Integrating Formal Verification and High-Level Processor Pipeline Synthesis

Eriko Nurvitadhi, James C. Hoe
Carnegie Mellon University
{enurvita, jhoe}@ece.cmu.edu

Timothy Kam, Shih-Lien L. Lu
Intel Corporation
{timothy.kam, shih-lien.l.lu}@intel.com

Abstract – **When a processor implementation is synthesized from a specification using an automatic framework, this implementation still should be verified against its specification to ensure the automatic framework introduced no error. This paper presents our effort in integrating fully automated formal verification with a high-level processor pipeline synthesis framework. As an integral part of the pipeline synthesis, our framework also emits SMV models for checking the functional equivalence between the output pipelined processor implementation and its input non-pipelined specification. Well known compositional model checking techniques are automatically applied to curtail state explosion during model checking. The paper reports case studies of applying this integrated framework to synthesize and formally verify pipelined RISC and CISC processors.**

I. INTRODUCTION

Motivations. The need to apply a high-degree of customization within a short time-to-market makes it intractable to develop application-specific processors (ASIPs) manually. To address this, prior works (e.g., [11][13][15]) have presented design frameworks that can be used to automatically synthesize custom pipelined processor implementations (usually at the register transfer level) from a precise high-level specification. These frameworks drastically shorten the time to arrive at an implementation. However, an overall reduction in time-to-market requires not only saving the time to create an implementation but also cutting down the time to verify that the implementation is correct.

Unfortunately, even starting from a presumably correct specification and assuming hands-free automatic synthesis, there are ample opportunities for bugs to be introduced in the many rounds of synthesis and translation that stand between a high-level specification and its final realization. We can group the bugs into: (1) a fundamental error in the synthesis algorithms, or (2) a programming bug in the coding of the synthesis algorithms. This is not a new problem. An analogous problem has long existed for the now industry-standard RTL-downward synthesis flows. In less critical designs, one may simply put faith in the correctness of the synthesis tools; for critical designs, one must perform extensive functional design validation at the lowest practical intermediate representations and even on the final parts.

Background Technologies. Taking advantage of the precise semantics of high-level design frameworks, one should utilize formal verification technologies to ensure functional equivalence between the initial high-level specification and the output of subsequent synthesis and translation. With recent advances in combining model checking and theorem proving to curtail state explosion, compositional model checking [12] has been applied successfully to verify the functional equivalence between non-trivial pipelines and their specifications [8][10]. Unfortunately, the required expertise and manual effort are reported to be very high [10].

In this paper, we present an effort to integrate formal verification with the T-piper high-level pipeline synthesis framework [13]. To reduce the error prone and tedious pipelining effort, T-piper automatically generates a pipelined implementation from an abstract transactional specification (T-spec) of a non-pipelined datapath. Many implementations can be derived from a T-spec, as long as the T-spec transactional semantics is correctly preserved. From a T-spec, T-piper creates a pipeline implementation that overlaps the executions of multiple transactions in different pipeline stages for better performance. A designer can direct T-piper to rapidly create many pipelines by specifying the pipeline stage boundaries and selecting from a complete set of hazard resolution options identified by T-piper. As such, T-spec/T-piper is well suited for ASIP design explorations and development.

Contributions. The pipelined implementation synthesized by T-piper, with its concurrent execution of transactions and the intricacies of hazard resolutions, nevertheless should result in the same execution as if the transactions were executed one-at-a-time as prescribed by the T-spec transactional semantics. To formally prove this, we propose extending T-piper to automate the verification of the functional equivalence between the input T-spec and the output pipeline implementation. This automation eliminates the high manual effort and expertise required to generate the verification models and to apply abstraction and compositional reasoning techniques for compositional model checking. This fully automated approach enables users without expert-level formal verification knowledge to develop formally verified custom pipelined datapath. In our case studies, we were able to automatically verify a variety of custom processor pipelines, 9 for the MIPS ISA and 9 for a hypothetical ISA with CISC-like memory-to-memory instructions.

Outline. The rest of the paper is organized as follows. Section II provides a background in model checking. Section III gives an overview of the T-spec/T-piper design framework. Building on Section II and III, Section IV presents the formal verification integration with T-piper. Section V reports our case studies on automatic verification of processor pipelines. Section VI provides a survey of related work. Finally, Section VII offers concluding remarks.

II. Model Checking Overview

This section uses a simple example from the Cadence SMV model checker tutorial [12] to explain the process of compositional model checking and to highlight the high level of sophistication and manual effort involved. The left-portion of Figure 1(a) (designated "Specification") shows a simple non-pipelined 32-bit processor datapath that only supports ALU instructions. Each ALU instruction reads two operands from the register file (RF); performs an ALU operation; and writes the result back to the RF. The right-portion of Figure 1(a) (designated "Implementation") depicts a 3-stage pipelined datapath with maximal data forwarding support.

