Mechanical Certification of Loop Pipelining Transformations: A Preview

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

[1] Department of Computer Science, Portland State University, Portland, OR 97207.
[2] Strategic CAD Labs, Intel Corporation, Hillsboro, OR 97124.

Abstract. We describe our ongoing effort using theorem proving to certify loop pipelining, a critical and complex transformation employed by behavioral synthesis. Our approach is mechanized in the ACL2 theorem prover. We discuss some formalization and proof challenges and our early attempts at addressing them.

Keywords: Behavioral Synthesis, Theorem proving, Electronic System Level Design, Equivalence Checking

1 Introduction

Behavioral synthesis is the process of compiling an Electronic System Level (ESL) specification of a hardware design into RTL. ESL facilitates fast turnaround time in production of hardware designs by rasing design abstraction; the designer only specifies the design functionality in a high-level language (*e.g.*, SystemC, C, C++, etc.), from which RTL is automatically synthesized. However, its adoption critically depends on our ability to certify that the synthesized design indeed implements the ESL specification. Given the abstraction difference between ESL and RTL, this certification is non-trivial.

Loop pipelining is a critical transformation implemented in most behavioral synthesis tools. Unfortunately, it is also one of the most complex transformations [1]. Furthermore, sequential equivalence checking (SEC) techniques are not directly applicable for certification of synthesized loop pipelines. In this paper, we describe how we are using interactive theorem proving (ACL2) to facilitate this certification. We discuss our early efforts with the proof, some of the challenges and complexities, and the approaches we are exploring to address them.

This work is a part of a project to develop a scalable certification framework for behavioral synthesis. The project has been ongoing for some time, with several mature components; however, the focus of previous work was on automated SEC. The work described here is its first serious "foray" in theorem proving.

2 Background and Context

Behavioral Synthesis and an SEC Framework: Behavioral synthesis transformations are classified into three categories: (1) compiler transformations,

(2) scheduling (mapping each operation to a clock cycle), and (3) Resource allocation and control synthesis (allocating registers to variables, and generating an FSM to implement the schedule). Loop pipelining is a part of scheduling. Given the abstraction gap between ESL and RTL, there are no obvious mappings between internal variables, rendering SEC ineffective. Applying theorem proving is also challenging: (1) *verifying each synthesized design* requires prohibitive human effort; (2) *verifying a synthesis tool* is infeasible since tool implementations are proprietary (and closely guarded), in addition to being highly complex.

Previous work [2–4] resulted in the following observations. (1) SEC can compare RTL with the intermediate representations (IRs) after compiler and scheduling operations; correspondence between internal variables is preserved, and identified from resource mappings. (2) While transformation *implementations* are proprietary, IRs after successive transformations are available from reports generated during synthesis. (3) IRs are structurally similar across synthesis tools *viz.*, graphs of operations with explicit control/data flow and schedule. Consequently, a formalization called *Clocked Control Data Flow Graph* (CCDFG) was developed for IRs, together with two SEC algorithms, respectively to compare (1) a CCDFG with RTL, and (2) two CCDFGs corresponding to IRs after each successive transformation. However, the latter is effective only if the difference between IRs is small. Loop pipelining substantially changes control/data flow and introduces controls (*e.g.*, to eliminate hazards), making SEC infeasible.

Certifying Loop Pipelining: Our key observation is that it is *not* necessary to verify the implementation of any synthesis tool. Instead, we can (1) develop a *reference* algorithm $\mathcal{A}$ that takes a sequential CCDFG $\mathcal{C}$ and generates a pipelined CCDFG $\mathcal{P}$, (2) use SEC to compare $\mathcal{P}$ with the synthesized RTL $\mathcal{R}$, and (3) prove the correctness of $\mathcal{A}$. The algorithm $\mathcal{A}$ can be much simpler than that of any synthesis tool, since it can use the synthesis tool's report to determine the values of the key parameters (*e.g.*, pipeline interval, number of iterations pipelined, etc.). Viability of this flow was justified previously [5] by developing such an algorithm and using it to compare several synthesized pipelines. However, the algorithm was not verified (indeed, not formalized), rendering the "certification" flow unsound; in fact, we already found errors in that algorithm merely by attempting formalization. Furthermore, since it was not written with reasoning in mind, it is a non-trivial target for mechanical proof. Our current work is a deconstruction of that algorithm, developed from ground up to account for necessary invariants. Note that we are free to choose *any* verifiable implementation without losing the ability to certify designs synthesized by commercial tools.[3]

