PDL

A High-Level Hardware Design Language for Pipelined Processors

Drew Zagieboylo
Department of Computer Science
Cornell University
dz333@cornell.edu

Charles Sherk
Department of Computer Science
Cornell University
cs897@cornell.edu

G. Edward Suh
School of Electrical and Computer Engineering
Cornell University
suh@ece.cornell.edu

Andrew C. Myers
Department of Computer Science
Cornell University
andru@cs.cornell.edu

Abstract

Processors are typically designed in Register Transfer Level (RTL) languages, which give designers low-level control over circuit structure and timing. To achieve good performance, processors are pipelined, with multiple instructions executing concurrently in different parts of the circuit. Thus even though processors implement a fundamentally sequential specification (the instruction set architecture), the implementation is highly concurrent. The interactions of multiple instructions—potentially speculative—can cause incorrect behavior.

We present PDL, a novel hardware description language targeted at the construction of pipelined processors. PDL provides *one-instruction-at-a-time* semantics; the first language to enforce that the generated pipelined circuit has the same behavior as a sequential specification. This enforcement facilitates design-space exploration. Adding or removing pipeline stages, moving operations across stages, or otherwise changing pipeline structure normally requires careful analysis of bypass paths and stall logic; with PDL, this analysis is handled by the PDL compiler. At the same time, PDL still offers designers fine-grained control over performance-critical microarchitectural choices such as timing of operations, data forwarding, and speculation. We demonstrate PDL's expressive power and ease of design exploration by implementing several RISC-V cores with differing microarchitectures. Our results show that PDL does not impose significant performance or area overhead compared to a standard HDL.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PLDI '22, June 13–17, 2022, San Diego, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9265-5/22/06…$15.00
https://doi.org/10.1145/3519939.3523455

CCS Concepts • **Hardware → Hardware description languages and compilation**; • **Software and its engineering → Compilers**.

ACM Reference Format:
Drew Zagieboylo, Charles Sherk, G. Edward Suh, and Andrew C. Myers . 2022. PDL : A High-Level Hardware Design Language for Pipelined Processors. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3519939.3523455

1 Introduction

To achieve high performance, processors parallelize the execution of sequential instruction streams through pipelines, achieving high throughput via microarchitectural optimizations such as bypassing, speculation, and out-of-order execution. Processor designs are inherently complex since they must respect the sequential semantics of the instruction set architecture (ISA) despite aggressively executing operations in parallel. Processors are usually designed using hardware description languages (HDL) that operate at the register transfer level (RTL), providing low-level control but at the cost of highly parallel semantics that make reasoning difficult. This combination of complexity and RTL abstraction makes it difficult to achieve high confidence in the correctness of processor implementations. In practice, RTL processors are usually validated via simulation or bounded model checking: techniques that have seen practical success but cannot expose all bugs in large designs [13, 20].

We propose a new approach, a Pipeline Description Language (PDL) that raises the level of abstraction to specifically target the construction of processor pipelines. PDL allows designers to easily specify the intended functionality of a processor, while still giving them fine-grained control over its microarchitecture and performance. Designers can demarcate stage boundaries, ensuring each stage executes in a single clock cycle. PDL introduces *hazard locks*, which abstract different implementations of stalling and bypass logic

```
1  pipe cpu(pc)[rf, imem, dmem] {
2    acquire(imem[pc], R); //IFETCH STG
3    insn <- imem[pc];
4    release(imem[pc]);
5    --- //DECODE STG
6    op = insn{6:0};
7    //decode logic for rs1,rs2,etc.
8    acquire(rf[rs1], R); acquire(rf[rs2], R);
9    rf1 = rf[rs1]; rf2 = rf[rs2];
10   release(rf[rs1]); release(rf[rs2]);
11   if (writerd) reserve(rf[rd], W);
12   --- //EXEC STG
13   alu_out = alu(alu_op, alu_arg1, alu_arg2);
14   offset = calc_offset(op, pc, imm, alu_out);
15   //start next instruction
16   call cpu(pc + offset);
17   --- //MEM STG
18   acquire(dmem[alu_out]);
19   if (isStore(op)) { dmem[alu_out] <- data; }
20   if (isLoad(op)) { rddata <- dmem[alu_out]; }
21   else { rddata = alu_out; }
22   release(dmem[alu_out]);
23   --- //WB STG
24   if (writerd) { block(rf[rd]);
25                  rf[rd] <- rddata;
26                  release(rf[rd]); }
27 }
```

Figure 1. Abbreviated PDL code for a 5-stage RISC pipeline.

to prevent data hazards. Additionally, PDL offers a speculation API that enables pipelines to flexibly initiate and resolve branch prediction. Lastly, PDL supports a limited form of out-of-order execution.

Despite this microarchitectural control, PDL provides an easy-to-understand *one-instruction-at-a-time* semantics. The realized behaviors of pipelines are consistent with an execution that runs each instruction completely in sequence. This strong assurance allows designers and static analysis tools to easily reason about the behavior of a design with respect to a sequential specification, facilitating design space exploration.

As such, PDL does not directly support architectures with relaxed consistency guarantees, such as the memory models of multicore architectures. Nor can PDL express all pipelined architectures, such as superscalar or 2D systolic arrays. Lastly, PDL does not provide strong guarantees about the *timing* of updates to architectural state, and thus cannot reason precisely about timing channels. Supporting more relaxed definitions of correctness, microarchitectural expressivity, and precise reasoning about timing are interesting potential future extensions to PDL.

In this work we present the following:

- An overview of the PDL language and its microarchitectural abstractions for pipeline structure, data hazard resolution, and speculation.
- An informal presentation of PDL's semantics, correctness assurance, and advantages over RTL.
- A description of the PDL compiler implementation.
- Evidence of PDL's expressivity, practicality, and utility in design-space exploration. To do so we evaluate the performance of several RISC-V cores, implemented in PDL with differing microarchitectures.

2 Pipeline Description Language

In an RTL implementation, the designer must explicitly instantiate registers to store each pipeline stage's inputs and must manually coordinate the communication between stages. PDL, in contrast, only requires the user to specify the core functionality as an imperative-style program; then they can employ a few key microarchitectural primitives to control the pipeline's structure and performance. The PDL compiler automatically generates the registers and control logic necessary to split the pipeline into multiple, concurrently executing stages. The PDL compiler also makes it safe and easy to alter the pipeline structure or to move functionality across stages, without worrying about introducing bugs.