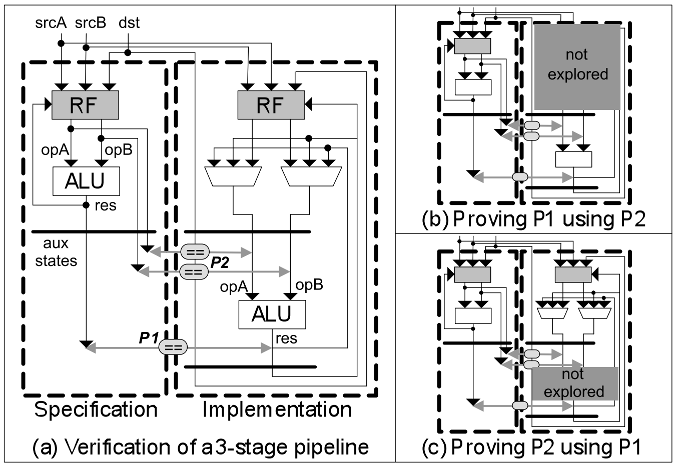

Fig. 1. A simple verification example from the Cadence SMV model checker tutorial [12].

A. Model Checking

To verify that the Specification and the Implementation are functionally equivalent, we first create cycle-accurate and bit-true RTL models for both the Specification datapath and the Implementation datapath.

Next, we devise a pipeline correctness property to be checked. To prove functional equivalence of the Specification and the Implementation, we can set a property stating that following all possible instruction execution sequences, the Specification and the Implementation make the same RF state updates. Since the RF state update value is produced by the ALU, which depends on the RF state as input, the correctness property (let us call this P1) can instead require the ALU outputs in Specification and Implementation to be the same.

Because the timing of Specification and the Implementation are different, we need to create a refinement map that relates the ALU output in the Implementation to the ALU output in the Specification. In this case, we introduce an auxiliary pipeline register in the Specification model to provide a delayed ALU output value that corresponds in timing with the ALU output in the implementation model.

We next declare certain control signals to be "free" variables, indicating to the model checker to consider all possible combinations of values of those variables. In the current example, the read and write indices (srcA, srcB, dst) of the RF would be declared as free variables so that the model checker considers in all possible execution sequences all combinations of reading and writing the different RF entries.

B. Abstraction and Decomposition

Given correctly formulated (1) Specification and Implementation models, (2) a refinement map, and (3) a correctness property, a capable model checker should either prove that the property is true or produce a counter example. In practice however, even the simple pipeline in the current example would cause today's model checkers to run out of memory due to the large number of states that need to be explored (a.k.a., state explosion).

The complex functionality of the ALU (supporting a large number of 2-to-1 functions, such as multiply-and-shift) is one cause of state explosions. A standard workaround in model checking is to assume that the corresponding ALU blocks in the Specification and the Implementation are identical. Thus, they can be captured as *uninterpreted functions* [12] so that the model checker does not have to consider their internal details. We can further abstract other details such as the exact word-size of the datapath. For example, *data type reduction* [12] can be applied to the ALU operands and output to verify the correctness property generally for unbounded word-size (which is actually much cheaper to verify than an explicit word-size).

A property that depends on many signals (i.e., has a large cone of influence) can also lead to state explosion. The correctness property P1 posed in Section II.A has a cone of influence that covers the entirety of the Specification and the Implementation. *Compositional reasoning* [12] allows a property to be decomposed, so multiple smaller (more manageable) properties can be checked instead. For example, we can introduce another property P2 that states that the ALUs in the Specification and Implementation receive the same operands. Instead of proving P1 as a standalone property, we prove separately P1 assuming P2, and then P2 assuming P1. When proving P1 assuming P2, the cone of influence is greatly reduced from before since it is no longer necessary to consider the RF fetch logic in the Implementation. (Figure 1(b) illustrates the part of the pipeline that can be left out when proving P1 assuming P2; Figure 1(c) shows the same for when proving P2 assuming P1.)

Another well-known decomposition is *case analysis* [12], which splits a proof into multiple proofs according to different assignments to a set of variables. For example, we can split P1 into multiple (smaller) cases that consider separately different combinations of ALU output and input operands. Furthermore, symmetry can be used on the 32-bit ALU's input operands to reduce the number of cases that need to be checked explicitly.

As the example shows, the manual effort needed in compositional model checking is significant. Expert knowledge both in pipeline design and model checking is needed to determine the appropriate abstractions and decomposition strategies to apply. A similar sentiment was reported in a recent case study that verified RISC processor pipelines using compositional model checking [10].

III. T-SPEC AND T-PIPER REVIEW

A. *Transactional Datapath Specification (T-spec)*

A T-spec is a textual "netlist" that comprises of a set of architectural states and next-state logic blocks. An architectural state (register or array) has explicit state-read and state-write interfaces. Each next-state compute block is treated as a black-box for synthesis, except for multiplexers which are primitives understood by T-piper analysis.

Figure 2(c) shows an example T-spec datapath with a single state element R and a network of next-stage logic blocks (op1, op2, op3, op4, op5, and m1); R is represented as its separate read and write interfaces. Note that we chose to use this simple example for the sake of brevity. Details on T-spec can be found in [13].

