Automatic Pipelining from Transactional Datapath Specifications

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*—We present a transactional datapath specification (T-spec) and the tool (T-piper) to synthesize automatically an in-order pipelined implementation from it. T-spec abstractly views a datapath as executing one transaction at a time, computing next system states based on current ones. From a T-spec, T-piper can synthesize a pipelined implementation that preserves original transaction semantics, while allowing simultaneous execution of multiple overlapped transactions across pipeline stages. T-piper not only ensures the correctness of pipelined executions, but can also employ forwarding and speculation to minimize performance loss due to data dependencies. Design case studies on RISC and CISC processor pipeline development are reported.**

I. INTRODUCTION

Pipelining is a widely applied microarchitectural performance optimization. However, pipelining a datapath by hand is tedious and error prone, as it requires the designer to reason about subtle corner cases when sequentially dependent operations are processed concurrently by different pipeline stages. Non-trivial logic must be added to a pipeline to detect and resolve (by stalling, forwarding, and/or speculation) read-after-write (RAW) hazards that can possibly occur from such pipelined execution.

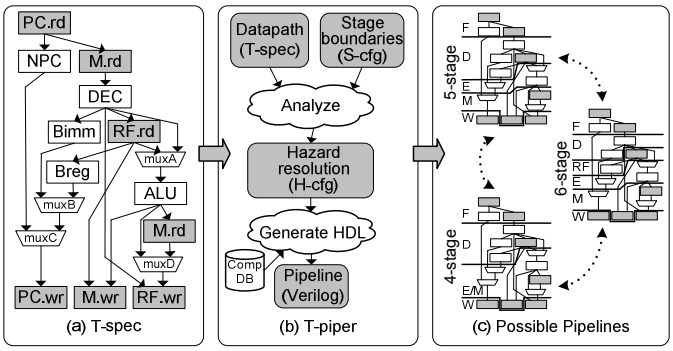

Figure 1. Pipeline development using T-spec and T-piper.

This paper presents a transactional datapath specification (T-spec) and its accompanying synthesis tool (T-piper) to simplify in-order pipeline development. Using T-spec and T-piper, the designer is only concerned with capturing in T-spec (e.g., Fig. 1.a) a non-pipelined version of the datapath that executes transactions one at a time. Each transaction reads the state values left by the previous transaction, computes a new set of state values, and commits them for the next transaction to see. To arrive at a pipelined implementation, T-piper analyzes the T-spec (Fig. 1.b) along with pipeline-stage boundaries specified by the designer (S-cfg), and reports available opportunities for applying forwarding and speculation to resolve hazards (H-cfg). Based on the designer's selection of which forwarding and speculation optimizations to include as well as user-supplied datapath components (Comp. DB), T-piper generates an RTL-Verilog implementation of the desired pipeline, which preserves the transaction semantics of the T-spec datapath. From the same T-spec, the designer can rapidly explore the pipeline design space by submitting different pipeline configurations to T-piper. We demonstrate the effectiveness of T-spec and T-piper on RISC and CISC processor pipeline development through design case studies.

The rest of the paper is organized as follows. Section II summarizes relevant work. Section III presents the T-spec transactional datapath specification. Section IV explains T-piper pipeline synthesis. Section V presents the results from our case studies. Section VI offers concluding remarks.

II. RELATED WORK

Prior studies (e.g., [1][2][4][5][6][7][8][9][10]) have proposed automated pipeline synthesis to alleviate the manual effort of pipeline development. However, they still require manual effort to identify forwarding opportunities and place the forward paths. Furthermore, they only accommodate restricted forms of speculation, or none at all. [6][9] support custom hand-written predictor modules but require manual implementation of resolution and recovery. Finally, some prior efforts [4][6][9][10] focus only on instruction set processors, and can not handle arbitrary sequential datapaths.

III. TRANSACTIONAL DESIGN

A. Conventional Thinking in Pipelining

A typical pipeline design development begins with creating an initial non-pipelined (so-called "single-cycle") reference implementation where each system state is instantiated explicitly and, in each clock cycle, a set of combinational logic operations computes the next-state based on the current state. For example, in a prototypical development of a RISC processor pipeline, the ISA states are instantiated in the single-cycle implementation, where they are transformed according to the execution of one instruction per cycle [3]. Starting from this reference design point, the pipelining transformation first establishes the desired pipeline stage boundaries, dividing the next-state logic datapath into multiple segments as pipeline stages. To support overlapped execution of multiple operations, hazard detection and stall logic are introduced to maintain correctness of operations in the overlapped executions.