2.1 Language Design

Figure 1 demonstrates some features of PDL by presenting an abbreviated RISC processor. The code in this example is more similar to an imperative program than typical RTL code, and can mostly be understood via the straightforward imperative interpretation. Syntax for combinational logic in PDL is mostly standard, with support for sized integers and typical operators such as bit selection and concatenation. Variables are declared and assigned exactly once, like Verilog [27] wires. Array access notation denotes a request to a *memory*, which is any stateful, addressed data structure, including registers; it need not be implemented as SRAM/DRAM. Memories may be declared as providing either combinational or synchronous read access[1]. The read at line 3 is made with the arrow (`<-`) notation because `imem` has a synchronous interface: its data cannot be used until the next pipeline stage. Modern processors often contain pipelined subcomponents: PDL thus supports a **`call`** statement that allows one pipeline to make a synchronous request to another, e.g.:

```
int<32> divres <- call multi_cycle_div(arg1,arg2);
```

The above example sends a request to a pipelined divider, `multi_cycle_div`. The subsequent stage in the primary pipeline will wait for the divider's response before executing.

Pipeline Structure Stage separators (`---`)[2] control the structure of a pipeline, breaking up combinational logic across multiple clock cycles.

[1]Synchronous means that requests and responses are coordinated via a clock-edge aligned protocol, such as a ready-valid interface.

[2]This notation is inspired by Dahlia's [16] ordered composition operator.

Table 1. The hazard-lock interface with summarized requirements for using each operation.

Operation	Description	Requirements
`reserve(m[a], R/W)`	Defines the intended order of memory ops	Lock has not been acquired and is executed during an in-order stage.
`block(m[a])`	Stall the current thread until it can execute the associated op	Lock is "reserved".
`acquire(m[a], R/W)`	Syntactic sugar for reserve;block	Same as reserve.
`read/write(m[a])`	Execute the op, potentially forwarding data	Lock is "acquired" via block().
`release(m[a])`	Release lock resources associated with op and commit	Read or write has executed, and release is executed during an in-order stage.
`checkpoint(m)`	Create a checkpoint of lock state	Automatically inserted with final reservation.
`rollback(c)`	Reset lock state to checkpoint c	Automatically inserted with verify statements.

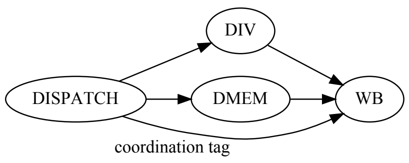


Figure 2. Stage graph of a pipeline with unordered DIV and DMEM stages. Each stage can execute different instructions in parallel: an in-order issue, out-of-order execute pipeline. The coordination tag is used to re-establish the original execution order in the WB stage.

Although each instruction flows through pipeline stages in sequence, stages actually execute in parallel and can process multiple instructions at a time. For the most part, separators can be placed wherever the designer wishes, to tune the critical path of the realized design without affecting the functionality; PDL rejects any design that could violate one-instruction-at-a-time semantics.

Out-of-Order Stages PDL is not limited to fully in-order pipeline descriptions; placing stage separators inside conditional branches describes a pipeline as a directed acyclic graph. While instructions travel through the *ordered* stages of the pipeline in the same order they were started, this is not true for *unordered* stages. Consider the following CPU design, which utilizes the aforementioned **call** statement to execute division in a separate pipeline, but still allows memory access operations to execute in parallel:

```
//DISPATCH
if (isDiv) {
  --- //DIV
  int<32> res <- call multi_cycle_div(arg1,arg2);
} else {
  int<32> addr = arg1 + off;
  --- //DMEM
  int<32> res <- dmem[addr];
}
--- //WB
rf[rd] <- res;
```

Figure 2 visualizes the pipeline generated by this code snippet. The DIV and DMEM stages may execute in parallel, despite being unordered. PDL ensures that the code following the branch (the WB stage) *does* execute in order. PDL generates coordination signals that record the original execution order. The DISPATCH stage enqueues a tag indicating which branch an instruction took; the WB stage uses this queue to determine from which stage to receive its next inputs, and stalls until that stage has completed execution.

Pipeline Threads In PDL, a pipeline body describes how to sequentially process a *single* instruction. To initiate execution of the next instruction, a *recursive* **call** is used, as at line 16. The one-instruction-at-a-time semantics allows the designer to think of these recursive calls as tail calls that occur at the end of the pipeline body. Semantically, a pipeline is a loop that processes one instruction per iteration.

The placement of the recursive **call** does not affect the semantics of the generated circuit, but it does have an impact on performance: it introduces concurrency. At this point in the pipeline, the pipeline begins processing the subsequent (called) instruction, *in parallel* with the rest of the current instruction. We borrow the term *thread* from concurrent software; each instruction is executed by a single thread that travels through the pipeline independently, and potentially in parallel with other threads. *Thread order* refers to the order in which threads are initiated; it is equivalent to program order in processors.

2.2 Preventing Data Hazards with Hazard Locks

A key to one-instruction-at-a-time semantics is ensuring that pipelines are free of data hazards. Data hazards occur when read and write operations on memories do not respect thread order, and are typically prevented by explicit stall logic or by bypassing values from writes to reads. For instance, in a standard 5-stage processor pipeline, stall and bypass logic are needed to prevent the following RISC-V [29] instruction sequence from creating a read-after-write hazard:

```
lw a0, 0(sp) //load data from stack into a0
addi a0, a0, 1 //increment a0 register
```

Absent this logic, the read from register `a0` in the second instruction would occur *before* the load into `a0`, so the final value of `a0` would effectively ignore the load instruction.

Hazard Locks PDL introduces a novel *hazard lock* abstraction to prevent data hazards. Hazard locks encapsulate data hazard prevention in a separate hardware module, whose usage in the pipeline can be checked by the compiler. This design contrasts strongly with traditional RTL development, where bypass and stall logic is explicitly described by the designer and manually integrated into the entire pipeline. RTL hazard resolution logic is also *non-modular* and brittle; it cannot be re-used across designs, and must often be changed if the pipeline structure is modified. On the other hand, hazard locks are a general abstraction that can express a variety of different microarchitectural designs, from simple stall logic to the renaming used in complex out-of-order pipelines.

As with traditional software locks, a thread must *acquire* a hazard lock before accessing the associated memory location:

```
acquire(rf[rs1], R); //acquire READ lock for rs1
int<32> x = rf[rs1]; //OK: lock acquired
int<32> y = rf[rs2]; //ERROR: acquire missing
```

Similarly, hazard locks must eventually be *released*. To support implementations with a variety of performance characteristics, PDL allows acquisition to be split into two phases: *reservation*, and *blocking* until the reservation is fulfilled.

```
reserve(rf[rd], W); //reserve WRITE lock for rd
---              //(READ locks can be reserved too)
block(rf[rd]); //STALL this stage until OK to exec
```

The **acquire** operation is actually just syntactic sugar for the sequence **reserve** followed by **block** in the same stage.

A key insight is that, even in highly speculative, out-of-order processors, there is an in-order execution point where the CPU establishes and records sequential data dependencies in some data structure. We abstract this record-keeping point as lock *reservation*; it must execute in thread order, but still allows execution to proceed freely. *Blocking* represents the point in the pipeline when a stage may be forced to stall lest it observe a stale value or incorrectly overwrite state. *Writing* data makes it available for bypassing, and *releasing* the lock represents the actual, in-order commit point.