A T-spec captures an abstract datapath, whose execution semantics is interpreted as a sequence of "transactions" where each transaction reads the state values left by the preceding transaction and computes a new set of state values for the next transaction to see (Figure 2(b)). Many valid implementations may be derived from a T-spec, as long as the aforementioned transactional semantics is preserved. The proposed formal verification approach aims to prove that the in-order pipeline implementation synthesized by T-piper executes and performs the same order of transactions and state updates as its T-spec specification.

B. *Pipeline Synthesis (T-piper)*

To arrive at a pipelined implementation, T-piper analysis (Figure 2(a)), based on designer-specified pipeline-stage boundaries (S-cfg), informs the designer of the available opportunities for applying forwarding and speculation to resolve hazards. Next, based on the designer's choice of forwarding/speculation optimizations to include (H-cfg), T-piper generates an RTL-Verilog of the desired pipeline, which preserves the transactional execution semantics of the T-spec datapath. From a single T-spec, the designer can rapidly explore the pipeline design space by submitting different pipeline configurations to T-piper.

Figure 3 shows three example pipelines that can be synthesized from the T-spec in Figure 2(c) by specifying a 4-stage S-cfg where op1, R.rd, op2, and op3 are assigned to stage 1, op4 to stage 2, op5 and m1 to stage 3, and R.wr to stage 4. Figure 3(a) shows a simple synthesized pipeline with stalling, where T-piper generates hazard detection and stall logic for resolution (dashed lines).

Figure 3(b) shows a synthesized pipeline with data forwarding. During analysis (Figure 2(a)), T-piper identifies all possible forwarding points (FwdPt) for R. A forwarding point represents an output port or a pipeline register where the write-data value for R is available from an in-flight transaction downstream in the pipeline, and therefore can be forwarded to a younger transaction reading R. Based on the available forwarding points, the designer can select which ones to implement. In Figure 3(b), only forwarding points 2 and 4 are enabled. The dashed lines illustrate the forward paths and logic generated by T-piper in the final pipelined implementation.

Finally, Figure 3(c) adds a predictor logic pred to the datapath. T-piper identifies opportunities for forwarding such prediction (PredFwdPt), and lets the designer choose which ones to enable. A prediction can be resolved at any forwarding points in the pipeline, using the actual data that has become available. T-piper requires the designer to select sufficient set of prediction resolution points (PredResPt) to ensure a prediction is resolved fully before any transaction retires. In Figure 3(c), the prediction is forwarded using PredFwdPt 1, and resolved in the last stage. The dashed lines depict the prediction forwarding and resolution logic introduced by T-piper. More details on T-piper synthesis are available at [13].

IV. AUTOMATIC VERIFICATION

Compositional model checking can be used to prove that a T-piper synthesized pipelined implementation is functionally equivalent to its non-pipelined T-spec specification. Specifically, given the inputs of T-spec, S-cfg and H-cfg files, T-piper generates a design file that can be submitted to

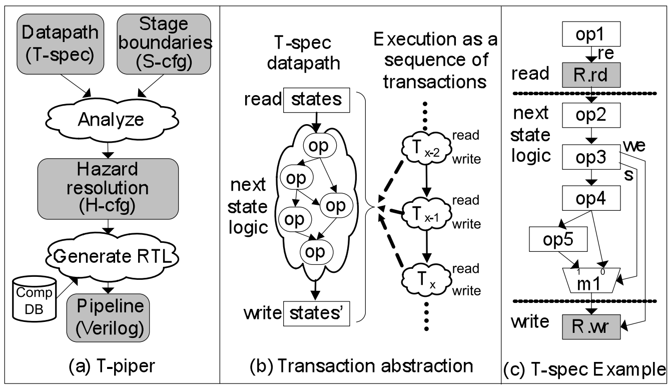

Fig. 2. Pipeline development using T-spec and T-piper.

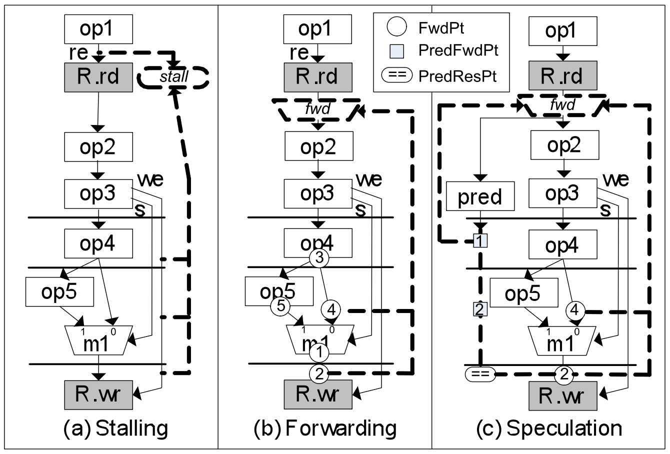

Fig. 3. Examples of hazard resolution in T-piper pipelines.

Cadence SMV [12]. Ideally, we would like to model check the RTL Verilog design directly. The current choice of the SMV language is simply because Cadence SMV is the only capable compositional model checker that we have access to. As is, the Verilog and SMV descriptions are generated from a common RTL internal representation at the final step of the synthesis process.