This work was supported in part by a grant from the Intel Corporation. We thank Scott Robinson from Intel and our colleagues in the Computer Architecture Lab at Carnegie Mellon for their interaction and feedback.

Forwarding and/or speculation may be added to minimize performance loss due to stalls. The basic methodology for pipelining is well established but nevertheless tedious and error-prone if applied manually and haphazardly.

A single-cycle datapath is much simpler to specify and implement than the final pipelined version, since the designer is only concerned with next-state computation that occurs in a single step, avoiding the need to reason about interactions among multiple concurrent overlapping operations.

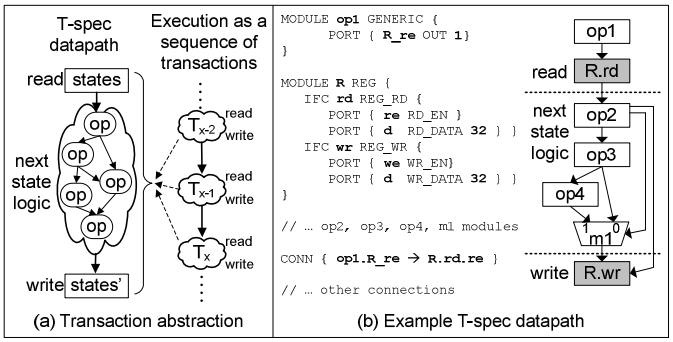

Figure 2. An example datapath and its T-spec.

B. Transactional Datapath Specification (T-spec)

We propose a transactional specification (T-spec), which adopts and expands on the basic thinking on datapath specification discussed in the previous section. Like a single-cycle design, T-spec is a textual "netlist" that consists of state elements and next-state compute operations implemented by a network of logic blocks. Unlike a single-cycle implementation, a T-spec captures an abstract datapath whose execution semantics are interpreted as a sequence of "transactions". Each transaction reads the state values left by the preceding transaction and computes a new set of state values to be read by the next transaction (Fig. 2.a.). Many valid implementations may be derived from a T-spec as long as they preserve the transaction semantics. This paper describes how an in-order pipeline is synthesized from a T-spec.

State Elements. Without loss of generality, the T-spec netlist can include register- or array-type state elements, with explicit read and write interfaces. An unusual feature of T-spec is that a state-read interface includes an explicit "read-enable" control signal, which is a bookkeeping signal to assist T-piper in RAW hazard analysis by letting the designer indicate exactly when a transaction must see the valid value of a state in order to proceed. Similarly, an explicit "write-enable" is included in a state-write interface.

Next-State Logic Blocks. An acyclic network of logic blocks computes the next-state updates for the write-interfaces of the state elements based on the current state values received from the read-interfaces of the state elements. Except for multiplexers, T-piper treats all such next-state logic blocks as black-boxes during analysis. A multiplexer is a built-in logic primitive understood by T-piper and used for hazard analysis.

Unlike the single-cycle design where the next-state logic blocks are implemented using combinational logic, each next-state logic blocks in T-spec can either be a combinational block or a fixed/variable multi-cycle (MC) block (e.g., iterative divider). A MC block implements a handshake interface based on ready, start, and done signals [1]. Each handshake interface can only be used once by each transaction. Asserting start implicitly resets any internal state so no history can be carried from one transaction to the next. Although not discussed in detail here, T-spec also supports state elements with handshake interfaces, which can represent structures like a memory cache.

T-spec Example. Fig. 2.b shows an example design with a single state element R and a network of logic blocks (op1, op2, op3, op4, and m1), with R represented as its separate read and write interfaces. It also shows an example T-spec excerpt for the design, which begins with a GENERIC module declaration for op1, a black-box combinational block with a 1-bit output named R_re. (This input-less block is for a hardwired constant.) Then, the module declaration for a REG-type 32-bit state R is shown, which has explicit read (rd) and write (wr) interfaces with enable and data ports. Lastly, a connection is declared between R_re output of op1 and rd.re input of R.

IV. Pipeline Synthesis

A. Pipeline-Stage Boundaries

The desired pipeline stage boundaries are expressed in T-spec by declaring the stages, and assigning each module (or interface in the case of a state element) to a stage. For example, the pipeline boundaries in Fig. 3.a (shown in solid lines) is accomplished by assigning op1, op2 and R.rd to stage 1; op3 to stage 2; op4 and multiplexer m1 to stage 3; and R.wr to stage 4.