Table 1 lists a summary of the hazard lock interface and how the PDL compiler restricts its use. For brevity, we often refer to hazard locks as "locks" in the remainder of the paper.

2.3 Refining the Hazard Lock Abstraction

While the lock abstraction allows PDL to reason about whether a design is free of data hazards, different lock implementations have different performance characteristics. PDL is bundled with a small library of lock implementations reflecting different microarchitectural designs; designers can also implement and use their own unique locks in RTL. Here we present the implementations we have developed.

Queue Lock The simplest lock implementation is a First-In-First-Out (FIFO) queue of reservation requests for a given memory location. *Reserve* enqueues a request. *Block* stalls until the associated reservation is at the head of the queue. *Read* and *write* access memory normally. *Release* dequeues the reservation. The implementation refines the specification required by PDL, but assumes we have a separate queue for each memory location: an obvious impracticality for large memory. To efficiently implement queue locks, we provide a fully associative array of queues. In this way, any location can be associated with any queue and is disassociated once the queue is completely empty (and is therefore reusable by another location). The size of the associative array and the depth of the queues are design parameters that may influence performance; for instance, attempting to reserve an unused location when all queues are in use could cause pipeline stalls. This lock represents a simple but low-performance design: it has stall logic but no bypassing paths between conflicting writes and reads.

Bypass Queue To support in-order cores with bypassing, we implemented a lock which commits writes to the memory in reservation order, but allows write values to be bypassed to reads by storing them in a temporary buffer. We implement this lock as a queue of write addresses, values, and valid bits. *Reserve write* enqueues the address and sets the associated valid bit to 0. *Block write* is a no-op, and writes update the data and valid bits of the associated queue entry. *Release* then commits the write to the actual memory.

Reserve read checks for conflicting writes and updates a register with the entry number of the given write. *Block read* stalls until the conflicting write has executed (if there is one), and reading either forwards data from the write or reads directly from the memory. *Release read* frees internal state for future read reservations. This implementation also buffers read data so that access to the memory occurs in the same cycle as reservation, and includes combinational bypass paths so that writes are observable to reads in the same cycle. With this implementation, we can fully bypass a standard 5-stage in-order core.

Renaming Register File We also implemented the lock interface with a renaming register file of the kind used in modern out-of-order processors. A renaming register file maintains a table that maps architectural register addresses (a.k.a. names) to physical names, and stores data in a traditional register file indexed by physical names. Lock *reservation* translates to physical name allocation for writes, and physical name lookup for reads. A vector of per-register valid bits tracks their status: they are set to 0 on allocation, and 1 once data is written. *Block* operations are no-ops for writes and check the appropriate valid bit for reads. *Release* operations are no-ops for reads, but for writes they add the old name mapping to a free list for future allocation. Like the Bypass Queue, this implementation can fully bypass a

Table 2. Speculation operations supported by PDL.

Operation	Description
`s <- spec call pipe(pred)`	Spawn a speculative thread using value `pred`.
`update(s, npred)`	Update the prediction for speculation `s` using `npred`.
`verify(s, real) {pred(...) }`	Mark speculative thread `s` as correctly or mispredicted by comparing the value of `real` to the original prediction; optionally, update external predictor module `pred`.
`spec_check()`	Kill the current thread if it has been mispredicted.
`spec_barrier()`	Stall until this thread’s status is known. If mispredicted, then kill this thread.

5-stage pipeline, although it is also general enough to be a good fit for a Tomasulo-style out-of-order machine [28].

2.4 Speculation

Speculation is critical for processor performance, and PDL enables a large class of speculation through *speculative call* statements. As with locks, PDL offers a modular abstraction for speculative operations, summarized in Table 2. Designers can initiate a speculative thread, mark it as (mis)predicted, update a prediction, and kill speculative threads. In PDL, all speculation is made explicit, even speculation that is often overlooked in processors: the typical `pc + 4` prediction that instructions usually execute sequentially. The following snippet implements this speculation:

```
spec_check(); //Kill this thread if misspeculated
s <- spec call cpu(pc + 4); //Spawn a new thread
```

The **`spec call`** speculatively spawns a new thread with the argument `pc + 4` and produces a handle, `s`, used to later reference this speculation. We refer to the thread making the speculative call as the *parent* thread, and the thread created by the speculative call as the *child* thread. In a pipeline that uses speculation, every thread has the potential to be both a parent *and* a child. For that reason, we use the operation **`spec_check`** to kill the current thread if it is misspeculated. Note that this check does not prevent “nested” speculation (i.e., speculation initiated by an already speculative thread); this check just ensures that *already misspeculated* threads do not continue to speculate.

Eventually, the parent thread needs to *verify* whether its prediction was correct:

```
s <- spec call cpu(pc + 4);
...             //later in the pipeline
spec_barrier(); //blocking version of spec_check()
verify(s, npc); //check that npc == pc + 4
```

The parent thread first ensures that it itself is non-speculative with a blocking version of the speculation check. Then it marks reference `s` with a single bit defining its correctness; PDL automatically propagates the original prediction and inserts a comparison with the given value. In this instance the **`verify`** operation marks `s` as correct if `npc == pc + 4`. If the prediction was *wrong*, the child thread will be killed once it executes a **`spec_check`** or **`spec_barrier`** operation (often in the same cycle). In this case, **`verify`** also causes the parent to spawn a new, non-speculative, thread with the correct value.

PDL also supports an **`update`** operation that can be used to compose both termination and speculation, by spawning a new thread if the new (presumably more accurate) prediction does not match the original and marking the old child thread for termination.

PDL allows predictors to be implemented as modules in RTL safely: predicted values cannot affect functional correctness! Predictor accuracy has significant impact on processor performance, so the ability to integrate custom predictors without compromising PDL’s correctness assurance is critical for efficiency. The following example shows how to use an external branch history table (BHT) for branch prediction:

```
s <- spec call cpu(pc + (bht.req(pc) ? imm : 4));
...
verify(s, npc) { bht.upd(pc, brTaken) }
```

The module `bht` must be declared earlier in the PDL program as externally implemented with a `req` interface that produces a boolean. A true value indicates the branch should be taken; false means it should not. The predictor `bht` also provides an `upd` interface that receives a pc and a “was taken” bit to update its own internal state. Whenever **`verify`** executes, the branch history table is also updated.

Implementation Like locks, speculation in PDL places few restrictions on the structure and timing of the pipeline. To achieve this, PDL stores speculation state in a table, which threads can use to update and read speculative status. Spawning a thread allocates a new identifier. **`verify`** and **`update`** statements mark a child thread’s entry as correct or incorrect. Entries are freed whenever a child thread learns its speculative status through **`spec_check`** and **`spec_barrier`** statements. Importantly, whenever a **`verify`** or **`update`** marks an entry as mispredicted, it also marks *all newer entries* as mispredicted too. In this way, all threads will eventually be notified of their status, even if their parent is killed before it calls **`verify`**.