3 Pipelinable loop and Correctness Formalization

Pipelinable Loop: A *pipelinable loop* [5] is a loop with (1) no nested structures, (2) one *Entry* and one *Exit* block; and (3) no branching between scheduling

[3] One caveat is that we must synthesize pipelines *using the parameters reported by the synthesis tool*; otherwise we may fail to certify correct designs. We have not found this to be a problem in practice.

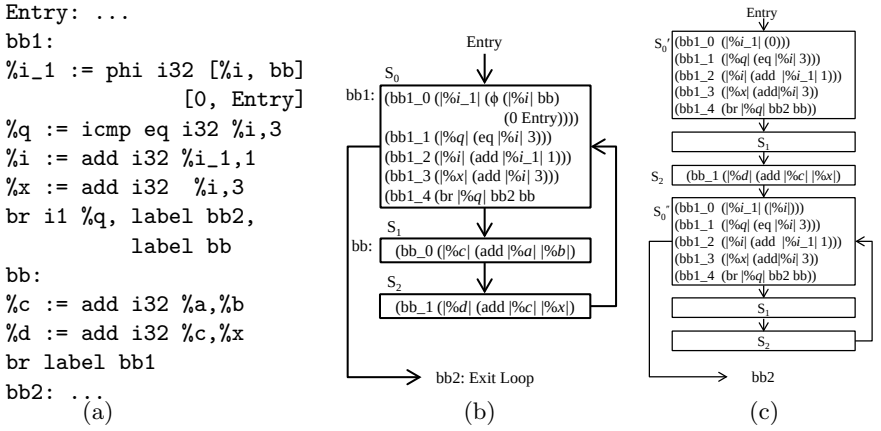

Fig. 1. (a) Loop in LLVM Assembly (b) Fragment of CCDFG corresponding to loop. Scheduling step S_0 has a ϕ-statement (c) ϕ-elimination operation. $\%i\_1$ is assigned `0` in S_0' and $\%i$ in S_0''.

steps. These restrictions are not for simplifying reasoning, but reflect the kind of loops that are actually pipelined, *e.g.*, synthesis tools unroll inner loops (by a compiler transformation) before applying pipelining to the outer loop.

Correctness Statement : Let L be a loop in CCDFG C, and let L_α be the pipelined implementation generated by our algorithm using pipeline parameters α. 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. The algorithm introduces additional variables, eg, *shadow variables* (cf. Section 4). The values of these variables in f' are irrelevant since they are not accessed subsequently.

CCDFG : Formalizing the correctness statement entails defining the semantics of CCDFG. A CCDFG is a control/data flow graph with a schedule. Control flow is broken into basic blocks. Instructions in a basic block are grouped into *microsteps* that are executed concurrently. A schedule is a grouping of microsteps which can be completed within one clock cycle. The instruction language we support is a subset of LLVM [6] which is a front-end for many behavioral synthesis tools [7, 8]; we support assignment, load, store, bounded arithmetic, bit vectors, arrays, and pointer manipulations. As is common with ACL2, we use a state-based operational semantics [9, 10]. Assigning meanings to most instructions is standard; one exception is the ϕ-statement "`v := phi [`σ `bb1] [`τ `bb2]`". If

reached from basic block `bb1`, it is the same as the assignment statement `v :=` σ; if reached from `bb2`, it is the same as `v :=` τ; the meaning is undefined otherwise. Reasoning about ϕ-statement is complex since after its execution from state s, the state reached depends not only on s but previous basic block in the history. We need to handle it since it is used extensively to implement loop tests. A key step in loop pipelining is ϕ-elimination, *viz.*, unrolling the loop once and replacing the ϕ-statement with assignment statements (cf. Fig. 1).

