Automatic Multithreaded Pipeline Synthesis from Transactional Datapath Specifications

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

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

ABSTRACT

We present a technique to automatically synthesize a multithreaded in-order pipeline from a high-level unpipelined datapath specification. This work extends the previously proposed transactional specification (T-spec) and synthesis technology (T-piper). The technique not only works with instruction processors but also flexible enough to accept any sequential datapath. It maintains previously proposed non-threaded pipeline features and is enhanced with multithreading features. We report a design space exploration study of 32 multithreaded x86 processor pipelines, all synthesized from a single T-spec.

Categories and Subject Descriptors

B.5.2 [**Hardware**]: Design Aids – *Automatic synthesis.*

General Terms

Design.

Keywords

Datapath specification, multithreading, automatic pipelining, hardware synthesis, design exploration of x86 processor pipelines.

1. INTRODUCTION

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 [4-8, 11, 13, 14, 16], to the best of our knowledge there has not been any for synthesis of in-order *multithreaded* pipelines. Prior works [1, 3, 9, 10, 12] have also presented multithreaded processor pipelines for FPGA prototyping, but they are manually developed.

In this paper, we propose extensions to the recently proposed transactional datapath specification (T-spec) and its in-order pipeline synthesis technology (T-piper) [13] to support multithreading. Our proposal not only works well with instruction processor pipelines but also is flexible enough to accept any sequential datapath. It maintains the synthesis features for non-threaded pipelines proposed previously (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 our work, we report a case study, using 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 rest of the paper is organized as follows. Section 2 gives background on relevant prior work. Section 3 presents a motivating example to be used for discussion in the later sections. Section 4 summarizes the recently proposed transactional datapath specification (T-spec) and pipeline synthesis from it (T-piper). Section 5 discusses extensions for T-spec and T-piper to support multithreading. Section 6, presents a design exploration study of x86 processor pipelines. Section 7 offers concluding remarks.

2. RELATED WORK

Many previous studies [4-8, 11, 13, 14, 16] have investigated ways to automate pipeline development. However, all these studies target in-order pipelines that are *not* multithreaded.

The only automation work we could find on multithreading is pipeline generation from a parameterized in-order multithreaded pipeline *template* [2], which is very restricted. For example, the template is fixed to a 4-stage pipeline design for a processor with MIPS ISA as baseline.

Other recent works [1, 3, 9, 10, 12] also presented design case studies of multithreaded processor pipelines for FPGA prototyping. However these pipelines are *manually developed*.

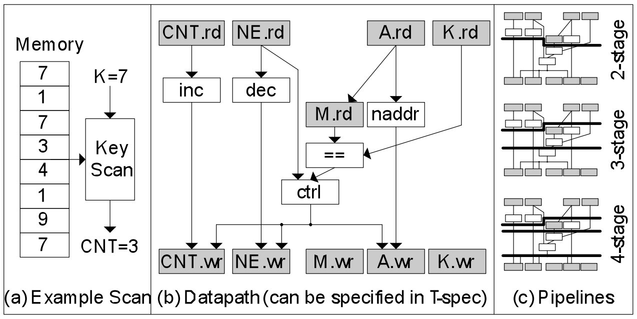
We believe our work is the first to fully automate the synthesis of *multithreaded* in-order pipelines from an unpipelined datapath specification. Furthermore, it is very flexible. Not only it allows synthesis of instruction processors with multithreading features found in previously mentioned FPGA prototyping studies, but it also allows capturing larger design space of any sequential system

DAC'10, June 13-18, 2010, Anaheim, California, USA
Copyright 2010 ACM 978-1-4503-0002-5 /10/06...$10.00

datapath beyond instruction processor as well as enabling new multithreading features (e.g., states shared by a group of threads).

3. MOTIVATING EXAMPLE: KEY SCAN

To illustrate pipelining and multithreading usage scenarios to be discussed in this paper, here we present a simple example of a key scanner that finds the number of occurrences of a given 32-bit key value in an array of words in memory. Figure 1a 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.