This table is a straightforward circular buffer, which is also synthesized with combinational bypass paths between the status updates and speculation checks. These paths are necessary when pipelines both speculate and resolve every cycle (the typical case). However, representing this structure as a registered table allows it to function even in loosely timed scenarios, where threads may not check their speculative status in *every* pipeline stage. This implementation requires no strict assumptions about the timing of speculative checks and verification operations, and frees PDL from the in-order requirements of other tools that generate pipeline speculation [18]. This design can contribute to the overhead of PDL when building very simple processors whose control logic can be manually optimized by reasoning about global invariants. Nevertheless, it generalizes to more complex processors which do not broadcast speculation results to every stage.

2.5 Supporting Speculative Reservation

For any given lock implementation, allowing speculative lock reservation could lead to bugs; a thread holding a lock may be killed, leading to an inconsistent lock state. To increase the expressivity of PDL, we extend the lock abstraction with **`checkpoint`** and **`rollback`** primitives that can be used to safely undo speculative lock operations. A **`checkpoint`** causes the lock to logically snapshot its current internal state and returns a handle referencing this snapshot; **`rollback`** indicates that the snapshot is no longer needed, and/or that the lock should revert its internal state to the given snapshot.

Unlike the other lock operations, these need not be exposed to the designer; these operations *must* be executed exactly at certain points in the pipeline, and the compiler can automatically insert them. Specifically, a checkpoint must be taken atomically as each thread completes its reservations; thereby making a checkpoint *between* the reservations of a parent thread and its (speculative) child. Rollback invocations must coincide with **`verify`** and **`update`** operations; whenever a parent terminates its child thread, it should revert all memories to the state that captures the parent's reservations, but not subsequent speculative ones. To illustrate this, we annotate the following snippet where the compiler would invoke **`checkpoint`** and **`rollback`** operations:

```
s <- spec call cpu(pc + 4); //...
if (writerd) { reserve(rf[rd],W); }
//c <- checkpoint(rf);
//take checkpoint after last reservation
...
verify(s, npc); //rollback(rf, c);
//undo speculative ops on rf if misprediction
//release lock state associated with checkpoint
```

Checkpoint Implementations We extend both the BypassQueue and the Renaming Register File with standard rollback mechanisms. The former requires little additional state; the head of the write queue (i.e., most recently reserved write) itself serves as a checkpoint. *Rollback* simply requires invalidating all entries newer than that point and moving back the write queue head. For the rename file, we replicate the mapping table and free list; *rollback* resets the main mapping table and free list to the indicated replica.

3 Sequential Pipeline Behavior

A central draw of PDL is that it allows designers and static analysis tools to describe and reason about pipelines as *sequential programs* that process instructions one at a time. Any PDL program accepted by the compiler can be automatically translated—via a straightforward erasure process—to a sequential program that serves as a specification of correctness, which the pipelined circuit generated by PDL refines.

RTL languages have no such canonical sequentialization; many hardware designs do not implement a sequential specification and thus may exploit the unfettered parallelism of RTL. However, PDL's one-instruction-at-a-time semantics greatly simplify reasoning and can likely alleviate the scalability problems of traditional RTL testing and verification. For instance, to achieve soundness, bounded model checking of RTL pipelines requires considering instruction sequences long enough to saturate the pipeline [20]; applying this technique to PDL programs would only require analyzing sequences of length 1. Sequential software proof techniques, such as Hoare logic [10], which are not easily adapted to RTL languages, can also be applied to sequential PDL programs.

Assumptions PDL's correctness relies on the correctness of the compiler itself, and the RTL implementations of the lock API. Namely, locks must ensure that reads and writes must be stalled (via **`block`**) if they would produce observations inconsistent with the reservation order. We plan to formalize this verification requirement, as well as the correctness of PDL's overall design, in future work.

Locks can be verified using existing hardware verification techniques [3, 13]. However, unlike verifying bypassing networks in RTL processors, locks enable modular verification: they can be verified in isolation, since PDL checks that they are used correctly by the main pipeline. Modularity also implies that locks may be reused across processor designs, amortizing the effort of correctness proofs. Importantly, a verifier (human or tool) only needs to reason about the software-visible architectural state, and *does not* need to supply any global invariants about a pipeline's *microarchitectural* state, a notoriously difficult verification task [5, 33].

3.1 Extracting a Sequential Specification

A PDL pipeline can be understood as a sequential program through a straightforward translation procedure. This program effectively defines the behavior of the given PDL pipeline as a sequence of updates to, and observations of, architectural state. As an example, Figure 3 includes a simple pipeline, its sequential interpretation, and a graph describing the circuit structure. The steps for this translation are straightforward:

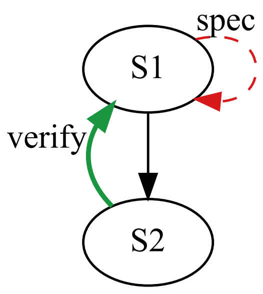
- Erase stage separators, speculation checks, initiation and invalidation, and lock operations.
- Replace **`verify`** statements with **`call`** statements
- Delay memory write and recursive **`call`** statements to the end of the pipeline.

```
pipe ex1(in)[m]: {
  spec_barrier();
  s <- spec call ex1(in + 1);
  reserve(m[in], R);
  acquire(m[in], W);
  m[in] <- in; release(m[in], W);
  ---
  block(m[in], R);
  a1 = m[in]; release(m[in], R);
  verify(s, a1);
}
```

(a) Original PDL Code

```
pipe ex1(in)[m]: {
  a1 = m[in];
  m[in] = in;
  call ex1(a1);
  //Erased ---
  //Erased Spec Operations
  //Erased Lock Operations
  //Delayed Write Operation
  //Replaced verify() with
      tail call
}
```

(b) Code representing sequential semantics



(c) Stage Graph Representing the Pipelined Circuit

Figure 3. Interpretations of a Sample PDL Pipeline

Erasing microarchitecure-controlling primitives is intuitive; by design they should have no impact on the intended functional behavior of the pipeline. Verifying speculation is the exception, as it *does* imply functionality; however, without any speculative events it reduces to unconditionally spawning a child thread (i.e., a recursive **call**).

We also apply reordering transformations on memory writes and recursive **call**s. Memory writes are delayed until after all reads, and recursive **call** statements are moved to the end of the program to become tail calls. In the realized pipeline, the placement of these statements have performance impact (and often placing them earlier is better); functionally, their location in the program should have no impact. For **call** statements, this property is obvious; since **call** initiates the next instruction, its behavior should be sequenced after all of the operations for the current thread. For memory writes, we made a simplifying design decision to declare that their effects are not visible to the current thread. This decision simplifies locks so that they do not need to consider dependencies between reads and writes from the same thread, as no thread may read its own writes. Conveniently, this transformation also produces programs that align with typical ISA semantics; when a location is both read and written by an instruction, the read appears to occur *before* the write.

