Automatic Pipeline Synthesis and Formal Verification from Transactional Datapath Specifications

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Eriko Nurvitadhi

B.S., Electrical and Electronic Engineering, Oregon State University
B.S., Computer Engineering, Oregon State University
B.S., Computer Science, Oregon State University
B.A., International Studies, Oregon State University
M.B.A., Business Administration, Oregon State University
M.S., Electrical and Computer Engineering, Oregon State University

Carnegie Mellon University
Pittsburgh, PA

December, 2010

Ph.D. Thesis Committee

Prof. James C. Hoe (Advisor)

Prof. Edmund M. Clarke

Prof. Donald E. Thomas

Dr. Timothy Kam (Intel)

Dr. Shih-Lien L. Lu (Intel)

Keywords: datapath specification, automatic pipeline synthesis, automatic pipeline formal verification, multithreading, x86 design space exploration.

Copyright © 2010 by Eriko Nurvitadhi. All rights reserved.

Abstract

Pipelining a datapath by hand is tedious and error prone, as it requires a designer to reason about overlapped concurrent execution of sequentially dependent operations in different pipeline stages. Nevertheless, doing so is often necessary to meet performance targets and improve efficiency. Automation techniques have been proposed to reduce the manual effort in designing, implementing, and verifying pipelines. Nevertheless, they are limited in the extent and form of automation that they can do, and the type and size of designs that they can handle.

This thesis presents the transactional datapath specification (T-spec) and the technology (T-piper) to automatically synthesize and formally verify in-order pipeline implementations from it. T-spec elevates design abstraction by allowing a designer to reason about a sequential system at the transactional level, where state transformations happen in a single step, thereby relieving designer's burden to resolve subtle corner cases from concurrent execution due to pipelining. Unlike previous works, the proposed approach can automatically identity and place forwarding paths, support general value speculation (i.e., on any state, with custom predictors), and automatically perform scalable verification using compositional model checking. Further, it improves upon existing processor-specific works since it can handle any sequential designs. Finally, it is extendable to do multi-treaded pipeline synthesis, a novel capability not achievable by any previous work. The technology has been made available online at www.t-piper.net, and its effectiveness has been demonstrated by various design case studies.

Acknowledgements

First of all, I would like to thank my academic advisor, Prof. James C. Hoe. His advice and guidance have made this thesis possible. Furthermore, he has provided an empowering and productive environment for me to grow as a researcher.

I also thank my thesis committee, Prof. Edmund M. Clarke, Prof. Donald E. Thomas, Dr. Shih-Lien L. Lu (Intel), and Dr. Timothy Kam (Intel), who gave insightful advice on this work. In particular, Dr. Lu has been my mentor even before I came to CMU. Thanks for being supportive of me all these years. Prof. Clarke and Dr. Kam have been very kind in helping me with the formal verification and design automation aspects of this work, respectively. Dr. Scott Robinson (Intel) has provided me with useful feedbacks as well. I also would like to express my appreciation to Prof. Babak Falsafi and Prof. Ken Mai, whom I interacted with in my earlier years at CMU, prior to starting this thesis work.

I am very thankful to the many fellow students at the computer architecture lab who have helped me tremendously in many ways, such as with research infrastructure development and debugging, proofreading papers, practice talks, etc. I would like to specifically acknowledge, in order of graduation dates and seniority, Tom Wenisch, Jared Smolens, Jangwoo Kim, Brian Gold, Stephen Somogyi, Nikos Hardavellas, Roland Wunderlich, Peter Milder, Eric Chung, Mike Ferdman, Vasilis Liaskovitis, and Michael Papamichael. Furthermore, I really appreciate my study buddies outside of the lab, Nutthanon Leelathakul, Namtarn Chaipah, Fabian Manan, Livinia Effendy, Chris Chong, Fenny Shintawaty, and Chutika Udomsinn.

I am forever indebted for the endless patience, unconditional love, and unwavering support of my family, Mardi, Oely, Didi, Irena, and Fanuel. And, last but not least, my sincere gratitude to Jo Choi for being there for me, especially in the last stretch towards the completion of this thesis.

This work was supported in part by a grant from the Intel Corporation.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Pipelining [31] is a widely-used microarchitecture performance enhancement technique. It divides the critical path of a sequential circuit into multiple stages separated by pipeline registers, thereby reducing the critical path delay and increases clock frequency. To improve pipeline throughput and efficiency, multiple operations are allowed to simultaneously execute at the different pipeline stages. However, if an operation is dependent on the result of an older operation that is still in the pipeline, a condition known as a data hazard, it has to stall the pipeline to wait for the older operation to complete execution and commit the result. Forwarding (or bypassing) and speculation are common pipeline optimization techniques that allow early resolution of data hazards, and therefore improve performance by reducing the amount of pipeline stalls. Multi-threading is another optimization technique that improves pipeline efficiency when executing multiple sequences of operations, or threads, by allowing pipeline resources to be shared among these threads.

Manual pipeline development is tedious and error prone, as it requires a designer to reason with concurrent execution of sequentially dependent operations in the different pipeline stages, involving many possible scenarios and complicated corner cases.

Nevertheless, doing so is often necessary in practice to meet performance targets and improve efficiency.

To address this issue, many existing studies have proposed techniques to automate pipeline development. Some studies [9][19][25][27][28][30][33][39][47][58][71] target the design and implementation phases, by proposing automatic synthesis of pipeline implementations directly from a high-level design specification. Others studies [1][2][10][26][37][42][43][57][68] focus on the verification phase, by proposing techniques to ensure that a pipeline implementation is functionally equivalent to its high-level specification. However, these studies are limited in the extent and form of automation that they can do, and the type and size of designs that they can handle.

More specifically, with regards to automatic pipeline synthesis, existing works suffer from the following shortcomings. First, some of them target only instruction-set processors, limiting the scope of the designs that can be pipelined. Second, they cannot automatically identify forwarding opportunities and place forwarding paths, an effort that can be prohibitively expensive for designs with large number of states and pipeline stages. Third, they only accommodate restricted form of speculation, or not at all, even though sophisticated form of speculation is commonly found in commercial pipelines. Finally, all the aforementioned works target only in-order pipelines without any multi-threading support, although multi-threading has gained popularity in practice, as is evident by its adoption in commercial pipelines, such as in Intel Atom® [24] and Sun Niagara® [32] processor pipelines.

In terms of automatic pipeline verification, various studies [1][2][68] have mostly focused on automatic generation of test cases to be used for simulation-based validation. However, validation typically suffers from long simulation time, resulting in the inability to test the entire design space in practice. Others [10][26][37][42][43][57] target formal verification, which promises full design space coverage but generally suffers from scalability limit and/or the need for large manual effort.

1.1. Transactional Datapath Specification, Synthesis, and Verification

This thesis presents the transactional datapath specification framework (T-spec) and the transactional design automation system (T-piper) to automatically synthesize and formally verify in-order pipeline implementations from it.

The basis of the automation work presented in this thesis is the novel T-spec, which makes pipeline synthesis problem solvable. 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 to be seen by the next transaction. T-spec exposes sufficient information about state accesses that can occur in a datapath, which is necessary for performing precise data hazards analysis, and eventually pipeline synthesis. Furthermore, not only T-spec makes pipeline synthesis possible, but its precise semantics also makes functional verification between the T-spec datapath and the synthesized pipeline implementation natural to do.

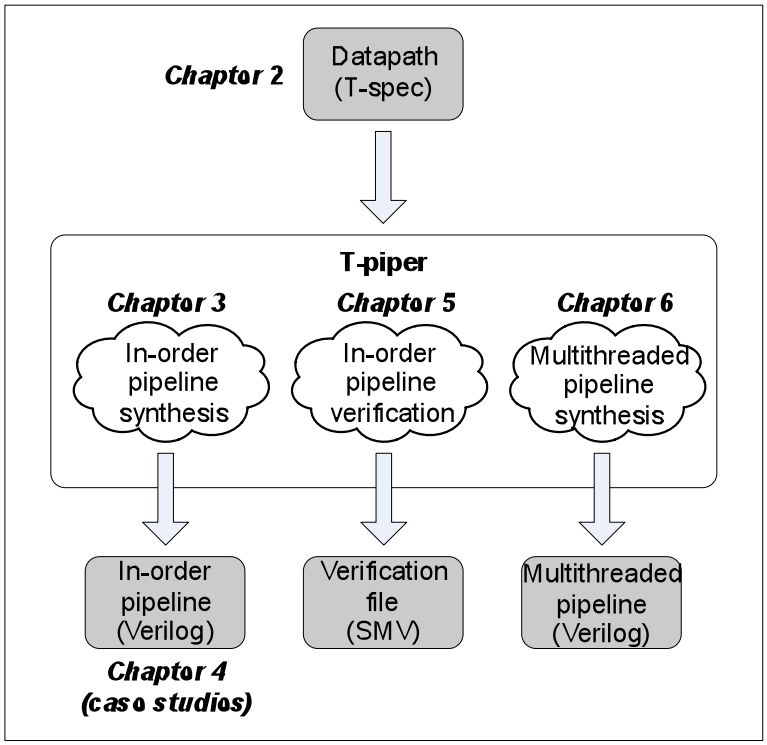


Figure 1. Overview of the thesis work.

1.2. Thesis Contributions

The overview of the thesis work is presented in Figure 1, which illustrates the T-piper pipeline synthesis and verification technologies enabled by T-spec. More specifically, the contributions of this thesis work are as follows.

The transactional datapath specification (T-spec) [47][48]. T-spec elevates design abstraction by allowing a designer to reason about a system at the transaction level, where state transformations happen in a single step. This relieves designer's burden from having to resolve subtle corner cases associated with the concurrent overlapped execution caused by pipelining. T-spec makes pipeline synthesis problem solvable, and is highly amenable to microarchitecture synthesis in general.

T-piper in-order pipeline synthesis [47][48]. Starting from a datapath specified in T-spec and the desired pipeline stage boundaries (P-cfg), automated analysis can be done to gather information about data hazards that can be used in pipeline synthesis and verification. The analysis generates a hazard resolution configuration (H-cfg), which describes all hazard resolution opportunities (i.e., forwarding, speculation) and the suggested resolution strategy according to some predefined schemes. Alternatively, a designer can manually modify H-cfg to target a particular resolution strategy of interest.

After the data hazard analysis, automatic pipeline synthesis can then generate the desired pipelined implementation. Unlike previous works, the proposed approach can pipeline any arbitrary datapath, automatically identify and place forwarding paths, and support general value speculation.

Case studies using T-spec and T-piper [48][51] demonstrate that: (1) a synthesized MIPS 5-stage pipeline is comparable in performance and area to a hand-made one, as well as one generated by an existing processor development framework; (2) rapid design space exploration of x86-subset processor pipelines varying in pipeline depths, forwarding schemes, and speculation schemes is achievable; and (3) the automation capability of T-spec and T-piper allows investigation of various application-specific customizations to significantly reduce the costs of supporting a sophisticated CISC ISA such as the x86.

T-piper in-order pipeline verification [49]. Alongside synthesis, a verification file can be automatically generated. The file contains verification models of the T-spec and the synthesized pipeline, along with the appropriate abstractions and proof

decompositions. The file can be submitted to a compositional model checker to formally verify that the synthesized pipeline is functionally equivalent to its T-spec, under the transactional execution semantics.

Unlike existing works on simulation-based validation, the proposed approach employs a formal technique, and therefore can cover the entire design space. Relative to other formal techniques, compositional model checking is more scalable, since it allows dividing the verification problem into smaller sub-problems that are individually manageable to handle.

Case studies on the verification of various non-trivial load-store and memory-memory processor pipelines demonstrate the usefulness of the proposed T-piper in-order pipeline verification approach.

T-spec and T-piper extensions for multithreaded in-order pipeline synthesis [50]. These extensions allow T-spec to capture a datapath that executes multiple threads of transactional executions, and provides the necessary information for T-piper to automatically synthesize a multithreaded in-order pipelined implementation from a given T-spec.

Furthermore, the extended T-piper maintains the original non-threaded pipeline synthesis features (e.g., forwarding, speculation) while supporting various multithreading features, consisting of those found in modern in-order multithreaded pipelines (e.g., global state sharing, replay on long-latency events) as well as novel ones (e.g., state sharing by thread groups).

A case study demonstrates the effectiveness of the approach in the design space exploration of x86-subset processor pipelines varying in their multithreading optimizations. All of the pipelines are synthesized from a single T-spec.

Note that no existing studies can synthesize multithreaded pipelines automatically from a high-level datapath specification. Therefore, this work is the first to provide such a capability.

1.3. Thesis Organization

The rest of the thesis delves in detail into the work behind each of the contributions, and is organized as follows. Chapter 2 presents the proposed transactional design approach, including the details on T-spec. Chapter 3 elaborates on the T-piper in-order pipeline synthesis approach. Chapter 4 summarizes the results of design case studies using T-spec and T-piper. Chapter 5 provides details on the automatic pipeline verification using compositional model checking. Chapter 6 discusses the extensions to T-spec and T-piper to support multithreading. Chapter 7 provides details on the relevant prior works. Finally, Chapter 8 offers concluding remarks and discussion on possible future works.

Chapter 2

Transactional Datapath Specification

Transaction is a well-known abstraction that can be used to reason about concurrent executions. It has been widely used in various areas, including databases [15] and (more recently) parallel programming [36]. In this work, we apply this abstraction to hardware design specification. This section provides details on our transactional datapath specification approach.

2.1. Conventional Thinking in Pipelining

Many pipelined design developments begin 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 RISC processor development, the instruction-set architectural states are instantiated in the single-cycle implementation, where they are transformed according to the execution of one instruction per cycle [20]. Starting from this reference design point, the pipelining transformation begins with first establishing 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 is introduced to maintain the correctness of the operations in the overlapped executions. As necessary, forwarding

and/or speculation are 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 version is much simpler to specify and implement correctly than the final pipelined version. The single-cycle version also serves very effectively as a functional specification of the final pipelined design, as well as a reference model utilized in many verification techniques (e.g., [43]). In fact, during design exploration, a single functional specification may be transformed into multiple pipeline implementations. Thus, it is useful to distinguish, respectively, the "what" from the "how" of pipeline designs. The simplicity of the single-cycle version is that the designer is only concerned with next-state computation that happens in a single step, avoiding the need to reason with the interactions between multiple concurrent overlapping operations. T-spec adopts and expands on this basic thinking on datapath specification.

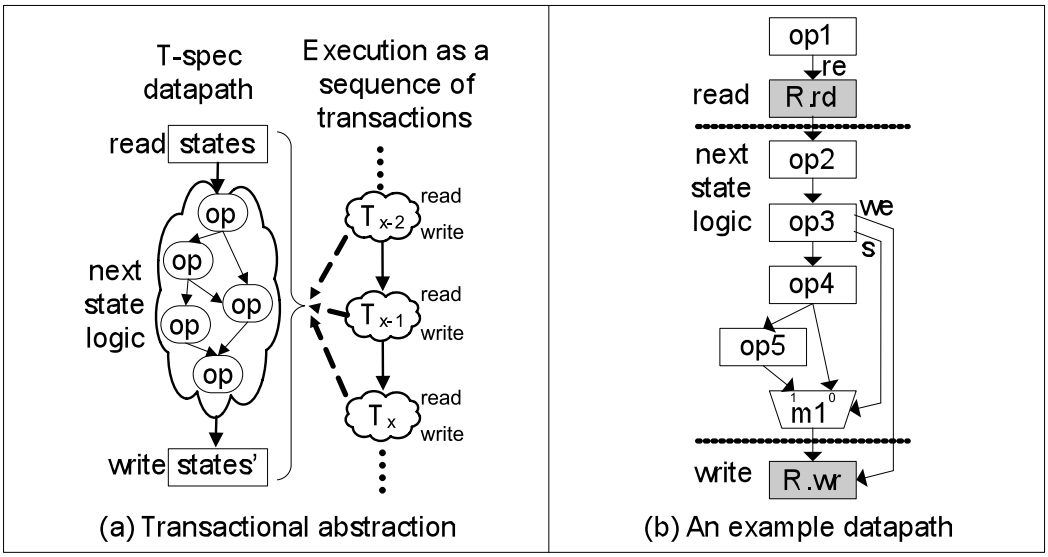


Figure 2. Transactional abstraction and an example datapath.

2.2. Generalizing to a Transactional Abstraction

For this work, we propose T-spec to abstractly describe a datapath. Figure 2 illustrates the transactional abstraction. Similar to a single-cycle design, T-spec is a textual "netlist" that comprises state elements and next-state compute operations implemented by a network of logic blocks. However, unlike a single-cycle implementation, a T-spec's 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 to be seen by the next transaction. Both single-cycle and T-spec datapaths do perform state transformations in a single step. However a T-spec datapath execution is sequenced by transactions, instead of clock cycles.

In result, T-spec decouples the datapath specification from a particular implementation. For example, a transaction may be mapped to an implementation that takes multiple clock cycles. This thesis work shows that it is possible to synthesize an in-order pipelined implementation that executes multiple overlapped transactions, yet maintains the transactional semantics of the datapath described in T-spec. In general, any implementation may be derived from a T-spec, as long as it preserves the transactional semantics.

In practice, T-spec retains the same type of information as that captured by a RTL description of a datapath, which consists of state elements and next-state logic blocks. However, T-spec adds to a typical RTL description several interface and type requirements, which are useful in capturing the state access behaviors of the transactions

that can be executed by the datapath. Such information is necessary to reason with data hazards that could happen in a pipelined implementation to be derived from the T-spec in a precise manner. Details on the data hazard analysis will be provided in Chapter 3. Below we elaborate on the information captured by a T-spec.

2.2.1. State Elements

Without loss of generality, the T-spec netlist only includes register-type state elements of arbitrary word-size, and array-type state elements of arbitrary size. Figure 2(b) gives the schematic netlist of an example design. This design comprises of a single state element R and a network of logic blocks (op1, op2, op3, op4, op5, and m1), with the register-type state element R represented by its separate read interface and write interface. In a valid T-spec, a particular register or array location can be written at most once by a transaction. The effect of a write is only observable starting with the next transaction.

An unusual feature of T-spec is that a state-read interface includes an explicit "read-enable" control signal. This read-enable is not a tristate control; rather it is purely a bookkeeping signal to help T-piper refine RAW hazard analysis by letting the designer indicate exactly when a transaction must see the valid value of a state element in order to proceed. Similarly, an explicit "write-enable" is included in a state-write interface.

2.2.2. Combinational Next-State Logic Blocks

An acyclic network of combinational logic blocks computes the next-state update for the write-interfaces of the state elements based on the current state values received from the read-interfaces of the state elements. Only the input and output ports of the

combinational logic blocks are declared in a T-spec. Except for multiplexers, T-piper treats all combinational logic blocks as black-boxes during analysis. Multiplexer is a built-in logic primitive understood by T-piper and used for hazard analysis.

2.2.3. Asynchronous Next-State Logic Blocks and State Elements

Besides combinational blocks, a T-spec netlist can also include next-state logic blocks with asynchronous interfaces. Beside data inputs and outputs, these blocks must also support an established set of handshaking signals: ready, start, and done.

The ready output signal indicates when an asynchronous block is ready to accept a new set of data inputs. A new calculation is performed by asserting the start input signal until the data output is valid, as indicated by the done output signal. Each asynchronous block can be executed at most once by each transaction. Asserting start implicitly resets any internal state so no history can be carried from one transaction to the next.

In the final synthesized clock-synchronous pipeline, an asynchronous block in T-spec is replaced by a corresponding library block that produces its output after a fixed or variable multi-cycle delay (e.g., an iterative divider).

Finally, T-spec also supports state elements with asynchronous interfaces whose read and write interfaces are not always available immediately. This is used to represent hardware structures such as a memory element that contains a cache, where access latency varies depending on whether there is a cache hit or not.

Chapter 3 provides details on how asynchronous interfaces are handled by our pipeline synthesis approach.

2.2.4. Inputs and Outputs

An external input to a T-spec datapath is associated with the read-interface of a special Input-type element. The usage of the Input read-interface is similar to that of a register-type element, except the value read from an Input element's read-interface is not associated to any prior state update operation. The value returned from reading an Input read-interface reflects directly the external environment that the input is combinationally connected to. An Input read-interface is not subjected to RAW hazard analysis during synthesis.

Similarly, an external output is associated with the write-interface of a special Output-type element that is connected to the external environment. The usage of the Output write-interface is similar a register-type element. Functioning like a register, the external output will hold the last written value until the next write. The Output write-interface is also not subjected to RAW hazard analysis.

2.3. A T-spec Example

Figure 3 depicts a T-spec excerpt for the datapath example in Figure 2(b). The T-spec begins with a GENERIC module declaration for op1, a black-box combinational block. It has a 1-bit output named R_re. (This input-less block represents a hardwired constant.)

The second module declaration is for a built-in REG-type state module named R. A REG-type state module has explicit read and write interfaces (i.e., named rd and wr in this case). The read (or write) interface comprises of a read-enable (or write-enable) port and an output read-data (or input write-data) port. The declared data-width of R is 32-bit.

Lastly, a connection declaration connects the R_re output port of op1 to the en input port of R's rd interface. The declarations for the remaining modules and connections are omitted for brevity.

Appendix A provides the complete T-spec language syntax, and Appendix C provides an example T-spec for a MIPS processor datapath.

```
// Black-box combinational module op1
MODULE op1 GENERIC {
    PORT { R_re OUT 1 }
}

// State element R, of type REG
MODULE R REG {
    IFC rd REG_RD {
        PORT { re RD_EN }
        PORT { d  RD_DATA 32 } }
    IFC wr REG_WR {
        PORT { we WR_EN }
        PORT { d WR_DATA 32 } }
}

// … op2, op3, op4, op5, m1 modules

// Connect output of op1 to the enable of R's read interface
CONN { op1.R_re → R.rd.re }

// … other connections
```

Figure 3. A T-spec example.

Chapter 3

In-order Pipeline Synthesis

This chapter presents the T-piper technology to automatically synthesize in-order pipeline from a given T-spec. The first part of the chapter discusses automatic data hazard analysis and resolution. Then, the second part of the chapter elaborates on the RTL generation procedure.

3.1. Data Hazard Analysis and Resolution

We present here the analytical approach for reasoning with data hazards and strategies to resolve them based on the information captured in a T-spec.

3.1.1. Pipeline Stage Boundaries

The desired pipeline stage boundaries are expressed in T-spec by declaring the number of stages and assigning each module (or interface in the case of a state element) to a pipeline stage. For example, Figure 4 depicts a possible set of pipeline stage boundaries (shown in solid lines) for the T-spec datapath in Figure 2(b), are 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.

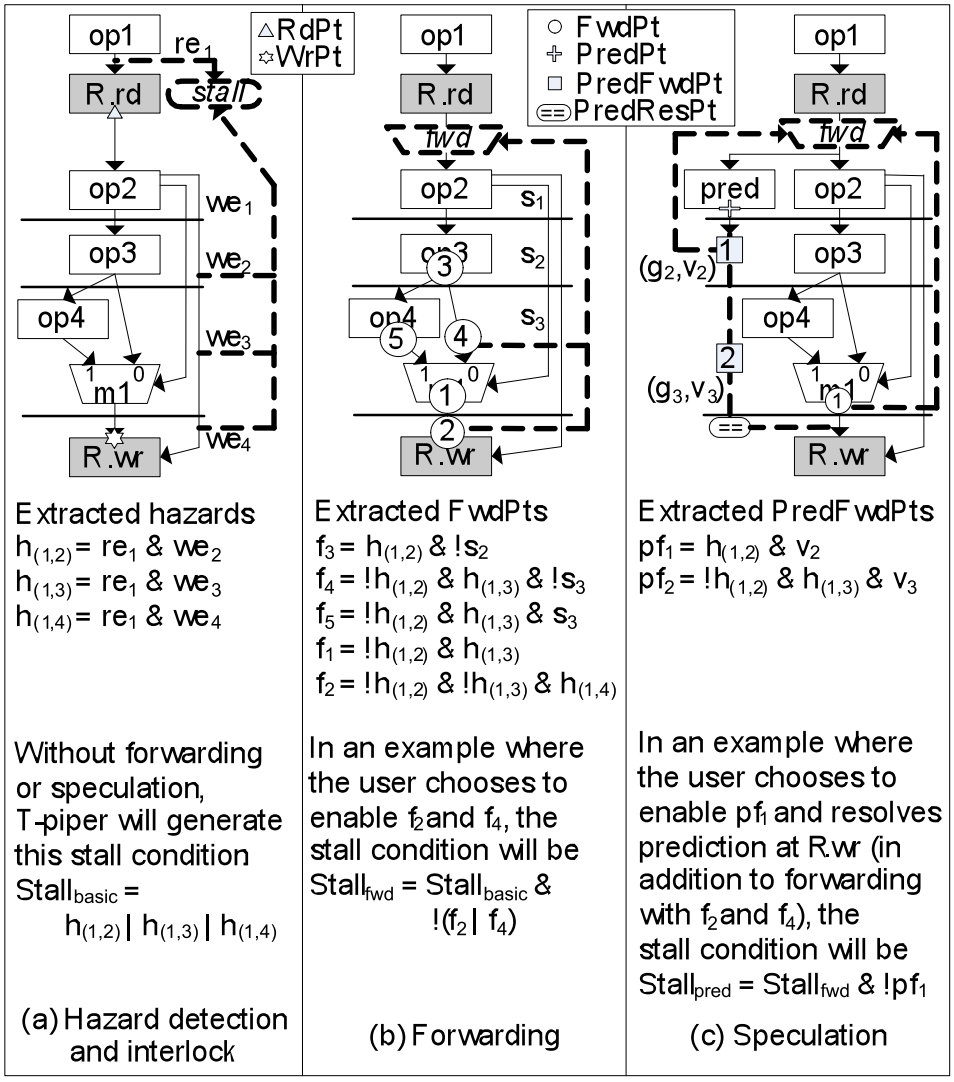


Figure 4. Data hazard analysis and resolution.

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 element also cannot be assigned to a stage earlier than the state element's read interface. If an array state element supports multiple write interfaces, they must be assigned to the same stage. These constraints on state read and write interface assignments exclude the possibility of write-after-write and write-after-read hazards.

3.1.2. 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. The T-spec datapath from Figure 2(b) is divided into four stages in Figure 4. As such, multiple versions of a signal (corresponding to different transactions in different stages) co-exist if the source and destination of a connection are not located at the same pipeline stage. For example in Figure 4(a), the we signal traverses all four stages since its source (op2) is located in the first stage, while its destination (R.wr) is located at the fourth stage. When op2 is computing we_1 for transaction T_1 in stage 1, we_2 is an older version of the same signal belonging to the older transaction T_2 in stage 2. Similarly, we_3 and we_4 belong to transactions T_3 and T_4, respectively.

For hazard analysis, T-piper identifies for each state element a "read-point" (RdPt) associated with the state element's read interface. In Figure 4(a), the read-point for the state element R is labeled with a "triangle"-symbol. A read-point is qualified by its state element's explicit read-enable; that is, a read-point is required to carry a valid state value only when its accompanying read-enable is asserted. Similarly, a "write-point" (WrPt) is associated with the write-data interface of the state element, and is qualified by the write-enable. In Figure 4(a), the write-point for the state element R is labeled with a "star"-symbol. A write-point carries the new state update value only if its write-enable is asserted.

In general, with respect to any state element E in a datapath, a hazard condition exists whenever there are two "in-flight" transactions in the pipeline, and the younger transaction is reading from E while the older transaction is planning to write to E. In

other words, hazard occurs when there is a younger transaction T_x and an older transaction T_{x+i} that occupy stage x where E's read-point resides and a later stage x+i between the read-point and the write-point of E, respectively. Furthermore, T_x asserts the read enable of E, and T_{x+i} asserts its write enable (i.e., re_x & we_{x+i} is true).

When hazard occurs, T_x will receive an incorrect state value if it reads from E directly because the update value 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. The expression for $Stall_{basic}$ in Figure 4(a) is constructed accordingly to indicate when the reading of the state element R in our example must be stalled.

When stalling the transaction T_x, the older transactions downstream in the pipeline must be allowed to proceed so T_{x+i} will eventually progress past the write-point of E, removing the hazard condition. It is possible for we_{x+i} to not exist if we is computed by a module assigned to a later stage. For the purpose of hazard analysis, we_{x+i} must be assumed true whenever the stage x+i is occupied by a transaction. For example, in Figure 4(a), if we were computed instead by op4 in stage 3, then we_1 and we_2 do not exist, and we must conservatively assume that the transactions in stages 1 and 2 will assert their we.