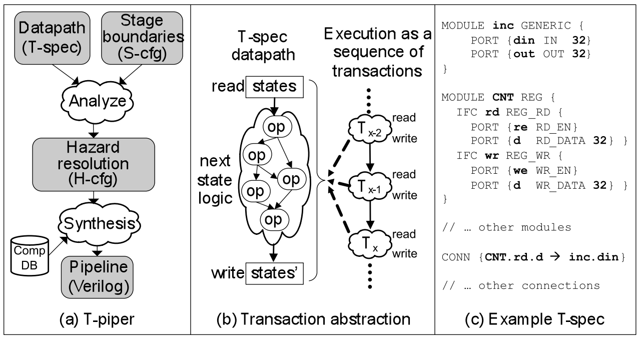


(a) Example Scan (b) Datapath (can be specified in T-spec) (c) Pipelines

Figure 1. Key scan example.

Figure 1b 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 1a 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.



```
MODULE inc GENERIC {
    PORT {din IN  32}
    PORT {out OUT 32}
}

MODULE CNT 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} }
}

// … other modules

CONN {CNT.rd.d → inc.din}

// … other connections
```

(a) T-piper (b) Transaction abstraction (c) Example T-spec

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

4. T-SPEC PIPELINE DESIGN REVIEW

We can pipeline the datapath in Figure 1b to reduce critical paths and improve frequency, by dividing the next-state logic blocks into multiple stages separated by pipeline registers. For example, Figure 1c shows three possible pipeline implementations of the datapath in Figure 1b. However, manually pipelining a datapath can be tedious and error prone. To address this issue, previous work [13] has presented a transactional design approach to automate pipeline development.

The approach works as follows. First, a non-pipelined version of the datapath that performs a set of next-state compute operations, or a transaction, one at a time is captured using transactional specification (T-spec). The example in Figure 1b can be specified in T-spec straightforwardly. Then, to arrive at a pipelined implementation, T-piper analysis (Figure 2a), based on designer-specified pipeline-stage boundaries (S-cfg), informs the designer the available opportunities for applying forwarding and speculation to resolve hazards (H-cfg). Next, based on the designer's selection of which forwarding and speculation optimizations to include, T-piper generates an RTL-Verilog implementation of the desired pipeline, which preserves the transaction semantics of the T-spec datapath. Starting from a T-spec, the designer can rapidly explore the pipeline design space by submitting different pipeline configurations to T-piper.

More specifically, T-spec is a textual "netlist" that comprises of architectural states and next-state compute operations implemented by a network of logic blocks. An architectural state (register or array) has explicit state-read and state-write interfaces. Note that *architectural* states are those in an unpipelined datapath in T-spec, which are different than the pipeline registers automatically inserted by T-piper in the implementation. A next-state compute logic block can be either a combinational block, or a fixed/variable multi-cycle (MC) block that implements a handshake interface based on ready, start, and done signals. All blocks are treated as black-boxes for analysis, except for multiplexers, which is a logic primitive understood by T-piper and used for hazard analysis.

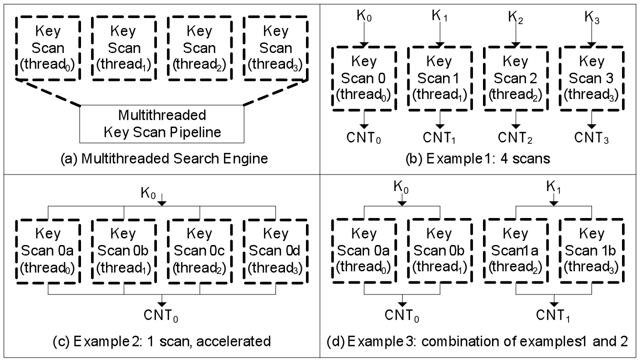
A T-spec captures an abstract datapath, whose execution semantics is interpreted as a sequence of "transactions" where each transaction reads the state values left by the preceding transaction and computes a new set of state values to be seen by the next transaction (Figure 2b). Many valid implementations may be derived from T-spec, as long as it preserves the transaction semantics. Previous work [13] presented a technique to synthesize a non-threaded in-order pipeline from a T-spec. This paper presents extensions to the work to synthesize an in-order multithreaded pipeline from a T-spec.

Figure 2c gives an example T-spec excerpt for Figure 1b datapath. It starts with a `GENERIC` module declaration for `inc`, a black-box combinational block with 32-bit input and output named `din` and `dout`, respectively. The second module declaration is for a built-in `REG`-type 32-bit architectural state `CNT`. A `REG`-type state module has two interfaces: read and write. The read (or write) interface comprises of a read-enable (or write-enable) and an output read-data (or input write-data) ports. Lastly, a connection declaration connects the `d` output port of `CNT`'s `rd` interface to the `din` input port of `inc`. The declarations for the remaining modules and connections are omitted for brevity in this example.