The obvious operational interpretation of these sequential specifications (such as Figure 3b), yields a definition of correctness for the generated circuit; the effects on memories referenced by the pipeline appear to happen one iteration at a time, in sequence. Thus, each iteration corresponds to a single instruction that may read and write shared memory, and lastly determines which instruction to execute next.

3.2 Informal Correctness of PDL

We briefly justify why the PDL compiler only generates pipelines whose concurrent execution is consistent with the behavior of their sequential interpretation.

Preventing Data Hazards In PDL, locks do the "heavy lifting" to prevent data hazards. As explained in Section 2.3, locks implement stall, bypass, and commit logic for each memory that the pipeline accesses, and expose this logic through the lock interface in Table 1. PDL confirms that this interface is used appropriately, and rejects pipelines in which data hazards could still occur. Specifically, lock implementations need to assume that:

- *Reservations* are made in the intended program order.
- Stages check that *block* returns true before accessing memory.
- *Write locks* are released (committed) in program order.

The PDL compiler rejects any pipeline description in which these three requirements may not be satisfied. To enforce them, it checks that each lock is used in the intended sequence (i.e., *reserve*; *block*; *access*; *release*), and that the *reserve* and *release* operations are guaranteed to execute in thread order. The former is easily checked with a path-sensitive analysis (see Section 4.3). The latter requirement necessitates reasoning about the possible parallelism in the compiled design. PDL does not reason about concrete timing of stage execution. Instead, by examining the structure of the stage graph, it proves that all threads must traverse, in thread order, the stages that contain *reserve* and *release write* statements for a given memory. For example, if all *reserve* statements for a given memory only occur in a single stage, in-order reservation is trivially satisfied. Section 4 discusses how the PDL compiler implements these checks in more detail.

Speculation Correctness We also argue that speculation does not influence the observations of threads in PDL pipelines. PDL guarantees this by validating the following conditions:

- All speculative calls are verified or killed accurately.
- Misspeculated threads are rolled back before committing writes

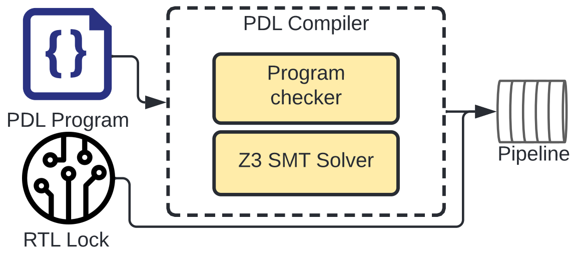


Figure 4. Each PDL program is checked for well-formedness and correct lock usage, relying on Z3 for its path-dependent analyses. If successful, it produces a BSV pipeline. Locks are implemented in RTL and not checked for correctness.

PDL establishes a key invariant that simplifies correctness reasoning around speculation: all speculative calls are resolved *in thread order*. PDL enforces this by restricting **verify** statements to non-speculative threads. If speculation were resolved out of order, then a *verified* thread might still be speculative and PDL would need a more complicated speculation tracking and resolution mechanism. Instead, PDL guarantees that each **verify** statement fully determines the speculative status of its child thread: either it was correctly predicted, or it *and its children* were all misspeculated. Through another path-sensitive analysis, the compiler ensures that all speculation is eventually resolved by the parent.

Intuitively, PDL checks that reservations to write locks (i.e., those that can change the observations of other threads) are rolled back before they can influence non-speculative threads, if they were misspeculated. Since speculation verification happens in order, the rollback event associated with that verification resets all speculatively updated locks to a point *right after the parent's reservations.* Effectively, in addition to terminating all speculative threads, verification signals for locks to undo all speculative modifications to their state. We also require that write locks are not released by speculative threads; this prevents writes from becoming permanent before misspeculation is discovered.

4 Rule Checking

This section expands on the details of the PDL compiler's program checking process which defends the intuition outlined in Section 3. Figure 4 visualizes the end-to-end checking and compilation process.

4.1 Lock Checking

In addition to standard type checking rules, PDL has a unique set of restrictions for locks to ensure that the realized, parallel design accesses locks correctly.

- Locks must be reserved *in thread order.*
- Each thread must use locks in the appropriate sequence: reserve; block; read/write; release.
- Write locks must be released non-speculatively *in thread order.*

The first and third restrictions are checked by proving that all of the reserve and release operations occur during in-order stages of the pipeline. We establish in-orderness by constructing the stage graph for the pipeline; if a stage is ordered with respect to *all* other stages, then it will be executed by threads *in order*. We slightly relax this restriction, allowing these operations for a given memory to occur inside *at most one branch* of an out-of-order region. For instance, in Figure 2 it is safe if all reservations for access to data memory occur in the DMEM stage. Although it is unordered with respect to the DIV stage, DIV does not make any reservations and thus races to reserve locks cannot occur.

Reservations must also happen *atomically*, meaning that thread i makes all of its reservations for a given memory before thread $i+1$ makes any. The PDL compiler ensures atomicity by annotating the start and end of a *lock region*: the set of stages in which the reservations for a given memory occur [3].

```
reserve(m[a], R); //Start Lock Region m
---
reserve(m[b], W); //End Lock Region m
```

The compiler inserts control logic to ensure that only a single thread may execute inside a lock region at a time. However, in practice the region is usually only a single stage and no extra logic is needed or used. The main use of multi-stage reservation is for indirect references:

```
acquire(m[a], R); //Start Lock Region m
b <- m[a];
---
acquire(m[b], W); //End Lock Region m
```

This pipeline cannot reserve all aliasable memory locations in a single stage, because it must read from m before knowing all the addresses to reserve. While this pattern can arise in certain pipelined circuits, it is uncommon for processors. Importantly, *atomicity* is only required for all of the reservations of a given memory; reservations for two different memories may occur in different stages without synchronization penalty, since those reservations cannot possibly alias each other.

4.2 Speculation Checking

PDL limits the use of speculation to ensure that the final design is equivalent to a non-speculative version. First, we check that all speculative calls are eventually verified across any program path; this uses the same machinery as checking

[3]The designer may also manually place these annotations and compiler will check that they actually wrap all of the reservation operations for the given memory.

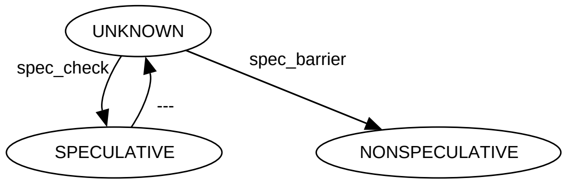


Figure 5. A state machine representation of the typestate used to check speculative status of pipeline threads.

that lock operations are called in the correct sequence and is described in more detail in Section 4.3. Second, PDL prevents speculative effects from being observable by non-speculative threads via restricting the set of operations that speculative threads may execute. We adapt typestate [26] to determine the speculative status of threads in any given stage. Threads can transition between three states: `Unknown`, `Speculative`, and `Nonspeculative`[4], beginning in state `Unknown` and using the speculation primitives to transition to other states. Figure 5 illustrates the relationship between typestates and these primitives.