If the state element E is an array, T-piper carries out hazard analysis at the granularity of individual locations. Given rd-idx and we-idx are the indices to the read and write interfaces of E, a hazard condition arises only if (re_x & we_{x+i}) and ($rd\text{-}idx_x$==$wr\text{-}idx_{x+i}$) are true. 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.

The aforesaid hazard analysis is repeated independently for each state element in the datapath. The stalling logic generated by the hazard analysis procedure is to be used in conjunction with an implementation-specific method that tracks the existence of valid transactions in pipeline stages. For this work, when synthesizing an implementation from a T-spec, a valid bit is added to each stage for this purpose. The bit is set and cleared accordingly as transactions enter and leave the pipeline.

3.1.3. Forwarding

Based on the simple analysis in the previous section, T-piper can already emit a correct pipelined implementation. The pipeline's effectiveness depends on how often the stall conditions are triggered at runtime.

In some cases, a stall 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.

In some cases, a stall 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.

```
1  // Inputs:
2  //    WrPt – write point of the state subjected to forwarding
3  //    Node – current node being analyzed
4  //    Parent – the parent of Node
5  //    MuxSelChain – conjunction of multiplexer select conditions
6  // Output: a database containing a set of forwarding points
7
8  extractFwdPt (WrPt, Node, Parent, MuxSelChain) {
9     // Create a forwarding point, if appropriate
10    for each stage s from Node.stage to Parent.stage {
11       if(  resolvable_we(WrPt, s) &&
12            resolvable_muxsel(MuxSelChain, s) )
13          addFwdPtDB(WrPt, Node, s, MuxSelChain);
14    }
15    // Recursive call, if necessary
16    if(Node is a MUX) {
17       for each mi input port of Node
18          extractFwdPt( WrPt, getSrc(mi), Node,
19                              updateMuxSelChain(MuxSelChain, mi) );
20    }
21 }
```

Figure 5. Forwarding-points extraction algorithm.

To ensure 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. (According to the transactional semantics, when multiple older transactions have outstanding writes to a state, the current reader of that state depends on the youngest of those transactions.) Forwarding-points can be traced backwards through a multiplexer to create conditional forwarding further qualified by the mux-select logic.

Figure 5 provides the pseudo-code for the forwarding-points extraction algorithm. We treat the network of operations in T-spec as a Directed Acyclic Graph (DAG) with a state write-point at the root of the graph. (The direction of the edges in the DAG is opposite of the dataflow.) The algorithm performs a depth first traversal from the root of the DAG, visiting each node where forwarding may happen.

The algorithm consists of two parts. The first part (i.e., the first for-loop) analyzes the pipeline stages between the node under analysis and its parent, and creates forwarding-points accordingly. The analysis involves checking whether the predicate of a potential forwarding-point is resolvable or not. In line 11, resolvable_we(WrPt, s) checks if the write enable signal for WrPt is available in all stages between s and the state's read-point. Dynamically, a forwarding is only valid if the write enable in stages earlier than s are all de-asserted (i.e., no writes by any transaction younger than the one in stage s). In line 12, resolvable_muxsel(MuxSelChain, s) checks if all of the mux-select conditions in MuxSelChain have been computed by stage s. If all the required predicates for forwarding are resolved, then the forwarding-point is inserted to a database (by addFwdPtDB() in line 13). The second part of the algorithm deals with the case when the node is a multiplexer. Because a multiplexer only passes data value, we can recursively

analyze each of the input paths for additional forwarding-points. However, each forwarding-point on the input path has to be further qualified by the mux-select signal. In line 18, a recursive call is made for each of the multiplexer input paths. For each input path, updateMuxSelChain() in line 19 adds the required select signal for the current multiplexer to the conjunction of the select conditions for the previously visited multiplexers between the current node and the root.

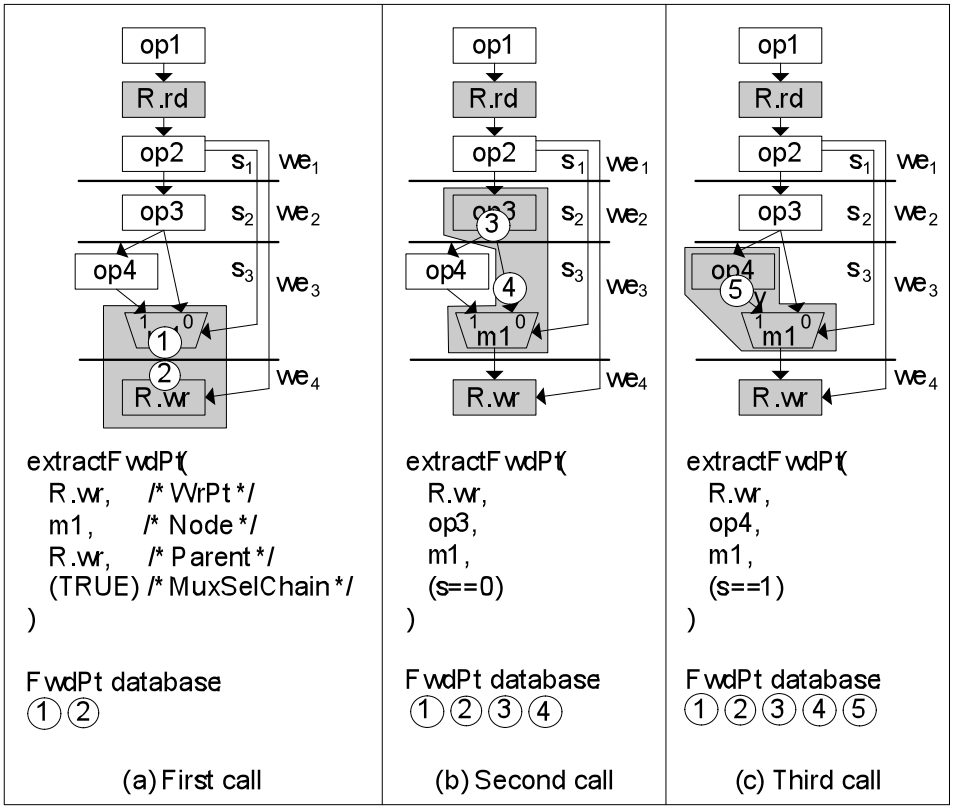


Figure 6. Forwarding-points extraction example.

Figure 6 illustrates the application of the algorithm to the datapath example in Figure 4(a). Figure 6(a) depicts the first call to extractFwdPt(). Since this is the first call, Node is the input source to the root of the DAG (R.wr), which is m1. Parent is R.wr itself. Since no multiplexer is encountered yet, MuxSelChain is initially TRUE. The shaded area

indicates the part of the DAG under analysis (i.e., between R.wr and m1). The first part of the algorithm analyzes the stages between m1 and its parent, R.wr. Since all the write enable signals (we_1, we_2, we_3, and we_4) between R.rd and R.wr are available, resolvable_we() returns TRUE for stages 3 and 4. Two forwarding-points (i.e., the "circle"-symbols labeled with 1 and 2) are added to the forwarding-points database. Next, since Node is a multiplexer, the second part of the algorithm invokes the recursive calls to each of the two inputs of m1.

Figure 6(b) shows the second (recursive) call to extractFwdPt() on input 0 of m1. The shaded area indicates the part of the DAG being analyzed. Node is now op3, which is the source of input 0 of m1 (obtained by getSrc() in the first call). Parent of op3 is m1. Since the algorithm traversed through input 0 of multiplexer m1 to get to this call, the condition (s==0) is added to MuxSelChain. The first part of the algorithm analyzes stages 2 and 3 (i.e., where Node op3 and Parent m1 belongs to, respectively). resolvable_we() and resolvable_muxsel() return TRUE for stages 2 and 3, since the required predicates are available (i.e., {s_2, we_2, w_1} and {s_3, we_3, we_2, we_1}, respectively). Two forwarding-points (3 and 4) are inserted to the forwarding-points database. Similarly, Figure 6(c) depicts the third call to extractFwdPt(), or the second recursive call at m1 for input 1. For the datapath in Figure 6 example, the depth-first traversal ends at the children of m1. However, further calls could have happened if there was a chain of multiplexers in the datapath (e.g., if an input to m1 was a multiplexer).

The five forwarding-points of R in Figure 4(b) are labeled by numbered "circle"-symbols. The numbers correspond to the traversal order by the algorithm in Figure 5. For each forwarding-point, Figure 4(b) gives the exact condition when the value at a

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_1 depends on T_4 with respect to R but not on T_3 or T_2. Likewise, forwarding-point 1 is valid iff T_1 depends on T_3 with respect to R but not on T_2. Points 5 and 4 are conditional forwarding-points corresponding to the two possible settings of m1's the mux-select (s_i). The condition for using forwarding-point 5 (or 4) is the same as 1 with the additional requirement that the multiplexer m1 in question is set to select the 1-path (or the 0-path). Forwarding-point 3 is an earlier version of forwarding-point 4, allowing forwarding from T_2 to T_1 when s_2 is not asserted.

After the analysis, T-piper reports to the user all forwarding-points. Based on the user's selection of which to include, T-piper generates a pipelined implementation with the selected forwarding paths. When a forwarding path is added, its exact trigger condition is subtracted from the stall condition. When the trigger condition is satisfied, the would-be RAW hazard is resolved by forwarding from the corresponding forwarding-point. The example in Figure 4(b) adds forwarding from forwarding-points 2 and 4, resulting in a new stall condition $Stall_{fwd}$ that subtracts f_2 and f_4 from $Stall_{basic}$.

It is important to note an injudicious selection of forwarding-points does not always help performance and could even hurt performance by creating unwanted long critical paths. For example, adding forwarding-point 1 in Figure 4(b) would create a critical path spanning two-stages worth of combinational logic (op4 and m1 in stage 3, and op2 in stage 1). Also, a forwarding path does not improve performance unless it is triggered frequently during execution.

3.1.4. Speculative execution

Forwarding can only be done if an older transaction already computes (but has not yet written) the value that a younger transaction depends on. Consider the example in Figure 4(b). If a younger transaction in stage 1 depends on the value of R to be produced by an older transaction currently in stage 2 via op4 eventually (i.e., mux select is 1), then the younger transaction 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 either forwarding-point 1 or forwarding-point 5.

In cases where forwarding cannot sufficiently reduce the RAW hazard stalls, T-spec supports a general-purpose framework for a designer to introduce a value predictor to resolve RAW hazards speculatively. Starting with a T-spec datapath with complete functionality, a designer can introduce auxiliary 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 and with less logic effort, 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. This valid signal should only be asserted when the confidence in a guess is high; otherwise stalling is preferred over speculation to avoid the misprediction recovery penalty.

The auxiliary states and logic blocks for making value predictions are specified using the same T-spec syntax and constructs 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 T-spec of the primary datapath should stay exactly the

same whether or not value prediction is added. In Figure 4(c), the auxiliary logic module pred is making a guess for the next-state value of register R in stage 1 whereas the true next-state value of R is not fully resolved until stage 3 (at the m1 output). In this example, a guess is generated combinationally based on the primary state elements only. In general, one could introduce and maintain new auxiliary state elements and logic blocks to describe arbitrarily elaborate history-based value predictors for any of the state updates.

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 Figure 4(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. Prediction forwarding can be done in the stages starting from the prediction-point to the corresponding write-point (or until forwarding-points are available). In Figure 4(c), the possible prediction-forwarding-points are labeled by the numbered "box"-symbols. The figure also shows an implementation that makes use of prediction-forwarding-point 1, resulting in a new stall condition $\mathsf{Stall}_{\mathsf{pred}}$.

If value prediction is used in a design, 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 will happen at the write-point of E (as shown in stage 4 of the example in Figure 4(c)), incurring minimum resolution logic, but maximum penalty for incorrect prediction. The user can also instruct

T-piper to check prediction in advance of the write-point by comparing against user-selected forwarding-points to reduce the misprediction penalty. If the prediction resolution is done behind a multiplexer, then T-piper makes sure that the resolution includes the set of forwarding-points that cover all possible paths to the multiplexer (e.g., forwarding-points 4 and 5).

During a prediction check, if the predicted value and the actual value agree, nothing more needs to be done. However, if they disagree, all younger transactions in flight following the mispredicting transaction must be squashed from the pipeline. The mispredicting transaction is allowed to complete fully since the transaction itself never made use of the prediction. The execution continues by restarting the next transaction using the now available 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 assignment constraint ensures that flushing the transient contents of just the pipeline registers is sufficient to recover to the restart state, without needing to undo any state changes to the system state elements.

The support for value prediction and speculative execution is particularly important to instruction processor pipelines. In the RISC pipeline we discuss in Section V, one can introduce a straightforward, combinational prediction of PC+4 as the next-state value of the program counter (PC) register; a deeper CISC pipeline requires more elaborate history-based prediction. Without PC prediction, it would be impossible to fetch a new instruction each cycle in either the RISC or the CISC pipeline.

3.1.5. Hazard Resolution Configuration (H-cfg)

H-cfg contains the specification of which value forwarding, prediction forwarding, and prediction resolution points should be implemented from all of the extracted design points. Currently, we provide MAX (enable all), ASAP (enable only earliest points), and ALAP (enable only latest points, which is used for prediction resolution) as default heuristics to automatically generate such a configuration. Alternatively, the designer can also provide a manually written H-cfg.

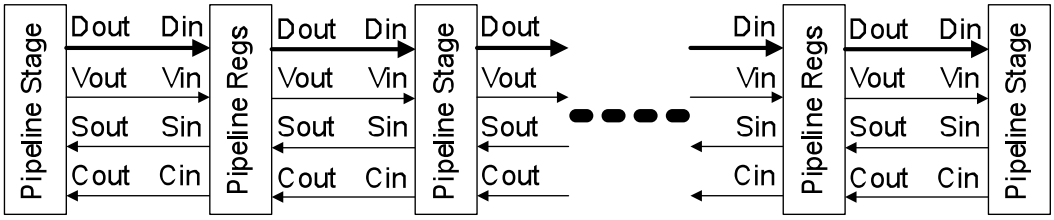


Figure 7. Pipeline model.

3.2. RTL Generation

We support synthesis of in-order pipelines that follows the model depicted in Figure 7. In this model, a pipeline consists of pipeline stages and register sets, connected via a communication channel based on Valid, Stop, and Cancel signals. The use of Valid and Stop signals is inspired by the protocols described in [9]. The Cancel signal is added to support speculative execution in our pipelines.

Note that the choice of pipeline model targeted by our synthesis process is orthogonal to the data hazard analysis. We chose the model based on Valid and Stop bits in our implementation for its simplicity.

3.2.1. Pipeline Flow Control Protocol

The communication channel among the pipeline stages and register sets works as follow. A Valid signal indicates data validity, and a Stop signal indicates whether a recipient (could be a pipeline stage or register set) has accepted the data. Thus if a sender asserts its Valid signal, and a recipient deasserts its Stop signal, a *Transfer* would occur. If the sender deasserts its Valid signal, then there is no data to be sent regardless of what the Stop signal condition is, in which case the communication channel would be *Idle*. If there is a valid data value to be sent, but the recipient is not yet ready to accept it (i.e. Stop is deasserted), then the communication channel status would be *Retry*, and the sender will persistently assert its Valid signal until a *Transfer* occur.

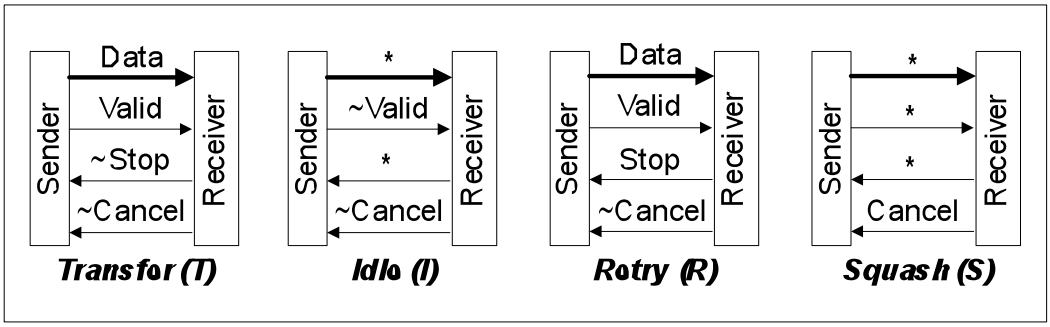


Figure 8. Pipeline communication modes.

Furthermore, we use the Cancel signal to qualify a data transfer. Whenever Cancel is asserted, the sender will nullify its operation, and therefore invalidates the data that it is trying to send. An asserted Cancel signal causes a *Squash* in the data transfer, regardless of the condition of the Valid and Stop signals. If Cancel is de-asserted, then the protocol operates as usual. Figure 8 shows the aforementioned communication modes.

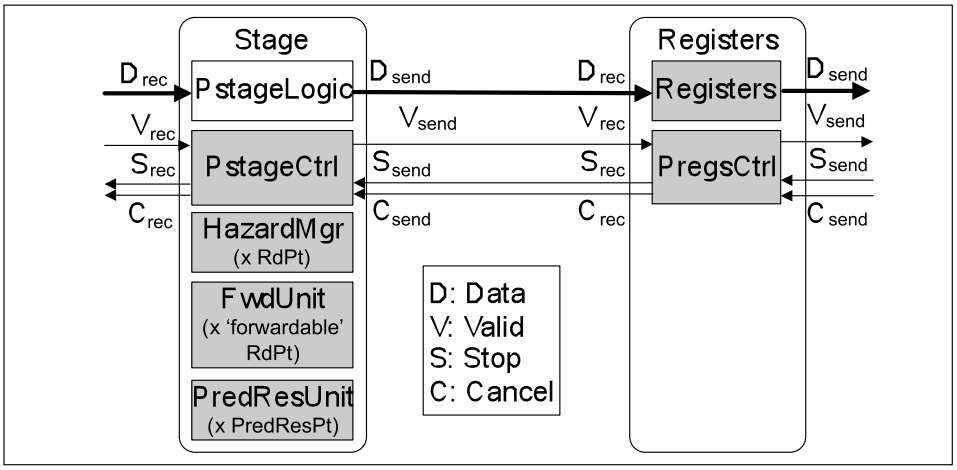


Figure 9. Pipeline internals.

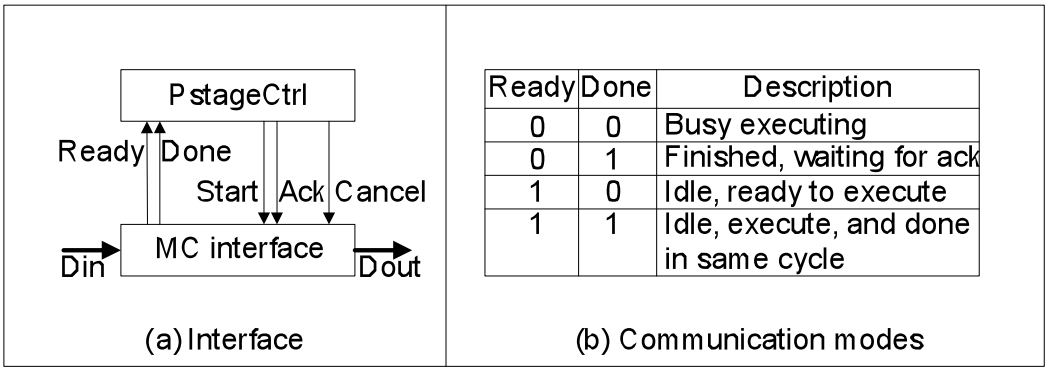
3.2.2. Pipeline Management Logic

The contents of a pipeline stage and a register set are shown in Figure 9. The pipeline stage logic (PstageLogic) contains the datapath modules and state access interfaces instantiated in T-spec, and are obtained from the component database during T-piper synthesis. The shaded components in the figure are pipeline management logic generated automatically by T-piper. The pipeline stage controller (PstageCtrl) is responsible for (1) monitoring and generating the hand-shaking signals for communicating with the neighboring pipeline register sets, and (2) interacting with other components (PstageLogic, etc) in the stage. There is one PstageCtrl per stage. The synthesis of this unit is straightforward. Stop output is synthesized by analyzing stall conditions due to hazards (from HazardMgr) and multi-cycle (MC) interfaces (discussed later in this section). Stop is asserted when the stage stalls. Valid is synthesized by analyzing the interfaces in the stage, and is asserted when the stage has completed execution. Cancel is synthesized by analyzing PredResPts, and it is asserted when later stages encounter any mispeculation (i.e., triggered by a PredResUnit).

The data hazard manager (HazardMgr) detects the existence of data hazards, and activates the appropriate hazard resolution logic. Since we detect hazards at the system state read-interfaces, one HazardMgr is generated for each read interface in the stage. It is synthesized by analyzing the RdPt, WrPt, and the enabled FwdPt and PredFwdPt.

The forward unit (FwdUnit) manages forwarding of both actual and predicted values. It acts as a proxy to a read interface, returning either the actual state value or a forwarded value. One forward unit is generated for each "forwardable" read interface (i.e., a read-interface with one or more enabled FwdPt or PredFwdPt). Synthesis of a FwdUnit is done by analyzing RdPts and FwdPts/PredFwdPt.

The prediction resolution unit (PredResUnit) contains the logic that compares a predicted value with the actual value, and triggers misspeculation when a mismatch occurs. One PredResUnit is generated for each enabled PredResPt in the stage. Synthesis of this unit is done by analyzing PredResPts.



Ready	Done	Description
0	0	Busy executing
0	1	Finished, waiting for ack
1	0	Idle, ready to execute
1	1	Idle, execute, and done in same cycle

Figure 10. Multi-cycle interface and operation modes.

3.2.3. Multi-Cycle Modules

Asynchronous next-state logic blocks (Chapter 2.2.3) are mapped to multi-cycle (MC) modules with the interface shown in Figure 10(a). The interface contains start, done, ready, ack, and cancel signals. The various communication modes using these signals are shown in the table in Figure 10(b). When the module first starts up, the ready output signal is asserted, which indicates the MC module is idle and ready for execution. Asserting the start signal can then activate the interface. Once the interface is activated, the MC operation would be performed. The assertion of the done signal would then indicate the completion of the operation. Next, the pipeline control (PstageCtrl) can assert the ack signal when the result of the operation has been consumed. From there, the ready signal of the MC module becomes asserted again.

This type of MC interface can also be applied to state read- and write-interfaces to allow the interface to respond with varying delay. This feature is useful, for example, to interface with the cache subsystem of a processor that responds in different number of cycles depending on whether there is a cache hit. Note that our T-piper prototype currently does not support chaining of MC modules in the same pipeline stage, though it should be possible to include such a support in the future.

Chapter 4

Case Studies

This chapter describes the prototype implementation of T-piper in-order pipeline synthesis tool, and the design case studies using the prototype on the development of MIPS and x86 processor pipelines.

4.1. T-piper Prototype

We have developed a prototype for the T-piper in-order pipeline synthesis tool, which supports all the synthesis features described in the previous chapter (i.e., data forwarding, speculation, and multi-cycle units).

Furthermore, we have also made a version of T-piper freely available online for research and academic uses at www.t-piper.net. Figure 11 shows the online user interface of T-piper, which accepts a T-spec file, a P-cfg file, and a forwarding scheme. By clicking on the "synthesize" button, T-piper will synthesize the target pipeline implementation in Verilog. Note that the online version of T-piper does not support speculation yet. At the website, we have also included a tutorial and design examples for new users to quickly get started with T-piper.

Figure 11. Online version of T-piper at www.t-piper.net.

4.2. MIPS Case Studies

4.2.1. Comparison Against a Hand-made MIPS Processor Pipeline

In this study, we trained an undergraduate student to develop a 5-stage MIPS pipeline using T-spec and T-piper. The student has prior experience in developing a textbook 5-stage MIPS pipeline [20] by hand. Including training, the T-spec was completed in under a week. The synthesized pipeline utilized the same datapath components as the hand-made pipeline the student had developed previously. Both pipelines support the same set of user-level MIPS instructions (e.g., ALU, memory, and branches). Thus, the difference between the two pipelines is only in the pipeline control logic, one of which is synthesized by T-piper and the other manually developed by the student. We found that the synthesized pipeline is within 2% in performance and area of the student's hand-made design. Furthermore, the student was also able to synthesize 3, 4, and 6-stage pipelines from the same T-spec by slightly modifying the pipeline stage configuration file (P-cfg).

4.2.2. Comparison Against Existing MIPS Processor Pipelines

We also compared the processor pipeline used in the previous section against the open-sourced design of the SPREE MIPS-I processor that is also a 5-stage pipeline with a full operand-forwarding network [15]. We found that our MIPS implementation could reach an 8% higher clock frequency than the SPREE implementation. Our MIPS implementation is however 4% larger in area then the SPREE implementation at SPREE's peak frequency. This comparison is inexact but should be sufficient to establish that our MIPS processor pipeline is a reasonable quality. Moreover, in [15], the quality of the SPREE MIPS-I implementation was successfully vetted against the commercial Altera NIOS RISC processor pipeline.

4.2.3. Developing a Complete MIPS I Processor

In the next study, a graduate student who is already familiar with T-spec and T-piper designed a complete user-level MIPS I processor from scratch. Within 5 days, he was able to synthesize MIPS pipelines that supported all of the user-level MIPS I instructions except for the platform-dependent co-processor instructions. In addition to basic RISC instructions, the MIPS I instructions include many variations of memory instructions (e.g., partial and unaligned load and stores), control instructions (e.g., branch and link), and integer multiply/divide (implemented using multi-cycle iterative divider/multiplier functional units) along with HI/LO register move instructions.

The breakdown of the development time is as follows:

1) One day was spent for reading the MIPS ISA manual and designing the non-pipelined datapath (e.g., the types of components, and how they are connected).

2) One day was spent to describe the datapath in T-spec, and to develop the target pipeline configurations (i.e., P-cfg files). This one day also includes the time spent ironing out bugs to make sure that the T-spec and the P-cfg files are well-formed.

3) Three days were spent for the rest of the development activities, which consists of implementing the datapath components in Verilog, creating the testbench, writing MIPS assembly test programs for validating correct functionalities of supported instructions, and debugging.

Notice by using T-piper to relieve the designer from the manual pipelining effort, the development time becomes dominated by supporting tasks not directly related to pipeline implementation (e.g., reading manuals, making test cases, etc).

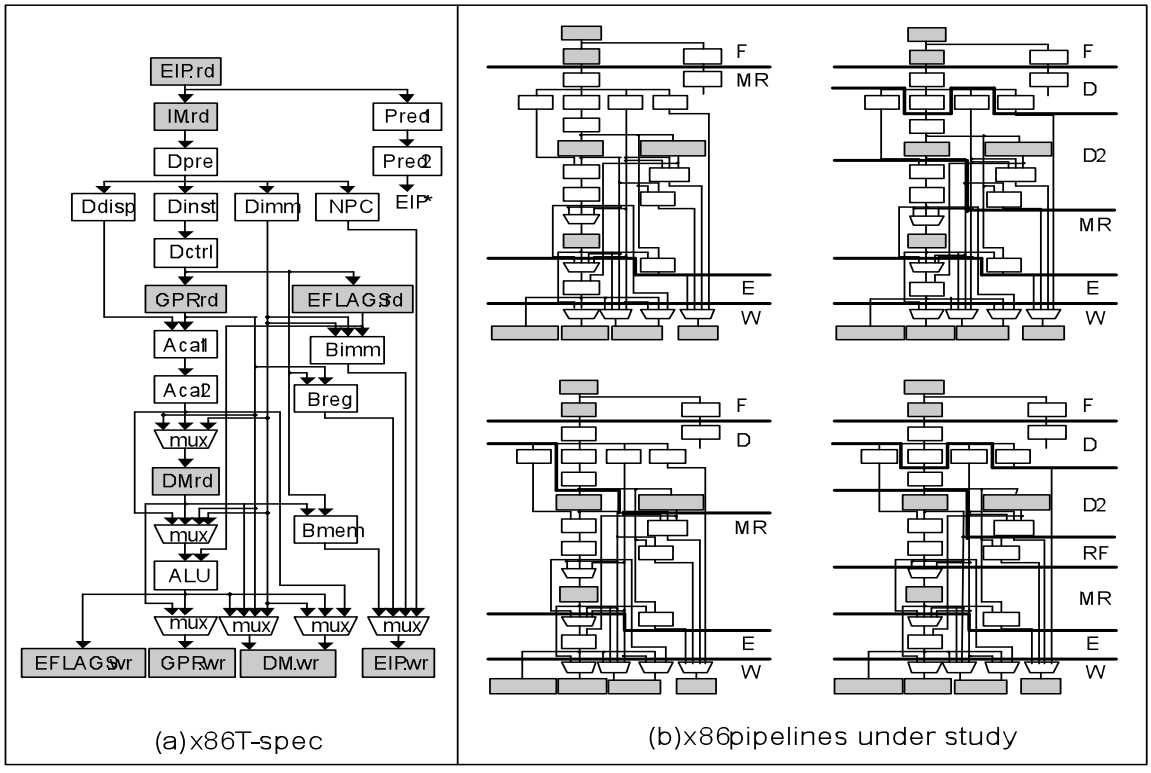


Figure 12. x86 T-spec and pipelines under study.