When assigning modules to stages, the destination module of a connection cannot be assigned to a stage earlier than the source module. The write interface of a state also cannot be assigned to a stage earlier than the state's read interface. If an array state supports multiple write interfaces, they must be assigned to the same stage. The constraints on state read and write interface assignments exclude the possibility of write-after-write and write-after-read hazards.

B. Hazard Detection and Interlock

Given a datapath in T-spec and pipeline-stage assignments, T-piper analyzes the input design for RAW hazards when transactions are executed in a pipelined fashion. For example, the T-spec datapath from Fig. 2.b is divided into four stages in Fig. 3.a. As such, multiple versions of a signal co-exist if the source and destination of a connection are not assigned to the same pipeline stage. These versions correspond to different transactions in different stages. In Fig. 3.a, the we signal generated by op2 to control R.wr traverses all four stages. When op2 is computing we_x for transaction T_x in stage 1, we_{x-1} is an older we version from the older transaction T_{x-1} in stage 2. Similarly, we_{x-2} and we_{x-3} belong to T_{x-2} and T_{x-3}, respectively.

For hazard analysis, T-piper identifies for each state element a "read-point" associated with the state element's read interface (e.g., the read-point for R is labeled with a "triangle"-symbol in Fig. 3.a). A read-point is required to carry a valid state value only when its accompanying read-enable is asserted. Similarly, a "write-point" is associated with the write-data interface of the state element, and is qualified by the write-

enable (e.g., the write-point for R is labeled with a "star"-symbol in Fig. 3.a,). A write-point carries the new state update value only if its write-enable is asserted.

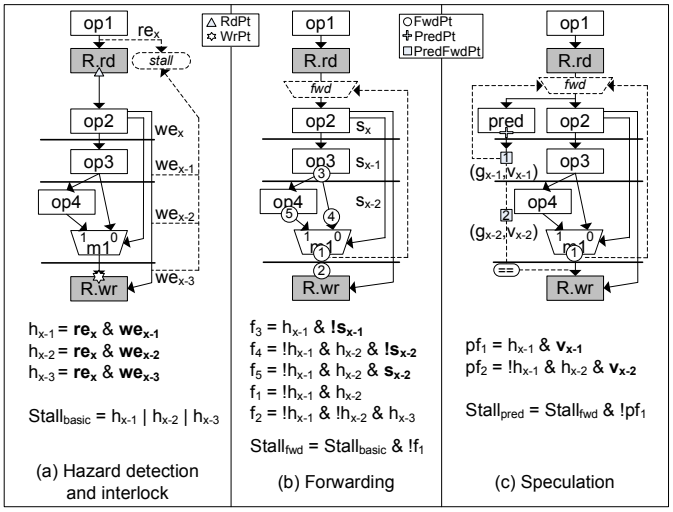

Figure 3. Data hazard analysis and resolution.

With respect to each state element E in a datapath, a hazard condition exist whenever re_x&we_{x-i} where T_x is the transaction occupying the stage of E's read-point and T_{x-i} is some older transaction in a stage between the read-point and the write-point of E. When re_x&we_{x-i}, T_x will receive an incorrect state value if it reads from E directly because the update by T_{x-i} has not yet been written to E. Thus, T_x must stall (i.e., delaying the reading of E) if re_x&we_{x-i} is true for any stage between the read-point and the write-point of E. For the example in Fig 3.a with a state element named R, the expression $Stall_{basic}$ indicates when the reading of R must be stalled. When stalling the transaction T_x, the older transactions in the pipeline must be allowed to proceed so T_{x-i} will eventually progress pass the write-point of E, removing the hazard condition. Note that we may not exist for a stage if it is computed by a module that is at a later stage. In this case, we must be conservatively assumed true whenever the stage is occupied by a transaction. The aforesaid hazard analysis is performed for each state element in the datapath.

If the state element E is an array, T-piper carries out hazard analysis at individual location granularity. If rd-idx and wr-idx are the indices to the read and write interfaces of E, a hazard condition arises only if re_x&we_{x-i} and $rd\text{-}idx_x$==$wr\text{-}idx_{x-i}$. In other words, the read and write of the array element E by T_x and T_{x-i} only conflict if they are to the same location in E.