A. Verification Objective

Since the implementation pipeline in our context is automatically generated from T-spec, any bug in the implementation would have to be caused by a bug in T-piper. As noted in the introduction, there are two classes of bugs that can occur in T-piper: (1) a fundamental bug in the pipeline synthesis algorithms itself, or (2) a programming bug introduced in coding the synthesis algorithms. Both types of bugs can be exposed by the formal verification approach described in this paper.

Starting from a T-spec netlist, T-piper introduces pipeline stage registers and pipeline control logic such that the overlapped transaction executions on the synthesized pipeline produces the same result as the one-at-a-time, sequential next-state update of the T-spec model under T-spec's transactional execution semantics. The pipelined implementation uses the same next-state compute blocks (e.g., op1, op2, op3, op4, op5, and m1 in Figure 2(c)) as the original T-spec. Since these blocks are a part of the specification, we assume they are correct and have been verified independently.

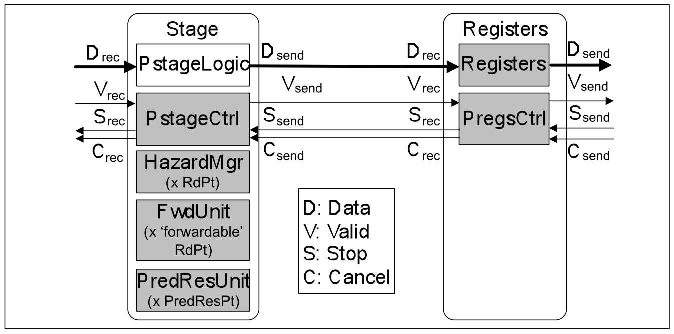

Fig. 4. Pipeline model.

The focus of our verification effort is the correctness of the pipeline register insertion and the pipeline control logic generated by T-piper. Figure 4 depicts the pipeline structure of a single pipeline stage generated by T-piper, with the pipeline control logic in the shaded blocks. In the figure, PstageLogic (pipeline stage logic) refers to the user-provided next-state logic blocks specified in T-spec. The pipeline control logic blocks introduced by T-piper are:

- Pipeline Stage Controller (PstageCtrl) interacts with the PstageLogic in a given stage and manages communication with the adjacent pipeline state registers.
- Data Hazard Manager (HazardMgr) detects data hazards and activates the appropriate resolution logic. Since hazard is detected at a state-read interface, one HazardMgr is generated for each state-read interface in a stage.
- Forward Unit (FwdUnit) manages the forwarding of both actual and predicted values. It includes a forwarding mux (e.g., fwd in Figure 3(b)) and acts as a proxy to a state read interface, providing either a forwarded or an actual state-read value. One FwdUnit is generated for each state-read interface with forwarding support.
- Prediction Resolution Unit (PredResUnit) compares a predicted value with the actual value eventually produced by the datapath. It triggers squash on a misprediction. One PredResUnit is generated for each PredResPt in the stage.

B. Verification Models

Prior to abstraction, the specification and the implementation models are cycle-accurate and bit-true representations of the original non-pipelined T-spec and the synthesized pipelined implementation, respectively. The specification model is automatically augmented with the necessary auxiliary states and logic to establish the refinement mapping for formal verification.

Uninterpreted Functions. To curtail state explosion, T-piper generates the simplest possible models while exposing sufficient details to facilitate the verification of the pipeline control logic. In the verification models, the internal details of the next-state compute blocks (PstageLogic in Figure 4) are abstracted as uninterpreted functions. Recall that these are user-provided blocks that are assumed correct as given.

On the other hand, the implementation model must expose faithfully the pipeline control logic (shaded units in Figure 4) introduced by T-piper. To do so, the abstraction retains the following details needed by the pipeline control logic for formal verification:

- State elements and their read/write interfaces, which expose all possible hazards in the implementation model.
- State write-data sources (e.g., outputs of op4 and op5 in Figure 2(c)) and write multiplexers (e.g., m1 in Figure 2(c)), which expose all forwarding in the implementation.
- Dataflow dependencies, which are required to maintain the correct original transactional semantics.

Free Variables. T-piper automatically declares control signals as 'free' variables, so that the model checker will explore all possible control conditions. More specifically, the following signals are declared as free variables by T-piper:

- The read and write enables of each state element (and index of an array state element). Freeing these signals allows model checking to cover for all possible data hazard scenarios.
- The select signal of a write multiplexer. This allows the model checker to explore the state writes from all the possible write-data sources, thus uncovering all the possible data forwarding scenarios.

There is no extra analysis required to identify these signals since they are already needed by T-piper for pipeline synthesis.

Auxiliary States and Coordination Logic. To establish formal equivalence, T-piper inserts auxiliary states to buffer values produced by the specification model. These buffered values are used to set the refinement maps and correctness properties. Details on the correctness properties produced by T-piper are discussed in Section IV.C.

In addition to the auxiliary states, T-piper also generates logic that coordinates the execution progress of the specification and the implementation models. Such coordination logic is needed to ensure proper alignment between the propagated reference values through auxiliary states and the values in the pipeline implementation.