4.3. x86 Case Study 1: Rapid Design Space Exploration

In this case study, we created the T-spec for an x86-based processor, and used T-piper to rapidly explore 60 different synthesized pipelines. We evaluated the impact of the various pipeline features and characterized their performance-area tradeoffs. To the best of our knowledge, our automatic pipeline synthesis approach is the first to be demonstrated with an ISA as complicated as the x86.

4.3.1. x86 Datapath

Figure 12 show the T-spec of the x86 processor and the pipelines under study, respectively. For brevity, we omitted several less important details of the datapath from the figures. For example, only one out of multiple GPR read interfaces is shown. The T-spec design supports all protected-mode general-purpose instructions that do not modify privilege states, except for ASCI/decimal adjustments (e.g., AAA, DAA), bit operations (e.g., bit scan, bit test), divide, swap (e.g., BSWAP, CMPXCHG), string, and I/O (e.g., IN, OUT) instructions. We also did not include interrupt and exception handling. The supported subset of x86 ISA is sufficient to execute the SPREE benchmarks [71] used in our study. The datapath includes a multi-cycle integer multiplier unit and variable-cycle memory interfaces capable of interfacing with memory systems with caches. For the evaluations in this study, however, we assume a memory system with a perfect cache (no misses) so we can focus on the effect of various pipeline features on performance.

4.3.2. Design Parameters Explored

We explored several manually chosen pipeline parameters in this study. While reasonably complete, these parameters do not span the complete design space. It is

possible in future work to develop orthogonally an automatic design space exploration system to replace the manual effort in selecting the pipeline parameters. Below we describe the major dimensions of the pipeline configuration parameters we considered.

Pipeline depth. For a baseline, we started with the shortest possible pipeline (i.e., 4-stage, where each multi-cycle interface is assigned to its own pipeline stage) without any forwarding or prediction. We then manually analyzed the critical path and added new pipeline stages to break the critical path and to improve frequency. We continued adding more pipeline stages, up to the point when adding an additional pipeline stage resulted in only a negligible improvement in frequency. We ended up with a collection of pipelines between 4 and 7 stages.

Note that there are many possible pipeline boundaries for a given number of pipeline stages. For example, for a 5-stage pipeline, one could put all the decode units in stage 2, or spread it across stage 2 and 3. If the decode unit that decides whether an instruction is a branch or not is placed in stage 2, forwarding of the next program counter value (i.e., to stage 1, where the program counter is read) can be done as early as stage 2. However, if the unit is placed on stage 3, then forwarding can be done only from stage 3 at the earliest. On the other hand, putting all the decode units in the same stage could adversely impact the critical path. We consider such intricacies when selecting the pipeline boundaries. In overall, we picked the pipeline boundaries that resulted in the best frequency improvement

Data forwarding scheme. T-piper allows the user to select any subset of forward paths from the forwarding opportunities reported. Thus, there are many possible

combinations of data forwarding configurations. To limit the scope of this study, we chose the set of forward paths based on two simple schemes. The first scheme forwards data only from the earliest possible forward points (ASAP), while the second performs forwarding whenever possible (MAX).

Speculation scheme. Although our system can generally handle value prediction on any system state, in this study we only attempt to predict the program counter (PC), which is the most common predictor for processor pipelines. In the baseline case, we added a next-PC (NPC) predictor that assumes branches are not taken and guesses the execution always proceeds to the instruction immediately following the current one. (In other words, the NPC predictor is effectively predicting the instruction length of the current instruction being fetched.) In the second case, we incorporated a bimodal predictor that uses 2-bit saturation counters and a branch target buffer (BTB) to guess the direction and target address of branch instructions. To determine an appropriate size for these predictors, we conducted trace-based simulations, and picked the size at the saturation point of the prediction accuracy curve. We ended up with 256 entries for the NPC predictor's memoization table, and 16 entries for the bimodal predictor's BTB.

We also needed to choose the schemes to forward predicted values. Since the PC is read and written by every transaction, a transaction NPC guess is needed by only the transaction immediately after it and no one else. Therefore, the MAX forwarding scheme does not add any additional benefit over the ASAP scheme. Thus, we evaluated only the ASAP scheme for forwarding scheme for NPC prediction.

We evaluated two different prediction resolution schemes. The first scheme resolves prediction at the latest possible point (ALAP), which minimizes the amount of resolution logic to compare the predicted value against the actual value. The second scheme resolves prediction at the earliest points (ASAP), which reduces misprediction penalty at the cost of increased resolution logic since there can be multiple places where the predicted value is compared against the actual value).

Table 1. Benchmark applications under study.

Name	Description
bitcnt	counts the number of bits in an array of integers
bubble_sort	performs the bubble sorting algorithm
crc	cyclic redundancy check
des	performs the 16 rounds in data encryption standard
fft	fixed point fast fourier transform
fir	finite impulse response filter
iquant	Inverse quantization algorithm for MPEG and JPEG encoding
quant	quantization algorithm used in JPEG compression
vlc	variable Length coding for JPEC and MPEG compression

4.3.3. Evaluation Framework

This study used 9 benchmark applications from the SPREE collections [71], which consists of benchmarks from Mibench [17], XiRisc [5], and RATES [59] that have been stripped of system and I/O instructions. Table 1 offers a description of each benchmark.

The evaluation framework is shown in Figure 13. First, we use Simics full-system simulator [38] to simulate an x86 system and generate reference traces containing final architectural states for each instruction executed by the benchmarks. The trace also contains initial architectural and memory states used to initialize the pipeline RTL model. Next, we use Verilator [61] to convert the RTL Verilog description of the pipeline to

C++. Then, we integrate this RTL C++ model with a C++ trace processing application that executes the RTL model and performs validation by comparing the traces from the model with the reference traces. For performance analysis we also collect the number of clock cycles and instructions executed by each benchmark. To obtain implementation costs, we synthesized the pipelines with Synopsys Design Compiler [63] targeting a commercial 180 nm standard cell library and memory compiler.

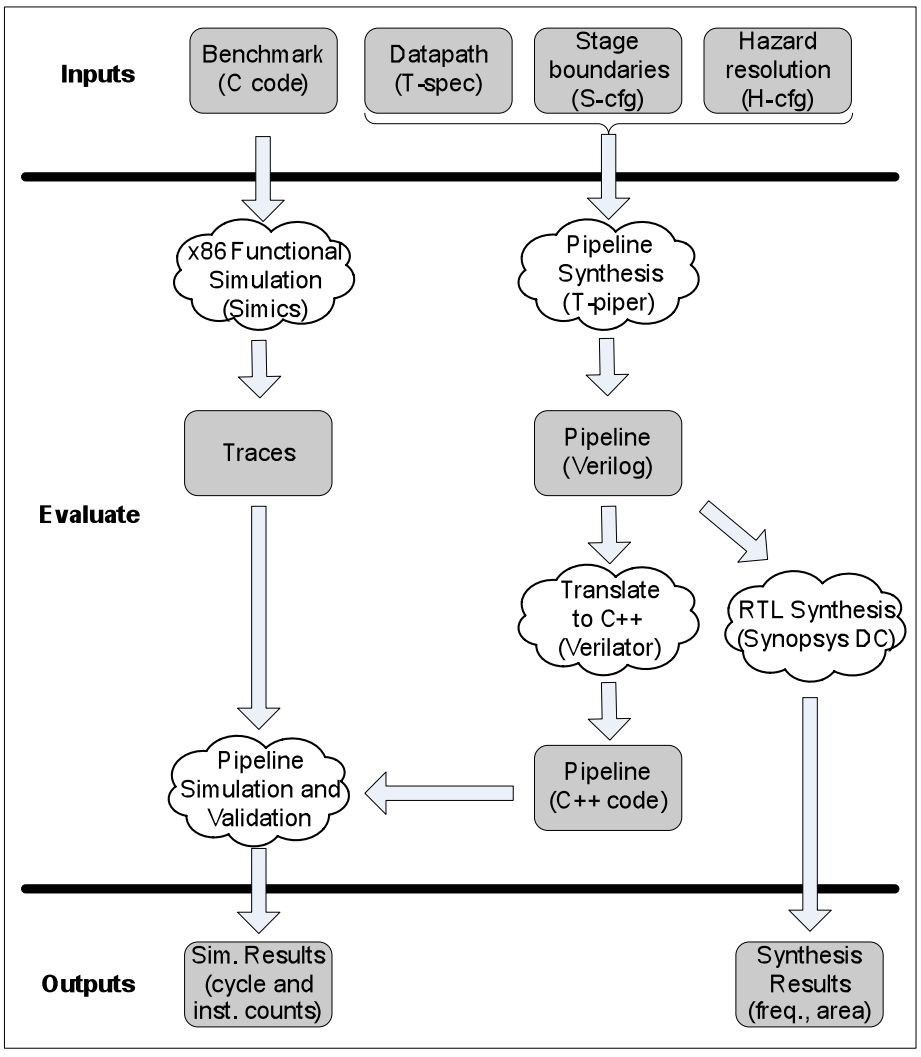


Figure 13. Evaluation framework.

Table 2. Cycle counts for each of the x86 processor pipelines evaluated.

Pipeline configuration				Pipeline stages			
Row	**P-type**	**P-res**	**D-fwd**	**4**	**5**	**6**	**7**
1	None	None	None	0.0	-23.7	-47.5	-71.2
2			ASAP	50.0	38.9	30.0	19.3
3			MAX	62.4	52.0	38.8	29.9
4	NPC	ALAP	None	29.2	15.5	5.1	-8.6
5		ASAP		29.2	15.5	5.1	-8.6
6		ALAP	ASAP	50.0	38.9	32.2	21.6
7		ASAP		50.0	38.9	32.2	21.6
8		ALAP	MAX	62.4	52.0	43.6	33.6
9		ASAP		62.4	52.0	43.6	33.6
10	Bimodal	ALAP	None	32.4	20.1	10.0	-2.6
11		ASAP		32.6	20.5	10.5	-2.3
12		ALAP	ASAP	50.0	38.9	31.9	21.3
13		ASAP		50.0	38.9	32.0	21.4
14		ALAP	MAX	62.4	52.0	43.1	33.3
15		ASAP		62.4	52.0	43.8	33.9

Table 3. Frequency for each of the x86 processor pipelines evaluated.

Pipeline configuration				Pipeline stages			
Row	**P-type**	**P-res**	**D-fwd**	**4**	**5**	**6**	**7**
1	None	None	None	0.0	28.8	47.7	59.5
2			ASAP	-9.0	-13.5	16.2	26.1
3			MAX	-15.3	-16.2	17.1	19.8
4	NPC	ALAP	None	-3.6	31.5	47.7	55.0
5		ASAP		-4.5	14.4	44.1	25.2
6		ALAP	ASAP	-9.9	-11.7	24.3	23.4
7		ASAP		-13.5	-14.4	17.1	20.7
8		ALAP	MAX	-17.1	-18.0	17.1	34.2
9		ASAP		-17.1	-17.1	14.4	16.2
10	Bimodal	ALAP	None	-2.7	31.5	55.9	53.2
11		ASAP		-4.5	17.1	51.4	47.7
12		ALAP	ASAP	-11.7	-10.8	19.8	21.6
13		ASAP		-11.7	-12.6	24.3	20.7
14		ALAP	MAX	-17.1	-19.8	14.4	16.2
15		ASAP		-17.1	-16.2	16.2	20.7

Table 4. MIPS performance for each of the x86 processor pipelines evaluated.

Pipeline configuration				Pipeline stages			
Row	**P-type**	**P-res**	**D-fwd**	**4**	**5**	**6**	**7**
1	None	None	None	0.0	5.4	2.3	-4.2
2			ASAP	75.2	39.0	64.6	56.6
3			MAX	115.9	69.8	87.2	69.3
4	NPC	ALAP	None	36.4	56.5	58.8	46.1
5		ASAP		35.1	36.1	54.9	18.0
6		ALAP	ASAP	73.4	41.9	80.7	57.3
7		ASAP		66.5	37.6	70.2	53.9
8		ALAP	MAX	111.3	66.1	104.4	101.3
9		ASAP		111.3	67.9	99.6	74.3
10	Bimodal	ALAP	None	44.2	65.8	77.4	53.1
11		ASAP		42.1	48.5	73.4	48.4
12		ALAP	ASAP	70.0	43.4	73.0	53.6
13		ASAP		70.0	40.5	80.2	53.2
14		ALAP	MAX	111.3	62.5	98.0	72.3
15		ASAP		111.3	69.8	104.1	81.8

Table 5. Area for each of the x86 processor pipelines evaluated.

Pipeline configuration				Pipeline stages			
Row	**P-type**	**P-res**	**D-fwd**	**4**	**5**	**6**	**7**
1	None	None	None	0.0	-9.3	-10.7	-14.8
2			ASAP	-18.6	-20.3	-20.4	-24.0
3			MAX	-23.9	-26.0	-30.1	-32.1
4	NPC	ALAP	None	-22.3	-32.6	-34.7	-32.7
5		ASAP		-23.3	-30.0	-36.8	-36.0
6		ALAP	ASAP	-40.2	-44.7	-35.9	-47.8
7		ASAP		-39.0	-43.4	-44.5	-48.5
8		ALAP	MAX	-45.5	-48.3	-53.5	-48.3
9		ASAP		-46.7	-50.2	-52.6	-56.0
10	Bimodal	ALAP	None	-34.6	-44.6	-44.8	-50.1
11		ASAP		-35.0	-43.3	-40.1	-51.7
12		ALAP	ASAP	-51.4	-56.0	-56.5	-59.9
13		ASAP		-50.8	-56.4	-54.5	-60.8
14		ALAP	MAX	-57.3	-60.7	-63.0	-67.2
15		ASAP		-61.0	-65.6	-58.8	-68.1

4.3.4. Results

The result of the design space exploration is summarized in Table 2, 3, 4, and 5. The data points in these tables show the relative percentage improvement of each pipeline variant over the baseline pipeline (i.e., a 4-stage design without any data forwarding or speculation). Note that the cycle count, frequency, performance, and area of the baseline 4-stage pipeline are 29,086 cycles, 111 MHz, 24.1 MIPS, and 3.09 mm2, respectively.

For all the tables, the first major column shows all the pipeline configurations we studied (excluding the different pipeline depths), while the second major column show the results categorized into minor columns for the different pipeline depths. Table 2 and 3 show the execution clock cycle count from RTL simulation and implementation frequency from synthesis, respectively. Table 4 shows the performance in terms of Million Instructions Per Second (MIPS), averaged over all the benchmarks we used in our evaluation. For each benchmark, the MIPS rating is calculated by considering the number of instructions executed, the clock cycle count needed to execute the benchmark (Table 2), and the implementation frequency (Table 3). Table 5 shows the implementation area obtained from Design Compiler synthesis. The key insights from the results in Table 2, 3, 4, and 5 are explained in the following subsections.

Impact of Pipeline Depth. The first row of Tables 2, 3, 4, and 5 show the evaluation results for the pipelines without any forwarding or speculation. As expected, having more pipeline stages breaks down the critical paths for improved frequency. However, cycle count increases for the longer pipelines due to the increased number of stall cycles that are needed to resolve data hazards. In terms of the overall MIPS performance, the 5-stage pipeline leads to the most improvement relative to the 4-stage pipeline; the 7-stage

pipeline actually performs worse than the 4-stage. Lastly, deeper pipelining leads to larger area, due to the extra resources to implement the additional stages.

Impact of Data Forwarding. The second and third rows of Table 2, 3, 4, and 5 show the evaluation results for the pipelines that have data forwarding, but do not have speculation. As expected, cycle count improves for the more aggressive forwarding schemes since data hazards are resolved earlier, thereby reducing the amount of stalls. However, the addition of forwarding logic can adversely impact critical path delay, as can be seen in the reduction in the implementation frequency. Nevertheless, data forwarding is still beneficial for overall performance, as indicated by the large improvement in MIPS. Also, the area increase due to the forwarding logic is small (up to 32% over baseline) relative to the MIPS performance improvement (up to 116%).

Impact of NPC Predictor. Rows 4 to 9 of Table 2, 3, 4, and 5 show the evaluation results for the pipelines with the NPC predictor. Relative to the pipelines without any forwarding or speculation (i.e., row 1 vs. row 4), the NPC predictor improves cycle count by allowing a correct new instruction to be fetched on most cycles. Furthermore, the addition of the NPC predictor does not have significant impact implementation frequency, resulting in up to 59% overall improvement in MIPS relative to the baseline pipeline. Accordingly, the addition of predictor logic incurs extra area, up to 35% in comparison with the baseline.

If we compare the addition of NPC predictor against the addition of data forwarding (i.e., row 4 vs. rows 2 and 3), having a NPC predictor does not improve cycle count as much as having data forwarding. This is because the NPC predictor allows for fast hazard

resolution only for program counter (i.e., EIP in x86), while data forwarding provides fast hazard resolution for all the architectural states. Furthermore, the NPC predictor requires some warm-up time before it can start making good predictions. During this period, data hazards on EIP are resolved by stalling the instruction fetch. In terms of frequency, data forwarding increases critical path more than NPC predictors. Overall, however, MIPS performance gain from only having data forwarding is higher than the gain from only having an NPC predictor in a pipeline. In terms of area, at the chosen table size, the NPC predictor consumes more implementation area than data forwarding. Thus, in this case study, having data forwarding is overall more efficient than having an NPC predictor.

The NPC predictor can also be combined with data forwarding (i.e., rows 6 to 9 in Table 2), which further improves cycle count in deeper pipelines (6- and 7-stage). However, for shorter pipelines (4- and 5-stage), data forwarding can already resolve hazards very early (i.e., EIP forwarded from stage 2 to stage 1), therefore the NPC predictor does not provide any additional benefit. In comparison with pipelines with only data forwarding (i.e., rows 2 and 3 in Table 3), having the additional NPC predictor (i.e., rows 6 to 9 in Table 3) does not worsen the implementation frequency significantly. Thus, the improvements in cycle count from having an NPC predictor in the longer pipelines do translate to overall MIPS performance gains.

Finally, the prediction resolution schemes do not affect cycle count in the case of NPC predictor (i.e., rows 4, 6, and 8 vs. rows 5, 7, and 9 in Table 2). This is because there is no use of self-modifying code in our benchmarks. The NPC predictor simply remembers the length of previously seen instructions to predict the next instruction to fetch. Without self-modifying code, the NPC predictions would be correct except in the

case of a taken branch. In terms of frequency (Table 3), the ALAP resolution scheme permits much higher frequency than the ASAP scheme. This is because in ASAP scheme, there are multiple prediction checks in the pipeline, which lead to increased critical path. With comparable effect on cycle count, the overall MIPS (Table 4) is also better with the ALAP schemes than the ASAP scheme. In terms of area (Table 5), one might expect that ALAP will consume less area, but in practice the total implementation area is dominated by interconnection resources rather than the prediction check logic. In this case, we did not see any consistent trends indicating whether ALAP is better than ASAP scheme in terms of area, or vice versa.

Impact of Bimodal Predictor. Adding a Bimodal predictor lets speculation on branch instructions to reduce the lost fetch cycles after a taken branch. Rows 10 to 15 of Table 2, 3, 4, and 5 show the evaluation results for pipelines with the Bimodal predictor.

For pipelines without forwarding (rows 1, 4, and 5), we can clearly see the benefits of having a Bimodal predictor in improving the cycle counts (Table 2). However, for the pipelines with data forwarding, this is not the case. As with the NPC predictors, the Bimodal predictor does not provide much benefit for the shorter 4- and 5-stage pipelines, since branch target calculation for these pipelines can already be directly forwarded from stage 2 to stage 1 (incurring no stalls). For the longer 6- and 7-stage pipelines, the benefit depends on the interplay between the amount of the misprediction penalty and the benefit of having a correct prediction. For the pipelines with ALAP prediction resolution (rows 12 and 14), the overall misprediction penalty is larger than the stalls avoided from having correct predictions. Thus, Bimodal predictor is not beneficial for these shorter pipelines. For 6- and 7-stage pipelines with the more aggressive ASAP prediction resolution

scheme and the MAX data forwarding scheme, the Bimodal prediction can improve cycle count, albeit only slightly.

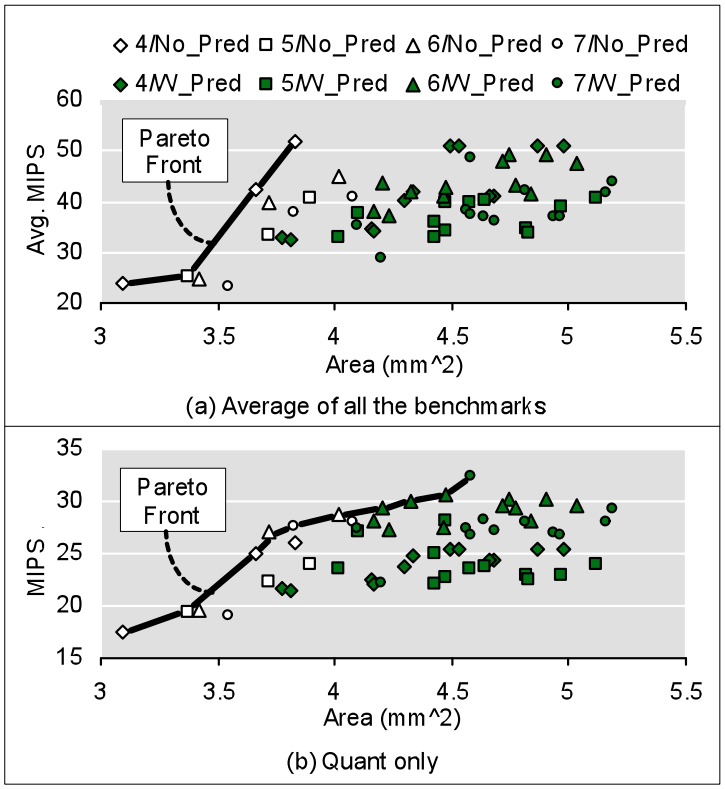


Figure 14. x86 cost-performance tradeoff.

Having a Bimodal predictor on top of an NPC predictor does not worsen the overall implementation frequency (i.e., rows 4 to 9 vs. rows 10 to 15 in Table 3). In terms of overall performance, the Bimodal predictor improves MIPS rating relative to pipelines without data forwarding (i.e., rows 10 and 11 vs. rows 4 and 5 in Table 4), but it does not help much for the pipelines with data forwarding already in place. Finally, the area (Table 5) increases accordingly as we add the Bimodal predictor.

Cost-performance Tradeoff. Figure 14(a) depicts the overall tradeoff between cost (area) and performance (Average MIPS over all of the benchmarks we studied). As

shown in the figure, the synthesized pipelines vary in their implementation cost and performance, providing a wide range of implementation alternatives to choose from depending on the desired target. In overall, we found that the Pareto optimal fronts consist of shorter pipelines (4- and 5-stage) without any speculation. Note that such insights on the design tradeoff for the pipelines and benchmarks we studied would have been difficult to learn without exploring many RTL designs by simulation and synthesis.

There is also an opportunity for application-specific pipeline customization since the Pareto fronts would change when optimizing for individual applications instead of an average. For example, Figure 14(b) shows the tradeoff graph for the Quant application only. As the figure shows, the Pareto fronts for Quant includes longer pipelines with speculation. The automatic pipeline synthesis capability of T-spec and T-piper makes it possible to customize and pick the best pipeline for a specific target application.

Automation benefit of T-spec and T-piper. Using T-spec and T-piper, we can fully automate the pipeline design flow starting from a non-pipelined datapath directly to an implementation. The approximate breakdown of time spent to obtain one data point in our study is as follows:

- Synthesizing a pipeline from a T-spec requires only seconds of T-piper execution.
- RTL simulation run can be done within one hour by simulating concurrently across multiple machines.
- Design Compiler synthesis of the Verilog of the pipeline takes a few hours.

Even then considering the lengthy synthesis time, without T-spec and T-piper, the design exploration cycle would be prohibitively bogged down by the manual RTL development time.

4.4. x86 Case Study 2: Application-Specific Processors

The classic 1991 paper by Bhandarkar and Clark measured a performance advantage of RISC over CISC in the context of in-order pipelined implementations of MIPS and VAX processors [3]. Beyond performance, CISC ISA processors also suffer from an increased cost in implementation area and power in order to capture the greater variety of instruction behaviors. These differences between RISC versus CISC have been largely neutralized in the realm of high-performance processors today where superscalar out-of-order execution is the norm. However, RISC ISAs still dominate the embedded processor domain where simple microarchitectures are common, with great attention paid to the design efficiency in terms of area and power. This case study investigates the opportunity to close the gap between CISC and RISC ISAs in in-order pipelined embedded processors when assuming a CISC processor can be customized to support only a particular target application's execution.

To maintain ISA compatibility, CISC processors are excessively burdened by a large number of instructions with a large variety of behaviors. In a very complicated CISC ISA like x86, many instructions, especially the complicated ones, are not used by the compilers. Some instructions are used only for OS bootstrapping. Some instructions are maintained solely for legacy compatibility. In a custom embedded processor tailored made for a particular application, the overheads associated with those unused instructions

can be avoided. Provided a sufficient subset of instructions is included, these customized processors with an incomplete native ISA support could nevertheless provide full ISA compliance by emulating the missing instructions in software (e.g., by trapping to PAL code [29]) when necessary, albeit at a large performance penalty.

Yiannacouras, et al. previously showed the opportunity to improve a RISC processor's performance-per-area metric by 25% by (1) pruning the processor's datapath to support only the instructions required by a particular application (i.e., ISA subsetting) and (2) tuning the processor's microarchitecture (pipeline depth, forwarding paths, etc.) to the application's specific execution behavior [71]. This case study evaluates the opportunity to improve performance, area, and power by making similar application-specific customizations in x86 processor implementations. The results (Section 4.4.4) show that application-specific customizations can significantly reduce the overhead in performance, area and power of an x86 processor relative to a RISC processor.

4.4.1. Application-Specific Customization Approach

This study considers the application-specific customization flow illustrated in Figure 15(a). Given the binary executable of a target application, an ISA specification, and an optimization objective, the goal of processor customization flow is to arrive at a processor implementation that can execute the unmodified binary of the target application and best maximizes the optimization objective.

At the ISA level, the subset of the ISA that is exercised by the target application is identified. The support for the unused instructions is later omitted from the implementation in hope to reduce implementation cost and clock cycle time. Such "ISA

subsetting" was previously shown to be effective in reducing the performance-per-area metric by 25% on average in the context of RISC processor design [71]. We expect that the benefit of ISA subsetting to be larger for a CISC ISA like the x86, since it encompasses much more unused instruction behaviors.