C. Forwarding

In some cases, stalling can be avoided if the required not-yet-committed state update values from an older transaction still in flight can be forwarded (bypassing the state element) to the younger dependent transaction. To determine forwarding opportunities, T-piper further identifies a set of "forwarding-points" (FwdPt) for each state element. Starting from each write-point, T-piper traces the write-data signal backwards across pipeline boundaries to find all points (output of a module or a pipeline-stage register) where the write-data signal and its accompanying write-enable signal are both available. When the associated write-enable is asserted, the value at a forwarding-point can be provided to the read-point for use by a dependent younger transaction in lieu of stalling. To make a valid forwarding, one must also ascertain that no other transactions between the read-point and the forwarding-point also want to write to the state element. Forwarding-points can be traced backwards through a multiplexer to create conditional forwarding further qualified by the mux-select logic. To automatically find forwarding-points, T-piper uses a recursive depth-first algorithm starting from the write-point of a state.

Fig. 3.b shows the 5 forwarding-points of R (labeled by "circle"-symbols) and the exact condition when the value at each forwarding-point can be used by the transaction at the read-point stage in lieu of stalling. For example, forwarding-point 2 is valid iff T_x depends on T_{x-3} with respect to R but not on T_{x-2} or T_{x-1}. Points 5 and 4 are conditional forwarding-points corresponding to the two possible settings of the m1 multiplexer select. The condition for using forwarding-point 5 (or 4) is the same as 1 with the additional requirement that the m1 multiplexer is set to select the 1-path (or the 0-path).

After the analysis, T-piper reports all forwarding-points to the user, and lets the user select which ones to include in the pipeline to be synthesized by T-piper. When a forwarding path is added, its exact trigger condition is subtracted from the stall condition. When the trigger condition occurs at run time, the would-be RAW hazard is resolved by forwarding from the corresponding forwarding-point. The example in Fig. 3.b implements forwarding-point 1, resulting in a new stall condition $Stall_{fwd}$ that subtracts f_1 from $Stall_{basic}$.

D. Speculative Execution

Forwarding can only be done if a transaction already computes (but has not yet written) the value that a younger transaction depends on. Consider the example in Fig. 3.b. If a transaction in stage 1 depends on the value of R to be produced by an older transaction currently in stage 2 via op4 (i.e., mux select is 1), then it has to wait for 1 cycle for the older transaction to reach stage 3 and utilizes op4 to compute the value, which can then be forwarded using forwarding-point 1.

T-spec supports a general-purpose framework for a designer to provide a value predictor to overcome this kind of performance limitations rooted in true data dependence. Starting with a T-spec datapath with complete functionality, a designer can introduce additional state elements and logic blocks for value prediction. In parallel to the original full determination of the next-state value of a given state element E, the auxiliary states and logic blocks are to compute, presumably faster, a "guess" for the next-state value of E. With each guess, the auxiliary logic also generates a Boolean valid signal to indicate whether the guess should be used for speculation (e.g., when a guess' confidence is low, stalling is preferred over speculation to avoid misprediction penalty).

The auxiliary states and logic blocks for making value predictions are specified using the same T-spec syntax as the original primary state and logic. They are allowed to depend on the value and output of the primary states and logic blocks, but not vice versa. In other words, the operations of the primary datapath are not to change. Fig. 3.c shows the auxiliary logic

module pred making a guess for the next-state value of R in stage 1, whereas the true next-state value is not known until stage 3 (at the m1 output). In this example, a guess is based on the primary states only. In general, an arbitrary history-based value predictor can be made for any state element by adding auxiliary states and logic blocks.

For each predictor in T-spec that guesses the next-state value of a state element E, T-piper automatically generates a pipelined implementation that incorporates the predicted value (when the associated valid bit is asserted) in speculative executions. A prediction-point (PredPt in Fig. 3.c) is the output of a value predictor (g), and is qualified by its valid signal (v) generated also by the predictor. The value of a valid prediction-point can be forwarded to the read-point at a "prediction-forwarding-point" (PredFwdPt) in the same way as a forwarding-point. Fig. 3.c shows the possible prediction-forwarding-points, labeled by "box"-symbols. The figure also shows an implementation that makes use of prediction-forwarding-point 1, resulting in a new stall condition $Stall_{pred}$.

T-piper generates automatically the mechanism to track and eventually verify a transaction's predicted next-state value for E against the dutifully calculated true next-state value for E. By default, the check is done at E's write-point of (e.g., stage 4 in Fig. 3.c), minimizing resolution logic at the cost of increasing misprediction penalty. The user can also instruct T-piper to check prediction earlier by comparing against user-selected forwarding-points to reduce misprediction penalty. During a prediction check, if the predicted and actual values agree, nothing more is done. However, if they disagree, all younger transactions in flight after the mispredicting transaction must be squashed from the pipeline. The mispredicting transaction itself can complete fully since it never made use of the prediction. The execution continues by restarting the next transaction using the correct state values. Due to the need to squash and restart, no write-points for any state elements may be assigned to a stage where unchecked value predictions remain. This ensures that flushing the pipeline registers is sufficient without needing to undo any state changes to the system state elements.