5. 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.



(a) Multithreaded Search Engine

(b) Example1: 4 scans

(c) Example2: 1 scan, accelerated

(d) Example3: combination of examples1 and 2

Figure 3. Multithreaded key scan.

5.1 Multithreading the Key Scan Example

Let us suppose that we would like to improve the example datapath in Figure 1b by pipelining and multithreading that supports 4 threads (Figure 3a). There are multiple possible multithreading scenarios that can be employed, three of which are shown in Figures 3b, 3c, and 3d. 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 3b). Second, only 1 scan is performed, but accelerated by having the 4 threads scanning different parts of the memory (Figure 3c). Lastly, the first and second scenarios can be combined, where there are two scans, each one performed by two threads (Figure 3d). 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.

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 Section 4), 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.

5.2 Multithreading Features

5.2.1 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) [15].

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 [9].

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 [9], and other custom-designed scheduling policies.

5.2.2 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.

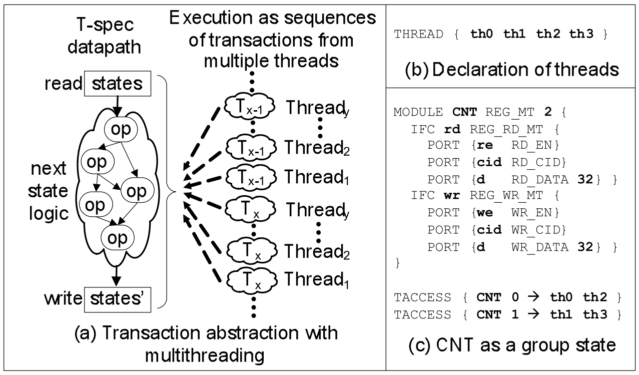
Our synthesis system supports the recently proposed approach to deal with long-latency events based on replay [9, 10]. The idea is to allow a pipeline stage to request a replay when it suffers from a long-latency event. Upon replay, the thread in that stage is canceled and re-executed at a later time. Meanwhile, other threads can use the pipeline and proceed with their execution.

A known shortcoming of replay [10] 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.

5.2.3 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 3b). 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 3c). Third, a group state has multiple contexts, each accessible only by a set of threads (e.g., *K* and *CNT* in Figure 3d).

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.



(a) Transaction abstraction with multithreading

```
THREAD { th0 th1 th2 th3 }
```

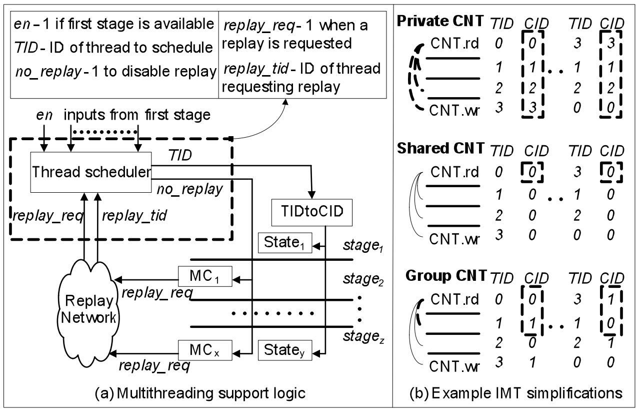
(b) Declaration of threads

```
MODULE CNT REG_MT 2 {
  IFC rd REG_RD_MT {
    PORT {re  RD_EN}
    PORT {cid RD_CID}
    PORT {d   RD_DATA 32} }
  IFC wr REG_WR_MT {
    PORT {we  WR_EN}
    PORT {cid WR_CID}
    PORT {d   WR_DATA 32} }
}

TACCESS { CNT 0 → th0 th2 }
TACCESS { CNT 1 → th1 th3 }
```

(c) CNT as a group state

Figure 4. Extending T-spec for multithreading.