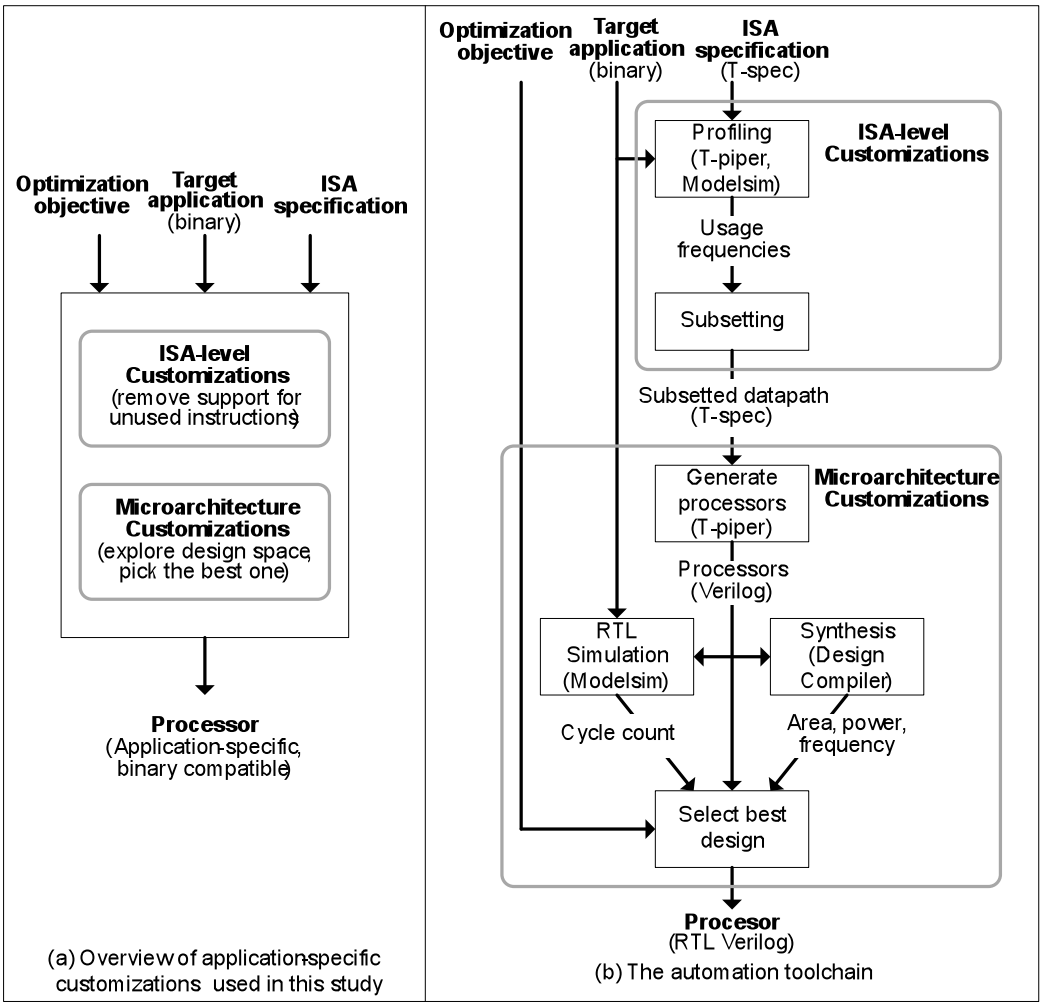


Figure 15. The application-specific processor customization approach.

At the microarchitecture level, the microarchitecture design space for the pruned ISA is explored to identify an instance that maximizes the optimization objective. This study considers the microarchitecture space of in-order pipelined processors with a well-

supported suite of options for data forwarding and speculative execution, which is suitable for "lean" processor implementations for applications with high-performance requirement, yet with strict constraints in power and area. As such, this choice of design space is relevant for many embedded application contexts.

Note that while these application-specific customization techniques by themselves are not new, this case study provides new insights by demonstrating their applications to reduce the overhead of supporting the popular x86 CISC ISA in in-order pipelined embedded processor designs. This study is made possible by T-spec and T-piper that enable the rapid design exploration of in-order pipelined processors, capable of supporting sophisticated ISA such as the x86.

4.4.2. Evaluation Framework

The application-specific customization flow outlined in Figure 15(a) could be realized in many ways. However, extensive use of design automation is needed to practically explore any nontrivial problem instances. Figure 15(b) shows the actual realization of the flow in a form of a design automation toolchain that leverages T-spec and T-piper.

Problem Inputs. The toolchain takes as its first input a checkpointed initial memory image, which includes the application binary. This initial memory image is prepared using the Simics simulator [38] so the execution of the application can be bootstrapped directly from the start of the application. The second input is the ISA specification, written in T-spec. The final input is an optimization objective to be maximized. In this study, we considered objectives that include wall-clock-time performance,

implementation area, power dissipation, as well as the normalized metrics of performance-per-area and performance-per-Watt.

ISA-Level Customization Step. The toolchain simulates the execution of a target application against the Verilog RTL implementation that corresponds directly to the T-spec specification (without any pipelining transformations or optimizations). The simulation is instrumented to collect profiling data about the usage frequencies of at different parts of the datapath (e.g., usage frequencies of different parts of the decoder module, functional units, etc). The profiling results are used to back-annotate the T-spec specification to identify unused portions of the datapath to be omitted from the pipelined implementations generated in the next step.

Microarchitectural Customization Step. This step explores the microarchitectural design space to select an implementation of the pruned T-spec datapath that best maximizes the chosen optimization objective. From a T-spec specification, rapid design space exploration (similar to what we did on the case study presented earlier in Section 4.3) is done by submitting different pipeline configurations to T-piper that controls the number and the positions of the pipelines stages and specifies where to apply forwarding and value-prediction optimizations to mitigate the penalty of data dependency stalls.

4.4.3. Processor Baselines, Target Applications, and Customization Options

Processor baselines. For the MIPS baseline processor, we used the 5-stage pipelined processor from the case studies previously discussed in Section 4.2, which has been shown to be comparable to a hand-made implementation, as well as one generated by SPREE.

For the x86 CISC baseline implementation, we used the T-spec datapath in the case study described earlier in Section 4.3. From the T-spec, T-piper was used to generate a 7-stage pipeline with a maximal data-forwarding network and a bimodal branch predictor based on 2-bit saturation counters. Unfortunately, there is not an open-source pipelined x86 implementation that would allow us to vet the quality of our x86 baseline implementation. We can glean some indications by comparing our x86 CISC baseline against our MIPS baseline. Bhandarkar and Clark [3] studied the relative overhead of an in-order pipelined CISC processor (Digital VAX 8700) against a RISC processor with a similar microarchitecture (MIPS M/2000). They reported a net cycle-based performance advantage of RISC over CISC (i.e., the RISC factor) of 2.7 on average, with a minimum of 1.8 and a maximum of 3.7. As a sanity check, we calculated the RISC factor between our x86 CISC baseline and the MIPS RISC baseline to be 2.1 on average, with a minimum of 1.4 and a maximum of 2.7 over the SPREE benchmark applications we tested. This comparison is again inexact but should give some support that our x86 processor pipeline is not grossly unreasonable for an in-order pipelined CISC processor.

Target Applications. We consider the same set of target applications from the SPREE collection [71] as in the previous case study (Section 4.3). Refer to Table 1 for a brief description of each application.

Customization options. For each application under study, we used the flow presented in Section 4.4.2 first to prune the unused instructions from the x86 ISA and then to generate a variety of implementations corresponding to different microarchitecture optimizations. During ISA subsetting, the following datapath support could become removed when they are made non-essential by pruned instructions.

- various parts of the decoder (i.e., decoder table, instruction length calculator, control logic generation, immediate value decoding) corresponding to pruned instructions
- various parts of the ALU (i.e., operand packing/unpacking logic, functional units, multiplier, divider) used by specific pruned instructions
- various parts of the flags calculation logic.
- various parts of the memory address calculation logic.
- multiplexers associated with aforesaid modules (e.g., pruning out a multiplier unit leads in removal of the multiplexer input that the multiplier is connected to).

From either the original full x86 T-spec or the pruned version, T-piper is used to generate various pipelined implementations using pre-prepared pipeline configuration scripts (i.e., a total of 240 pipelined implementations are studied in this paper). The configurations visited cover a design spanned by the orthogonal design choices used in the previous case study (Section 4.3), i.e., varying pipeline depths from 4 to 7 stages; forwarding schemes from no forwarding (None), forwarding as soon as the value is computed (Asap), and forwarding whenever possible (Max); and speculation schemes from no speculation (None), with a next-PC (NPC) predictor, and with a both NPC and bimodal predictors (Bimodal).

4.4.4. Results

ISA Subsetting. Let us first consider the effects of ISA subsetting only. The results from our evaluation are summarized in Figure 16 and Figure 17. For each target application, we created a corresponding subsetted x86 T-spec and used T-piper to create a

range of implementations according to the microarchitecture space described in Section 4.4. In this series of figures, the X-axis lists the different microarchitectures evaluated by our automatic development toolchain. The labels are in the format of {pipeline depth,

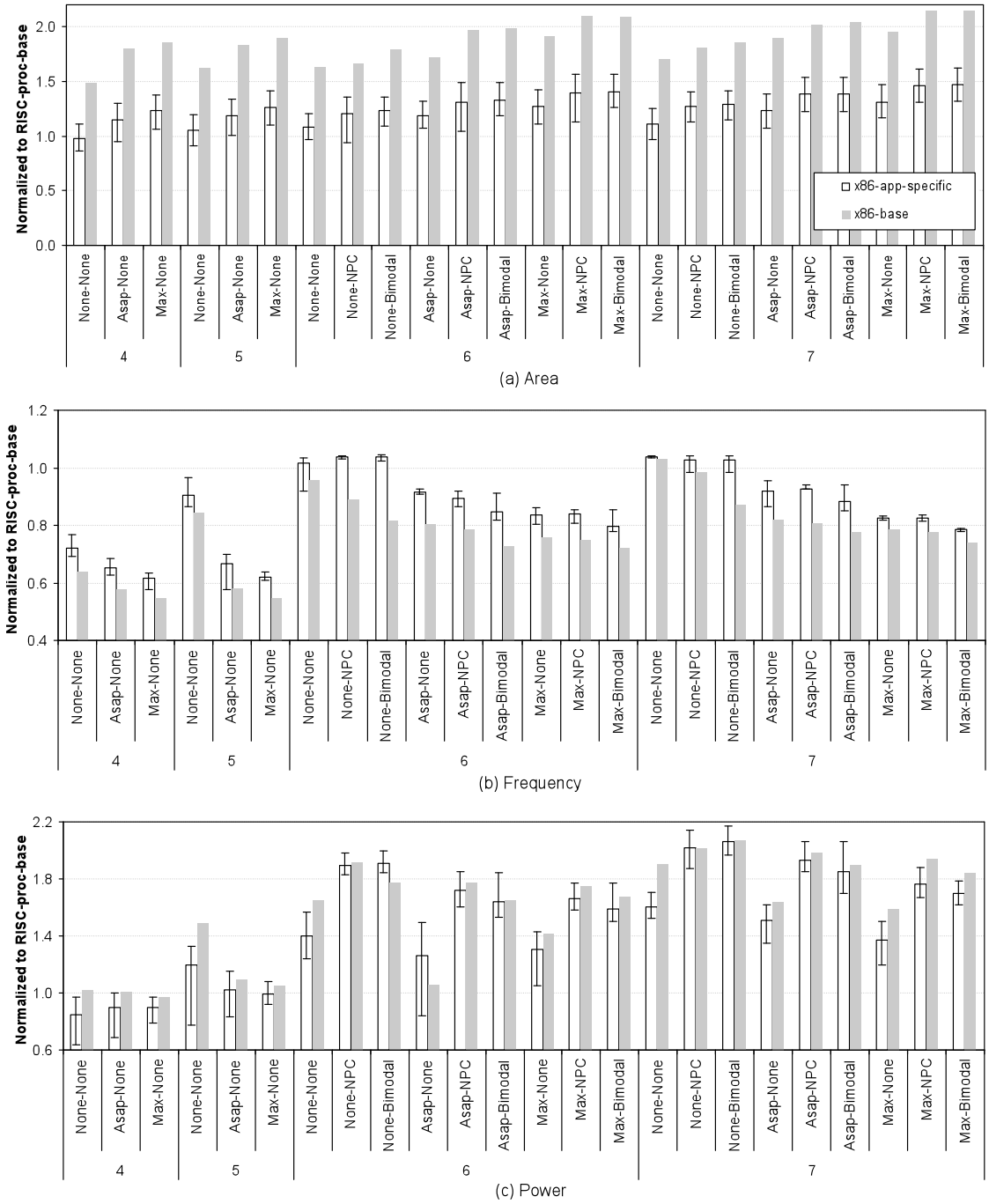


Figure 16. Implementation costs of the x86 processors under study.

forwarding scheme, speculation scheme}. For example, 7-Max-Bimodal refers to the 7-stage pipelined instance with maximum forwarding and a Bimodal predictor. The Y-axis shows the area, frequency, power, or performance averaged over the nine target applications and normalized to our baseline RISC processor. The x86-base bars show the values for the processor without any ISA subsetting, while the x86-app-specific bars show the processors with ISA subsetting given each target application. The x86-app-specific bars show the average value across each of the target application. The range markers represent the minimum and maximum values.

As shown in Figure 16(a), ISA subsetting effectively reduce implementation area relative to the x86-base, averaging in 33% of area reduction. For several microarchitectures, the reduction brings the area down to a level comparable to the area of our RISC processor baseline (e.g., pipelines without any forwarding nor speculation).

In terms of frequency, the improvement is not as much as area, averaging in 12% frequency improvement (Figure 16(b)). This is because the target applications still utilize several CISC instructions. Therefore, even with ISA subsetting, it is not possible to purely eliminate the support for CISC-style instructions. As such, the critical path of the CISC datapath does not get significantly affected by ISA subsetting.

As depicted in Figure 16(c), power dissipation of an ISA subsetted processor is not always lower than the non-subsetting x86-base processor (e.g., 6-stage, ASAP forwarding, no speculation). This is because the increase in frequency may lead to larger increase in power dissipation relative to the power savings afforded by the reduced area. In overall, ISA subsetting leads to an average of 6% reduction in power dissipation. Also,

for some microarchitectures (i.e., 4-stage pipelines), this reduction leads to power dissipation lower than that of the RISC baseline.

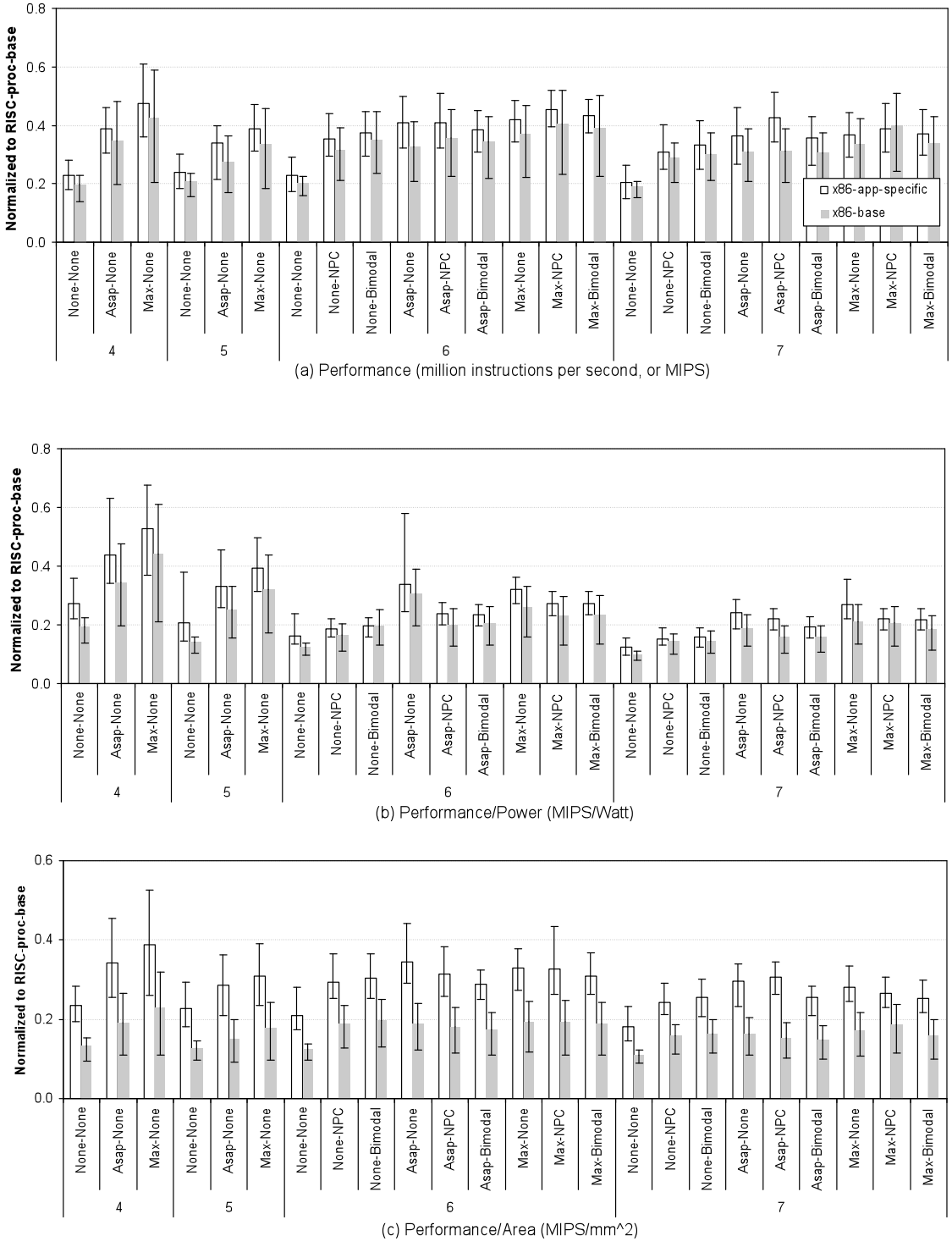


Figure 17. Performances of the x86 processors under study.

The performances for the processors under study are shown in Figure 17. We look at three performance metrics. The millions of instructions per second (MIPS) depends only on the implementation frequency and the cycle-per-instruction (CPI) performance of the microarchitecture, without any consideration to power or area overheads. The MIPS/Watt and MIPS/mm^2 consider the power and area overheads, respectively.

Since the ISA subsetting preserves datapath correctness and targets purely unused parts of the datapath, CPI of a subsetted processor remains the same to the non-subsetted x86-base processor (i.e., subsetting has no impact on microarchitecture effectiveness). Therefore, the improvement in MIPS performance is resulted from the frequency increase only. As such, the ISA subsetting reduces the CISC-to-RISC performance gap in MIPS (Figure 17(a)) by 12% on average (i.e., from an average of 3.1x the performance of the RISC processor baseline, down to 2.8x).

When power is considered (Figure 17(b)), the modest additional average power savings of 6% leads the larger reduction in the CISC-to-RISC gap in performance-per-Watt, averaging in 17% reduction (from 4.7x down to 3.9x). When area is considered (Figure 17(c)), the significant 33% savings from subsetting yields a 40% average reduction in the CISC-to-RISC performance-per-area gap, from an average of 5.9x down to 3.5x. Note that, as expected, this 40% improvement in performance-per-area metric is larger than the 25% improvement of the same metric that was reported in [71] when applying ISA subsetting on a RISC processor.

Microarchitecture customizations. Looking at the results in Figure 16 and Figure 17 more holistically, comparing the results across the X-axis, we can very clearly see the

dramatic effect of microarchitecture customization for both the x86-base and x86-app-specific sets of processors. The microarchitecture customizations explored by our toolchain are beneficial in providing a wide range of implementation options to choose from. The design space, starting from the simplest microarchitecture (4-None-None) and ending at the most aggressive one (7-Max-Bimodal), covers a large range of implementation costs and performances.

For example, for the x86-base processors (i.e., without ISA subsetting), the area, frequency, and power can vary by 1.4x, 1.9x and 2.1x, respectively. Performance in terms of MIPS, MIPS/Watt, and MIPS/mm2 vary by 2.2x, 4.4x, and 2x, respectively. These large variations exist in the x86-app-specific processors (i.e., with ISA subsetting) as well. Variations in area, frequency, and power for these processors are 1.5x, 1.7x, and 2.4x, respectively. For performance, they are 2.3x, 4.1x, and 2.1x in terms of MIPS, MIPS/Watt, and MIPS/mm2. The large variations provide customization opportunity by selecting the best microarchitecture instance to satisfy the desired optimization objective.

Bottom line: application-specific x86 vs. RISC. Tables 6, 7, and 8 show the characteristics of the best application-specific x86 processor generated by our toolchain for each of the application under study, when optimized for performance (MIPS), performance/power (MIPS/Watt), and performance/area (MIPS/mm^2), respectively. The tables provide the microarchitecture selected as the best processor, along with the power, area, and performance of the processor. Each of these metrics are shown both in terms of their absolute values normalized to RISC baseline, and as percentage of improvements relative to the non-subsetted x86 baseline processor with the most aggressive 7-Max-Bimodal microarchitecture. The absolute values indicate the CISC-to-RISC gap, while

the percentages show the benefits of applying the application-specific customizations relative to a non-customized x86 processor. Lastly, the table provides the performance range to provide insights of alternative design points (that gives median and minimum performance) aside from the best performing design. Notice that most of the best designs are at a significantly higher performance level than the median and minimum designs.

The application-specific customizations improves performance relative to the most aggressive x86 processor by 42%, 185%, and 151% on average for the MIPS,

Table 6. x86 application-specific processors with the best performance.

Target application	Micro-architecture	Power	Area	Performance	Performance range (median, min)
Bitcnt	4-Max-None	0.86 (+53%)	1.26 (+41%)	0.51 (+58%)	0.34, 0.17
Bubble_sort	4-Max-None	0.88 (+52%)	1.06 (+50%)	0.56 (+56%)	0.39, 0.18
Crc	4-Max-None	0.79 (+57%)	1.10 (+49%)	0.41 (+44%)	0.31, 0.16
Des	4-Max-None	0.89 (+52%)	1.15 (+46%)	0.46 (+57%)	0.30, 0.15
Fft	6-Max-Npc	1.72 (+7%)	1.50 (+30%)	0.52 (+22%)	0.45, 0.24
Fir	4-Max-None	0.96 (+48%)	1.23 (+43%)	0.51 (+35%)	0.42, 0.26
Iquant	7-Asap-Npc	2.01 (–9%)	1.47 (+31%)	0.44 (+23%)	0.39, 0.26
Quant	7-Asap-Npc	2.07 (–12%)	1.54 (+28%)	0.41 (+23%)	0.36, 0.23
Vlc	4-Max-None	0.90 (+51%)	1.31 (+39%)	0.61 (+59%)	0.41, 0.19

Table 7. x86 application-specific processors with the best performance/power.

Target application	Micro-architecture	Power	Area	Performance	Performance range (median, min)
Bitcnt	4-Max-None	0.86 (+53%)	1.26 (+41%)	0.59 (+238%)	0.23, 0.11
Bubble_sort	4-Max-None	0.88 (+52%)	1.06 (+50%)	0.64 (+228%)	0.25, 0.12
Crc	4-Max-None	0.79 (+57%)	1.10 (+49%)	0.52 (+239%)	0.20, 0.11
Des	4-Max-None	0.89 (+52%)	1.15 (+46%)	0.52 (+224%)	0.20, 0.10
Fft	4-Max-None	0.87 (+53%)	1.32 (+38%)	0.51 (+123%)	0.27, 0.14
Fir	6-Asap-None	0.84 (+55%)	1.10 (+49%)	0.58 (+182%)	0.26, 0.15
Iquant	4-Max-None	0.97 (+47%)	1.30 (+39%)	0.41 (+110%)	0.24, 0.15
Quant	4-Max-None	0.94 (+49%)	1.25 (+42%)	0.43 (+135%)	0.22, 0.13
Vlc	6-Asap-None	0.90 (+51%)	1.31 (+39%)	0.68 (+225%)	0.26, 0.12

Table 8. x86 application-specific processors with the best performance/area.

Target application	Micro-architecture	Power	Area	Performance	Performance range (median, min)
Bitcnt	4-Max-None	0.86 (+53%)	1.26 (+41%)	0.40 (+167%)	0.27, 0.16
Bubble_sort	4-Max-None	0.88 (+52%)	1.06 (+50%)	0.53 (+214%)	0.34, 0.18
crc	4-Max-None	0.79 (+57%)	1.10 (+49%)	0.37 (+182%)	0.25, 0.16
des	4-Max-None	0.89 (+52%)	1.15 (+46%)	0.40 (+191%)	0.25, 0.14
fft	6-Asap-None	1.49 (+19%)	1.32 (+38%)	0.38 (+90%)	0.31, 0.19
fir	6-Asap-None	0.84 (+55%)	1.10 (+49%)	0.44 (+149%)	0.33, 0.23
iquant	6-Asap-None	1.49 (+19%)	1.28 (+40%)	0.34 (+102%)	0.29, 0.22
quant	6-Asap-None	0.94 (+49%)	1.25 (+42%)	0.32 (+105%)	0.25, 0.18
vlc	4-Max-None	0.90 (+51%)	1.31 (+39%)	0.47 (+160%)	0.30, 0.16

MIPS/Watt, and MIPS/mm2 metrics, respectively. Even with such significant improvement, there is still exists CISC-to-RISC performance gaps of 2x, 1.9x, and 2.5x for the three aforesaid performance metrics, respectively.

In terms of power, an average saving of 33%, 52%, and 45% with respect to the most aggressive x86 processor was achieved when considering the MIPS, MIPS/Watt, and MIPS/mm2 metrics. More importantly, for some applications (e.g., bitcnt, bubble_sort, crc, des) such saving brings down the power dissipation to a level even lower than that of the baseline RISC processor. This is because these applications favor the shorter 4-stage pipeline, which typically consume less power than the deeper pipelined designs.

Finally, the area saving relative to the most aggressive x86 processor averages to ~40% for all the performance metrics we looked at. The end result is an average of 25% CISC-to-RISC area gap. For some target applications (e.g., bubble_sort, crc, fir), this area gap is as low as 10% or less.

Chapter 5

In-order Pipeline Verification

High-level (above RTL) design frameworks, like T-spec and T-piper, that employ design abstractions with precise semantics make it possible for designers to formally verify the properties and correctness of their initial design specifications. 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 implementation 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.

Taking advantage of the precise design semantics of high-level design frameworks, one should extend formal verification technologies to ensure not only the correctness of the initial specification but also its equivalence with the output of subsequent synthesis and translation. With recent advances in combining model checking [8] and theorem

proving techniques to curtail state-explosion, compositional model checking [42] has been applied to successfully verify functional equivalence between non-trivial pipelines and their specifications [26][37]. Unfortunately, the manual effort involved in compositional model checking (e.g., applying abstractions and compositional reasoning) was reported to be extremely high [37].

In this chapter, formal verification is integrated with our T-piper high-level pipeline synthesis framework. The integration allows formally proving that the pipeline RTL output of T-piper, with its concurrent execution of transactions and the intricacies of hazard resolutions, does result in the same execution as if the transactions were executed one-at-a-time as prescribed by the T-spec transactional semantics.

Specifically, T-piper is extended to use Cadence SMV compositional model checker [43] to automatically verify the functional equivalence between the input T-spec and the output pipeline implementation. Furthermore, to make the system practical, T-piper automatically applies abstraction and compositional reasoning techniques, therefore avoiding the need for manual compositional model checking effort. We demonstrate automatic verification of 9 processor pipelines for the MIPS ISA and 9 pipelines for a hypothetical ISA with CISC-like memory-to-memory instructions. We also discuss how integrated formal verification helped us uncover T-piper implementation bugs.

The rest of the chapter is organized as follows. Section 5.1 provides a background in model checking. Section 5.2 presents the integration of automatic verification using compositional model checking into T-piper. Finally, Section 5.3 reports our case studies on automatic verification of example processor pipelines.

5.1. Model Checking Overview

This section uses a simple example from [43] 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 18(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 18(a) (designated "Implementation") depicts a 3-stage pipelined datapath with maximal data forwarding support.

5.1.1. Model Checking

To verify that the Specification and the Implementation are functionally equivalent, we first create cycle-accurate and (at least initially) 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 the 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.

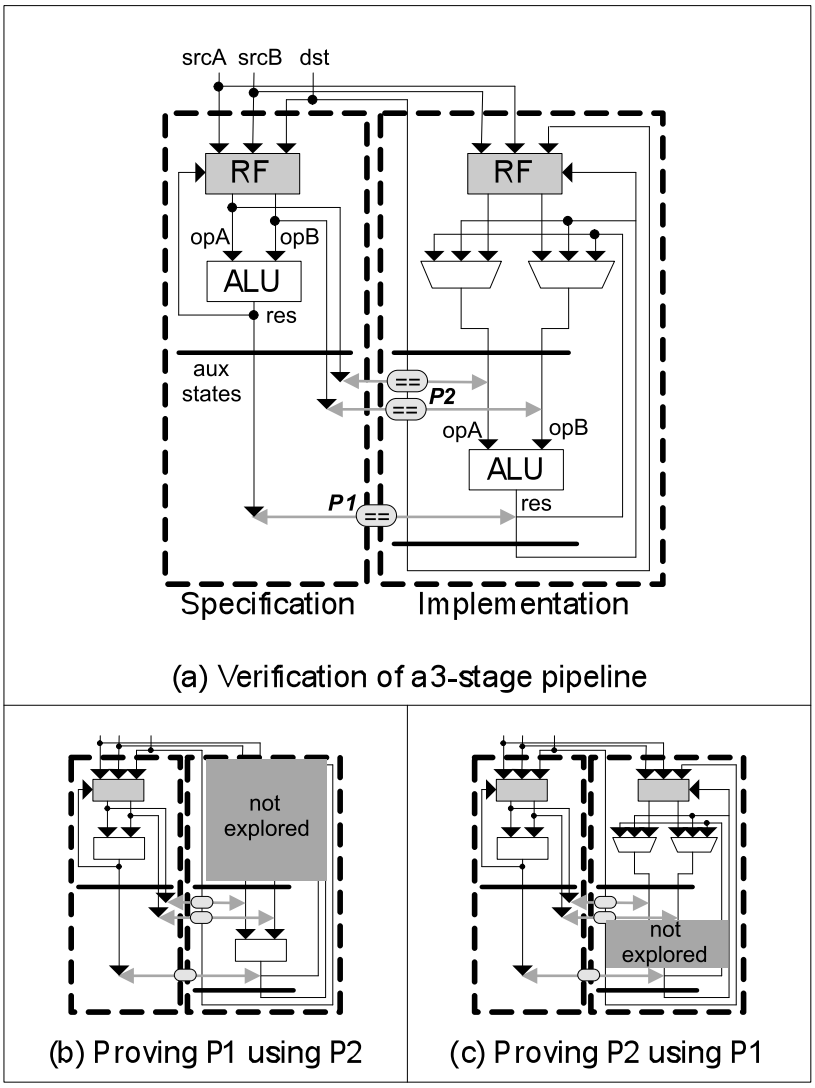


Figure 18. A simple verification example from [43].