4 Algorithm and Proof

Our algorithm includes (1) ϕ-elimination mentioned above, (2) shadow registers, and (3) superstep construction. Fig. 2 illustrates steps 2 and 3.

Shadow Registers: Consider the CCDFG in Fig. 1. Here `%x` is written in step S_0 but read in S_2. If the loop is pipelined such that a new iteration is initiated every cycle, then we must ensure that the write from the S_0 step of a subsequent iteration does not overwrite `%x` before it is read by the S_2 step of the current iteration. This is achieved by introducing a *shadow register* `%x_reg` that preserves a copy of the "old value of `%x`" and replacing reads of `%x` to use `%x_reg`.

Superstep Construction: We combine scheduling steps of successive iterations into "supersteps" which are scheduling steps for the pipeline. Supersteps account for hazards, *viz.*, if a variable is written in scheduling step S and read subsequently in S' then S' cannot be in a superstep that precedes S. S and S' can be in a single superstep since we implement data forwarding.

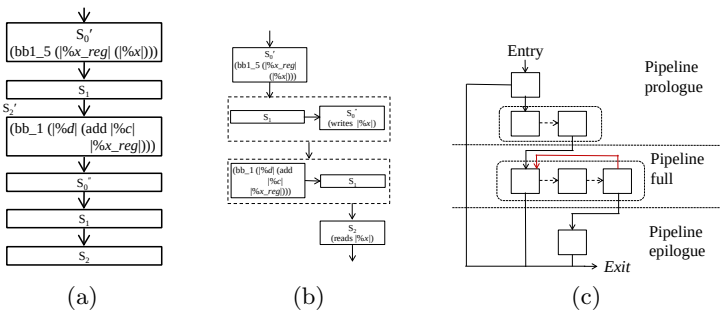

Fig. 2. (a) CCDFG of Fig. 1 after inserting shadow register $\%x\_reg$ for $\%x$. (b) Superstep construction. Horizontal arrows represent data forwarding. (c) Pipelined loop.

Correspondence Relation: Our planned proof involves defining a "correspondence relation" between loops of the sequential and pipelined CCDFGs and proving that it is preserved across loop iterations. The relation is informally paraphrased as follows. "Let S be a sequential loop and G be the pipelined loop

generated from our algorithm, constituting prologue G_p, full stage G_l, and epilogue G_e (cf. Fig 2(c)). Let s_l be any state of G poised to execute G_l, and let k be any number such that the loop of G is not exited in k iterations from s_l. Then executing k iterations of G_l from s_l is equivalent to executing k iterations of S together with a collection of "partially completed" iterations of S.[4]

Proof Sketch: The invariant, albeit non-trivial, admits a direct proof of the correctness statement in Section 3. Equivalence of CCDFG states after completing execution of G and S follows from the fact that the epilogue G_e constitutes the incomplete scheduling steps of S. To prove that the relation is invariant across pipeline iterations, note that each new iteration of G_l initiates a new (incomplete) iteration of S, and advances incomplete iterations by one scheduling step; the result follows by rearranging the incomplete iterations, since rearrangement of scheduling steps produces the same computation in the absence of hazards. Thus we need to show that our algorithm generates hazard-free pipelines, which reduces to structural properties of the three components of the algorithm.

5 Current State and Conclusion

As of this writing, we have formalized the correspondence relation, and finished the proof of key lemmas for ϕ-elimination and shadow register. We have also proven an implication chain from the correspondence relation to the correctness statement. Our current ACL2 script has 156 definitions and 300 lemmas, including many lemmas about structural properties of CCDFGs. We admit that the correspondence relation and the proof sketch above, while rather natural on hindsight, are outcomes of lessons learned from several false starts.

Microprocessor pipeline verification is a mature research area [11–13]. Our work, albeit analogous, is different, *e.g.*, we verify an *algorithm* to generate pipelines instead of a specific implementation. Also, recent work on translation validation for software pipelines [1] has parallels to our work. However, their correctness statement is contingent upon the equivalence of a certain symbolic simulation of the two designs, and they do not statically identify data hazards.