V. CASE STUDIES

We implemented the aforementioned pipeline synthesis in the T-piper tool, and used it in two design case studies. In the first study, we trained an undergraduate student to design a 5-stage MIPS pipeline in T-spec. The student has prior experience in developing a textbook 5-stage MIPS pipeline by hand [3]. Including training, the design was completed within a week. The synthesized pipeline is within 2% in performance and area of the student's hand-made design. Further, 3, 4, and 6-stage pipelines were synthesized from the T-spec only by slight modifications in the pipeline stage configuration file.

In a second study, we created a T-spec for an x86 processor (Fig. 6.a) that supports a sufficient subset of the x86 ISA to run 9 benchmarks from [10]. The T-spec included a bi-modal branch predictor. We used T-piper to synthesize 36 pipelines, varying in depths (4 to 7), forwarding (from earliest forwarding-points vs. whenever possible), and prediction resolution (resolve as soon as possible vs. as late as possible). Fig. 6.b shows the cost-performance tradeoff for the pipelines. The area is based on Synopsys DC synthesis targeting a commercial 180nm standard-cell library, and the average run-time is from RTL simulation cycle counts, adjusted by the frequency reported by Synopsys. Notice in Fig. 6.b that the Pareto optimal fronts are dominated by 4-stage pipelines because the increases in data stalls and EIP misprediction in deeper pipelines outweigh the frequency gains. Such a design tradeoff would have been difficult to learn without exploring many RT level designs.

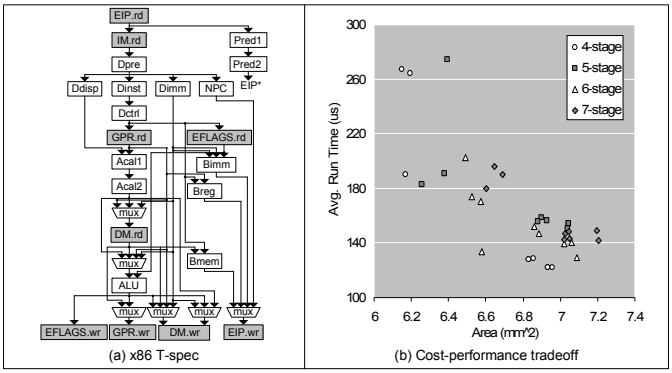

Figure 4. x86 design under study and results from the case study.

VI. CONCLUSION

We presented the transactional specification (T-spec) and pipeline synthesis technology (T-piper) to automate development of in-order pipelines. Two case studies show that (1) a synthesized MIPS pipeline is comparable to a manually designed one, and (2) this work enables rapid design exploration of x86 pipelines. Although the studies are for processors, this work can pipeline any sequential datapath.

REFERENCES

[1] J. Cortadella, M. Kishinevsky, B. Grundmann, "Synthesis of synchronous elastic architectures", Design Automation Conf., 2006.

[2] S. Hassoun and C. Ebeling, "Architectural Retiming: Pipelining Latency-Constrained Circuits", Design Automation Conference, 1996.

[3] J. Hennesy, D. Patterson, "Computer Architecture: a Quantitative Approach", Morgan Kauffmann, 1990.

[4] M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takeuchi, A. Kitajima, M. Imai, "PEAS-III: An ASIP Design Environment," International Conference on Computer Design, 2000.

[5] T. Kam, M. Kishinevsky, J. Cortadella, M. Galceran-Oms, "Correct-by-construction Microarchitectural Pipelining", International Conference on Computer-Aided Design, 2008.

[6] A. Kejariwal, P. Mishra, N. Dutt, "Synthesis-driven Exploration of Pipelined Embedded Processors", Int. Conf. on VLSI Design, 2004.

[7] D. Kroening, W. Paul, "Automated pipeline design", Design Automation Conference, 2001.

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

[9] O. Schliebusch, A. Chattopadhyay, R. Leupers, G. Ascheid, H. Meyr, et al., "RTL Processor Synthesis for Architecture Exploration and Implementation", Design Automation and Test in Europe, 2004.

[10] P. Yiannacouras, J. G. Steffan, J. Rose, "Exploration and Customization of FPGA-Based Soft Processors", IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, 2007.