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 of the RF would be declared as free variables so that

the model checker considers all combinations of reading and writing the different RF entries.

5.1.2. Abstraction and Decomposition

Given a correctly formulated set of (1) Specification and Implementation models, (2) the 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 could 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 explosion. A standard workaround in model checking is to assume that the ALU blocks in the Specification and the Implementation are identical. Thus, they can be captured as uninterpreted functions [43] in the verification model, and 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 [43] 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 5.1.1 has a cone of influence that covers the entirety of the Specification and the Implementation. Compositional reasoning [43] 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 18(b) illustrates the part of the pipeline that can be left out when proving P1 assuming P2; Figure 18(c) shows the same for when proving P2 assuming P1.)

Another well-known decomposition is case analysis [43], 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 [37].

5.2. Automatic Verification

This section describes the extensions to T-piper to enable automatic compositional model checking to prove that the in-order pipelined implementation synthesized by T-piper executes and performs the same order of transactions and state updates as its T-spec

datapath specification. In other words, the verification demonstrates that the synthesized pipelined datapath is functionally equivalent to the non-pipelined T-spec datapath under its transactional execution semantics.

More specifically, given the inputs of T-spec, P-cfg and H-cfg files, T-piper generates a verification file in the SMV language that can be directly submitted to Cadence SMV [43]. Ideally, we would like to model check the RTL Verilog design directly. The current choice of the SMV language is simply because Cadence SMV was 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. Below, after first clarifying the verification objectives, we explain T-piper's model generation process and the proof procedure.

5.2.1. Verification Objective

Since the implementation pipeline in our context is automatically generated, any bug in the implementation would have to be caused by a bug in T-piper. There are two classes of bugs that can occur in T-piper: (1) a fundamental bug in the pipeline synthesis algorithms, or (2) a programming bug in the coding of the synthesis algorithms (e.g., a bug in the synthesis code, a bug in the code that checks for the validity of the input T-spec or configuration files, etc). Both types of bugs can be exposed by the verification approach described in this section.

Starting from the T-spec netlist, T-piper introduces pipeline stage registers and pipeline control logic such that the overlapped transaction executions on the synthesized pipelines produce 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 synthesized pipelined implementation uses the same next-state compute blocks (e.g., op1, op2, op3, op4, op5, and m1 in the datapath example depicted in Figure 2(b)) as the original T-spec. Since these blocks are a part of the specification, we assume they are correct and have been verified independently.

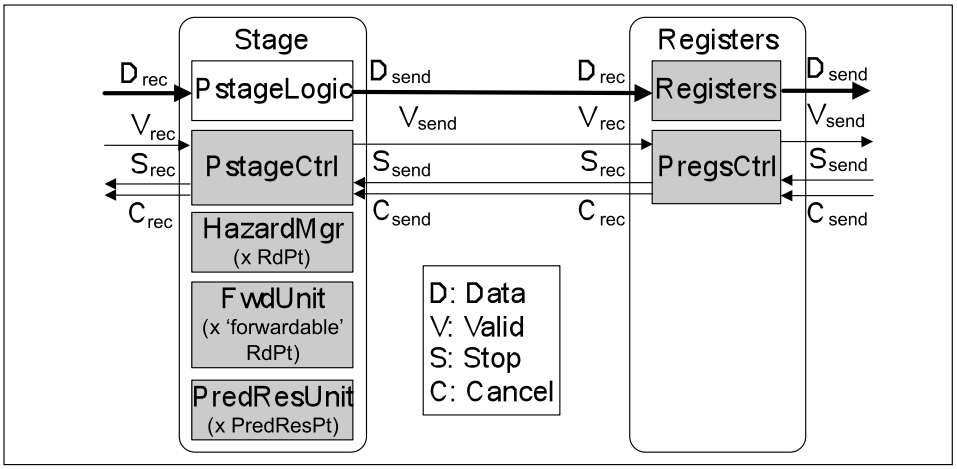


Figure 19. Pipeline internals.

The focus of our verification effort is the correctness of the pipeline register insertion and the pipeline control logic generated by T-piper. Chapter 3 has provided detailed discussion on the pipeline structures synthesized by T-piper. A brief summary is presented here for convenience. Figure 19 depicts the pipeline structure generated by T-piper, with the pipeline control logic in the shaded blocks. In the figure, PstageLogic (pipeline stage logic) refers to the original user-provided next-state compute blocks specified in T-spec. The pipeline control logic blocks introduced by T-piper are:

- PstageCtrl (Pipeline Stage Controller) interacts with the PstageLogic in a given stage and manages communication with the adjacent pipeline state registers.

- HazardMgr (Data Hazard Manager) 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.

- FwdUnit (Forward Unit) manages forwarding of both actual and predicted values. It includes a forwarding multiplexer (e.g., fwd in Figure 4(b)) and acts as a proxy to a state read interface, providing either a forwarded value or an actual state-read value. One FwdUnit is generated for each state-read interface where forwarding from a downstream stage is supported.

- PredResUnit (Prediction Resolution Unit) compares a predicted value with the actual value eventually produced by the datapath. The unit triggers squash on a misprediction. One PredResUnit is generated for each PredResPt in the stage.

5.2.2. Verification Models

This section describes how T-piper emits the models for use with the Cadence SMV model checker [43]. Prior to abstraction, the specification and the implementation models are cycle-accurate and bit-true representations of the non-pipelined input T-spec and the synthesized pipelined implementation, respectively. The specification model is automatically augmented with the necessary auxiliary states and logic for the refinement mapping.

Uninterpreted Functions (UFs). To curtail state explosion, T-piper generates the simplest possible models while exposing sufficient details to facilitate verification of the pipeline control logic. In the verification models, the internal details of the next-state

compute blocks (PstageLogic in Figure 19) are abstracted as uninterpreted functions. Recall that these blocks are provided by the user and are assumed to be correct as given.

On the other hand, the implementation model must expose faithfully the pipeline control logic (the shaded units in Figure 19) introduced by T-piper. To do so, the abstraction retains the following details needed by the pipeline control logic:

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

Figure 20 sketches the algorithm for automatic abstraction in T-piper. The algorithm produces a set of UFs and a list of state write sources that it represents. The first part (comprising the first 3 for-loops) considers all the write-data sources of an architectural state and allocates a minimal number of UFs to abstract the PstageBlocks of the stage(s) where the write-data originates. The final for-loop assigns a UF for the PstageBlock of any stage that has not been abstracted in the first part, ensuring that all PstageBlocks are abstracted.

To minimize the number of UFs, a single UF is used to represent multiple write-data sources where appropriate. For example, state A in a design may have write-data sources

src1 and src2 that are placed at pipeline stages s1 and s2, while state B may have write-data sources src3 and src4 at stages s2 and s3. In this case, only three UF's are needed to represent (1) src1 in s1; (2) src2 and src3 combined in s2; and (3) src4 in s3. Although only one UF is used to represent two write-data sources (src2 and src3) in s2, from the pipeline control logic's perspective, state B still sees two write-data sources (at s2 and s3); similarly, state A sees two write-data sources (at s1 and s2).

```
Inputs: T-spec, P-cfg
Output: a set of <uf, pstage, <write sources>>

// Find shared write sources, and track them in a database
for each ws1 write source of state S1 in T-spec {
     for each ws2 write source of state S2 in T-spec other than S1
       if(ws1 == ws2) // same source data, but write to different states
         add_to_shared_wr_src_db(ws1);
}

// Create UF to represent each shared write source
for each ws write source in shared_wr_src_db {
  create_new_uf(ws, get_pstage(ws, P-cfg), uf_db);
}

// Assign private write sources to existing uf, if any. Or, create new uf.
for each ws write sources not in shared wr_src_db {
  if(uf_exist_in_pstage(get_pstage(ws, P-cfg), uf_db))
    assign ws to existing uf;
  else
    create_new_uf(ws, get_pstage(ws, P-cfg), uf_db);
}
```

Figure 20. Algorithm to abstract the PstageBlocks by a minimum number of UFs.

Free Variables. T-piper automatically declares certain control signals as 'free' variables, such as with the RF read and write indices in the example in Section 5.1. 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 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 sake.

Auxiliary States and Coordination Logic. 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 synthesized by T-piper are discussed a later section.

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 was not needed in the example presented in Section 5.1, since the maximally forwarded pipelined implementation in the example never stalls. In general, a pipeline implementation may still need to stall when the available forwarding paths do not resolve all hazard conditions.

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 another downstream transaction and the value is not yet available through forwarding, automatic speculative mechanisms in a T-piper pipeline allow the transaction to utilize a predicted value generated by a user-provided value predictor in place of the actual state value. 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.

In the case of a correct prediction, we only 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 [26] 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. On the other hand, 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 the predicted value, T-piper uses the value generated by the specification model as an 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 always read and written by each instruction. To further aid in simplifying 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.

5.2.3. Proving Correctness

Correctness Properties. T-piper automatically places refinement maps and specifies properties to prove the correctness of:

- State write value (P-wr), which states that any architectural state updates made in the implementation model should be consistent to those in the specification model. The property P1 (i.e., correctness of RF writes) in Section 5.1 is this type.

- State read value (P-rd), which states that any architectural state reads made in the implementation model should be consistent to those in the specification model. The property P2 (correctness of RF reads) in Section 5.1 is this type. Note that in an implementation model with data forwarding, state read value is obtained at the output of the FwdUnit.

- Uninterpreted function (UF) output (P-uf-out), which states that the output produced by each UF in the implementation model should be consistent with the one in the specification model. The property P1 in Section 2 is also of this type (since ALU output is the RF write data).

The P-wr properties by themselves would be necessary and sufficient to prove 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 automatically performs decomposition utilizing the following heuristics:

- 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(b)), which are already identified by T-piper during pipeline synthesis. A write-data source is either an output of a UF (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 implementation 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 a UF 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 UF inputs should either be a state read data (assumed correct in P-rd) or a UF 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 UF.

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. If the state is an array, 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.

5.2.4. Verification Examples

Figure 21(a) depicts the verification model for the specification datapath derived from the T-spec in Figure 2(b). Three UFs are needed in these models, s1, s2, and s3, corresponding to first three stages of the target 4-stage pipelines from Figure 4(a), Figure 4(b), and Figure 4(c). No UF is needed to represent the last pipeline stage since it has only the write interface to state R (i.e., no computational block). Notice that the next-state compute blocks op2 and op3 in Figure 2(b) are abstracted away as a single UF s1. The figure also shows the correctness properties automatically placed by T-piper, which are used to decompose the verification problem into smaller sub-problems.

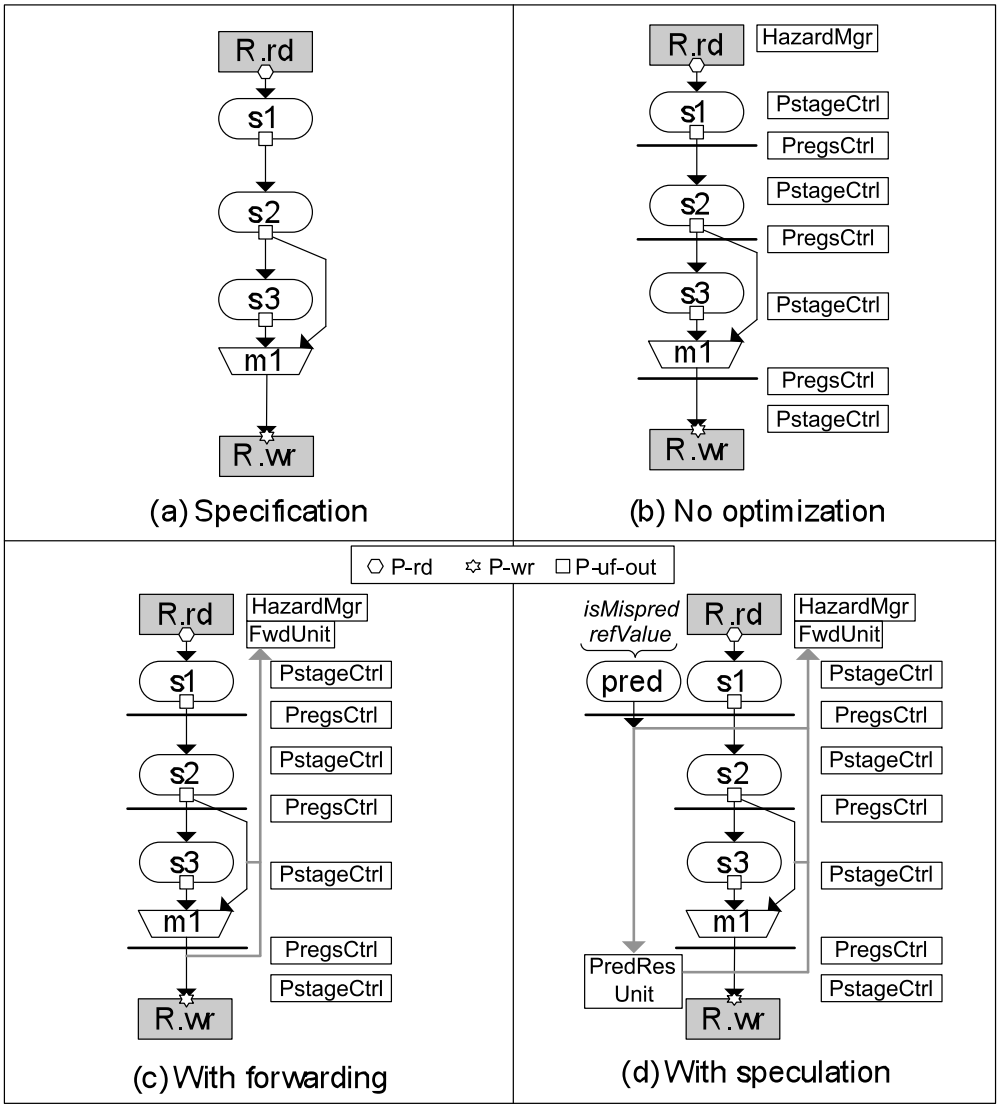


Figure 21. Verifying pipelines with stalling, forwarding, and speculative execution.

Figure 21(b), Figure 21(c), and Figure 21(d) show the implementation models generated by T-piper to verify the pipelines in Figure 4(a), Figure 4(b), and Figure 4(c), respectively. In Figure 21(b), the pipeline has no optimization. Thus, data 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. These control blocks are shown in white boxes in the figure. Also, although not shown in the figure,

the read and write enables of state R are declared as free variables to make sure that all possible transaction sequences in the pipeline are explored.

For the pipeline in Figure 21(c), T-piper includes the forwarding paths and the necessary logic (FwdUnit) in the verification 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 in the design. Although not shown, the model Figure 21(b) and Figure 21(c) has coordination logic to stall the specification model when the implementation model stalls.

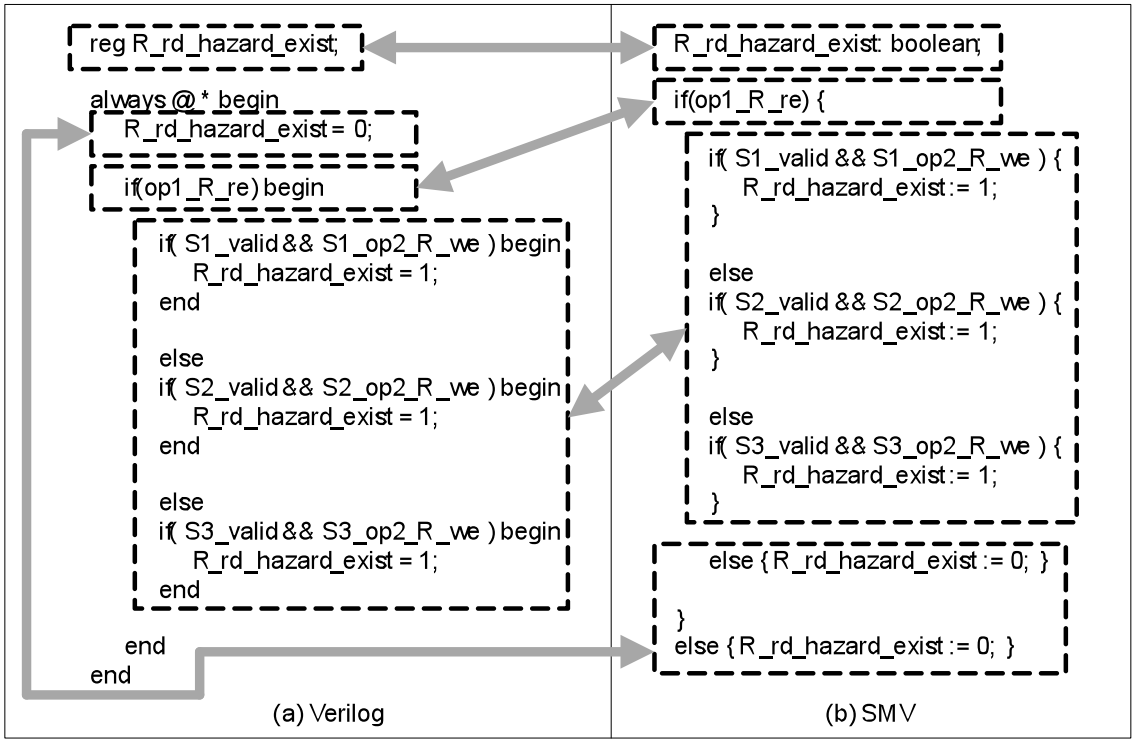


Figure 22. Verilog and SMV excerpts from verification example in Figure 21(b).

In Figure 21(d), a predictor logic block pred is added to enable 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 coordination logic to stall the specification model when the implementation is emulating a misprediction; and (3) the flag bits to track 'shadow' transactions in the pipeline (as described in Section 5.2.2).

As mentioned previously, the Verilog and SMV descriptions of the implementation are generated from a common RTL internal representation at the final step of the synthesis process. Thus, both the Verilog and SMV descriptions contain the exact same pipelining logic, except that they are written in different syntax. Figure 22 shows Verilog and SMV excerpts of the generated HazardMgr for the pipeline verification example in Figure 21(b). The figure highlights the various parts of the excerpts, using the arrows to indicate the equivalent parts between the Verilog and SMV descriptions.

In addition to the implementation model, the SMV file also contains the specification model, auxiliary states, coordination logic, and correctness properties to model check, as described in earlier sections. Figure 23 shows an SMV excerpt for three generated correctness properties for the example in Figure 21(b). The first property, property_R_rd_d, is of P-rd type (see Section 5.2.3), to check the correctness of state R read data. The comment in the excerpt shows the temporal logic description [43] of the property. The G is for "globally" operator, and the → sign indicates "imply" relationship. For property_R_rd_d property, "G (S1 done computing → R read data equal ref)" says that at all time, a completion of stage S1 (where R read interface is) implies that the read data of R in the implementation model is equal to its reference value produced by the specification model. The other two properties are of types P-uf-out and P-wr, and they specify the correctness of pipeline stage S2 output and R write data, respectively.

```
/* ---- State R read data correctness */
/* G (S1 done computing → R read data equal ref) */
layer property_R_rd_d: {
     if(S1_preg_update)
       impl_R_rd_d := ref_R_rd_d;
}

/* ---- Stage S2 output correctness */
/* G (S2 done computing → S2 output equal ref) */
layer property_S2_out: {
    if(S2_preg_update)
        impl_S2_out := aux_S1_ref_S2_out;
}

/* ---- State R write data correctness */
/* G (S4 done computing → R write data equal ref) */
layer property_R_wr_d: {
     if(S4_preg_update)
       impl_R_wr_d := aux_S3_R_wr_d;
}
```

Figure 23. SMV excerpts for the correctness properties from Figure 21(b) example.

5.3. Evaluation

The proposed automation approach described in the previous section has been integrated with our T-piper pipeline synthesis tool. Given the inputs of T-spec, P-cfg and H-cfg files, T-piper generates both a design file in RTL Verilog and a verification file in the SMV language, which can be directly submitted to Cadence SMV [43]. This section discusses the findings from several case studies in applying this approach. In particular, we collected the following key verification metrics from the case studies (similar to those in [37]):

- The number of correctness properties. This metric is suggestive of the manual effort and knowledge obviated by our automation approach.
- The number of state variables for the property with the largest cone of influence. This number indicates the likelihood of encountering state explosion.
- The number of BDD nodes allocated during the model checking. This runtime measure of SMV data size is most indicative of the likelihood of encountering state explosion.
- The execution time spent by the model checker to complete the verification.

We should remind the readers that without the abstractions and decomposition strategies automated in Section 4, SMV would have encountered state explosion on even simple examples like the ones in Figure 29.

5.3.1. Comparison with Manual Verification

In this case study, we compared the verification of the simple 3-stage pipeline example from Section 2, when done manually (as prescribed in the Cadence SMV tutorial [43] where the example is taken from) versus with our automatic approach. For the automatic approach, we created a T-spec for the specification datapath and used T-piper to synthesize and verify the target pipeline (i.e., 3-stage with maximal forwarding).

T-piper automatically arrives at the same correctness properties as stipulated by the manual counterpart (i.e., correctness of RF operands and ALU output), except for an additional P-wr type property (Section 5.2.3) only in the T-piper verification. It is included because T-piper conservatively assumes that there may be multiple write-data

sources to a state element, and places a P-wr property at each state write port during decomposition. The inclusion of P-wr properties leads to finer decomposition and does not affect the soundness of the functional equivalence proof.

The maximum number of state variables in the T-piper generated verification model is 40, whereas the manual effort required a maximum of 25. Correspondingly, the model checking time and the number of BDD nodes allocated are worse in the automatic approach than in the manual one (i.e., 0.44sec vs. 0.05sec; 98K vs. 12K). The differences are due to slightly less optimized (implementation-wise) pipeline control logic in the synthesized pipeline. During pipeline synthesis, T-piper could not infer (as in the manual design effort) the RF is read and written on every cycle and therefore could not make the associated simplification in the pipeline control logic. We have extended T-spec to allow users to provide these additional assumptions, but T-piper does not yet support automatic pruning of the pipeline control logic based on these user-provided assumptions.

Despite these quantitative disadvantages, we will next show that the automatic approach is in fact quite capable at handling non-trivial designs. It is also important to keep in mind that our automatic approach completely eliminates the manual effort in creating the verification models and in applying abstractions and decompositions. Even designers without any formal verification knowledge or experience can invoke the automatic approach.

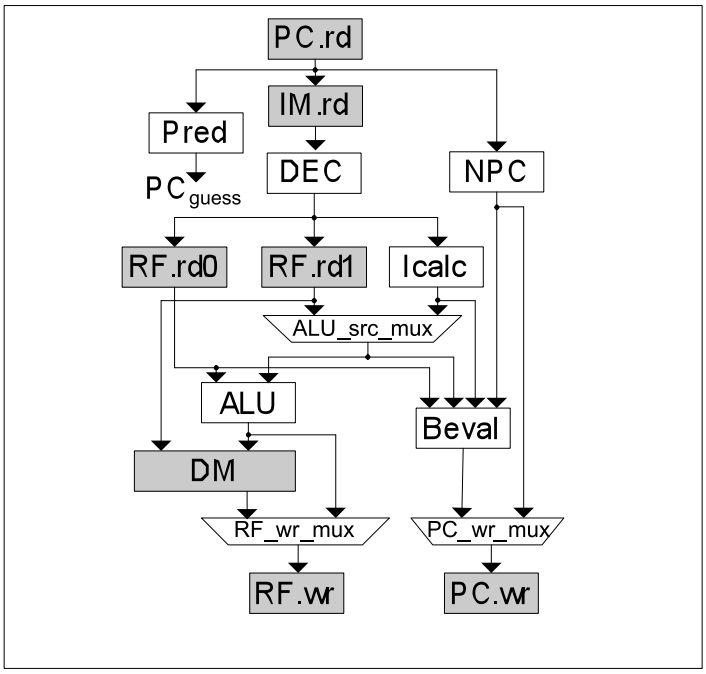


Figure 24. The T-spec of the load-store processor datapath under study.

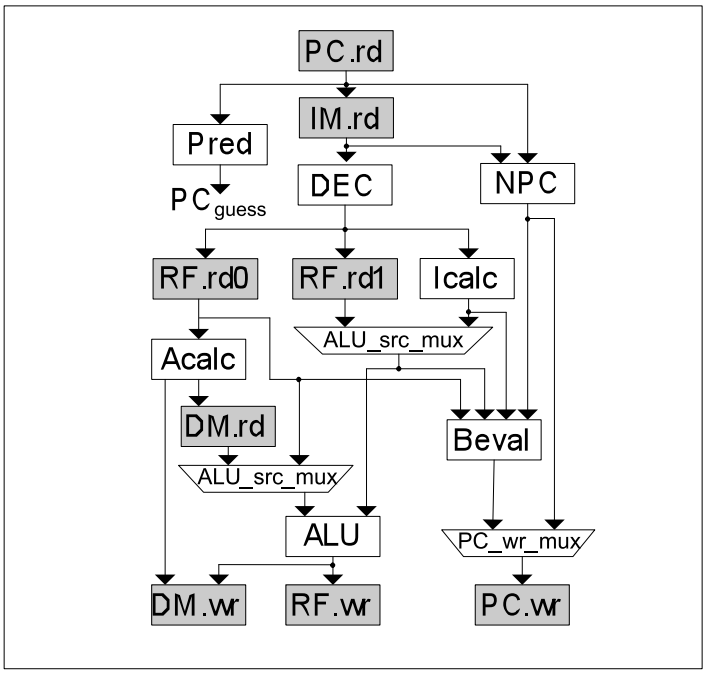


Figure 25. The T-spec of the memory-memory processor datapath under study.

5.3.2. Verification of Load-Store and Memory-Memory Processor Pipelines

We carried out two processor design case studies based on the two T-specs shown in Figure 24 and Figure 25. Figure 24 corresponds to the non-pipelined implementation of the MIPS RISC ISA; Figure 25 corresponds to the non-pipelined implementation of a hypothetical ISA with CISC-like memory-memory instructions. The datapath style in Figure 25 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 pipelines, varying in pipeline stages (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)).

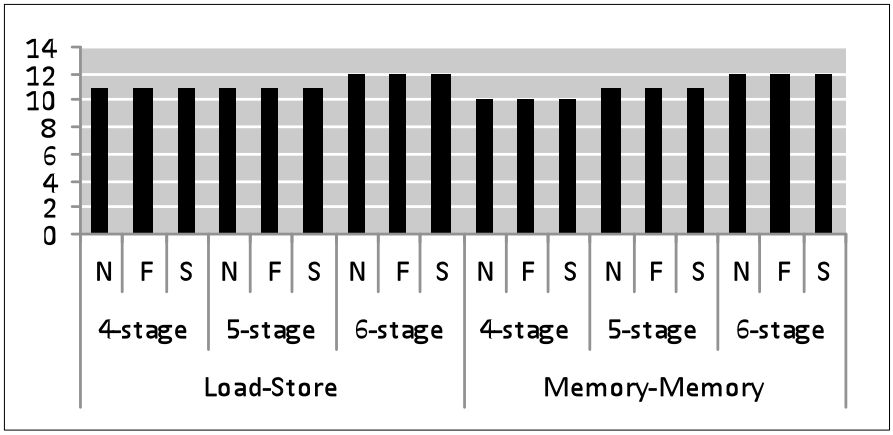


Figure 26. Number of correctness properties for each pipeline being verified.