Use of theorem proving on industrial flows typically involves either complicated reasoning about (optimized) implementations, or abstracting them significantly to facilitate proof. In contrast, we apply theorem proving on an algorithm that generates reference designs for SEC. This permits adjusting the algorithm (within limits) to suit mechanical reasoning while affording comparison with actual synthesized artifacts. We have made liberal use of this "luxury", *e.g.*, the three components of our algorithm were conceived from a reflection of our invariant and proof sketch. Indeed, we are currently refining the definition of superstep construction to facilitate proof of certain structural lemmas. We believe a similar approach is applicable in other contexts and may provide effective use of theorem proving without exposing confidential intellectual property.

[4] 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.

References

1. Tristan, J.B., Leroy, X.: A Simple, Verified Validator for Software Pipelining. In Hermenegildo, M.V., Palsberg, J., eds.: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010), Madrid, Spain, ACM (2010) 83–92
2. Ray, S., Hao, K., Xie, F., Yang, J.: Formal Verification for High-Assurance Behavioral Synthesis. In Liu, Z., Ravn, A.P., eds.: Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis (ATVA 2009). Volume 5799 of LNCS., Macao SAR, China, Springer (2009) 337–351
3. Hao, K., Xie, F., Ray, S., Yang, J.: Optimizing Equivalence Checking for Behavioral Synthesis. In: Design, Automation and Test in Europe, Dresden, Germany, IEEE (2010) 1500–1505
4. Yang, Z., Hao, K., Cong, K., Ray, S., Xie, F.: Equivalence Checking for Compiler Transformations in Behavioral Synthesis. In Byrd, G., Schenider, K., Chang, N., Ozev, S., eds.: Proceedings of the 31st International Conference on Computer Design (ICCD 2013), Asheville, NC, USA, IEEE (2013) 491–494
5. Hao, K., Ray, S., Xie, F.: Equivalence Checking for Behaviorally Synthesized Pipelines. In Groeneveld, G., Sciuto, D., Hassoun, S., eds.: Proceedings of the 49th International ACM/EDAC/IEEE Design Automation Conference (DAC 2012), San Francisco, CA, USA, ACM (2012) 344–349
6. Lattner, C., Adve, V.S.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: 2nd ACM/IEEE International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO 2004), San Jose, CA, USA, IEEE Computer Society (2004) 75–88
7. Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., Zhang, Z.: High-Level Synthesis for FPGAs: From Prototyping to Deployment. IEEE Transactions on CAD of Integrated Circuits and Systems **30** (2011) 473–491
8. Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J.H., Brown, S., Czajkowski, T.: LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems. In Wawrzynek, J., Compton, K., eds.: Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA 2011), Monterey, CA, USA, ACM (2011) 33–36
9. Liu, H., Moore, J.S.: Executable JVM Model for Analytical Reasoning: A study. Science of Computer Programming **57** (2005) 253–274
10. Boyer, R.S., Moore, J.S.: Mechanized Formal Reasoning about Programs and Computing Machines. In Veroff, R., ed.: Automated Reasoning and Its Applications: Essays in Honor of Larry Wos, MIT Press (1996) 141–176
11. Burch, J.R., Dill, D.L.: Automatic Verification of Pipelined Microprocessor Control. In: Proceedings of the 6th International Conference on Computer Aided Verification (CAV 1994). Volume 818 of LNCS., CA, USA, Springer-Verlag (1994) 68–80
12. Manolios, P.: Correctness of Pipelined Machines. In Hunt, Jr., W.A., Johnson, S.D., eds.: Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000). Volume 1954 of LNCS., Austin, TX, Springer-Verlag (2000) 161–178
13. Sawada, J., Hunt, Jr., W.A.: Verification of FM9801: An Out-of-Order Microprocessor Model with Speculative Execution, Exceptions, and Program-Modifying Capability. Formal Methods in Systems Design **20** (2002) 187–222