parameters (like pipeline interval, number of iterations pipelined etc.), this algorithm would merely use the values provided by its report. To certify a synthesized RTL with pipelines, it is sufficient to (1) check that the *given algorithm* can generate a pipeline $\mathcal{P}$ for the parameters reported by synthesis, (2) use SEC to compare $\mathcal{P}$ with the synthesized RTL, and (3) prove (using theorem proving) the correctness of this algorithm.

A previous work [6] justified the viability of steps 1 and 2 above; such a reference pipeline generation algorithm was developed and used to successfully compare a variety of pipelined designs across various application domains. This suggested

that the approach of using a reference implementation is viable for certifying industrial synthesized pipelines. However, a key (and perhaps the most complicated) component of the approach was missing. The algorithm was not verified (indeed, not implemented in a formal language), rendering the "certification" flow unsound.

The unsoundness mentioned above is not just an academic notion. In fact, merely by going through the formalization process and thinking about necessary invariants, we have already found a bug in the implementation of the algorithm (Discuss with Sandip Sir). Thus it is critical to develop a mechanized proof of correctness for this implementation. Unfortunately, it is not easy to verify the original pipeline generation algorithm as written. Its author was an expert in behavioral synthesis but not in program verification or theorem proving; consequently, the algorithm, while simpler than the one implemented in a synthesis tool, was still a highly complex piece of code. In particular, since it was not written with correctness certification in mind, it is difficult to decompose the algorithm into manageable pieces with nice invariants.

One way to address this problem is to "buckle down" and verify the pipeline generation algorithm (and fixing the bugs found in the process). However, a key insight in our case is that we can get away without verifying such a complex implementation. After all, there is nothing "sacred" about this specific algorithm for pipeline generation: given the steps described above, *any* verifiable pipeline generation algorithm would suffice.[1] Thus the approach of our dissertation can be viewed as a rational deconstruction of the pipeline synthesis algorithm of the previous work. We identify the key invariant that we need to maintain for proving computational equivalence between the pipelined and un-pipelined loops and

---

[1]Note that our algorithm *must* create a pipeline in accordance with the pipeline parameters obtained from the behavioral synthesis tools; otherwise we may fail to certify correct designs. However, in practice, we have not found this to be a problem.

design an algorithm to explicitly maintain that invariant.

We discuss the previously proposed algorithm in Chapter 4 such that we can draw a comparison and better understand the differences in our implementation of loop pipelining algorithm due to our need to formally certify it.

Chapter 3

FORMALIZATION

3.1 INTERMEDIATE REPRESENTATION: CCDFG

In order to formalize and prove the correspondence between pipelined and un-pipelined IRs, a first step is to define a formalization of the IRs themselves. We call our formalization of IRs *Clocked Control Data Flow Graph* (CCDFG). An informal description of CCDFG has been provided before [14]. It can be best viewed as a traditional control/data flow graph used by most compilers, augmented with a schedule. Control flow is broken into basic blocks. Instructions are grouped into microsteps which can be executed concurrently. A scheduling step is a group of microsteps which can be executed in a single clock cycle. The 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 each scheduling step. The underlying language we use is the LLVM [9]. It is a popular compiler infrastructure for many behavioral synthesis tools and includes an assembly language front-end. At the point of this writing we support a limited subset of LLVM, which however is sufficient to handle all the designs we have seen. Instructions supported include assignment, load, store, bounded arithmetic, bit vectors, arrays, and pointer manipulation instructions. We define the syntax of each type of statement by defining

an ACL2 predicate. For example, in our syntax, an assignment statement can be expressed as a list of a variable and an expression.

An expression can further be of multiple types, load expression (loading the value of a variable from memory), add expression (addition of two variables), xor expression (xor of two variables) etc., where each expression includes the operation applied to the appropriate number of arguments.

We provide semantics to these instructions through a state-based operational formalization as is common with ACL2. We define the notion of a CCDFG state, which includes the states of the variables, memory, pointers, etc. Then we define the semantics of each instruction by specifying how it changes the state. Thus, for an assignment statement we will have a function `execute-assignment` that specifies the effect of executing the assignment statement on a CCDFG state.

Defining the semantics of most supported statements is straightforward, with one exception. The exception is the so-called "ϕ-construct" available in LLVM. A ϕ-construct is a list of ϕ-statements. A ϕ-statement is $v := \phi[\sigma, bb1][\tau, bb2]$, where v is a variable, σ and τ are expressions, and $bb1$ and $bb2$ are basic blocks: if it is reached from $bb1$ then it is the same as the assignment statement $v := \sigma$; if reached from $bb2$, it is the same as $v := \tau$; the meaning is undefined otherwise. The construct is complex since the effect of executing this statement on a CCDFG state s depends not only on the state s but also on how s is reached by the control flow. Unfortunately, ϕ-statements are required in loop designs — they are used to evaluate the value of loop carried dependencies. Consequently, the complexity induced by this instruction cannot be avoided.
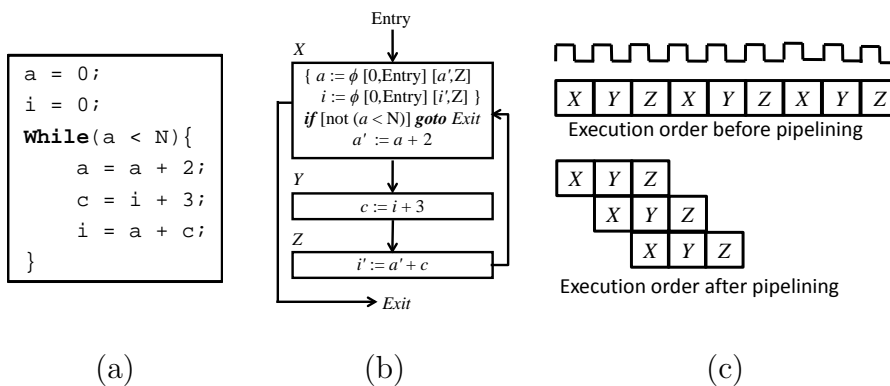
Figure 3.1: (a) Loop in C; (b) Loop CCDFG before pipelining (**Note for Disha: create a better figure. Discuss with Zhenkun**) ; (c) Pipelining increases throughput

3.2 CORRECTNESS OF LOOP PIPELINING

For the purposes of this paper, a *pipelinable loop* is a loop with the following restrictions [6]:

1. no nested loop;

2. only one *Entry* and one *Exit* block; and

3. no branching between the scheduling steps.