The non-blocking check **`spec_check`** transitions to `Speculative`, only establishing that the thread is *not definitely misspeculated.* After a stage separator, if `Speculative`, the typestate is reset to `Unknown` since its status may have been resolved by the time the thread executes the next stage. The only way to establish that a thread is `Nonspeculative` is to use a blocking check **`spec_barrier`**. `Unknown` threads may not make speculative calls or reserve locks, as these operations (if made by an already misspeculated thread) could cause races on starting new threads, and inconsistent lock state, respectively. Neither `Unknown` nor `Speculative` threads may **`verify`** their own speculation or release write locks, lest they permanently update lock state and write data that some non-speculative thread may read before they are rolled back.

4.3 Path-Sensitive Checking

PDL cannot rely on purely syntactic type checking to prove that locks transition through the correct sequence of states. Since the placement of operations and stage separators inside conditional branches can influence the structure of the pipeline, it is important to allow flexible placement of lock operations. For example, a purely syntactic type checker could not prove the following code snippet reserves the lock before blocking on it:

```
if (writerd) { reserve(rf[rd], W); }
---
if (writerd) {
  block(rf[rd]); rf[rd] <- wdata; release(rf[rd]);
}
```

[4]We do not need a `Misspeculated` state since misspeculated threads will automatically be terminated and will not execute code following the speculation check operations.

To permit such programs, we generate constraints to prove that locks are in the necessary state when they are used. The compiler runs an abstract interpretation over the program that approximates branch conditions using variable equality and boolean logic. To allow the compiler to precisely check lock usage, designers need only to simplify branch conditions into booleans or comparisons between variables. We then employ the Z3 SMT solver [8] to verify that the constraints are satisfied. The same code snippet follows, annotated with the information derived by our compiler and checked by Z3:

```
if (writerd) { //LockState: free
  reserve(rf[rd], W);
}              //Lockstate: writerd => reserved
               //          ^ !writerd => free
---
if (writerd) {
  block(rf[rd]); rf[rd] <- wdata; release(rf[rd]);
}             //Lockstate:  writerd => released
              //          ^ !writerd => free
```

PDL also uses this technique to confirm that speculative calls are resolved and that pipeline **`call`** statements are well-formed: each thread *either* makes a single recursive call, *or* outputs a value. In other words, each thread may either spawn a single child thread or terminate.

5 Implementation

Our prototype implementation of the PDL compiler is written in 10K lines of Scala; the lock implementations are written in 1.7K lines of Bluespec System Verilog (BSV) [17] and 1K lines of Verilog[5]. Given a PDL source program, the PDL compiler produces a BSV module that implements the specified design. From this module, the open-source BSV simulator and compiler can be used to run simulations or produce synthesizable Verilog. We chose BSV as a target language since it provides a natural translation from PDL stages to BSV rules. While this choice simplified code generation, it is not a fundamental requirement; it is certainly possible to implement a different back end for the PDL compiler targeting Verilog or another similar HDL. We implement locks in a combination of BSV and Verilog; the Queue Lock is implemented in BSV and the others are written in Verilog. The language choice at this level is for convenience; in principle, locks can be implemented in any RTL language.

5.1 Code Generation

In BSV, each rule is guaranteed to execute atomically in a single clock cycle; additionally, one can provide conditions that prevent BSV rules from executing. Given these conditional rules, BSV will automatically generate the control logic necessary to execute as many each cycle as possible. The PDL compiler's strategy is to represent each pipeline

[5]The PDL compiler is open-source and can be found at: https://github.com/apl-cornell/PDL.

Table 3. Performance in Cycles-Per-Instruction of multiple processor configurations on a selection of integer benchmark kernels. All processors implement the RV32I ISA, except for the PDL 5Stg RV32IM configuration.

Processor	coremark	aes	gemm	gemm-block	ellpack	kmp	nw	queue	radix	GeoMean
Sodor	1.441	1.201	1.530	1.525	1.380	1.496	1.355	1.332	1.282	1.37
PDL 5Stg	1.436	1.230	1.529	1.525	1.380	1.496	1.376	1.332	1.282	1.39
PDL 3Stg	1.205	1.101	1.265	1.262	1.190	1.247	1.188	1.118	1.108	1.18
PDL 5Stg BHT	1.367	1.154	1.413	1.414	1.269	1.255	1.306	1.231	1.202	1.28
PDL 5Stg RV32IM	1.384	1.230	1.421	1.226	1.280	1.496	1.376	1.332	1.282	1.32

stage as a single BSV rule, and to supply conditions to *stall* or *kill* a stage's execution when necessary, according to the appropriate PDL primitives.

We split the original PDL program into a DAG of pipeline stages as described in Section 2.1. Live variable analysis is used to annotate the edges between the stages with all variables needed by a later stage. Each stage translates to a single BSV rule, guaranteeing that all of its state-modifying operations occur in the same cycle. Each edge translates to a FIFO which stores the data communicated between stages. The FIFO is an abstraction over pipeline registers; the current compiler uses the default BSV FIFO implementation (which employs 2 registers), but it could be replaced with a single-register implementation. The only exception to this generation mechanism are the coordination edges generated to control out-of-order regions of the graph; these send only a single value, used by the downstream stage to determine which other input FIFOs to read from.

The combinational logic associated with each stage can be generated in a straightforward fashion and placed outside of the rules. The rule bodies contain all of the state-modifying or inter-stage operations: FIFO en/dequeues, lock operations, memory writes (and reads for synchronous memories), and updating speculation status. Lastly, we generate PDL's stall conditions to prevent BSV from scheduling rules erroneously. BSV automatically ensures rules do not execute when there is no valid data or there is back pressure from a later stage (i.e., the input FIFO is empty or the downstream one is full). Thus we only need to add stall conditions for `spec_check`, `spec_barrier`, and `block` commands and for stages that receive data from a variable-latency operation (e.g., responses from synchronous memories or other PDL pipelines).

BSV automatically generates a schedule that executes as many rules as possible within a single clock cycle; this corresponds to the control logic for stage activation. PDL does not guarantee that this schedule is optimal. PDL does automatically include two scheduling directives necessary for high performance. One indicates to BSV that it is safe to execute all stages that send data to the beginning of the pipeline (i.e., recursive `call` and `verify` statements) and appropriately muxes the correct values based on misprediction logic. The second ensures that BSV will make speculation results combinationally available to earlier stages (i.e., the misprediction signal is propagated immediately to early stages that contain `spec_check` or `spec_barrier` statements). With these directives, PDL can generate speculative pipelines that execute one instruction per cycle.