When the coordination logic detects a pipeline stall in the implementation model, the coordination logic needs to artificially throttle the progress of the specification model to keep the two models' progress in synchronization. Since T-piper is synthesizing the pipeline control logic, it knows which signals correspond to pipeline stalls and need to be monitored by the coordination logic.

Modeling Speculation. When a transaction requires an architectural state value that is due to be updated by an older transaction and the value is not yet available through forwarding, automatic speculative mechanisms in a T-piper pipeline allow the younger dependent transaction to utilize a predicted value generated by a user-provided value predictor in place of the actual state update value of the updating older transaction. T-piper automatically generates the logic to eventually check the predicted value against the actual value and, in the case of a misprediction, to restart the pipeline after squashing any affected transactions. (The transaction for which the prediction is made for never made use of the prediction itself and therefore is always correct.)

In the case of a correct prediction, we need to verify that the predicted value is forwarded correctly. This is essentially the same as verifying the data forwarding logic, except with a different type of data source (i.e., value produced by the predictor logic instead of a logic block from a downstream stage). In the case of a misprediction, we need to verify both that the prediction resolution unit (PredResUnit) appropriately informs the pipeline control units (PstageCtrl) of the event and that the pipeline is correctly squashed and restarted.

T-piper implements the approach in [8] to model speculative execution for verification. First, we utilize a free Boolean flag (e.g., isMispred) to indicate whether a misprediction happened at a given execution step. If isMispred is asserted, then the specification model stalls to wait for the implementation model to detect the misprediction and squash the affected transactions in the pipeline. If isMispred is not asserted, the specification model advances normally.

Second, the transactions following a misprediction in the implementation model are marked as 'shadow'. They will eventually be squashed when the misprediction is detected. Therefore, they do not have any correspondence to those transactions executed by the specification model. These shadow transactions are tracked with a shadow bit, which is an auxiliary state that is set when the isMispred flag is asserted and cleared when the implementation has handled the misprediction. The refinement maps are set to ignore these shadow transactions accordingly.

Finally, to verify the correctness of the forwarding logic for a predicted value, T-piper uses the value generated by the specification model as the input to the implementation model. This value is carried along the pipeline using auxiliary states, and is used to model the forwarding of a correct prediction.

User-Defined Constraints on State Accesses. By default, T-spec requires each state access to be predicated by an explicit enable signal (i.e., a read/write occurs only when its enable signal is asserted). However, in some cases, a state is constrained by design to always be read (or written) by every transaction. For example, the program counter (PC) in an instruction processor is read and written by every instruction. To simplify the verification models, we extended T-spec to allow the user to annotate such constraints. T-piper incorporates these constraints in the verification model. In the instruction processor example, the constraint that sets PC to always be read/written allows pruning the scenarios where PC is not read/written, which reduces the number of state transitions to be explored during model checking.

C. *Proving Correctness*

Correctness Properties. T-piper automatically sets up refinement maps and the following correctness properties:

- State write value (P-wr): any architectural state updates made in the implementation model should be consistent to those in the specification model.
- State read value (P-rd): any architectural state reads made in the implementation model should be consistent to those in the specification model. Note that in an implementation model with data forwarding, state read value is obtained at the output of the FwdUnit.
- Uninterpreted function output (P-uf-out): the output produced by each uninterpreted function in the implementation model should be consistent with the one in the specification model.

The P-wr properties by themselves are necessary and sufficient to prove the functional equivalence between the specification and the implementation models. The P-rd and P-uf-out properties are included to facilitate decomposition.

Decomposition. T-piper automates similar decomposition heuristics as those applied manually in [12].

- A P-wr property is proven by assuming that all the write-data sources are correct. Write-data sources are the earliest forwarding points (e.g., op4 and op5 in Figure 2(c)), which are already identified by T-piper during pipeline synthesis. A write-data source is either an output of an uninterpreted function (assumed correct in P-uf-out) or a state read interface (assumed correct in P-rd). The write-data source correctness assumption removes from the P-wr's cone of influence the pipeline logic that computes the write-data.

- A P-rd property is proven by assuming that all the possible forwarding sources to that read interface are correct. As mentioned earlier, T-piper already identified all forwarding sources. Therefore no additional analysis is needed to determine these sources. Each forwarding source should either be an uninterpreted function output (assumed correct in P-uf-out) or a state read interface (assumed correct in P-rd). T-piper will not create cyclic dependency in P-rd. The forwarding source correctness assumption removes from the P-rd's cone of influence the pipeline implementation logic that computes the forwarded values.
- A P-uf-out property is proven by assuming that all its inputs are correct. Each of the uninterpreted function inputs should either be a state read data (assumed correct in P-rd) or an uninterpreted function output (assumed correct in P-uf-out). T-piper will not create cyclic dependency in P-uf-out. The assumption removes from the P-uf-out's cone of influence the pipeline implementation logic that produces the inputs to the uninterpreted function.