Discuss with Sandip Sir: We should mention that there is only one conditional branch and one unconditional branch related to loop, others can be handle via previous compiler transformations. Also, there is only one phi statement.

These restrictions are not meant to simplify the problem, but 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 previous compiler transformation) in order to pipeline the outer loop.

Figure 3.1(a) illustrates the C code (ESL description) for a loop. The C code does not have a schedule or the concept of a clock cycle. Figure 3.1(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 $Exit$. Note that there is a ϕ-statement in the first scheduling step of the loop. This ϕ-statement accounts for those variables whose values are dependent on the variables evaluated in a previous iteration.

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). One iteration of the sequential design takes three clock cycles. Observe in Figure 3.1(c) that with the pipeline interval of one, 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 overall latency.

Correctness Statement: Let L be a loop in CCDFG C, and let L_α be the pipelined implementation generated by a pipeline algorithm using pipeline parameters α. Let V be the set of variables 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) ensures that variables in C that are not part of the loop are not affected by L_α. The value of any new variables introduced by the algorithm in f' are irrelevant since they are not accessed subsequently.

Chapter 4

RESEARCH CHALLENGES

4.1 PREVIOUSLY PROPOSED ALGORITHM

Given a set of microsteps M, a set of edges E, and a schedule S of a sequential CCDFG, and pipeline parameters (pipeline interval I and number of scheduling steps in the loop N), the algorithm [6] generates new microsteps, edges and schedule for the pipelined CCDFG. Values of these inputs are readily available from intermediate feedback reports from the behavioral synthesis tool. Given CCDFG C, this algorithm replaces each loop L in C with the pipelined refinement of L. The steps of the algorithm are explained below with the help of the example introduced earlier in Figure 3.1(a).

Generate scheduling steps: Figure 4.1(a) shows the addition of scheduling steps of new iterations of a loop according to the pipeline interval. Here, the loop has three scheduling steps and a pipeline interval of one. The new scheduling steps are generated till the pipeline is full. This step generates a new schedule. For the sake of easily identifying microsteps later, every microstep in the unrolled loop is given a unique name even though a microstep is the same across different iterations. We would later see how this trivial step affects the theorem proving process adversely.

Generate shadow registers: In order to pipeline a loop, we have to remove data hazards. We first identify all variables that can be overwritten when executing overlapped iterations and then introduce new temporary variables called shadow
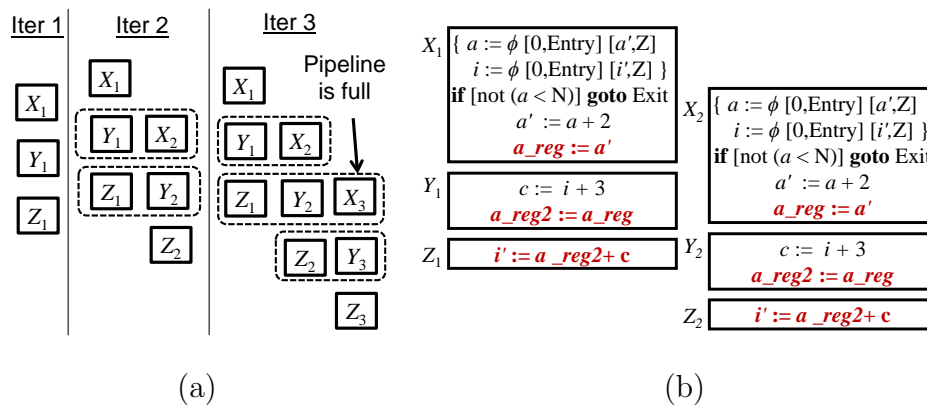
Figure 4.1: **Note to Disha: Create better figures** (a) Generate scheduling steps. After the pipeline is full, adding new iteration simply repeats the pipeline full stage. (b) Generate shadow registers.

registers to avoid overwrites. In Figure 4.1(b), the value of a' will be overwritten in X_2 before Z_1 can read it. So, we introduce a new pipeline register a_reg which gets assigned the value of a' and replace subsequent reads of a' with reads of a_reg. In the next scheduling step, a_reg2 is assigned the value of a_reg and further subsequent reads of a' are replaced with reads of a_reg2. Introducing shadow registers in such a way removes the possibility of data hazard as the value of the old variable is stored in a new temporary shadow variable every scheduling step.