6 Evaluation

To demonstrate PDL's expressivity and efficiency, we present a number of different implementations of the RISC-V32 ISA, and compare their performance and area to a baseline implemented in Chisel [2]. To measure software-visible performance (cycles per instruction—CPI), we use RTL simulations for the designs that simulate cache hits for every memory access (single-cycle responses). To measure processor area, we target a 100 MHz clock frequency using 45nm FreePDK [25] technology and execute synthesis and place-and-route. In our measurements we consider only the processor cores and exclude caches and any other parts of the memory hierarchy since PDL was not used to generate that part of the microarchitecture.

We compare the PDL-designed processors with the Sodor processor in its fully bypassed configuration. Sodor is implemented in Chisel and represents a standard 5 stage RV32I processor [21]. First, we show that our processors can implement a similar architecture. The PDL 5 Stage processor divides functionality across stages in the same way as Sodor and uses the Bypass Queue lock (see Section 2.3) to bypass write data. The PDL processor also implements the same speculation logic as Sodor, always predicting that branches are not taken and suffering a 2-cycle stall on taken branches and jumps. Both processors also experience the same stalls for data hazards thanks to their bypassed designs; they stall for 1 cycle, but only on load–use dependencies[6].

The first two rows of Table 3 present the CPI of the designs when executing a number of small benchmarks. The first benchmark, coremark, is from the eembc [7] embedded benchmark suite. The remainder of the benchmarks are the selection of integer kernels from the MachSuite [19] that we could successfully execute on the Sodor processor. There is a small variance between Sodor and PDL 5Stg, especially in

[6]A load instruction whose value is used by the subsequent instruction.

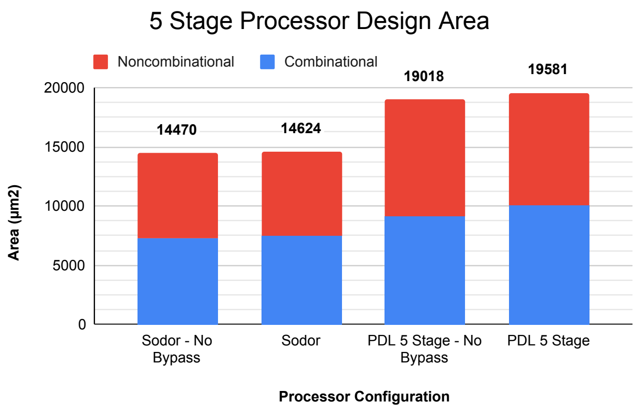


Figure 6. Design cell area for 5 Stage processors both with and without bypassing logic. Sodor is a baseline implemented in Chisel. Results achieved using 45nm technology targeting a 100 MHz clock frequency.

the aes benchmark; this is the result of a minor difference in the benchmark binaries, which was required to be compatible with the test benches. We manually confirmed that exactly the same stalls occurred in both pipelines and that this CPI difference does not signal a difference in processor performance.

Figure 6 shows the synthesis results for the Sodor and PDL 5 stage processors. We also include the areas for non-bypassed versions of both processors to measure the overhead caused by including bypassing logic. The PDL processor requires more area than the Sodor processor, and bypassing induces a larger percentage overhead than in Sodor (2.96% increase vs. 1.06%). Some of this area is due to less efficient stall logic and pipeline register representation. Specifically, the FIFO implementations consume significant combinational and non-combinational area. These overheads are all artifacts of the immaturity of the PDL compiler, and are not fundamental to PDL's language design. Bypassing, on the other hand, is more expensive in PDL than in the hand-written version, because PDL assumes nothing about the timing or coordination of stages and pays for this generality. In particular, the Bypass Queue requires a dynamic priority calculation to determine which write is the most recent, and stores some information redundant with data in pipeline registers.

Nevertheless, we are not concerned with this overhead for two reasons. First, these cores are small and thus cache areas would likely dominate costs in a complete chip. We used CACTI [30] to estimate the area overhead when using even tiny (4KB, 2-way associative) L1 data and instruction caches. For this configuration, PDL induces only a 5% overhead when considering the total area of core and L1 caches together; this provides an upper bound on chip area overhead, as real systems often use significantly larger caches. Second, in designs that support any out-of-order execution (even mostly in-order CPUs such as Ariane [31]), the complexity of PDL locks is required by the implementation. The bypass logic in Sodor is simple because the designer can statically know which stages might contain the bypass data *and* in which order to prioritize them. As soon as out-of-order behavior is introduced, a dynamic mechanism (such as those implemented in the PDL Bypass Queue and Renaming Register File) becomes required to correctly forward write data.

6.1 Expressivity

Lastly, we highlight the expressivity of, and design exploration enabled by, PDL. In addition to the 5 stage RV32I processor, we present:

- A 3-stage RV32I core
- A 5-stage RV32I core with a branch history table
- An RV32IM core with parallel, pipelined multiplication and division units

To demonstrate their microarchitectural differences, Table 3 also contains performances results for these three processors. Deriving these other designs from the original required far less effort than in a conventional design process. Reducing the number of stages from 5 to 3 required eliminating two stage separators, and modifying read locks to be reserved and acquired in the same cycle; we used a slightly simplified version of the Bypass Queue to support this efficiently. Adding a custom branch predictor required almost no change to the PDL design since the same speculation primitives can be linked with an external, RTL-implemented predictor. This did involve changing some logic to `update` predictions in the second pipeline stage once we determined an instruction was a branch. We needed to modify only about 20 lines of code from the original 5-stage design to implement each of these microarchitectures.

The RV32IM processor required noticeably more effort—it introduced new functionality and made the pipeline structure not fully linear. Similar to the design of the Ariane [31] processor, the execute stages of this processor are split based on functional unit (multiply, divide, ALU/memory), can execute in parallel, and write back data out of order (when using the Bypass Queue or Renaming Register File locks). The multi-cycle divider and multiplier are both also implemented as PDL pipelines. The former computes 1 bit of an integer division per cycle and supports only 1 concurrent operation; the latter takes two stages to implement integer multiplication and is fully pipelined, supporting 2 concurrent operations. Together, these were written in 32 lines of PDL code. A non-linear pipeline allows all execution units to run in parallel without increasing pipeline depth, providing a slight CPI benefit over a 5-stage in-order CPU, and this structure reflects the microarchitecture necessary for high performance in deeper and/or wider pipelines. Modifying

```
pipe cache(addr, dataIn, isWr)[entry, main]: int {
  idx = getIdx(addr);
  acquire(entry[idx], R);
  cline = entry[idx]; release(entry[idx]);
  hit = //cline is valid and matches addr
  if (!hit || isWr) { reserve(entry[idx], W); }
  if (hit || isWr) {
    dout = //cline data for rd, else default val
    output(dout); //enqueue response
  }
  maddr = alignAddr(addr);
  if(!hit) { newline <- main[maddr]; }
  ---
  if (!hit || isWr) { //update cache entry
    newCline = //construct from newline and data
    block(entry[idx]);
    entry[idx] <- newCline; release(entry[idx]);
  }
  //queue response for cache miss
  if (!hit && !isWr) {
    output shiftOut(newline, getOffset(addr));
}}
```

Figure 7. Abbreviated PDL code for a direct mapped, write-allocate, write-through cache.

the decode logic to support these new instructions and altering the pipeline structure required an additional 30 lines of PDL code.