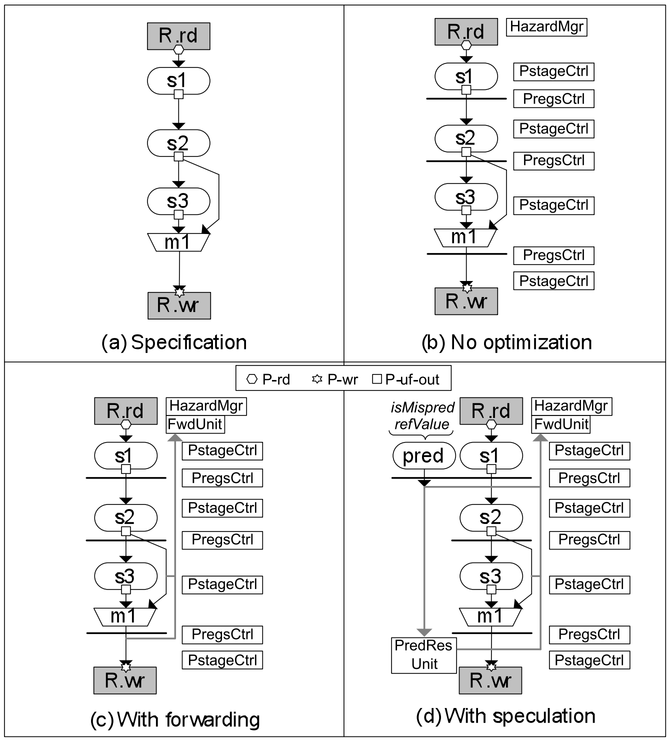

Fig. 5. Examples of verifying pipelines with stalling, forwarding, and speculative execution.

Case Splitting. T-piper automatically performs case splitting on the aforementioned properties, as follows:

- A P-wr property is split into cases that consider all possible values of each of the write-data sources to the state write-data being proven.
- A P-rd property is split into cases that consider all possible values of the state read-data. For an array state, the split also considers all possible values of the array read index.
- A P-uf-out property is split into cases that consider all possible values of its output and its inputs.

T-piper defines the variables involved in the case splitting as symmetric so that the model checker can perform data type reduction accordingly.

D. *Examples*

Figure 5(a) depicts the verification model for the specification datapath from the T-spec in Figure 2(c). 3 uninterpreted functions are used in these models, s1, s2, and s3. Notice that the next-state compute blocks op2 and op3 in Figure 2(c) are abstracted as a single uninterpreted function s1. Figure 5(b), 5(c), and 5(d) show the implementation models generated by T-piper to verify the pipelines in Figure 3(a), 3(b), and 3(c), respectively. These figures also show the correctness properties automatically placed by T-piper, which are used to decompose the verification problem into smaller sub-problems. Note that without abstractions and decompositions, SMV would encounter state explosion on even these simple examples.

In Figure 5(b), the pipeline has no optimization (hazards are resolved by stalling). Despite the simplicity, the pipeline still needs to implement a variety of control blocks (i.e., PstageCtrl, HazardMgr, PregsCtrl) for correctness (shown in white boxes in the figure). Also, the read and write enables of R are declared as free variables so that all possible transaction sequences in the pipeline are explored.

For the pipeline in Figure 5(c), T-piper includes the forwarding paths and logic (FwdUnit) in the model. Additionally, the select signal for the multiplexer m1 is defined as a free variable, so that the model checker would explore all possible forwarding scenarios. Although not shown, the models in Figure 5(b) and 5(c) have coordination logic to stall the specification model when the implementation model stalls.

Finally, in the case in Figure 5(d), a predictor logic block pred is added for speculative execution, with the predicted value forwarded from stage 2 to stage 1, and the prediction resolved in stage 4. T-piper exposes the prediction forwarding path as well as the prediction resolution unit (PredResUnit). It also augments the model with (1) the isMispred free variable to emulate speculative execution, (2) the reference RF state update value (refValue) from the specification that is used to model the correct prediction forwarding, (3) the coordination logic to stall the specification model when the implementation is emulating a misprediction, and (4) the flag bits to track 'shadow' transactions in the pipeline (Section IV.B).

E. *Applicability to Other Pipeline Synthesis Frameworks*

This section has described the proposed automatic verification approach in the context of T-spec/T-piper. However, it may be possible to apply the approach to other high-level pipeline synthesis frameworks, provided that these frameworks capture at the least the same amount of precise details of the input specification and the target pipelines as the

T-spec/T-piper framework (e.g., state read/write interfaces, hazard resolution scheme to apply, etc).

V. Case Studies

We have applied the formal verification integration in Section IV so that T-piper generates both an RTL-Verilog and an SMV input file for Cadence SMV [12]. We conducted case studies based on the processor T-specs in Figure 6. Figure 6(a) shows a processor T-spec for the MIPS RISC ISA; Figure 6(b) shows T-spec for a hypothetical ISA with CISC-like memory-memory instructions. The datapath style in Figure 6(b) is inspired by the Intel Atom® processor pipeline, which does not break x86 memory-memory instructions into multiple RISC-like micro-operands. True to CISC-style architectures, the pipeline also handles variable length instructions. From these two T-specs, we used T-piper to synthesize a total of 18 pipelined processor implementations, varying in their pipeline depths (4, 5, and 6 stages) and hazard resolution schemes (with stalling only (N), with maximal data forwarding (F), and with both forwarding and speculative execution (S)).