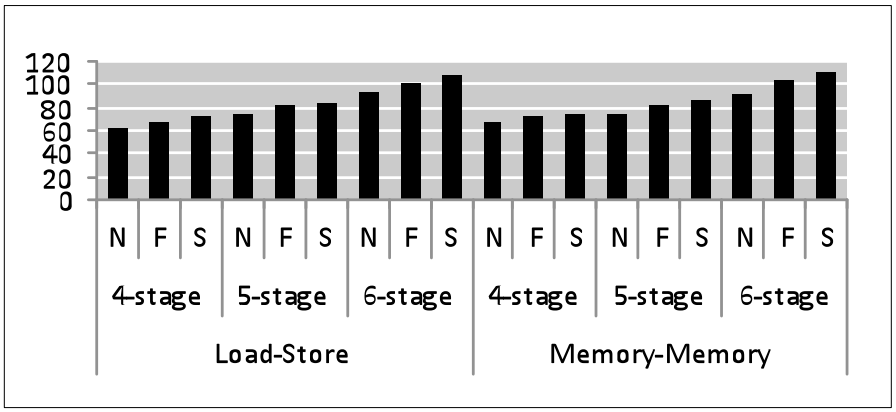


Figure 27. Number of states in the largest property for each pipeline being verified.

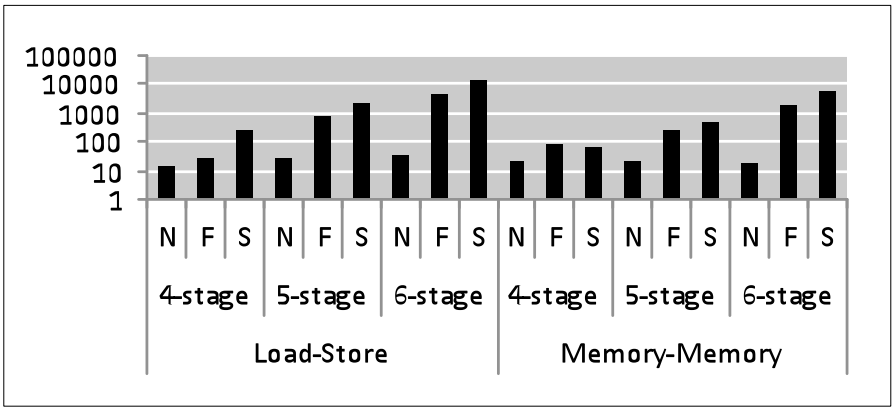


Figure 28. Model checking time (seconds) for each pipeline being verified.

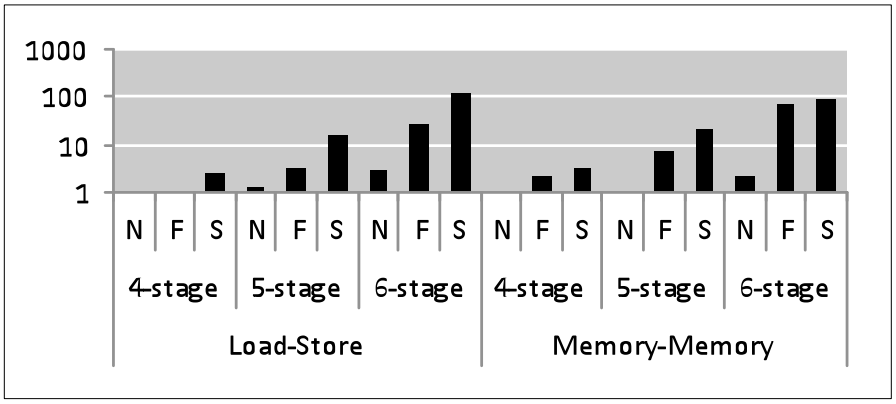


Figure 29. The BDD nodes allocated (in millions) for each pipeline being verified.

Results. All of the designs were checkable using Cadence SMV running on a 2 GHz PC with 4 GB of DRAM. Figure 26, Figure 27, Figure 28, and Figure 29 summarize the verification results. To the first order, deeper pipelines are more expensive to verify, as indicated by the higher cost metric in the number of states (Figure 27), model checking time (Figure 28), and number of BDD nodes (Figure 29). This is because each additional pipeline stage adds more pipeline registers and control logic. Furthermore, the finer partitioned PstageLogic blocks are abstracted by more UFs.

The more complicated hazard resolution schemes (i.e., forwarding and speculative execution) increase the number of state variables only slightly (Figure 27) but increase the model checking time significantly (Figure 28). This is because forwarding and speculative execution logic only requires a few additional state variables but introduce many more possible state transitions (e.g., all the data forwarding scenarios, pipeline squash conditions due to misprediction, etc). Such an increase in the exploration space also leads to higher number of BDD nodes used (Figure 29). The choice of hazard resolution does not affect the number of correctness properties (Figure 26), since it only impacts the hazard control logic, and does not introduce any additional next-state compute blocks or architectural states, which are the main concerns of the correctness properties placed by T-piper.

Finally, somewhat surprisingly, the MIPS and the CISC-like memory-memory processor pipelines incur a similar level of verification effort. One might expect that a memory-memory pipeline would require a higher effort due to the additional logic to manage data memory hazards since instructions can read from and write to the memory in a single pipeline pass. This is not the case because we assume a simple memory array

that reads combinationally and writes synchronously just like a register file array. Once abstracted, the larger capacity of the memory does not increase verification effort beyond what would be required to handle a register file array. Similarly, the extra complexity from the additional memory address compute block and the more sophisticated next PC compute block are also abstracted away as uninterpreted functions and hence do not have a large impact on the verification cost. Verification cost is much more affected by the complexity of the pipeline control logic injected by T-piper.

Catching Real Bugs. In the course of the case studies, we did in fact uncover a number of real "programming" bugs in T-piper and fortunately no "algorithmic" bugs so far (see Section 5.2.1). Two examples of the bugs are described below:

1) T-piper requires that all predictions be resolved before any architectural state is written. We had a bug in our hazard configuration file (H-cfg) where a prediction resolution point is placed later than a state write point. Further, we had not included a check for such a condition in T-piper. Thus, T-piper ended up taking the buggy H-cfg and synthesizing a pipeline. When we ran the verification, a counter example was generated for the scenario where an architectural state is written even though there is an unresolved incorrect misprediction.

2) Another bug results in a failed verification due to T-piper generating a (slightly) incorrect SMV model, whereas the Verilog RTL emitted was correct (upon later manual examination). In this case, model checking successfully produced a counter example to help us pinpoint the typo in the SMV model and track down a programming bug in T-piper. However, one can certainly also imagine an

opposite scenario where the typo is in the Verilog RTL and not in the SMV model. This speaks strongly for formal verification technologies that can be applied directly to the implementation design file. (But then again, we will still be at the mercy of the correctness of the model checker and the entire downstream synthesis flow.)

Chapter 6

Multithreaded Pipeline Synthesis

Multithreading is a microarchitecture optimization technique that allows multiple threads of execution to share a pipeline, thereby improving efficiency. Although multithreading can be applied to any pipelined datapath, the most common adoption of this technique has been for instruction processor pipelines. Various commercial processor pipelines are multithreaded, such as Intel® Atom and Sun® Niagara.

Developing a non-threaded pipeline by hand is already a difficult effort by itself, let alone with the complication of multithreading. There are many additional aspects to consider (e.g., thread scheduling policy, state sharing attributes among threads, throughput enhancing schemes on long-latency events) which exacerbate the pipeline development effort. While there are existing works on automatic synthesis of in-order pipelines [9][19][25][27][28][30][33][39][47][58][71], to the best of our knowledge there has not been any for synthesis of in-order multithreaded pipelines. Prior works [7][13][34][35][44] have also presented multithreaded processor pipelines for FPGA prototyping, but they are manually developed.

In this section, we present extensions to the transactional datapath specification (T-spec) and its in-order pipeline synthesis technology (T-piper) previously discussed in

Chapter 2 and Chapter 3 to support multithreading. Our approach not only works well with instruction processor pipelines but also is flexible enough to accept any sequential datapath. It maintains the synthesis features of T-piper for non-threaded pipelines (e.g., forwarding, speculation) while supporting various multithreading features, consisting of those found in modern in-order multithreaded pipelines (e.g., state sharing, replay on long-latency events) as well as novel ones (e.g., state sharing by thread groups).

To demonstrate the usefulness of the approach, we report a case study at the end of this section, which uses multithreading-capable T-spec and T-piper, on rapid design space exploration of 32 multithreaded processor pipelines supporting a subset of x86 ISA. The pipelines are all synthesized from a single T-spec, and they vary in pipeline depths, forwarding capabilities, thread scheduling policies, and mechanisms for handling long-latency events.

The chapter is organized as follows. Section 6.1 presents a motivating example to be used for discussion in the later sections. Section 6.2 discusses extensions for T-spec and T-piper to support multithreading. Section 6.3, presents a design exploration study of x86 processor pipelines utilizing the extended T-spec and T-piper.

6.1. Motivating Example: Key Scan

To illustrate pipelining and multithreading usage scenarios to be discussed in this chapter, here we present a simple example of a key scanner that counts the number of occurrences of a given 32-bit key value in an array of words in memory. Figure 30(a) shows an example, where the key K is 7, and 8 words are in the memory M. Count CNT should be 3 at the end of the scan.

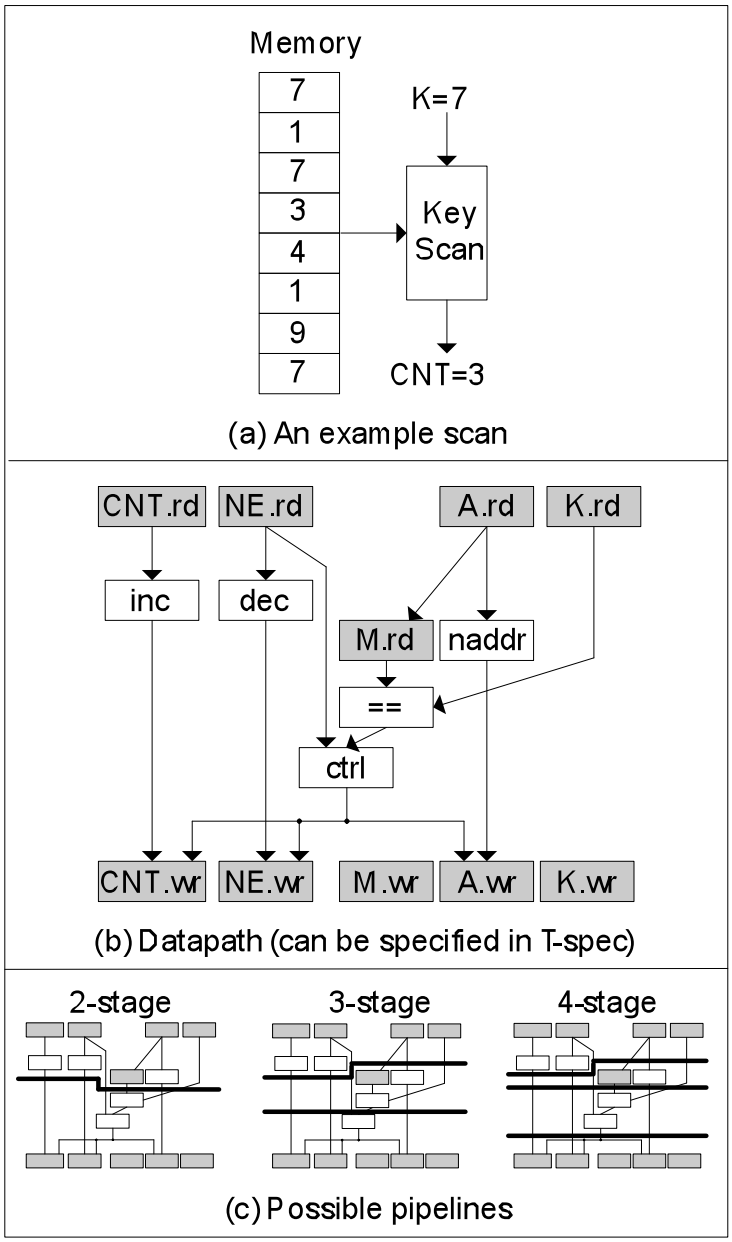


Figure 30. Key scan example.

Figure 30(b) depicts a sequential datapath for such a key scanner, which consists of state elements (registers and a memory, shown in shaded boxes) and combinational logic blocks (white boxes) that compute next-state values for each state within a clock cycle.

Note that the states are drawn with separate read and write interfaces, illustrating the read-compute-write cycle that happens in the datapath within each clock cycle.

The datapath operates as follows. The memory M contains an array of words to be scanned, with NE initially holding the number of words in M (e.g., 8 for Figure 30(a) example). The register K holds the keyword. Every clock cycle, the word in M pointed to by the address A is read and compared with keyword K. If there is a match, then count CNT is incremented by inc. Also, A is updated by naddr to point to the next word in M, and NE decremented by dec. When NE reaches 0, the scan is completed. The state updates are managed by ctrl, which monitors NE to check for scan completion (NE is 0), and the K and M.rd comparison result to check for when a K is found in M.

We can pipeline the datapath in Figure 30(b) using the T-piper in-order pipeline synthesis in Chapter 3 to reduce critical paths and improve frequency, by dividing the next-state logic blocks into multiple stages separated by pipeline registers. For example, Figure 30(c) shows three possible pipeline implementations of the datapath in Figure 30(b).

6.2. Multithreaded Pipeline Design

Multithreading is a microarchitecture optimization technique that allows multiple threads of execution to share a single pipeline. Each thread of execution is associated with a set of states and a sequence of transformations on those states. Adding multithreading to a non-threaded pipeline typically requires the following logic. First, architectural state elements need to be replicated to hold multiple contexts. Second, logic

for scheduling and managing the threads need to be added. The rest of the non-threaded pipeline resources can be shared in a time-multiplexed manner by all the threads.

There are two main benefits of multithreading. First, it saves area, at the expense of performance, relative to having multiple full pipelines to execute multiple threads. Second, when a thread experiences a long stall (e.g., due to data dependence, or long-latency event like a memory access), it may be possible to let other threads to proceed, thereby improving pipeline utilization.

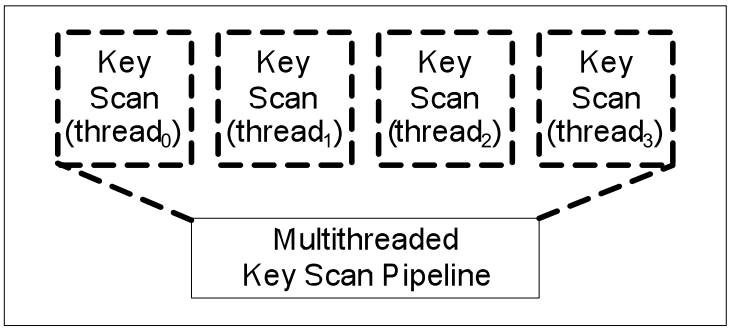


Figure 31. Multithreaded key scan.

6.2.1. Multithreading the Key Scan Example

Let us suppose that we would like to improve the example datapath in Figure 30(b) by pipelining and multithreading that supports 4 threads, as illustrate in Figure 31. There are multiple possible multithreading scenarios that can be employed, three of which are shown in Figure 32. First, each thread can be used to perform an individual scan, of which case the multithreaded key scanner will accept and return 4 different keywords and counts, respectively (Figure 32(a)). Second, only 1 scan is performed, but accelerated by having the 4 threads scanning different parts of the memory (Figure 32(b)). Lastly, the first and second scenarios can be combined, where there are two scans, each one

performed by two threads (Figure 32(c)). To facilitate these scenarios, the way threads access states have to be adjusted appropriately. In the first scenario, K and CNT support 4 contexts, each privately accessed by a thread. In the second scenario, they support only a single context that is accessible by all threads. In the last scenario, they support 2 contexts, each of which is shared by two threads.

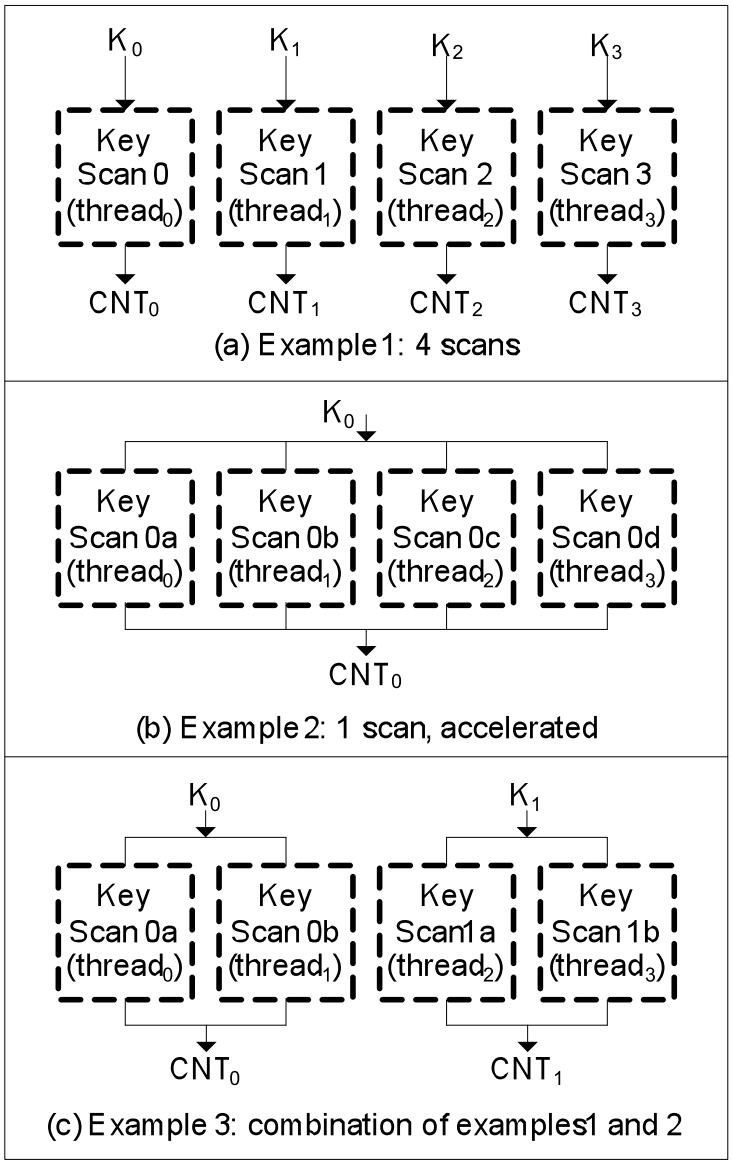


Figure 32. Examples of multithreading configurations applicable to key scan.

Another aspect of multithreading to consider is thread scheduling, which decides on which thread gets to use the pipeline at a given time. For our key scan, we may want to add an architectural state (e.g., STATUS) and the logic to set it to indicate if a thread is active (i.e., is scanning) or inactive (not scanning). A thread scheduler then monitors STATUS and skips inactive threads. Furthermore, suppose that the implementation of the memory M utilizes caches to improve overall latency (i.e., specified using a MC handshake interface mentioned in Chapter 3), such that an access to it may happen right away (cache hit) or after multiple clock cycles (cache miss). In this case, we may want to allow a thread suffering from a cache miss to be replayed at a later time while allowing other threads to continue to execute, so that the cache miss does not block all the threads from progressing.

Our synthesis supports all the multithreading features discussed in this key scanner example, and more. The next section summarizes the various multithreading features we support. Following that we present the details of the extensions we propose to T-spec and T-piper to support these multithreading features.

6.2.2. Multithreading Features

Thread Scheduling. A thread scheduler selects the thread that should be allowed to use the pipeline at a given time. The two most common scheduling policy for in-order multithreaded pipelines are interleaved multithreading (IMT) and block multithreading (BMT) [67].

In IMT, a thread switch happens in a fine-grained manner, whenever the first pipeline stage becomes available. The next thread to enter the pipeline is typically selected based

on a round-robin policy. The main benefit of IMT is the potential simplification that can be made to the hazard management logic, since it may be possible to guarantee that each stage in the pipeline is occupied by a different thread, making it impossible for certain data hazards to happen.

In BMT, a thread executes successively until a particular event occurs in the pipeline, which triggers a context switch to a new thread. The main benefit of BMT is the ability to deliver a good single-thread performance because BMT lets a thread to execute continuously, obtaining full access to the pipeline for a certain time period, before switching to another. However, continuous execution requires full hazard management logic, making it impossible to perform any simplification as in the case of IMT. An example for thread switch triggering event in BMT in the case of instruction processor is when a thread enters a critical section, which would need to be executed as fast as possible [34].

The simpler IMT policy is supported by default by our synthesis system. Furthermore, we also support custom-made thread scheduler by using a well-defined thread scheduler interface, which can be used to implement BMT policy, critical section acceleration, and other custom-designed scheduling policies.

Dealing with Long-Latency Events. When a thread encounters a long-latency event (e.g., a cache miss), it is often useful to allow other threads to proceed. This way, the stall experienced by one thread can be hidden by the execution of other threads.

Our synthesis system supports the recently proposed approach to deal with long-latency events based on replay [34][35]. The idea is to allow a pipeline stage to request a

replay when it suffers from a long-latency event. Upon replay, the thread in that stage is canceled and re-executed at a later time. Meanwhile, other threads can use the pipeline and proceed with their execution.

A known shortcoming of replay [35] is that it may lead to a live-lock when the service for a long-latency event for a thread that requested a replay keeps being cancelled by the service of another long-latency event for another thread that also requested a replay (e.g., conflicting cache misses where two cache line requests evict one another). To prevent this, we support a mechanism to turn off replay capability dynamically to guarantee forward progress.

State Sharing Attributes. There are a few possible attributes that an architectural state can have with respect to the way threads access the state. First, a private state has multiple contexts, each accessible only by a thread (e.g., K and CNT in Figure 32(a)). Second, a global state is shared by all the threads, and it has only one context accessible by any thread (e.g., K and CNT in Figure 32(b)). Third, a group state has multiple contexts, each accessible only by a set of threads (e.g., K and CNT in Figure 32(c)).

Our system supports all these attributes, allowing for generic sharing, where sharing can be applied to any arbitrary state and group of threads. Note for instruction processors, the most common attributes are private (e.g., PC, RF) and global (e.g., memory). Thus, a group state is a new kind of attribute enabled by our synthesis technology.

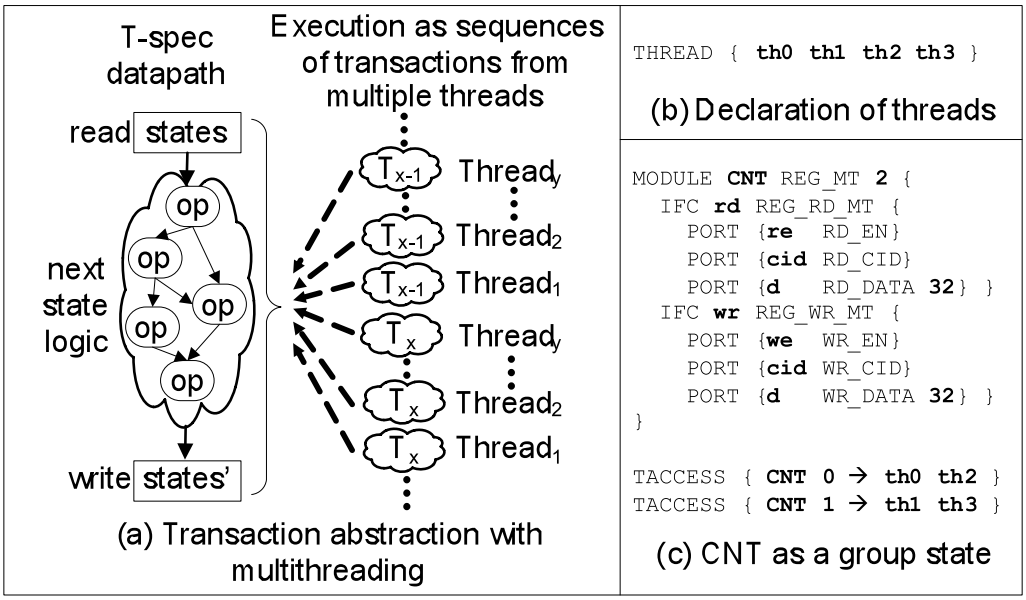


Figure 33. Extending T-spec for multithreading.

6.3. T-spec for Multithreading

6.3.1. Extending the Transaction Abstraction

We extend the transaction abstraction captured by T-spec (as previously explained in Section 3) to support multithreading. Figure 33(a) provides an illustration, where there exist multiple sequences of transactions, each belongs to a thread of execution

The original transaction abstraction semantics is still preserved, where the datapath executes one transaction at a time, and each transaction reads the state values left by the preceding transaction and computes a new set of state values to be seen by the next transaction. Except now each transaction is also associated with a thread (e.g., Thread_1 to Thread_y in Figure 33(a)), and a state may have multiple contexts.

A thread is associated with a transaction sequence (e.g., T_x, T_{x-1}, and so on in Figure 33(a)), which corresponds to the original sequence of execution of the thread in a non-

threaded system (i.e., correspond to the program order in the case of instruction processor, where a transaction is equivalent to an instruction).

A thread is also associated with state sharing attributes (see section 6.2.2), indicating, for each state, which context it can access. For example, a state update made by a transaction from a thread can be read by a subsequent transaction from a different thread, if both threads access the same context of the state.

Finally, having multiple threads also raises a question on the orders of the thread execution that should be considered valid. Here, we consider any possible thread execution order to be valid (Figure 33(a) shows a round-robin order, but any order is valid).

6.3.2. T-spec Language Extensions

We incorporated the following extensions to T-spec to capture multithreaded datapaths based on the aforementioned abstraction.

- First, we add a way to declare threads. Figure 33(b) shows declarations of 4 threads (th0, th1, th2, th3) in our key scan example. Each declared thread will be assigned a unique thread ID (TID) by T-piper during synthesis.

- Second, since a state now can contain multiple contexts, the state-read and state-write interfaces are extended with an additional input, context ID (CID), to indicate the context to access. T-piper will automatically synthesize logic that drives this input. Figure 33(c) shows the declaration for 2-context CNT in the key scan example. Note

that any multi-context state element implementation can be used, as long as it has appropriate read and write interfaces.

- Third, we add a way to specify state sharing attributes. Figure 33(c) shows an example of making CNT a group state, where context 0 is accessible only by th0 and th2, and context 1 by th1 and th3. T-spec can specify accessibility to any state context by arbitrary set of threads, so it can also specify private (a set of one thread) and global (a set of all the threads) states.

- Fourth, we add a module type to specify a custom thread scheduler implementation. T-piper synthesis will place the thread scheduler module at the first stage of the pipeline, so it can select the next thread to enter the pipeline, as illustrated in Figure 34. The figure also shows the interface to the thread scheduler module, which includes mandatory inputs and outputs (i.e., en, TID, no_replay, replay_req, and replay_tid) as well as any arbitrary inputs from interfaces specified in T-spec that are assigned to the first pipeline stage in P-cfg.

- Finally, the handshake interface of a multi-cycle block is extended with a replay_req output, and a no_replay input, to make it replay-capable. When the block needs multiple clocks to complete, it can ask for a replay by asserting its replay_req output unless its no_replay input is not asserted.

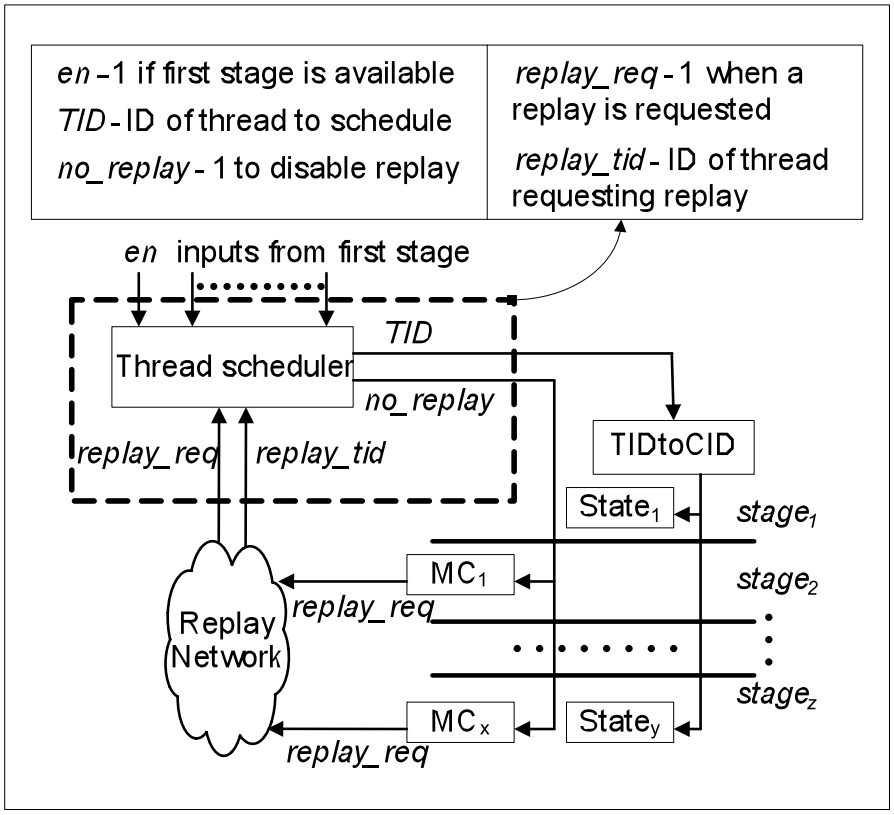


Figure 34. Multithreading support logic.