Generate edges: The algorithm adds the edges for data and control flow as shown in Figure 4.2(a). Control edges are edges from one scheduling superstep to the next. Data forwarding edges forward data from one scheduling step to the next in a single scheduling superstep. Loop edge denotes the repetition of the pipeline full stage. Exit edges are from the scheduling steps to the *Exit* block. Note that a pipelinable loop has only one *Exit* block.
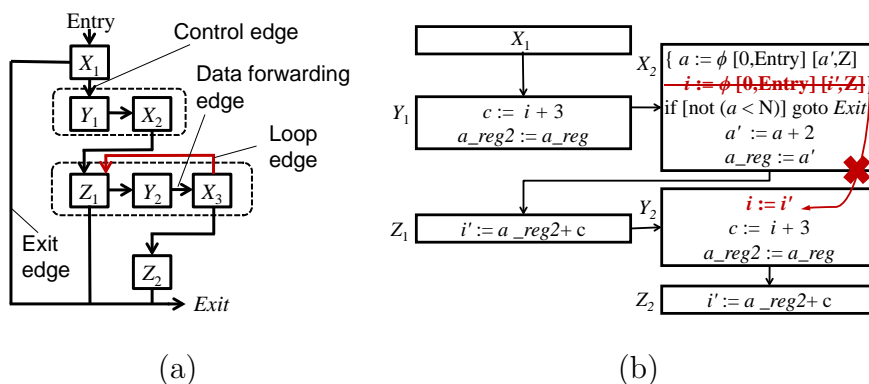
Figure 4.2: (a) Generate edges for pipelined CCDFG (b) Data propagation.

Generate data propagation: In Figure 4.2(b), we require the value of i' in X_2 to execute the following statement:

$$i := \phi[0, Entry][i', Z]$$

But, if we execute X_2 before executing Z_1, the value of i' has not yet been produced. To avoid such a situation, the algorithm relocates the assignment statement $i := i'$ to Y_2 just before it needs to be read. The assignment statement $i := i'$ is obtained from the ϕ-statement $i := \phi[0, Entry][i', Z]$ since in the sequential execution, we would have entered X_2 from Z block.

This relocation is however incorrect. Note that we have moved the assignment statement across the conditional branch.

$$if[not(i < N)]gotoExit$$

If we would have executed the sequential CCDFG and the exit condition in the second iteration is true, we would have executed the microstep $i := i'$ before exit. However, now in the pipelined CCDFG, if we exit in X_2, we do not execute the microstep $i := i'$. Hence, the value of i is not same in the sequential and pipelined CCDFGs. This is a bug that we found in this algorithm: it incorrectly allows

relocation of microsteps across conditional branch statements. We realized that the authors of the previous proposed algorithm had not accounted for the exit conditions while testing the algorithm. As a result, this bug was not found prior to our formalization. We provide a fix to this bug in our loop pipelining algorithm in Section 5.

Even after we fix the bug in the proposed algorithm, it is not easy to verify this algorithm as it is. We elaborate below on the challenges in verifying the proposed algorithm and in general, any complex algorithm not written while keeping theorem proving in mind.

4.2   CHALLENGES IN VERIFYING PREVIOUS ALGORITHM

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.

For instance, to prove the correctness statement in the previous algorithm, we want to prove that the complete algorithm follows the invariant that the semantic run of input CCDFG is equal to semantically running the output CCDFG. Since the algorithm is composed of four concrete steps – generate scheduling steps, add

shadow register, add edges and data propagation, we intuitively expect the individual steps or at least a combination of steps in sequence to follow this invariant. However, since the algorithm has not been designed keeping theorem proving in mind, that is not the case. For example, if we consider the first step of the proposed algorithm – **generating new scheduling steps** by overlapping executions of an unrolled loop, we know that the semantic run of the sequential scheduling steps is not the same as the run of new scheduling steps unless we prove that there are no data hazards. But, data hazards are not completely eliminated till the last step of the algorithm. Note, that the complete algorithm does follow the invariant as expected, but reasoning about the structure of the complete algorithm at once is not easy.

In addition, the previous algorithm does not take a CCDFG as input, but rather works with microsteps, edges and schedule of a CCDFG. That can get tricky because to analyze the semantic run of a CCDFG, we have to now simultaneously analyze all the three inputs. In our approach, we work with the CCDFG itself and define the run of a CCDFG semantically.

Furthermore, the previous algorithm initially unrolls the loop and later adds a back edge to mimic the pipeline full stage. Unrolling loop with unique names for each microstep makes it easier to design the algorithm but formally we lose the notion that each microstep is part of a loop and not different in each iteration. Besides, unrolling loop makes it very difficult to reason for the correctness of the back edge in the full pipeline stage.

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 [10].

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, this dissertation focuses 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.

Chapter 5

OUR APPROACH

Pipeline synthesis is based on the key observation that execution of successive iterations can be overlapped as long as no data hazard is introduced. Thus, the two main activities of a pipeline synthesis algorithm are to (1) identify and remove possible hazards, and (2) overlap the successive iterations according to the pipeline interval. In our case, the identification of data hazards is simplified since the synthesis tool provides a pipeline interval; thus, instead of *discovering* a pipeline interval ourselves so that no hazard is introduced, we merely need to work with the provided interval. We have developed a framework of certified pipelining primitives which allows us, among other things, to prevent possible data hazards. Our framework also provides a primitive to overlap successive iterations. We now discuss the framework in detail.

5.1   FRAMEWORK OF PROVABLE PIPELINING PRIMITIVES

We believe that the following primitives are 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
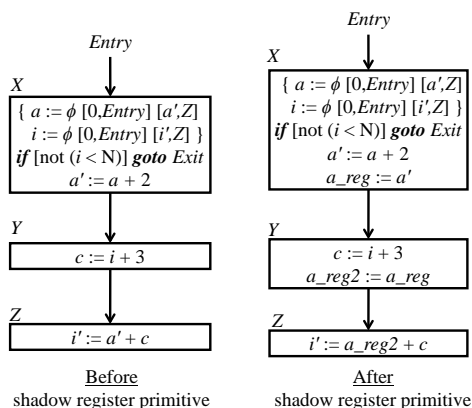
Figure 5.1: Shadow Register Primitive