We collected verification metrics similar to those in [10]:

- The number of correctness properties—this is an indication of the required level of verification effort and knowledge.
- The number of state variables for the property with the largest cone of influence—this static number is an indication of the likelihood of encountering state explosion.
- The number of BDD nodes allocated during model checking—this runtime measure of SMV data size is also an indication of the likelihood of encountering state explosion.
- The execution time spent by the model checker to complete the verification.

A. Results

Figure 7 summarizes the results from verification using Cadence SMV running on a 2 GHz PC with 4 GB of DRAM. The results indicate the following:

- Not surprisingly, deeper pipelines are more expensive to verify (i.e., states, time, and BDD nodes increase), since each additional stage adds more pipeline registers and control logic.
- More complicated hazard resolution schemes (forwarding, speculation) only require a few additional state variables, but introduce many more possible state transitions (e.g., various data forwarding scenarios) that result in significant increase in model checking time and BDDs.
- Surprisingly (at least at first), MIPS and the CISC-like memory-memory processor pipelines incur a similar level of verification effort. One might expect that the CISC-like pipeline would require a higher verification effort since the datapath requires a complex memory address compute module to facilitate instructions reading from and writing to the memory in a single pipeline pass, as well as a more sophisticated next PC compute module to accommodate variable length instructions. However, T-piper abstracts these memory address and next PC compute modules as black-boxes using uninterpreted functions during formal verification. Thus, verification cost is more affected by the complexity of the pipeline control logic injected by T-piper.

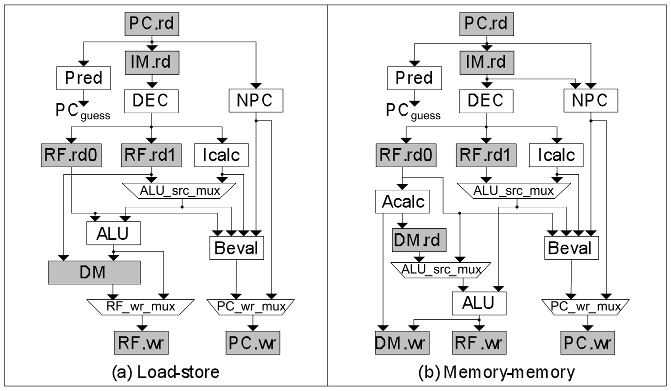

Fig. 6. T-specs for processor datapaths under study.

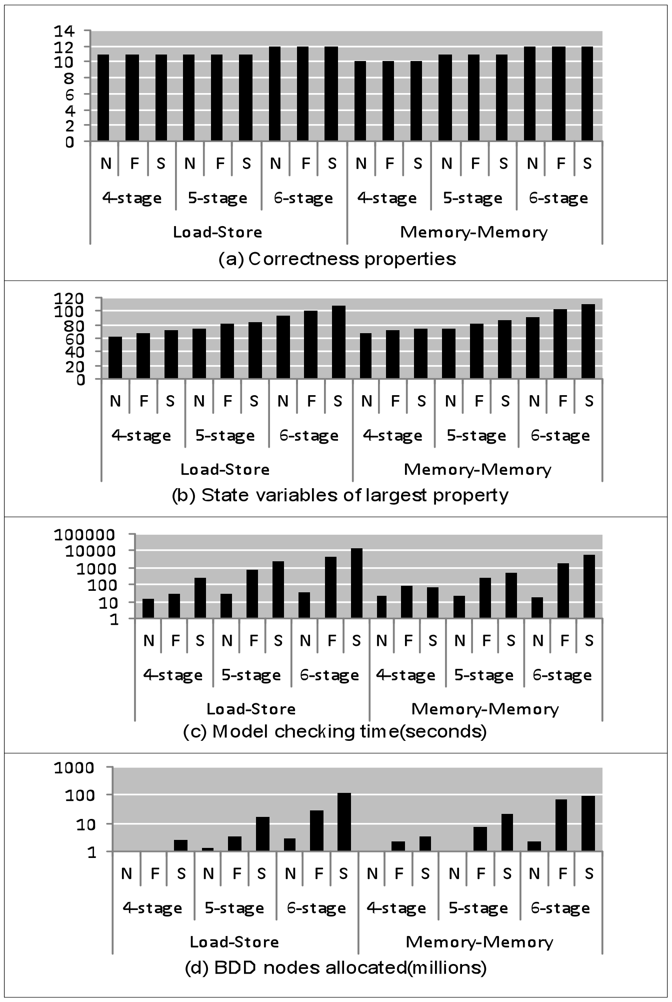

Fig. 7. Verification of load-store and memory-memory processor pipelines. Note that the N, F, and S labels correspond to pipelines with no forwarding and speculation, with forwarding only, and with both, respectively.

B. *Catching Real Bugs*

Our case studies did uncover a number of real "programming" bugs in T-piper and fortunately no "algorithmic" bugs so far (Section IV.A). One example of a failed verification was caused by an error in the coding of the RTL-to-SMV printing function, resulting in an incorrect SMV model from what is a correct internal representation. Model checking successfully produced a counter example to help us pinpoint a T-piper programming bug. While in this case the internal representation and the Verilog output were both correct, one can also imagine an opposite scenario where the typo is in the Verilog instead of in the SMV model. This speaks strongly for formal verification technologies that apply directly to the implementation design file.