The thread scheduler interface works as follows. When the first stage becomes available, en is asserted indicating that scheduling decision is needed. At this point, TID outputs the decision on the thread that should enter first stage. A replay can be requested whenever a thread in the pipeline encounters a long-latency event performed by a multi-cycle block (i.e., MCs in Figure 34) through the block's handshake interface. T-piper synthesis connects all replay-capable multi-cycle blocks to the replay network, which will assert replay_req input of the thread scheduler when a replay occurs in the pipeline and supply the TID of the thread requesting a replay to the replay_tid input. These inputs can be used in thread scheduling decision (e.g., the thread requesting a replay do not get re-scheduled right away). To ensure forward progress in the case of a live-lock, the thread

scheduler can assert its no_replay output whenever necessary (e.g., assert periodically to guarantee overall forward progress). Lastly, the scheduler interface can also be connected to any arbitrary inputs from the first pipeline stage. An example usage is to have the scheduler in our key scan example to monitor STATUS, and de-schedule any thread that is inactive.

The thread scheduler module is allowed to contain persistent states to help perform scheduling functions. For example, to implement a BMT scheduling, the thread scheduler may maintain an internal counter, that is incremented each time its en input is asserted. When the counter saturates, a thread switch is triggered.

The aforementioned thread scheduler specification strategy allows for flexibility, since any thread scheduler implementation can be used, as long it implements the appropriate interface.

6.4. Pipeline Synthesis Details

6.4.1. Multithreaded Data Hazard Management

In a non-threaded pipeline implementation, T-piper synthesizes hazard management logic for each state in T-spec to ensure Read-After-Write (RAW) hazards are detected and resolved accordingly (as explained in Chapter 3). To support hazard management in multithreaded pipeline, we extend such hazard management logic with a context ID (CID) check logic, which ensures that if there is a RAW hazard on a state, the hazard targets the same context of that state.

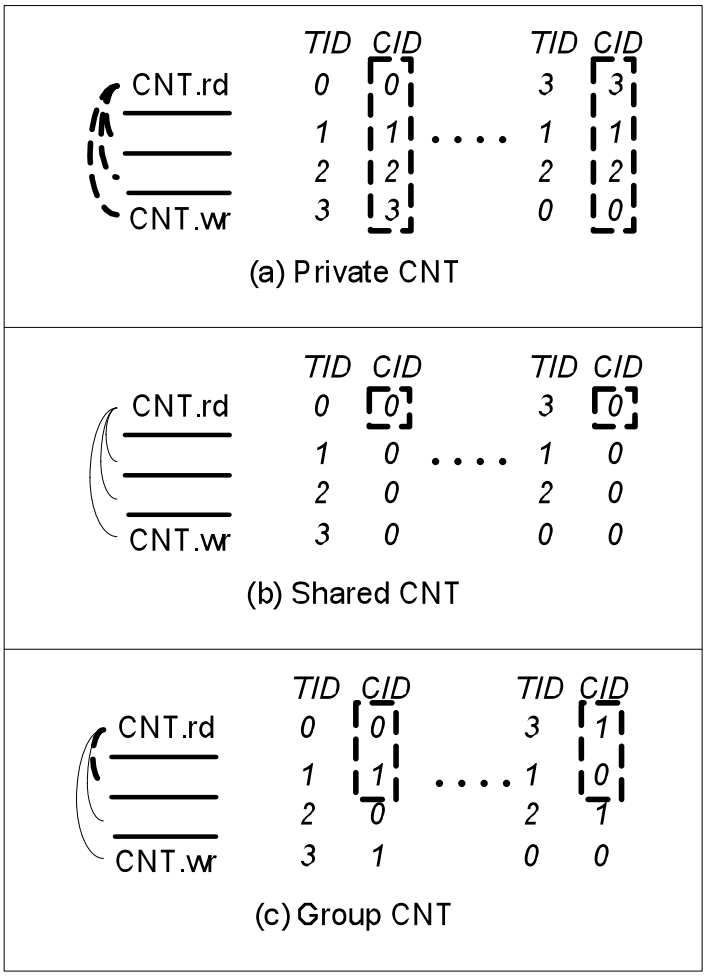


Figure 35. Interleaved multithreading hazard management logic simplification.

T-spec has already provided the information on how a thread accesses each state element (e.g. Figure 33(c)). T-piper uses this information to synthesize the logic that translates a thread ID (TID) to a CID for each state element and propagates the CID along the pipeline (Figure 34). The CID is used to access the appropriate context during a state access, and is incorporated to the hazard management logic, as mentioned above.

Beyond this baseline multithreaded hazard management logic, two types of simplifications can be made. First, for a global state, the CID check is always true. So, the logic can be optimized away.

Second, if the default round-robin IMT scheduler is used, it may be possible to make simplifications since certain hazards could never happen. Figure 35 provides an example IMT hazard logic simplifications to the CNT architectural state for the key scan implementation scenarios shown in Figure 32, assuming a 4-stage pipeline target where CNT is read and written back in the first and last stage, respectively. The lines on the left side of the pipeline show all possible hazards in the non-threaded pipeline, with the dashed lines showing the hazards that can never happen given IMT scheduling and the CNT sharing attribute.

To do this, T-piper enumerates all possible thread orders in the pipeline, given IMT scheduling. Next, for each thread order, it translates the TID of the thread occupying the stage to its CID, for each state element. Figure 35 shows the enumerated TIDs and the associated CIDs for CNT in the key scan example. Note that the enumerations deliberately do not consider stalls in the pipeline, so they represent the most aggressive schedule that could happen. From the enumerations, T-piper determines the minimum distance for which hazards can happen. For example, with private CNT, hazards cannot happen within 4 stages away from the state-read interface of CNT (i.e., in first stage). Since there are only 4 stages in the pipeline, the hazard management logic can be eliminated entirely. For global CNT, the minimum distance is 1. So, no simplifications can be made, since hazard can happen between the stage where the state-read is and any of the later stages. Lastly, for group CNT, the distance is 2. Simplification can be made here, since hazard can never happen between stage 1 and 2.

6.4.2. Thread Scheduler and Replay Support

The synthesis process integrates the thread scheduler (either the default IMT or a custom-defined one) to the pipeline by connecting its inputs and outputs to the appropriate pipeline control signals, as illustrated in Figure 34.

The en input of the scheduler is connected to the control signal of the first stage that indicates when a new transaction should enter the pipeline. The TID output is connected to the logic that translates TID to CID (i.e., TIDtoCID). The synthesis also creates the logic for propagating the CID through the rest of the pipeline, as well as connecting the CID to appropriate state access interfaces.

For replay, the replay_en and replay_tid inputs are connected to the network of replay signals from all the replay-capable MC blocks in the pipeline. This replay network is also automatically synthesized. It is a simple logic that detects a reply request, and selects one from the latest stage in case of simultaneous multiple replay requests. It outputs the TID of the thread requesting replay (replay_tid) and asserts the replay_en to indicate to the thread scheduler that a replay is requested. Lastly, the no_replay output of the scheduler is propagated through the pipeline and is connected to each replay-capable MC block, indicating to the block when a replay is prohibited.

6.5. A Case Study with x86 Pipelines

The key scan example discussed previously is very simple and was intended for illustration purposes only. In this section, we present a non-trivial case study on the design space exploration of various multithreaded x86 processor pipelines.

The pipelines are all synthesized from a single x86-subset T-spec using T-piper within minutes. The x86 T-spec is based on the case study presented in Chapter 4, and is extended to support 4 threads, a shared memory, and private architectural states. The memory uses a cache with a hit serviced right away and a miss serviced in 10 implementation clock cycles. The T-spec also includes a custom non-x86 1-bit private state (STATUS) and a custom instruction to set the state to indicate whether a thread is active (running a benchmark) or inactive (finished running, in idle loop). Each pipeline is evaluated by running a mix of 4 benchmarks (DES, Quant, VLC, Bitcount) from [71]. The benchmarks are of different lengths, each running as its own thread. At the end of each benchmark, the custom instruction is used to set the custom state to inactive. Then, the benchmark enters an idle loop. Thus, overall execution is completed when all threads have set their STATUS to inactive.

We evaluated a total of 32 pipelines with T-piper, varying the following parameters:

- pipeline depths varied from 4 to 7 stages
- with (F) and without (NF) inclusion of maximal forwarding
- with (P) and without (NP) inclusion of a thread scheduler that prioritizes for active threads by monitoring STATUS, on top of a round-robin IMT policy
- with (R) and without (NR) the capability to replay on a cache miss

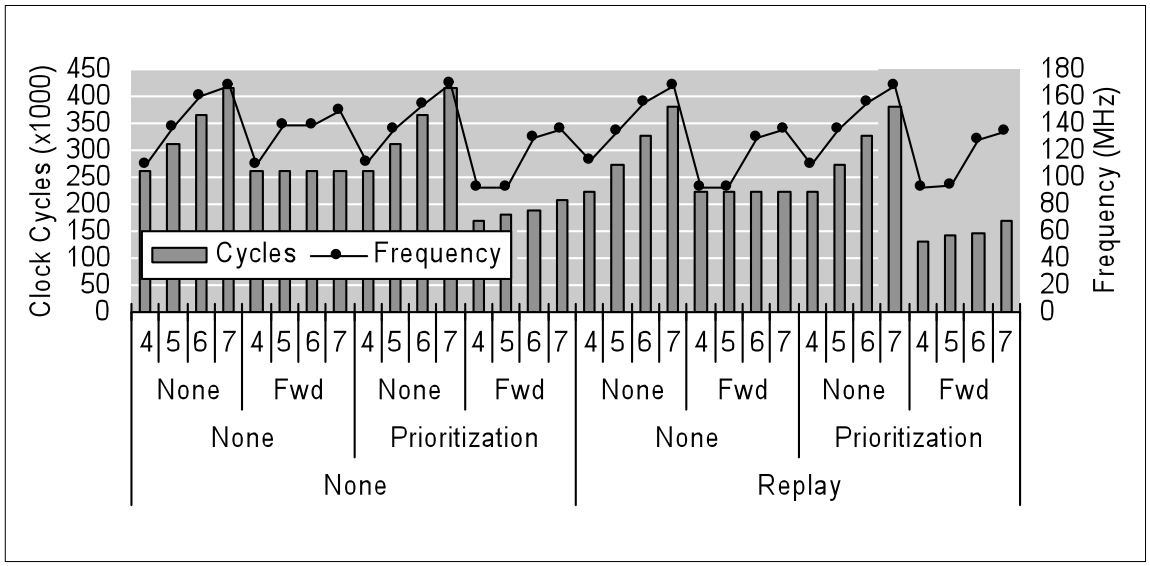


Figure 36. Cycle count and frequency of each multithreaded pipeline under study.

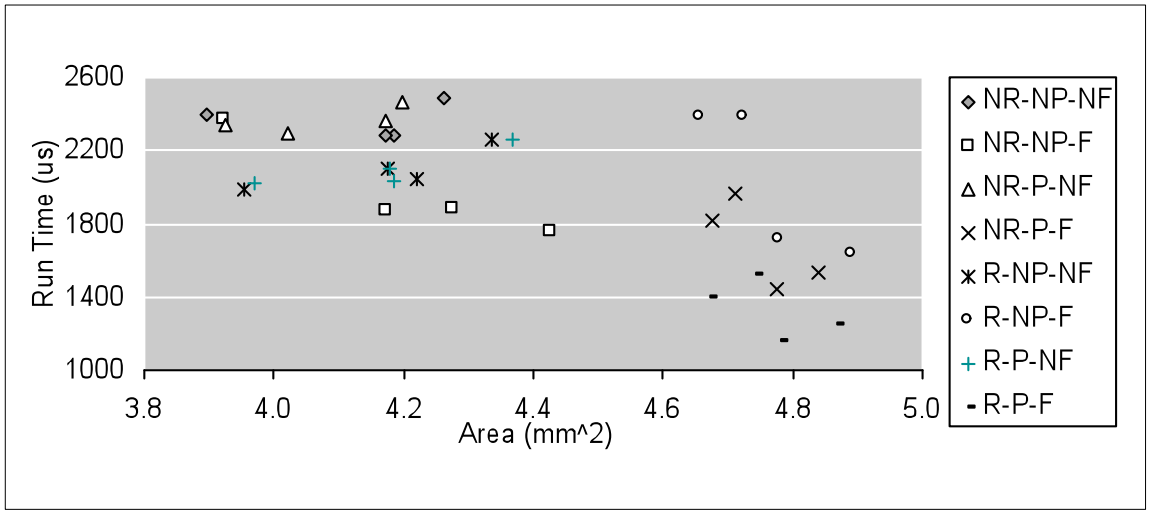


Figure 37. Cost-performance tradeoff for the multitheaded pipelines under study.

Figure 36 shows the cycle count from RTL simulation of each pipeline. Forwarding improves cycle count since stalls due to RAW hazards are reduced, allowing the pipeline to host multiple instructions from the same thread. Thread scheduler that skips inactive threads also helps cycle counts since completed shorter-running benchmarks that are in

idle loop are no longer scheduled to use the pipeline, thus accelerating forward progress of the longer-running still-active threads. Finally, replay improves cycle count since a cache miss does not block the entire pipeline.

We also synthesized the pipelines using Synopsys DC targeting a commercial 180nm standard-cell library. Figure 36 shows the implementation frequency for each pipeline. The improvement trend is generally the opposite of that of the cycle count because features that improve cycle count introduce additional implementation overheads that can result in reduced frequency. Notice also that deeper pipelines do help improve frequency.

Figure 37 shows the cost-performance tradeoff for the pipelines we studied. For each pipeline, the area is obtained from Synopsys, and the run-time is based on the RTL simulation cycle counts, adjusted by the implementation frequency. Notice that no single design parameter dominates the pipelines in the Pareto optimal design points (e.g., 2 points with all R, P and F optimizations; 2 with only F; 1 with only R; 1 with only P; and 1 without optimization). It would have been impossible to do such characterize such design points without exploring a large number of different designs at the RT level.

Previously in Chapter 4, we have shown that a non-threaded in-order pipeline generated by T-piper is comparable to a manually designed one via a case study with a MIPS pipeline. As a sanity check, we compared the simplest non-threaded pipeline (4-stage, without optimization) with its 4-threaded counterparts. The findings are as follows:

- Multithreading without any optimization (i.e., NR-NP-NF) incurs 26% area increase and 4% frequency decrease relative to the non-multithreaded version of the pipeline,

while shortening run-time only by 18%. This indicates that the only a modest additional amount of logic is needed to support multithreading.

- When all optimizations are considered (i.e., R-P-F), the area and frequency overheads are only 51% and 21%, respectively, while run-time improves by 2x. This illustrates the effectiveness of the multithreading optimizations synthesizable by T-piper, where the performance improvement achieved is significantly higher than the required implementation overheads needed to support them.

Chapter 7

Related Work

This chapter elaborates on the prior works that are relevant to this thesis. Specifically, the chapter discusses about the existing works on datapath specification techniques, automatic in-order pipeline synthesis and verification, and multithreaded in-order pipeline development.

7.1. Datapath Specification Techniques

Register Transfer Level (RTL) Descriptions. The most common datapath design practice is to create register transfer level (RTL) descriptions of the desired datapath using a standard Hardware Description Language (HDL) such as Verilog [66] and VHDL [65]. From such HDLs, commercial tools can be used to synthesize the datapath to specific implementation targets. Examples of such synthesis tools are Synopsis Design Compiler [63] for ASIC implementations and Xilinx ISE [69] for FPGA implementations. It is not possible to automatically pipeline an RTL design in general due to the low-level semantics of an RTL description. In contrast, T-spec captures the desired functionality of the datapath at a higher level transactional abstraction that makes automatic pipelining possible.

C and C++ Specifications. Several recent commercial and research efforts utilize software programming languages for high-level specification of hardware datapaths (e.g., [6][16][22][46][55][64][70]). In these works, functional and algorithmic specifications are expressed using a software programming language, such as C or C++. Optimizing compilers map the computations specified by the programs to equivalent RTL implementations. T-spec can complement these program-to-datapath frameworks by serving as an internal intermediate target for capturing first a non-pipelined implementation compiled from the program-level specifications. T-piper can next be used as a back-end to produce the final high-performance pipelined implementation or to explore the design space of pipeline configurations.

Operation-centric frameworks. An operation-centric description framework [21] describes concurrent hardware behaviors in terms of state transition rules that are guarded atomic actions. Each rule prescribes a set of state transformations that should be applied to the datapath atomically when the rule's guarding predicate is true. Synthesizing an implementation from a set of operation-centric rules involves creating a "merged" datapath that supports the execution of one or multiple non-conflicting rules in each clock cycle. This resulting merged datapath implementation of an operation-centric specification has a natural correspondence to a T-spec transactional execution system. Therefore, T-piper can be used as an optimizing backend to derive pipelined implementations that continue to obey the original operation-centric semantics.

Processor-specific Specifications. Prior works [12][14][18][25][30][53][56][58][71] have focused on processor-specific design specifications akin to ISAs. The specifications usually involve describing an ISA's architectural states and instructions. Such ISA

specifications map very well to operation-centric specifications discussed earlier; each instruction roughly corresponds to an atomic rule. Thus, an ISA specification, following a simple translation to an operation-centric language, could be subjected to the same synthesis flow we discussed in the last paragraph to produce an in-order pipelined processor implementation.

Other frameworks. Finally, there are specification frameworks that are focused on formal verification. For example, HAWK [40] can be used to specify pipelined processors. From a HAWK specification, equivalence can be shown between the pipelined processor in the specification against a non-pipelined version, which is derived using a sequence of formal microarchitectural transformations [41]. PVS [10] can also be used to specify pipelined processors for theorem proving purposes.

7.2. Automatic In-order Pipeline Synthesis

There are a few major differences between existing automated pipeline synthesis studies and this thesis work, which are explained below.

Pipelining for any arbitrary datapath. Many prior work in automatic pipelining focus specially on developing pipelined instruction set processors [25][28][58][71]. Although we use instruction-set processing as the case studies in this thesis, our work is applicable to pipelined datapath design in general. The systems presented in [19][27][39] do support automatic pipelining in general by applying transformations to RTL netlists. Because their algorithms must preserve the low-level semantics of the original RTL netlist, they are more limited in their opportunities for optimizations.

Manual forwarding path identification and placement. In particular, they do not automatically identify forwarding opportunities and place forwarding paths like T-piper. There are other automatic pipelining systems, that like T-piper, start from a high-level specification but nevertheless offer no support for forwarding [25] or still largely rely on the designer to manually identify and place the forwarding paths [28][58][71]. An exception is [39], which may be able to automatically identify forwarding opportunities, but no detailed explanation is provided in their paper.

Manual implementation of forwarding paths may still be quite manageable for the simple pipelines usually used as test cases for the abovementioned work. For example, the basic 5-stage textbook pipeline [20] used in [25][28][33][71] use only up to 4 forwarding paths (i.e., from MEM and WB stages to EXE stage, for both rt and rs operands). Interesting and practical designs are likely to contain many more forwarding opportunities to consider. For example, in our x86 case study, our system identified and placed up to 44 forwarding paths, which would have been very challenging to insert manually and make it prohibitive to explore more than a few design alternatives.

Generic speculation support. The use of value prediction has also been limited in prior automatic pipelining systems, some supporting only a very restricted form of prediction [19][25][28][33][39][47][58][71] or not at all [9]. Their support for prediction also still requires error-prone manual efforts in adding the logic for handling the predicted values and resolving the predictions. The PEAS-III processor-specific system [25] supports prediction automatically, but only for the special case of predict-not-taken branches (i.e. a default prediction of PC+4 always). Examples like [28][58] allow more general predictors but offer little in automation; they require the predictors to be designed

as low-level hand-written modules and requires manual introduction of the prediction and resolution logic in the pipeline. Lastly, examples like [19][33][39][47] allow speculation on any system state and offer some level of correctness guarantee through automation but do not permit the use of arbitrary user-defined custom predictors.

Our generic framework supports speculation on any system state and allows custom predictor logic to be expressed in the same high-level abstraction. Furthermore, we provide a stronger assurance that the incorporation of prediction will not affect correctness (will only affect performance) since (1) our approach never commits a predicted value (i.e., system states at any point in time always based on actual computed values), and (2) the prediction check and value management logic are automatically synthesized.

Demonstrated to work on a complex ISA. Another important distinction of our effort is the complexity of our test cases. Most prior works are demonstrated using only relatively simple pipelines, either pipelines for simple ISAs made specifically for their paper [27][39] or the textbook 5-stage RISC pipelines [28][33][71]. Among the examples we surveyed, only PEAS-III has been used to generate a (non-x86) CISC pipeline [30]. To the best of our knowledge, our work is the first to demonstrate automatic pipelining for an extensive CISC processor pipeline based on the x86 ISA. Although we only implemented a subset of the x86 ISA, the subset is quite extensive and able to run non-trivial benchmarks.

7.3. Automatic In-order Pipeline Verification

In terms of pipeline verification, prior works can be broadly classified in terms of simulation-based validation and formal verification approaches, which are discussed below.

Simulation-Based Validation. A common verification approach used in industry is 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 [1][2][68] have proposed generating these tests automatically to ensure adequate coverage.

Formal Verification. An alternative and promising approach to validation is formal verification. In this case, a formal procedure is employed to prove that certain properties of the design are correct for all possible execution paths. Thus, whenever possible, formal verification is preferable than validation since it can provide correctness guarantee for all possible behaviors.

There are two main approaches to formal verification, model checking and theorem proving. Theorem proving involves deriving mathematical description of the system and coming up with proofs to show that certain properties hold. Existing work has shown that it can be applied to pipeline verification [10][57]. The limitation here is the large manual effort that is needed in deriving the mathematical description, and defining and guiding the proofs.

On the other hand, model checking [8] can accept description of transition systems. It explores the state space of the given system to check that certain properties hold for all possible execution scenarios. Unfortunately, the amount of possible scenarios grows exponentially with the amount of storage elements in the design. Such state explosion problem limits the size of design that can be handled by model checking.

Compositional model checking [42] attempts to obtain the benefits of both worlds, by combining model checking and theorem proving. In this case, the state explosion is dealt with by spending manual effort to specify correctness properties to decompose the verification problem into sufficiently small sub-problems for model checking to handle. Cadence SMV [43] is a tool that provides compositional model checking technology. An existing study [26] has shown that even an out-of-order processor can be verified using SMV, when appropriate decompositions are applied.

Although compositional model checking is promising, the manual effort needed to do the verification set up procedure (e.g., developing verification models and the decomposition strategies for the design at hand) has been reported to be extremely challenging [37]. In this thesis work, we have developed an automatic approach to avoid the need for such challenging manual effort. Our approach automatically applies compositional model checking to verify that the pipeline logic generated by T-piper is functionally correct with respect to its T-spec datapath, under the transactional execution semantics.

7.4. Synthesis of In-order Pipelines with Multithreading

Although multithreading (MT) has been gaining in popularity and has been adopted in commercial pipelines, no existing automation system has yet provided the capability to automatically synthesize a multi-threaded pipeline from a high-level specification.

The only relevant work we could find is CUSTARD [11], which proposes a generic MT processor template that can be synthesized into an implementation by configuring certain parameters such as number of threads, threading type, etc. However, as with any other design templates, CUSTARD is quite restricted. For example, it has to abide to a 4-stage pipeline structure and a MIPS ISA baseline.

Other recent works [7][13][34][35][44] also presented design case studies of multithreaded processor pipelines for FPGA prototyping. However these pipelines are manually developed.

We believe our work is the first to fully automate the synthesis of multithreaded in-order pipelines from a non-pipelined datapath specification. Furthermore, it is very flexible. Not only it allows synthesis of instruction processors with multithreading features found in previously mentioned FPGA prototyping studies, but it also allows capturing larger design space of any sequential system datapath beyond instruction processor as well as enabling new multithreading features (e.g., states shared by a group of threads).

Chapter 8

Conclusion

This thesis develops a novel transactional specification framework (T-spec) to capture a non-pipelined datapath using the transactional abstraction. From a T-spec, any implementation can be synthesized, as long as the transactional abstraction semantics are preserved. This allows designers to focus on the datapath development at the transactional-level, relieving them from the burden of having to do the tedious and error-prone task of applying microarchitecture optimizations, such as pipelining, to the datapath by hand.

Based on T-spec, this thesis further investigates and proposes an automatic pipeline synthesis technology (T-piper) for in-order pipelines that support forwarding, speculation, and multi-cycle units. Its effectiveness has been evaluated by two design case studies, which demonstrates that: (1) a synthesized MIPS 5-stage pipeline is comparable in performance and area to a hand-made one, and (2) rapid design space exploration of various x86 processor pipelines is achievable. Furthermore, a version of T-piper along with example T-specs has been made available online at www.t-piper.net for academic and research usages.

Next, the thesis enhances T-piper with an automatic approach to verify that a pipeline synthesized by T-piper is functionally equivalent to its T-spec specification under the

transactional execution semantics. The approach utilizes compositional model checking, and automates the challenging task of developing verification models, applying appropriate abstractions, and determining the proof decomposition strategies. It is capable of capturing any bugs caused by T-spec and T-piper, including the fundamental bugs in the synthesis algorithms themselves as well as bugs due to programming mistakes. Its effectiveness has been demonstrated by a case study on automatic verification of various non-trivial pipelines synthesized by T-piper from a load-store processor datapath and a memory-memory processor datapath T-specs.

Then, the thesis presents extensions to T-spec to capture multiple threads of transactional execution, and T-piper to synthesize a multithreaded in-order pipelined implementation from such a T-spec. Not only that the extensions preserve the original non-threaded in-order pipeline features (e.g., forwarding, speculation), they also support various multithreading-specific features, consisting of those found in modern in-order multithreaded pipelines (e.g., interleaved thread scheduling, global state sharing, replay on long-latency events) as well as novel ones (e.g., custom thread scheduling, state sharing by thread groups). The effectiveness of these extensions has been demonstrated with a case study that evaluates multithreaded x86 processor pipelines with various multithreading optimizations.

Finally, please note that even though the case studies used in this thesis are for instruction processors, T-spec and T-piper can handle sequential datapath in general, since any datapath with state elements and next-state logic blocks can be captured using T-spec.

8.1. Future Work

The T-spec transactional datapath specification framework and the associated T-piper synthesis and verification technology developed in this thesis open up many possible future research possibilities, which are discussed below.

Automatic synthesis for other microarchitecture types. The T-piper pipeline synthesis technology presented in this thesis focuses only on the in-order pipelines, both non-threaded and with multithreading. It should be possible to extend the work to synthesize other types of microarchitectures. An example would be to synthesize superscalar out-of-order pipelines [60]. Another example would be to synthesize redundant pipelines, such as in [45], for reliable execution.

Automatic functional verification using other techniques. The automatic verification approach presented in Chapter 5, while very scalable in the context of formal verification, is still limited by the state explosion. Since T-spec and T-piper have the precise knowledge of the specification and the to-be-synthesized target implementation, it should be possible to automate functional verification using other techniques, such as automatic test case generation [1][2][68] and automatic insertions of correctness assertions (to be used with commercial assertion-based formal verification tools [4][54]).

Performance verification. The automatic verification approach presented in Chapter 5 addresses only the functional correctness of the T-piper synthesized pipeline. However, a correctly functioning pipeline may still not be "performing" as intended, for example, by stalling unnecessarily even though forwarding is available. Fortunately, unlike functional correctness, a well-hidden performance bug that can only be exercised by a

rare corner case in fact does not impact performance much (in accordance to Amdahl's law), such that performance validation can be well served by a conventional simulation-based testing. T-piper already synthesizes RTL Verilog that can be used for simulation.

Furthermore, since T-piper does generate SMV models. Designers can specify temporal properties to define performance correctness to be check formally. (For example, if data forwarding condition is true, the pipeline should not stall.) We are also investigating extending T-piper's front-end to automatically derive performance correctness properties based on the given T-spec, pipeline configuration (P-cfg) and hazard configuration (H-cfg), and include them in the verification models generated by the T-piper back-end. This allows the verification of the performance of the final pipelined implementation synthesized by the T-piper back-end against the performance expected by the T-piper front-end. (This does not help, however, if the bug is in the front-end itself.)