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 explicitly 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. In Figure 5.1, if we assign a shadow register a_reg value of a' at the end of X block, shadow register a_reg2 value of a_reg in Y and replace the read occurence of a'
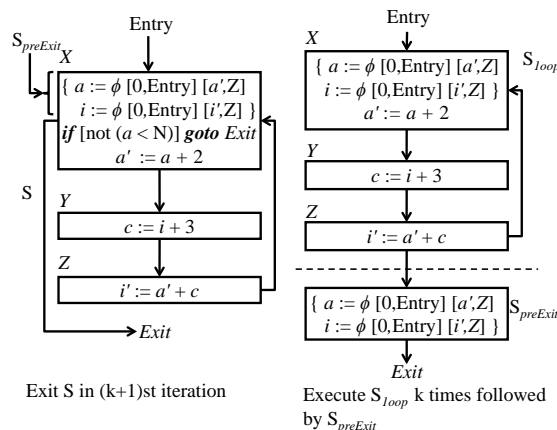
Figure 5.2: Conditional Branch Primitive

in Z with a_reg2, the sequential execution remains same. But, because of the addition of these shadow registers, the value of a' is stored in a new temporary variable in every new scheduling step which prevents data hazards.

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.

Conditional branch primitive – Branch instructions are required to determine the control flow. However, reasoning semantically about conditional branch instructions in a loop everytime we apply a primitive can make proof very complex. We have to always keep track of condition variable to make sure that loop

exits if the condition is true. 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 conditional branch primitive (c.f. Figure 5.2), 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 $S_{preExit}$.

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.

5.2 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.

Algorithm 1 Certifiable loop pipeline

1: **procedure** PIPELINELOOP(S, I)

2: $\quad S_1 \leftarrow RemoveBranches(S)$

3: $\quad S_2 \leftarrow UnrollLoopOnce(S_1)$

4: $\quad S_3 \leftarrow \phi - Elimination(S_2).$

5: $\quad S_4 \leftarrow DataPropagation(S_3, I).$

6: $\quad S_5 \leftarrow GenerateShadowRegisters(S_4, I).$

7: $\quad S_6 \leftarrow SuperstepConstruction(S_5, I).$

8: $\quad P \leftarrow AddBranches(S_6)$

9: $\quad$ **return** $(P).$

10: **end procedure**

Below we describe the steps to convert a sequential loop CCDFG to a pipelined loop CCDFG in detail:

Remove Branches: We apply the conditional branch primitive on S (c.f. Figure 5.3(a)) to remove the conditional branch by explicitly defining the control flow in S. The output is a sequence of two CCDFG's S_{loop} and S_{Exit} connected through an edge as shown in Figure 5.3(b). Note, that S_{loop} does not contain the conditional branch originally present in S. Executing S such that S exits in the $(k+1)$st iteration is semantically same as executing S_{loop} k times followed by $S_{preExit}$.

Unroll Loop Once: We have already established that the first iteration behaves differently than the rest of the iterations due to ϕ-construct. So, here we simply unroll the loop S_{loop} once. We call the first iteration S_{pre} as shown in Figure 5.3(c).

ϕ-elimination: We apply the ϕ-elimination primitive on S_{pre}, S_{loop} and S_{Exit} to
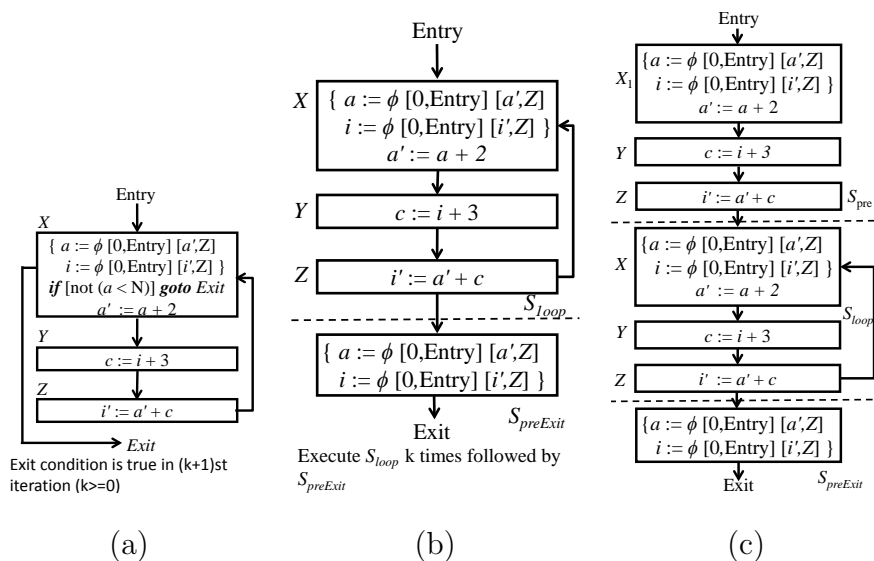
Figure 5.3: (a) Sequential CCDFG with conditional branch (b) Sequential CCDFG without conditional branch. Note the addition of S_{Exit} to explicitly define the control flow (c) Unrolling the loop once to separate the first iteration

return a CCDFG in which all the ϕ-statements have been replaced with their corresponding assignment statements. Figure 5.4(a) shows the CCDFG after applying the ϕ-elimination primitive. 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, $RelocateMSteps$ works in two steps. It first relocates the microstep which reads the variable in an iteration to the starting of S_{loop}. This step can be proved by the interchange primitive since we have already established that the value has not been written yet so there are no read write hazards in between. In the next step, we relocate the microstep to the end of S_{loop}. Note, to maintain the invariant that executing CCDFG semantically before and after this relocation is the same, we need to add the microstep at the end of S_{pre} as well and remove it from S_{Exit}. 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_{Exit}, 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 5.3(c) we found that the loop carried dependency i' in X would create a conflict when we would move X before Z while pipelining. So, first we relocate the microstep $i := i'$ to the beginning of S_{loop} using interchange primitive in Figure 5.4(b). Then, we move the microstep to end of S_{pre} and S_{loop} and remove the microstep from S_{Exit} in Figure 5.4(c). Note that this preserves semantic equivalence.