(a) Multithreading support logic

Private CNT

	TID	*CID*	*TID*	*CID*
CNT.rd	0	0	3	3
	1	1	1	1
	2	2	2	2
CNT.wr	3	3	0	0

Shared CNT

	TID	*CID*	*TID*	*CID*
CNT.rd	0	0	3	0
	1	0	1	0
	2	0	2	0
CNT.wr	3	0	0	0

Group CNT

	TID	*CID*	*TID*	*CID*
CNT.rd	0	0	3	1
	1	1	1	0
	2	0	2	1
CNT.wr	3	1	0	0

(b) Example IMT simplifications

Figure 5. Multithreading support logic and IMT hazard management logic simplification examples.

5.3 T-spec for Multithreading

5.3.1 Extending the Transaction Abstraction

We extend the transaction abstraction captured by T-spec (as previously explained in Section 3) to support multithreading. Figure 4a 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 4), 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 4), 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 5.2.3), 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 4a shows a round-robin order, but any order is valid).

5.3.2 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 4b 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 4c 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 4c 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 5a. 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 S-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.

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 5a) 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.

5.4 Multithreaded Pipeline Synthesis

5.4.1 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 [13]. 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.

T-spec has already provided the information on how a thread accesses each state element (e.g. Figure 4c). 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 5a). 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 5b provides an example IMT hazard logic simplifications to the *CNT* architectural state for the key scan implementation scenarios shown in Figure 3b, 3c, and 3d, 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.

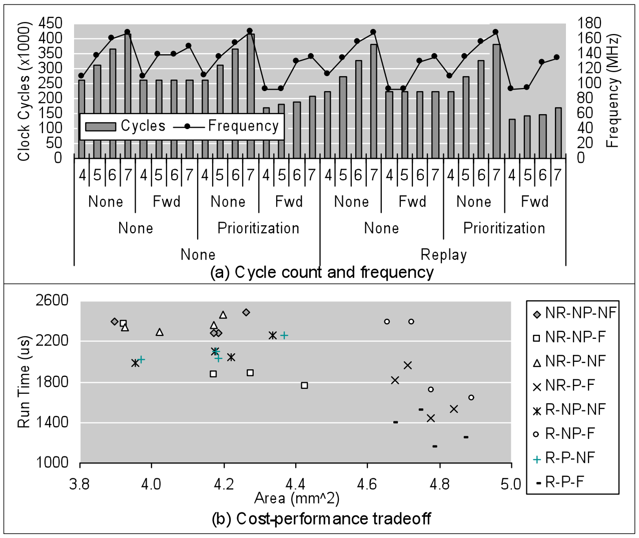
To do this, T-piper enumerates all possible thread orders in the pipeline, given IMT scheduling. Next, for each thread order, it translates the TID of the thread occupying the stage to its CID, for each state element. Figure 5b shows the enumerated TIDs and the associated CIDs for *CNT* in the key scan 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.

5.4.2 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 5a.

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 TIDtoCID translation logic. 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.



(a) Cycle count and frequency

(b) Cost-performance tradeoff

Figure 6. Design space exploration of x86 pipelines.

6. 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 [13], and 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 [16]. 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: (1) pipeline depths varied from 4 to 7 stages; (2) with (F) and without (NF) inclusion of maximal forwarding; (3) 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; and, (4) with (R) and without (NR) the capability to replay on a cache miss.

Figure 6a 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 6a 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 6b 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.

Previous work [13] has 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. We found that multithreading without any optimization (i.e., NR-NP-NF) resulted to only 26% area increase and 4% frequency decrease, while shortening run-time by 18%. 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.

7. CONCLUSION

This paper presented a novel approach for automatically synthesizing in-order *multithreaded* pipelines, by extending the recently proposed transactional datapath specification (T-spec) and the associated pipeline synthesis technology (T-piper). T-spec and T-piper greatly reduce the time and effort to develop in-order pipelines, with and without multithreading. The effectiveness of this work was demonstrated through a design space exploration study of 32 multithreaded processor pipelines supporting an x86 ISA subset, showcasing the benefits of several multithreading features supported by the proposed synthesis technology.

8. REFERENCES

[1] 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.

[2] R. Dimond, O. Mencer, W. Luk, “Application-specific Customization of Multithreaded Soft Processors”, IEE Computers and Digital Techniques, Vol. 153, Issue 3, 2006.

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

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

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

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

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

[8] D. Kroening, W. Paul, “Automated pipeline design”, Design Automation Conference, 2001.

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

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

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

[12] 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.

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

[14] 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.

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

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