Automatic design space exploration. T-piper offers new "tunable" parameters for designers to choose from, such as the pipeline boundary placements, forwarding and speculation schemes, and multithreading optimizations. These parameters capture a very large design space that is impossible to evaluate manually. Thus, there is a need for automatic design space exploration system for these parameters. In terms of pipeline boundary placement and forwarding path selection, there are existing studies (e.g., [39][62]) that have investigated automatic approach for it. However, no existing work exists in the automatic design space exploration of the speculation- and multithreading-related parameters. Furthermore, the exploration system should consider the global

optimization of these various parameters, instead of just optimizing them in isolation, which makes the problem even more challenging.

T-spec and T-piper as the back-end to an even higher-level design system. Even though the transactional abstraction captured by T-spec is already higher than the RTL, there is still opportunity to utilize T-spec and T-piper as the back-end to an even higher-level design system. Such a system would output a datapath described in T-spec, instead of the final RTL description. Thus, the microarchitectural optimizations can be left for T-piper to do. This approach would allow for decoupling of the datapath synthesis, and the synthesis of the microarchitecture.

There are already several existing high-level design systems that are prime candidates for integration with T-spec and T-piper, such as those systems that start from an Instruction Set Architecture specification [12][14][18][25][30][53][56][58][71], a C/C++ description [6][16][22][46][55][64][70], or an operation-centric description [21]. These systems have been previously mentioned in the discussion on relevant works in Chapter 7. Alternatively, a new type of a high-level design system may be devised in-order to utilize the full capabilities of T-spec and T-piper at its back-end.

Usage of T-spec and T-piper in other design case studies. First, in-terms of processor design, the processor case studies presented in this thesis still consider only subsets of their target ISAs. Even though these ISA subsets are not small, there are still certain ISA behaviors, such as exceptions and interrupts, which are not yet considered. These behaviors can be complicated. Thus, for future work, it would be interesting to

perform a case study that considers the entire suite of possible ISA behaviors (even exceptions and interrupts).

Second, the case studies presented in this thesis target only processor datapaths. However, as mentioned previously, T-spec and T-piper can accept any sequential datapath, and are not limited to just processors. Therefore, it would be interesting to see more case studies using T-spec and T-piper that target non-processor datapaths.

References

[1] A. Aharon et al., “Test Program Generation for Functional Verification of PowerPC Processors in IBM”, Design Automation Conference, 1995.

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

[3] D. Bhandarkar and D. Clark, “Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization”, International Conference on Architectural Support for Programming Languages and Operating Systems, 1991.

[4] Cadence Design Systems. Inc, “Assertion-based verification flow”, http://www.cadence.com/products/fv/pages/abv_flow.aspx, 2010.

[5] F. Campi, R. Canegallo, and R. Guerrieri, “IP-reusable 32-bit VLIW RISC core”, European Solid-State Circuits Conference, 2001.

[6] Catapult C Synthesis. http://www.mentor.com/products

[7] E. S. Chung, E. Nurvitadhi, J. C. Hoe, B. Falsafi, K. Mai, “A Complexity-Effective Architecture for Accelerating Full-System Multiprocessor Simulations Using FPGAs”, International Symposium on Field-Programmable Gate Arrays, 2008.

[8] E. Clarke, O. Grumberg, and D. A. Peled, “Model Checking”, The MIT Press, 2000.

[9] J. Cortadella, M. Kishinevsky, and B. Grundmann, “Synthesis of Synchronous Elastic Architectures”, Design Automation Conference, 2006.

[10] D. Cyrluk, "Microprocessor Verification in PVS: A Methodology and Simple Example," Technical Report SRI-CSL-93-12, SRI Computer Science Laboratory, 1993.

[11] R. Dimond, O. Mencer, and W. Luk, “Application-Specific Customization of Multi-Threaded Soft Processors”, Field Programmable Logic and Applications, 2006.

[12] A. Fauth, J. Van Praet, and M. Freericks, “Describing instruction set processors using nML”, European Design and Test Conference, 1995.

[13] B. Fort, D. Capalija, Z. G. Vranesic, S. D. Brown, “A Multithreaded Soft Processor for SoPC Area Reduction”, Field-Programmable Custom Computing Machines, 2006.

[14] R. E. Gonzales, “Xtensa: A Configurable and Extensible Processor”, IEEE Micro, vol. 20, 2000.

[15] J. Gray and A. Reuter, “Transaction Processing: Concepts and Techniques”, Morgan Kaufmann, 1993.

[16] S. Gupta, R. K. Gupta, N. D. Dutt, A. Nicolau, “SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits”, Kluwer Academic Publishers, 2004.

[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. and B. Brown, “MiBench: A Free, Commercially Representative Embedded Benchmark Suite”, Workshop on Workload Characterization, 2001.

[18] G. Hadjiyiannis, S. Hanono, S. Devadas, “ISDL: An Instruction Set Description Language for Retargetability”, Design Automation Conference, 1997.

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

[20] J. Hennesy and D. Patterson, “Computer Architecture: a Quantitative Approach”, Morgan Kauffmann, 1990.

[21] J C. Hoe. “Operation-Centric Hardware Description and Synthesis”, PhD Thesis, MIT, June 2000.

[22] Impulse C Website. http://www.impulseaccelerated.com/

[23] Intel Corp., “Intel ® 64 and IA-32 Architectures Software Developer’s Manuals”, http://www.intel.com/products/processor/manuals/

[24] Intel Corp., “Intel® Atom™ Processor: Intel’s Smallest Chip”, http://www.intel.com/technology/atom/

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

[26] R. Jhala, K. L. McMillan, “Microarchitecture Verification by Compositional Model Checking”, Conference on Computer Aided Verification, 2001.

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

[28] A. Kejariwal, P. Mishra, and N. Dutt, “Synthesis-driven Exploration of Pipelined Embedded Processors”, VLSI Design 17th International Conference, 2004.

[29] R. E. Kessler, “The Alpha 21264 Microprocessor”, IEEE Micro, Vol. 19, Issue 2, 1999.

[30] A. Kitajima, T. Sasaki, Y. Takeuchi, and M. Imai, “Design of Application Specific CISC Using PEAS-III”, International Workshop on Rapid System Prototyping, 2002.

[31] P. M. Kogge, “The Architecture of Pipelined Computers”, Taylor & Francis, 1981.

[32] P. Kongetira, K. Aingaran, and K. Olukotun, “Niagara: A 32-Way Multithreaded SPARC Processor”, IEEE Micro Magazine, 2005.

[33] D. Kroening and W. Paul, “Automated Pipeline Design”, Design Automation Conference, 2001.

[34] M. Labrecque, J. G. Steffan, “Fast critical sections via thread scheduling for FPGA-based multithreaded processors”, Field-Programmable Logic and Applications, 2009.

[35] M. Labrecque, P. Yiannacouras, J. G. Steffan, “Scaling Soft Processor Systems”, Field-Programmable Custom Computing Machines, 2008.

[36] J. Larus and R. Rajwar, “Transactional Memory (Synthesis Lectures on Computer Architecture)”, Morgan & Claypool Publishers, 2007.

[37] A. Lungu and D. J. Sorin. "Verification-Aware Microprocessor Design." International Conference on Parallel Architectures and Compilation Techniques, 2007.

[38] P. Magnusson et al., “Simics: A Full System Simulation Platform”, Computer, vol. 35, no. 2, 2002.

[39] M. V. Marinescu and M. Rinard, “High-level Automatic Pipelining for Sequential Circuits”, International Symposium on Systems Synthesis, 2001.

[40] J. Matthews, J. Launchbury, and B. Cook, “Microprocessor specification in HAWK”, International Conference on Computer Languages, 1998.

[41] J. Matthews and J. Launchbury, “Elementary Microarchitecture Algebra”, Computer-Aided Verification, 1999.

[42] K. L. McMillan, “A Methodology for Hardware Verification Using Compositional Model Checking”. Science of Computer Programming, Vol. 37, Issue 1-3, 2000.

[43] K. L. McMillan. “Getting Started with SMV”, Cadence Berkeley Laboratories, 2001.

[44] R. Moussali, N. Ghanem, M. A. R. Saghir, “Supporting Multithreading in Configurable Soft Processor Cores”, International Conference on Compilers, Architecture and Synthesis for Embedded Systems, 2007.

[45] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, “Detailed Design and Evaluation of Redundant Multithreading Alternatives”, ISCA 2002.

[46] NISC Technology. http://www.ics.uci.edu/~nisc/

[47] E. Nurvitadhi, J. C. Hoe, T. Kam, S. L. Lu, “Automatic Pipelining from Transactional Datapath Specifications”, Design Automation and Test in Europe (DATE), 2010.

[48] 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 (TCAD), to appear.

[49] E. Nurvitadhi, J. C. Hoe, T. Kam, S. L. Lu, “Integrating Formal Verification with High-Level Processor Development Framework”, submitted for publication.

[50] E. Nurvitadhi, J. C. Hoe, S. L. Lu, T. Kam, "Automatic Multithreaded Pipeline Synthesis from Transactional Datapath Specifications", Design Automation Conference (DAC), 2010.

[51] E. Nurvitadhi, J. C. Hoe, S. L. Lu, T. Kam, "Reducing the Cost of CISC ISA Support in Application-Specific Custom Embedded Processors", submitted for publication.

[52] M. G. Oms, J. Cortadella, M. Kishinevsky, "Speculation in elastic systems", Design Automation Conference, 2009.

[53] S. Onder and R. Gupta, "Automatic generation of microarchitecture simulators" In IEEE International Conference on Computer Languages, 1998.

[54] OneSpin Solutions, "The OneSpin 360® MV Product Family", http://www.onespin-solutions.com, 2010.

[55] PICO C Synthesis. http://www.synfora.com/

[56] I. Pyo, C. Su, I. Huang, K. Pan, Y. Koh, C. Tsui, H. Chen, G. Cheng, S. Liu, S. Wu, and A. M. Despain, "Application-driven design automation for microprocessor design", Design Automation Conference, 1992.

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

[58] 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.

[59] L. Shannon and P. Chow, “Standardizing the Performance Assessment of Reconfigurable Processor Architectures”, Field-Programmable Custom Computing Machines, 2003.

[60] J. P. Shen, M. Lipasti, “Modern Processor Design”, McGraw Hill Higher Education, 2002.

[61] W. Snyder, “Verilator”, http://www.veripool.org/wiki/verilator

[62] A. Srivastava, S. Park, E. Earlie, N. D. Dutt, A. Nicolau, Y. Paek, “Automatic Design Space Exploration of Register Bypasses in Embedded Processors”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 26, No. 12, 2007.

[63] Synopsys Design Compiler. http://www.synopsys.com

[64] System C. “IEEE 1666 Open SystemC Language Reference Manual”. http://www.systemc.org

[65] The Institute of Electrical Electronics Engineers, Inc., New York. IEEE Standard VHDL Language Reference Manual, 1988.

[66] D. E. Thomas and P. R. Moorby. The Verilog Hardware Description Language. Kluwer Academic Publishers, 3rd edition, 1996.

[67] T. Ungerer, B. Robic, and J. Silc, “A Survey of Processors with Explicit Multithreading”, ACM Computing Surveys, 2003.

[68] S. Ur and Y. Yadin, “Micro Architecture Coverage Directed Generation of Test Programs”, Design Automation Conference, 1999.

[69] Xilinx ISE Design Suite. http://www.xilinx.com/tools/designtools.htm

[70] xPilot System. http://cadlab.cs.ucla.edu/soc/

[71] P. Yiannacouras, J. G. Steffan, and J. Rose, “Exploration and Customization of FPGA-Based Soft Processors”, IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Vol. 26, No. 2, 2007.

Appendix A

T-spec Language Syntax

A.1. T-spec

T-spec ::= ***Components TopIOs*** [***Predictors***] ***Connections***

A.2. Components

Components ::= ***Component***

| ***Component Components***

Component := ***ModuleGeneric***

| ***ModuleState***

| ***ModuleMux***

A.2.1. Generic Modules

// --- Generic modules can be combinational, or multi-cycle

ModuleGeneric := ***ModuleGenericCombinational***

| ***ModuleGenericMultiCycle***

// --- Combinational block

ModuleGenericCombinational

:= `MODULE` ***ModuleName*** `GENERIC {` ***PortGenerics*** `}`

PortGenerics := ***PortGeneric***

| ***PortGeneric PortGenerics***

PortGeneric := ***PortIn***

| ***PortOut***

PortIn := `PORT` ***PortInName*** `IN` ***PortWidth***

PortOut := `PORT` ***PortOutName*** `OUT` ***PortWidth***

// --- Multi-cycle block

ModuleGenericMultiCycle

:= `MODULE` ***ModuleName*** `GENERIC_MC {` ***PortGenericMCs*** `}`

PortGenericMCs :=

`PORT` ***PortName*** `CLK`

`PORT` ***PortName*** `RST`

`PORT` ***PortInName*** `START`

`PORT` ***PortName*** `READY`

PORT ***PortName*** DONE

PORT ***PortName*** ACK

PORT ***PortName*** SQUASH

PortGenerics

A.2.2. State Modules

// --- States can be REG or ARRAY

ModuleState := ***StateReg***

| ***StateArray***

// --- REG state can be single- or multi-cycle

StateReg := ***StateRegSC***

| ***StateRegMC***

// --- Single-cycle REG state

StateRegSC :=

MODULE ***StateName*** REG {

```
            PORT PortName CLK

            PORT PortName RST

            StateRegSC_RdIfc StateRegSC_WrIfc

        }

// --- Single-cycle REG state – Read Interface

StateRegSC_RdIfc    :=

        IFC IfcName REG_RD {

            PORT PortName REG_DATA PortWidth

            PORT PortName REG_RD_EN

        }

// --- Single-cycle REG state – Write Interface

StateRegSC_WrIfc    :=

        IFC IfcName REG_WR {

            PORT PortName REG_DATA PortWidth

            PORT PortName REG_RD_EN

        }
```

// --- Multi-cycle REG state

StateRegMC :=

MODULE ***StateName*** REG_MC {

PORT ***PortName*** CLK

PORT ***PortName*** RST

StateRegMC_RdIfc StateRegMC_WrIfc

}

// --- Multi-cycle REG state – Read Interface

StateRegMC_RdIfc :=

IFC ***IfcName*** REG_RD_MC {

PORT ***PortName*** READY

PORT ***PortName*** DONE

PORT ***PortName*** ACK

PORT ***PortName*** SQUASH

PORT ***PortOutName*** REG_RD_DATA ***PortWidth***

PORT ***PortInName*** REG_RD_EN

}

// --- Multi-cycle REG state – Write Interface

StateRegMC_WrIfc :=

IFC ***IfcName*** REG_WR_MC {

PORT ***PortName*** READY

PORT ***PortName*** DONE

PORT ***PortName*** ACK

PORT ***PortName*** SQUASH

PORT ***PortInName*** REG_WR_DATA ***PortWidth***

PORT ***PortInName*** REG_WR_EN

}

// --- ARRAY state can be single- or multi-cycle

StateArray := ***StateArraySC***

| ***StateRegMC***

// --- Single-cycle ARRAY state

StateArraySC :=

```
MODULE StateName ARRAY {
    PORT PortName CLK
    PORT PortName RST
    StateArraySC_RdIfc StateArraySC_WrIfc
}
```

// --- Single-cycle ARRAY state – Read Interface

StateArraySC_RdIfc :=

```
IFC IfcName ARRAY_RD {
    PORT PortOutName ARRAY_RD_DATA PortWidth
    PORT PortInName REG_RD_EN
    PORT PortInName ARRAY_RD_IDX PortWidth
}
```

// --- Single-cycle ARRAY state – Write Interface

StateArraySC_WrIfc :=

```
IFC IfcName ARRAY_WR {
    PORT PortInName ARRAY_WR_DATA PortWidth
    PORT PortInName ARRAY_WR_EN
```

PORT ***PortInName*** ARRAY_WR_IDX ***PortWidth***

}

// --- Multi-cycle ARRAY state

StateArrayMC :=

MODULE ***StateName*** ARRAY_MC {

PORT ***PortName*** CLK

PORT ***PortName*** RST

StateArrayMC_RdIfc StateArrayMC_WrIfc

}

// --- Multi-cycle ARRAY state – Read Interface

StateArrayMC_RdIfc :=

IFC ***IfcName*** ARRAY_RD_MC {

PORT ***PortName*** READY

PORT ***PortName*** DONE

PORT ***PortName*** ACK

PORT ***PortName*** SQUASH

PORT ***PortOutName*** ARRAY_RD_DATA ***PortWidth***

PORT ***PortInName*** ARRAY_RD_IDX ***PortWidth***

PORT ***PortInName*** ARRAY_RD_EN

}

// --- Multi-cycle REG state – Write Interface

StateArrayMC_WrIfc:=

IFC ***IfcName*** ARRAY_WR_MC {

PORT ***PortName*** READY

PORT ***PortName*** DONE

PORT ***PortName*** ACK

PORT ***PortName*** SQUASH

PORT ***PortInName*** ARRAY_WR_DATA ***PortWidth***

PORT ***PortInName*** ARRAY_WR_IDX ***PortWidth***

PORT ***PortInName*** ARRAy_WR_EN

}

A.2.3. Multiplexers

ModuleMux :=

MODULE ***ModuleName*** MUX {

PORT ***PortOutName*** MUX_OUT ***PortWidth***

PORT ***PortInName*** MUX_SEL

PortMuxIns

}

PortMuxIns := ***PortMuxIn***

| ***PortMuxIn PortMuxIns***

PortMuxIn := PORT ***PortInName*** MUX_IN ***PortWidth MuxInOrder***

A.3. Top I/Os

TopIOs := ***TopIO***

| ***TopIO TopIOs***

TopIO := ***TopIn***

| ***TopOut***

TopIn := TOP ***TopInName*** IN ***TopInWidth***

TopOut := TOP ***TopOutName*** OUT ***TopOutWidth***

A.4. Predictors

Predictors := ***Predictor***

| ***Predictor Predictors***

Predictor :=

```
MODULE PredictorName PRED StateName {
    PORT PortName PRED_VALID
    PORT PortName PRED_VALUE PortWidth
}
```

A.5. Connections

Connections := ***Connection***

| ***Connection Connections***

Connection := `CONN {` ***ConnectionSrc*** → ***ConnectionDst*** `}`

ConnectionSrc ::= ***ModuleName.PortOutName***

| ***StateName.IfcName.PortOutName***

| ***TopInName***

ConnectionDst ::= ***ModuleName***.***PortInName***

| ***StateName***.***IfcName***.***PortInName***

| ***TopOutName***

A.6. Miscellaneous

ModuleName ::= `[a-z][a-z0-9]*`

StateName ::= `[a-z][a-z0-9]*`

TopInName ::= `[a-z][a-z0-9]*`

TopOutName ::= `[a-z][a-z0-9]*`

PredictorName ::= `[a-z][a-z0-9]*`

IfcName ::= `[a-z][a-z0-9]*`

PortName ::= `[a-z][a-z0-9]*`

PortWidth ::= `[1-9][0-9]*`

TopInWidth ::= `[1-9][0-9]*`

TopOutWidth ::= `[1-9][0-9]*`

MuxInOrder ::= `[1-9][0-9]*`

Appendix B

P-cfg Language Syntax

B.1. P-cfg

P-cfg ::= ***StageDeclarations StageBindings***

B.2. Stage Declarations

StageDeclarations ::= ***StageDeclaration***

| ***StageDeclaration StageDeclarations***

StageDeclaration := `PSTAGE` ***StageName***

B.3. Stage Bindings

StageBindings ::= ***StageBinding***

| ***StageBinding StageBindings***

StageBinding := `PBIND` ***StageName ModuleName***

| `PBIND` ***StageName StateName.IfcName***

| `PBIND` ***StageName PredictorName***

| `PBIND` ***TopInName***

| `PBIND` ***TopOutName***

Note: ModuleName, StateName, IfcName, PredictorName, TopInName, and TopOutName refer to the components and top I/Os specified in the T-spec (see Appendix A).

B.4. Miscellaneous

StageName ::= `[a-z][a-z0-9]*`

Appendix C

MIPS Processor Example

C.1. T-spec

C.1.1. States

```
MODULE pc REG {

      PORT clock CLK

      PORT reset RST

      IFC rd REG_RD {

            PORT en REG_RD_EN

            PORT d REG_RD_DATA 32

      }

      IFC wr REG_WR {

            PORT en REG_WR_EN

            PORT d REG_WR_DATA 32

      }
```

```
}

MODULE gpr ARRAY {

      PORT clock CLK

      PORT reset RST

      IFC rd0 ARRAY_RD {

            PORT en ARRAY_RD_EN

            PORT d ARRAY_RD_DATA 32

            PORT idx ARRAY_RD_IDX 5

      }

      IFC rd1 ARRAY_RD {

            PORT en ARRAY_RD_EN

            PORT d ARRAY_RD_DATA 32

            PORT idx ARRAY_RD_IDX 5

      }

      IFC wr ARRAY_WR {

            PORT en ARRAY_WR_EN

            PORT d ARRAY_WR_DATA 32
```

```
            PORT idx ARRAY_WR_IDX 5
      }
}
MODULE imem ARRAY_MC {
      PORT clock CLK
      PORT reset RST
      IFC rd ARRAY_RD_MC {
            PORT ready READY
            PORT done DONE
            PORT ack ACK
            PORT squash SQUASH
            PORT d ARRAY_RD_DATA PortWidth
            PORT addr ARRAY_RD_IDX PortWidth
            PORT en ARRAY_RD_EN
      }
}
MODULE dmem ARRAY_MC {
```

```
PORT clock CLK

PORT reset RST

IFC rd ARRAY_RD_MC {

      PORT ready READY

      PORT done DONE

      PORT ack ACK

      PORT squash SQUASH

      PORT d ARRAY_RD_DATA 32

      PORT addr ARRAY_RD_IDX 32

      PORT en ARRAY_RD_EN

}

IFC wr ARRAY_WR_MC {

      PORT ready READY

      PORT done DONE

      PORT ack ACK

      PORT squash SQUASH

      PORT d ARRAY_WR_DATA 32
```

```
        PORT addr ARRAY_WR_IDX 32

        PORT en ARRAY_WR_EN

    }

}
```

C.1.2. Next-state Compute Blocks

```
MODULE constants GENERIC {

    PORT zero OUT 1

    PORT one OUT 1

}

MODULE decoder GENERIC {

    PORT inst IN 32



    PORT gpr_rd0_idx OUT 5

    PORT gpr_rd1_idx OUT 5

    PORT gpr_wr_idx OUT 5

    PORT gpr_rd0_en OUT 1

    PORT gpr_rd1_en OUT 1
```

```
    PORT gpr_wr_en OUT 1

    PORT gpr_wr_mux_sel OUT 1


    PORT pc_wr_en OUT 1

    PORT pc_wr_mux_sel OUT 2


    PORT dmem_rd_en OUT 1

    PORT dmem_wr_en OUT 1


    PORT imm16 OUT 16

    PORT imm26 OUT 26


    PORT br_eval_op OUT 6

    PORT alu_op OUT 6

    PORT alu_src1_mux_sel OUT 1

    PORT imm_calc_op OUT 1

}
```

```
MODULE alu GENERIC {
      PORT op IN 6
      PORT din0 IN 32
      PORT din1 IN 32
      PORT dout OUT 32
}
MODULE imm_calc GENERIC {
      PORT imm_calc_op IN 1
      PORT imm16 IN 16
      PORT imm OUT 32
}
MODULE npc GENERIC {
      PORT pc_in IN 32
      PORT npc_out OUT 32
}
MODULE j_eval GENERIC {
      PORT npc IN 32
```

```
      PORT imm26 IN 26
      PORT j_target OUT 32
}
MODULE b_eval GENERIC {
      PORT npc IN 32
      PORT imm IN 32
      PORT din0 IN 32
      PORT din1 IN 32
      PORT br_eval_op IN 6
      PORT b_target OUT 32
}

MODULE pc_wr_mux MUX {
      PORT sel MUX_SEL 2
      PORT in0 MUX_IN 32 0
      PORT in1 MUX_IN 32 1
      PORT in2 MUX_IN 32 2
```

```
      PORT dout MUX_OUT 32
}
MODULE alu_src1_mux MUX {
      PORT sel MUX_SEL 1
      PORT in0 MUX_IN 32 0
      PORT in1 MUX_IN 32 1
      PORT dout MUX_OUT 32
}
MODULE gpr_wr_mux MUX {
      PORT sel MUX_SEL 1
      PORT in0 MUX_IN 32 0
      PORT in1 MUX_IN 32 1
      PORT dout MUX_OUT 32
}
```

C.1.3. Connections

```
// -- to pc
CONN { constants.one → pc.rd.en }
```

```
CONN { decoder.pc_wr_en → pc.wr.en }

CONN { pc_wr_mux.dout → pc.wr.d }


// -- to gpr

CONN { decoder.gpr_rd0_en → gpr.rd0.en }

CONN { decoder.gpr_rd0_idx → gpr.rd0.idx }

CONN { decoder.gpr_rd1_en → gpr.rd1.en }

CONN { decoder.gpr_rd1_idx → gpr.rd1.idx }

CONN { decoder.gpr_wr_en → gpr.wr.en }

CONN { decoder.gpr_wr_idx → gpr.wr.idx }

CONN { gpr_wr_mux.dout → gpr.wr.d }


// -- to imem

CONN { constants.one → imem.rd.en }

CONN { pc.rd.d → imem.rd.addr }


// -- to dmem
```

```
CONN { decoder.dmem_rd_en → dmem.rd.en }

CONN { alu.dout → dmem.rd.addr }

CONN { decoder.dmem_wr_en → dmem.wr.en }

CONN { alu.dout → dmem.wr.addr }

CONN { gpr.rd1.d → dmem.wr.d }


// -- to decoder

CONN { imem.rd.d → decoder.inst }


// -- to alu

CONN { decoder.alu_op → alu.op }

CONN { gpr.rd0.d → alu.din0 }

CONN { alu_src1_mux.dout → alu.din1 }


// -- to imm_calc

CONN { decoder.imm_calc_op → imm_calc.op }

CONN { decoder.imm16 → imm_calc.imm16 }
```

```
// -- to npc

CONN { pc.rd.d → npc.pc_in }


// -- to j_eval

CONN { npc.npc_out → j_eval.npc }

CONN { decoder.imm26 → j_eval.imm26 }


// -- to b_eval

CONN { npc.npc_out → b_eval.npc }

CONN { imm_calc.imm → b_eval.imm }

CONN { gpr.rd0.d → b_eval.din0 }

CONN { alu_src1_mux.dout → b_eval.din1 }

CONN { decoder.br_eval_op → b_eval.op }


// -- to pc_wr_mux

CONN { decoder.pc_wr_mux_sel → pc_wr_mux.sel }

CONN { npc.npc_out → pc_wr_mux.in0 }
```

CONN { *j_eval.j_target* → *pc_wr_mux.in1* }

CONN { *b_eval.b_target* → *pc_wr_mux.in2* }

// -- to alu_src1_mux

CONN { *decoder.alu_src1_mux_sel* → *alu_src1_mux.sel* }

CONN { *gpr.rd1.d* → *alu_src1_mux.in0* }

CONN { *imm_calc.imm* → *alu_src1_mux.in1* }

// -- to gpr_wr_mux

CONN { *decoder.gpr_wr_mux_sel* → *gpr_wr_mux.sel* }

CONN { *dmem.rd.d* → *gpr_wr_mux.in0* }

CONN { *alu.dout* → *gpr_wr_mux.in1* }

C.2. P-cfg for a 5-stage pipeline

C.2.1. Stage Declarations

PSTAGE *fetch*

PSTAGE *decode*

PSTAGE *exe*

PSTAGE *mem*

PSTAGE *wb*

C.2.1. Stage Bindings

PBIND *fetch constants*

PBIND *fetch pc.rd*

PBIND *fetch imem.rd*

PBIND *fetch npc*

PBIND *decode decoder*

PBIND *decode imm_calc*

PBIND *decode gpr.rd0*

PBIND *decode gpr.rd1*

PBIND *decode j_eval*

PBIND *decode b_eval*

PBIND *decode alu_src1_mux*

PBIND ***exe alu***

PBIND ***mem dmem.rd***

PBIND ***mem dmem.wr***

PBIND ***wb gpr_wr_mux***

PBIND ***wb gpr.wr***

PBIND ***wb pc_wr_mux***

PBIND ***wb pc.wr***