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

---

Algorithm 2 Data propagation

1: **procedure** DATAPROPOGRATION(L)

2: $\quad msteps \leftarrow GetLoopCarriedDependencies(L).$

3: $\quad$ **for each** mstep **in** msteps **do**

4: $\quad\quad$ **if** $CheckConflict(L, mstep, N, I) \neq 0$ **then**

5: $\quad\quad\quad L \leftarrow RelocateMStep(L, mstep).$

6: $\quad\quad$ **end if**

7: $\quad$ **end for**

8: $\quad$ **return** $(L).$

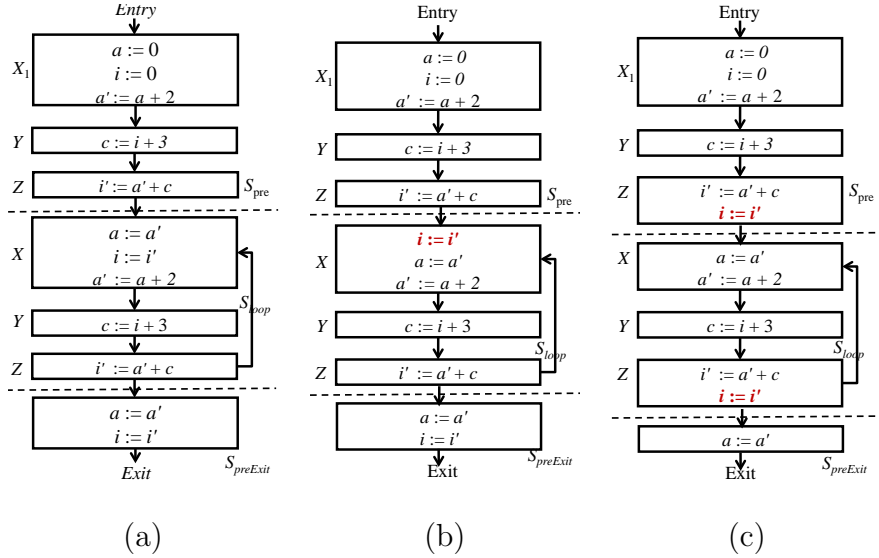9: **end procedure**

Figure 5.4: (a) After ϕ-removal transformation (b) Data propagation - first step (c) Data Propagation - second step

I, 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 5.5 (a), we introduce a shadow register a_reg in X and a_reg2 in Y.

Algorithm 3 Generate shadow registers

1: **procedure** GENERATESHADOWREGISTERS(L)

2:     $V \leftarrow GetAllVariables(L)$.

3:     **for each** v **in** V **do**

4:         $w_v \leftarrow WriteVariable(v, L)$.

5:         $r_v \leftarrow LastReadVariable(v, L)$.

6:         **if** $RequireShadowRegister(r_v, w_v, I) \neq 0$ **then**

7:             $L \leftarrow AddShadowRegister(w_v, L)$.

8:         **end if**

9:     **end for**

10:    **return** (L).

11: **end procedure**

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
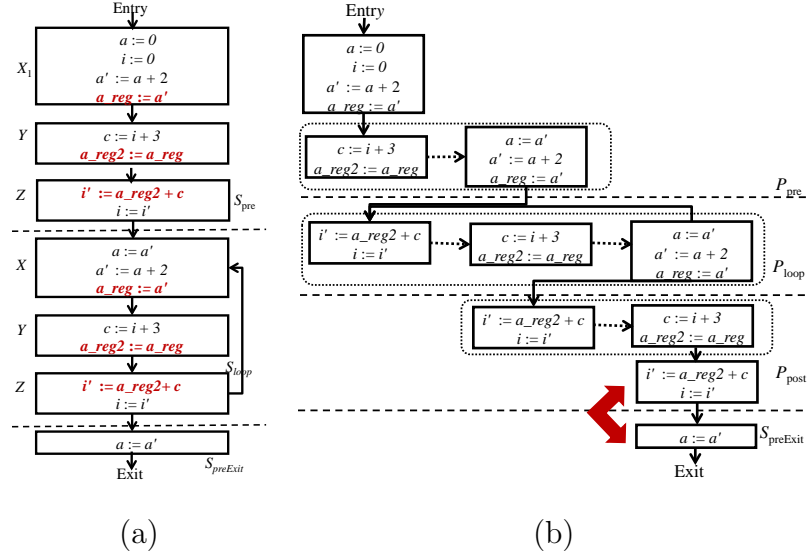
Figure 5.5: (a) After shadow register transformation (b) After superstep construction

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 5.5 (b). 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.

Add Branches: We remove S_{Exit} and add the conditional branches by using the interchange primitive and the conditional branch primtive. We have not fully formalized this step yet. But, the idea can be explained as follows. Note in Figure 5.5(b), if there are no read write hazards in between the last scheduling step Z of P_{post} and S_{Exit}, we can interchange them using interchange primtive. Now recall from the conditional branch primitive that if there is a loop structure,
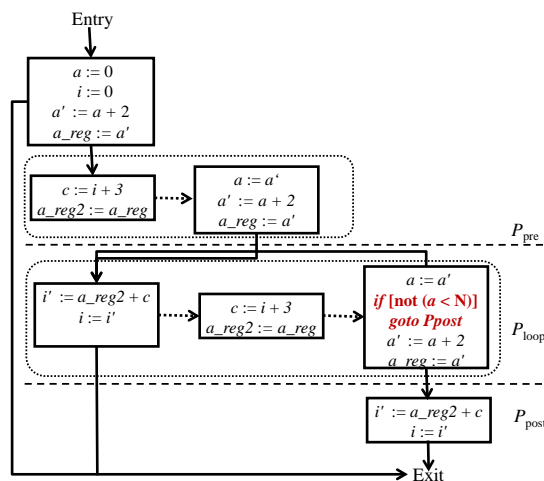
Figure 5.6: Final Pipelined CCDFG