C. *Sanity Check*

Lastly, the automatic verification quality of T-piper was validated against manual effort. We created a T-spec for a simple 3-stage pipeline example from Cadence SMV tutorial [12] and automatically verified it using T-piper. We found that the verification metrics collected from our automated approach are comparable to those gathered from the original example from the tutorial, which were done manually.

VI. Related Work

The most common functional verification approach in use today is still simulation-based validation, where an RTL description of the design is simulated with test inputs, and its output is checked for correctness. However, simulation-based validation can only ensure correctness of the behaviors exercised by the test inputs. In practice, simulation-based validation cannot achieve full coverage by brute force due to the prohibitively long simulation time required. Existing studies (e.g., [1][4]) proposed generating the test inputs formally to ensure adequate coverage.

Existing high-level design frameworks (e.g., [11][13][15]) provides correct-by-construction property, but do not have integrated formal verification to ensure the correctness of the output implementation that it generates. Thus, they are prone to aforesaid "programming" and "algorithmic" bugs. [3] has integrated assertion-based verification with [15]. However, to the best of our knowledge, no existing high-level framework has an integrated automatic compositional model checking capability.

In terms of formal verification approach that can be integrated with these high-level frameworks, besides SMV [12], theorem proving involves deriving a mathematical description of the whole system and creating proofs that certain properties hold. Burch and Dill [2] derive an abstraction function by "flushing" the implementation, and proves a commutative diagram. The proof in [9] leverages logic with equality, uninterpreted functions, etc. Such theorem proving techniques including [7][14] can automate the more menial steps in a proof but require user to figure out a strong invariant. In contrast, model checking computes the reachable states automatically, so requires less manual effort in deriving the formal model, and defining and guiding the proofs. Given just an implementation, [6] extracts its pipeline control logic for property verification with its abstracted datapath. This work leverages knowledge from T-spec and T-piper for more direct formal verification.

VII. Conclusion

This paper presents an integrated automatic synthesis and verification framework useful in ASIP development. The approach formally verify custom pipelined processors synthesized by T-piper from a given T-spec. T-piper automatically generates input file to Cadence SMV, which verifies that the pipeline synthesized by T-piper is functionally equivalent to its T-spec specification, under the T-spec transactional semantics. A case study on automatic verification of various custom RISC and CISC-like pipelined processors synthesized by T-piper demonstrated the effectiveness of the proposed approach. The usefulness of the approach in uncovering real-life implementation bugs in T-piper was discussed.

Acknowledgment

We thank Prof. Edmund M. Clarke from School of Computer Science at Carnegie Mellon, Scott Robinson from Intel, and our colleagues in the Computer Architecture Lab at Carnegie Mellon for their interaction and feedback.

References

[1] M. Behm, et al., "Industrial Experience with Test Generation Languages for Processor Verification", Design Automation Conference, 2004.

[2] J. R. Burch and D. L. Dill. "Automated verification of pipelined microprocessor control", Computer Aided Verification, 1994.

[3] A. Chattopadhyay, et al., "Integrated Verification Approach during ADL-Driven Processor Design", Rapid System Prototyping, 2006.

[4] M. Chen and P. Mishra, "Functional Test Generation using Efficient Property Clustering and Learning Techniques", IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, 2010.

[5] E. Clarke, et al., "Model Checking", The MIT Press, 2000.

[6] P. Ho, A. J. Isles and T. Kam, "Formal Verification of Pipeline Control using Controlled Token Nets and Abstract Interpretation", International Conference on Computer-Aided Design, 1998.

[7] R. Hosabettu, et al., "Verifying advanced microarchitectures that support speculation and exceptions", Computer Aided Verification, 2002.

[8] R. Jhala, K. L. McMillan, "Microarchitecture Verification by Compositional Model Checking", Computer Aided Verification, 2001.

[9] S. Lahiri, S. Seshia and R. Bryant, "Modeling and Verification of Out-of-Order Microprocessors in UCLID", Formal Methods in Computer-Aided Design, 2002.

[10] A. Lungu and D. J. Sorin. "Verification-Aware Microprocessor Design", Parallel Architecture and Compilation Techniques, 2007.

[11] M. V. Marinescu and M. Rinard, "High-level Automatic Pipelining for Sequential Circuits", International Symposium on System Synthesis, 2001.

[12] K. L. McMillan. "Getting Started with SMV", Cadence Berkeley Laboratories, 2001.

[13] E. Nurvitadhi, J. C. Hoe, T. Kam, S. L. Lu, "Automatic Pipelining from Transactional Datapath Specifications", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2011.

[14] J. Sawada, J. W. A. Hunt, "Trace Table Based Approach for Pipelined Microprocessor Verification", Computer Aided Verification, 1997.

[15] O. Schliebusch, et al.,"RTL Processor Synthesis for Architecture Exploration and Implementation", Design Automation and Test in Europe, 2004.