here P_{loop} followed by a collection of microsteps from the beginning of P_{loop} in sequence, say P_{Exit} (here, a collection of Z, Y and S_{Exit}), then we can add an exit conditional branch in P_{loop} after the microsteps P_{Exit} which exits the loop and points to the next scheduling step after the loop if the exit condition is true. We can add the conditional branch as shown in Figure 5.6. We expect that when we formalize and prove this transformation, we would have to define more conditions on the variables being evaluated at the exit conditional branch. Later, we would also formalize adding the exit conditional branches on the iterations when the number of iterations is the sequential loop at run time is less than three in this example, which means that we exit even before we are in the pipeline loop.

Chapter 6

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.

6.1 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 execution of a ϕ-construct is the same as executing the corresponding assignment statements. Note that this is not trivial since given a microstep in a scheduling step containing the ϕ-construct, the algorithm has to use static analysis to deduce the previous scheduling step.

Shadow register primitive: We prove that adding a shadow register microstep $x\_reg = x$ does not change the value of any variable except the shadow variable. Also, we prove that now since value of x_reg would be equal to value of x, executing a statement which reads x has the same effect on the state as executing a statement which reads x_reg till the next write of x. We determine the variables read and written in a statement by analyzing the execution semantics.

Interchange primitive: We prove that we can interchange two scheduling steps which do not have read-write conflict. Given an initial state, the state after executing scheduling steps 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 scheduling steps 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.

Conditional branch primitive: The proof of the primitive follows from the definition of conditional branch primtive itself as explained in Section 5. This proof would involve semantically analyzing the branch statements and induction on the steps of the CCCDG.

Superstep construction primitive: This primitive is proved using the interchange primitive and our key invariant described in detail below.

6.2   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. The relation can be informally paraphrased as follows [12].

> Let S be a sequential loop and P be the pipelined loop generated from our algorithm. The pipelined loop after superstep construction consists of three stages before $S_{preExit}$ as depicted in Figure 5.5(b): prologue P_{pre}, full stage P_{loop}, and epilogue P_{post}. Let s_l be any state of P poised

to execute P_{loop}, and let k be any number such that the loop of P is not exited in k iterations from s_l. Then executing P_{pre} followed by k iterations of P_{loop} is equivalent to executing first iteration of S, say S_1 followed by $(k-1)$ iterations of S together with a collection of "partially completed" iterations of S.[1]

The partially completed iterations can be determined by the length of the first iteration in P_{pre} and the pipeline interval. Suppose the length of the first iteration in P_{pre} is `m` and the pipeline interval is `i`. Note that we can calculate the value of `m` based on the number of scheduling steps in a CCDFG and the pipeline interval. The partially completed iterations mean m scheduling steps of S followed by $(m-i)$ scheduling steps of S, by $(m-2i)$ scheduling steps of S, etc. while $(m-ni)$ is positive.

In our example, `m` is 2 and `i` is 1. The invariant implies that starting from the same initial state, executing P_{pre} and `k` iterations of P_{loop} is the same as executing k iterations of S, followed by $m = 2$ scheduling steps of S, followed by $(m-i) = 1$ scheduling steps of S.

As is standard with proofs involving invariants, there are two obligations to prove the correctness, *viz.*, that it is indeed an invariant, and that its invariance is sufficient to imply the desired correctness theorem. Here we give a sense of our envisioned proof.

Our invariant is defined specifically to make the proof sufficiency straightforward. Equivalence of CCDFG states of P and S follows from the invariant by noting that the epilogue P_{post} exactly constitutes the incomplete scheduling steps of S specified by the invariant (cf. Figure 6.1).

---

[1]The formalization actually characterizes each incomplete iteration, *e.g.*, if the pipeline includes d iterations and successive iterations are introduced in consecutive clock cycles, then the i-th iteration has $i-1$ incomplete scheduling steps.
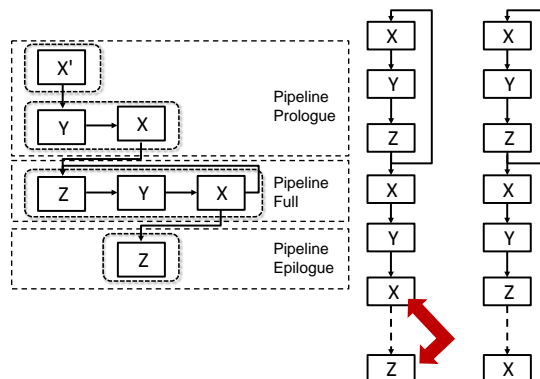
Figure 6.1: Correctness of invariant implies the correctness statement

The proof of invariance of this predicate is, of course, the main "work horse" in this exercise. The proof depends on our interchange primitive which in turn is based on a fundamental idea for pipelining, *viz.*, commutability of independent instructions.

Suppose that the set of variables written and read by two consecutive operations a and b is disjoint. Then executing a followed by b generates the same result as executing b followed by a.

If we view the scheduling steps in Figure 3.1 as arranged in a matrix, then the sequential execution proceeds column-wise along the matrix while the pipelined execution proceeds row-wise. Thus the core proof obligation involves the following two proof requirements.

– Our pipelining algorithm correctly combines the "appropriate" scheduling supersteps which do not have read-write hazards.

– Given that there are no read-write hazards at appropriate places, executing
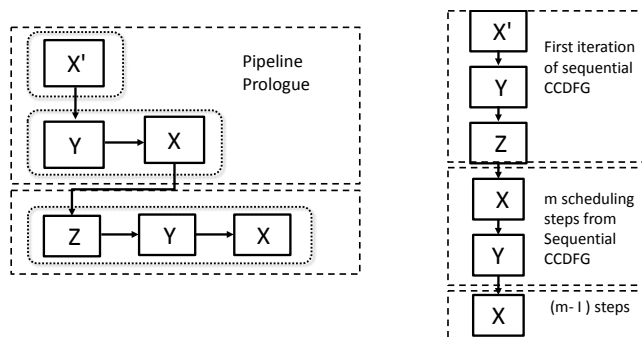
Figure 6.2: Invariant base case where $k = 1$, executing pipeline prologue and one pipeline full stage is the same as executing S_{pre} followed by a sequence of partially completed sequential loop CCDFG.
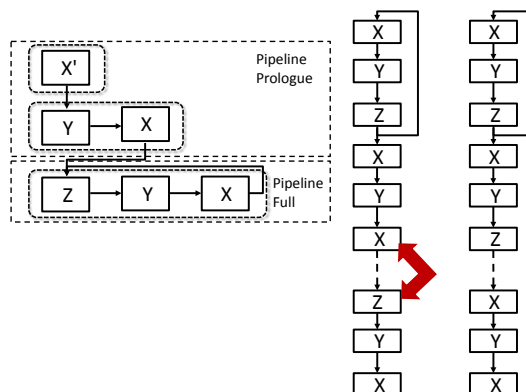
Figure 6.3: Assuming that invariant is true for k steps, executing one pipeline full stage on both sides gives us $(k+1)$ iterations of sequantial loop CCDFG followed by partially completed sequences as expected.

scheduling steps row-wise is same as executing those scheduling steps column-wise in the pipelined CCDFG. This requires the use of interchange primitive.

Although these requirements justify that our correspondence relation is an invariant, they are used somewhat differently in the base case (when the number of

iterations k of the pipelined loop is 1) and inductive step (assume the invariant holds for k iterations of the pipeline and prove that it holds for $(k+1)$ iterations). Their usage is pictorially shown in Figures 6.2 and 6.3. For the base case, we commute operations in the loop prologue of the pipeline (which corresponds to the first iteration after unrolling) with the loop body, while for the inductive step we work with two consecutive iterations of the loop.

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. Most microprocessor pipeline verification work went the other way. For instance, Sawada and Hunt's invariant [15], expressed through an intermediate structure called MAETT, "tracks" the instructions as they pass through different pipeline stages to ensure that hazards are not introduced. One difference in our case is that we are not working with a concrete pipeline with a fixed set of operations but an algorithm that generates pipelines with an arbitrary sequence of scheduling steps; a construction like MAETT is thus not directly applicable. However, there is a deeper reason for defining our invariant the way we did. Suppose we simply unroll the loop in the sequential design three times, and then use a technique similar to MAETT to track scheduling steps in this "unrolled loop body" in the pipeline execution. Unfortunately, this does not work, because of the back edge. There is no direct correlation between this edge and any edge in the sequential loop. In fact, it is interesting to observe what its introduction achieves: completion of one scheduling step in each of the three partially executed, overlapping loop iterations. This suggests that the invariant must explicitly capture the state of the executions that have been partially completed

during each iteration of the pipeline (*ie*, each traversal of the back edge).

6.3   CORRECTNESS OF OUR ALGORITHM

: **Note to Disha: This section needs to be rewritten and would expand further**

: **Note to Disha: Elaborate on this proof: One of the major things is proving that we can indeed define the semantics of run of CCDFG as a straightline run under the restrictions we have considered**

: **Note to Disha: Explain the induction involved in data propagation step where we are moving the step out of iteration in the previous iteration, it is a tricky induction**

: **Complexities involved in applying shadow register primitive to the algorithm**

The algorithm is essentially built from ground-up using primitives as shown in Section 5. 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. The correctness of each primitive discussed above, entails a so-called "assume-guarantee" reasoning: the prmitive is guaranteed to maintain the desired invariant if and only if it is applied under certain well-formed conditions. To use these correctness statements to verify the algorithm, we must therefore prove that the algorithm applies each primitive appropriately, maintaining the well-formedness condition required for the correctness of the primitive.

Note that verifying this requires an inductive proof relating the states of the CCDFG C' generated after the application of the transformation with the original

CCDFG C. The induction is on the lengths of execution of C and C'. Note that the induction is non-trivial because transformations have significant "global" effect on a CCDFG. These include one or more of the following:

1. Replacing one microstep of C with more than one microsteps in C' (*e.g.*, ϕ-elimination), or

2. Interchanging scheduling steps (*e.g.*, interchange), or

3. Changing the variable being read or written in several microsteps (*e.g.*, shadow register)

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.

6.4   LESSONS FROM PREVIOUS FALSE STARTS

: Note to Disha: Explain how because of theorem proving, we avoided certain pitfalls even though it looked perfectly reasonable to design the algorithm that way

Before we came up with our approach of building a pipelining algorithm using a framework of certified pipelining primitives, we tried a few other intuitive approaches. From each false start, we were able to learn something valuable.

In our initial approach we had decided to simplify the problem by ignoring the back edge and proving the correspondence between an unrolled loop and the pipeline. Only after substantially completing this proof and in attempting to extend it to the pipeline with the back edge did we realize that the extension does not work. Section 6 describes a key invariant we defined to deal with this problem.

Also, we attempted initially to stick to the previously proposed algorithm and try to prove that the semantic run of the input is equal to the semantic run of the output for the complete algorithm. To do that, we need to claim that the output pipeline does not introduce any data hazards. Hazard freedom entails showing the following. "Suppose a variable v is written by a scheduling step S and read subsequently by a scheduling step S' in the sequential CCDFG. Then in the pipelined CCDFG, there is no scheduling step P that writes v and is executed between S and S'." Originally, we defined this notion directly for each variable, *viz.*, with a function that statically analyzes the CCDFG to identify the range of scheduling steps between a write and subsequent read of each variable. However, this does not work. For example, proving this property for variable x may require a similar property to hold for another variable y (perhaps because x is assigned an expression involving y). But the range of scheduling steps in which x and y are read and written are different, and the extension of the property to all the variables cannot be easily specified by an invariant for any specific scheduling step. When we realized the challenges involved in proving the complete algorithm, it led us to propose our framework of pipelining primitives. Also, our current approach succinctly captures an "on-track property", *viz.*, that the state after k pipeline

iterations is equivalent to partial execution of a certain number of iterations in the sequential CCDFG (in addition to completion of k' iterations) which avoids this problem and can indeed be specified as an invariant.

Chapter 7

PLANNED EXPERIMENTS

Note to Disha: Look at Kecheng's thesis and include all the designs he has tested his algorithm for if possible. To discuss with kecheng further

We plan to test the pipeline reference model generated by our tool with the pipelined RTL generated by a behavioral synthesis tool using SEC for a variety of pipelined designs across different domains. Our test cases include memory operation designs, cryptography designs and designs for data, signal and image processing with varied pipeline intervals and depths. The test designs are non-trivial to pipeline and require data forwarding and use of temporary shadow registers to remove data hazards.

Chapter 8

CURRENT STATUS AND FUTURE PLAN

8.1 CONCLUSION

: Note to Disha: Need to rewrite this section We have formalized the syntax and semantics of our intermediate representation (CCDFG) in ACL2. We have successfully identified and formalized a framework of succinct and provable primitives essential for loop pipelining algorithms. These primitives include ϕ-elimination, shadow-register, interchange, conditional branch and superstep-construction primitive. Each primitive follows the invariant that the semantic runs of CCDFGs before and after application of the primitive are the same. As of this writing, we have certified ϕ-elimination primitive, shadow-register primitive and superstep-construction primitive. We have formalized the interchange and conditional branch primitives but have not proved them yet. We have formalized our algorithm except the last component of adding conditional branches. We have also proved the application of ϕ-elimination primtive in the algorithm.

Initially, we proposed a proof sketch certifying that an unrolled sequential loop is equivalent to its unrolled pipelined counterpart. We had planned to extend the unrolled loop to include loops with backedges. However, we realized that there is no direct link between the backedges of the two CCDFGs. While the backedge in the sequential CCDFG involves scheduling steps of a particular iteration in order, the backedge of the pipelined CCDFG includes scheduling steps across different iterations. This led us to identify and formalize our key invariant required for the

correspondence between the sequential loop with the backedge and the pipelined loop with the backedge. As of this writing, we have proved that the corresponding relation is true for our algorithm and we have proved the implication chain from this relation to the correctness statement for our algorithm.

Our current ACL2 script has 262 definitions and 258 lemmas, including many lemmas about structural properties of CCDFGs (but not counting those from the false starts). We are still working on proving some key components of the proof obligations to show that the algorithm does not introduce data hazards.

Now that we have a complete executable loop pipelining algorithm in ACL2, we first plan to show that our algorithm is practical and can be used for industrial strength designs with tens of thousands of RTL. We plan to evaluate on a variety of designs across different application domains. Given a sequential CCDFG and pipeline parameters from the behavioral synthesis tool, we will generate the pipeline reference model using our pipelining algorithm. Then, we will compare the pipeline reference model with the pipelined RTL generated by the behavioral synthesis tool using SEC. If we are able to generate the pipeline reference model using the same pipeline parameters as synthesis tools and can enable successful SEC, we can claim that our approach is practical and can be used for certifying industrial strength designs across different application domains.

Next, we will certify the conditional branch and interchange primitive to complete certification of our framework of primitives. The conditional branch requires us to ensure that the control flow before and after removing the conditional branch is the same. The interchange primitive involves proving that if there are no read-write hazards between two adjacent scheduling steps m and n, then executing m followed by n semantically equals to executing n followed by m. The proof would involve statically analyzing all types of statements.

For certification of the loop pipelining algorithm, we will first formalize the addition of conditonal branches in the pipelined loop. Then, we need to prove that each component of our algorithm described in Section 5 maintains the invariant that the semantic runs of CCDFGs before and after that component are same. Even though each component essentially decomposes into proving that our primitive is correct, we still have to prove that every application of our primitives maintains certain assumptions and does not disrupt the certification flow. Also, we need to prove by induction that applying a primitive in the context of a CCDFG is correct. For example, proving that the shadow register primitive is correct in a CCDFG involves proving that only the microsteps required for the shadow register primitive are changed while others remain the same. Also, even if we can prove that the addition of shadow registers does not change the state of our CCDFG, we still need to prove that the execution of CCDFG after this step is the same, considering the data and control flows.

With his dissertation, we expect to make the following major contributions:

– *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.

– *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.

– *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.

8.2 FUTURE WORK

: Note to Disha: Need to write this section

REFERENCES

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

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

[3] 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.

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] D. Puri, S. Ray, K. Hao, and F. Xie. Using acl2 to verify loop pipelining in behavioral synthesis. In *International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2014*, 2014.

[14] 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.

[15] 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.

[16] 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.

[17] University of Toronto. *LegUp Documentation Release 3.0*, 2013.

[18] 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.

[19] Xilinx. *Vivado Design Suite Reference Manual*, 2014.

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