Non-processor Designs PDL is not inherently limited to building processors; as described above, the multiplier and divider in our RV32IM CPU were also implemented in PDL as pipelines to which the main CPU issued requests. Furthermore, PDL can express pipelined modules that carry their own state, and guarantees that requests make updates and observations to that state in order. To demonstrate this feasibility, we built a 2-stage direct-mapped data cache with a write-allocate, write-through policy, in PDL. An abbreviated version of the cache is shown in Figure 7. The cache contains two memory references, its `entry` array of cache lines, which would typically be implemented with SRAM, and a `main` interface to DRAM. The first stage is responsible for reading the appropriate cache line and issuing a request to main memory on a cache miss, or responding to the "caller" on a hit. The second stage waits for the response from main memory and then updates the appropriate cache line. We use the PDL Queue Lock to protect the data cache entries, effectively stalling concurrent accesses to the same cache line. We expressed this design in about 50 lines of PDL code.

7 Future Work

PDL provides the foundation for a new methodology of processor development that can provide high-level semantics as well as low-level control over hardware design. One potential opportunity enabled by PDL is to explore the development of high-assurance microcontrollers for safety-critical systems, such as pacemakers [24]. While PDL's *current* features are not extensive enough to implement a modern high-performance processor, it should enable rapid but safe microcontroller development.

We also believe the PDL approach of abstracting common microarchitectural structures can be extended to support more advanced designs. For instance, PDL currently only allows out-of-order execution inside branches; new extensions that abstract reorder buffers and instruction schedulers could enable instruction reordering at any point in the pipeline. Another potential extension would be to generalize speculation from branch prediction to arbitrary value speculation.

To increase confidence in PDL's claims, the semantics and correctness guarantees of PDL could be formalized precisely, and the compilation strategy (or compiler itself) could be proven correct. We also believe that PDL's correctness guarantees could be extended to provide *security* guarantees. PDL's explicit treatment of speculation as a language construct may allow simpler reasoning about hardware speculation security properties, such as strictness-ordering [1] or non-interference [32] of speculative state.

8 Related Work

An old but short line of work aims to automatically generate pipelined processors from sequential specifications. Paul and Kroening [14] generated the stall and forwarding logic for an ordered list of stages with register assignments and combinational logic. Similarly, Nurvitadhi et al. [18] translate "transactional specifications" into pipelines; their work supports speculation and allows developers to selectively enable bypass paths via an iterative design tool. More recently, Liu et al. [15] demonstrated, with their ASSIST framework, how to synthesize high-performance, customized RISC architectures from a micro-op language. All of these projects are limited to strictly in-order pipelines, and can only generate specific implementations of speculation and bypassing.

Although the ASSIST framework can search through different timings with varied number of stages and bypass paths, it operates at a higher level of abstraction than PDL; the designer has no control over pipeline organization or optimization. This design makes autotuning tractable, but greatly limits the space of possible processor designs. PDL, on the other hand, admits more general pipeline DAGs that do not require fully in-order execution, and is a language, rather than a one-off tool. This enables static analysis and other techniques to improve PDL and provide further guarantees about PDL pipelines. PDL's hazard locks and speculation API offer more flexibility, demonstrating that processor components with a variety of implementations can be abstracted behind a single checkable interface.

TL-Verilog [11] is a language for designing pipelines with abstract timing. As with PDL's stage separators, TL-Verilog allows the designer to split combinational logic into sequences of stages with annotations. However, TL-Verilog's focus is on ensuring an equivalent semantics between the abstractly timed design and the physically timed implementation. TL-Verilog provides designers with low-level control of timing but unlike PDL, does not prevent data hazards.

BSV [17], Koika [4] and the BSV-based Kami [5] are considered high-level HDLs, because their transactional "one-rule-at-a-time" semantics can simplify correctness reasoning. Rules in these languages must execute in a single cycle, and thus their atomicity guarantees cannot automatically provide the *one-instruction-at-a-time* semantics of PDL; in particular, their compilers cannot automatically detect data hazards like PDL can. However, PDL is targeted more specifically at pipeline development, and thus these languages are more general-purpose.

High-level synthesis (HLS) tools [6] might appear similar but aim to solve a very different problem: automatically generating a timed hardware implementation from a sequential, untimed algorithm description. Because HLS primarily focuses on statically scheduling dataflow operations across hardware resources, it is unsuitable for synthesizing processors, which exhibit dynamic data dependencies. There are some recent examples of simple HLS processors [23], but these require the designer to explicitly denote bypassing logic as if they were writing RTL. Researchers have recently proposed using dynamic scheduling to improve the performance of HLS pipelines [12], using load–store queues to both enforce dynamic data dependencies and schedule memory operations. Unlike all of these tools, which rely on automatic scheduling, PDL gives hardware designers direct control over pipeline design and timing. Additionally, PDL supports integration with custom RTL implementations for breaking data dependencies via its lock API, rather than requiring a fixed implementation.

Type systems have been used in recent HLS languages [6] to provide other forms of static guarantees, focusing on *performance* rather than correctness. Dahlia [16] uses affine types to ensure *predictable* performance and to avoid synthesizing complex arbitration logic. Aetherling [9] automatically compiles data-parallel programs to streaming accelerators, using a type-directed search for hardware scheduling. Both of these tools target development of statically scheduled hardware accelerators rather than processors.

The Jade [22] language for distributed computing contains primitives similar to PDL's hazard lock `reserve()`. While they are also used to manage concurrent access to shared state, Jade locks do not support speculation and are not statically checked for correct use.

9 Conclusion

PDL is a *Pipeline Description Language* that raises the abstraction of RTL to provide one-instruction-at-a-time semantics while enabling efficient, parallel execution by modularizing bypassing and speculation logic. PDL still gives architects low-level control over the microarchitecture and timing of their processor, and is compatible with RTL implementations of modules for branch prediction and hazard resolution. We have shown that a variety of RISC-V microarchitectures can be implemented in PDL with acceptable overhead. Through its flexible abstractions for hazard resolution and speculation, PDL promises to ease the burden of processor verification while still allowing out-of-order and speculative microarchitectures.

Acknowledgments

This work was supported by the Department of Defense (DoD) through the National Defense Science & Engineering Graduate (NDSEG) Fellowship Program, and by NSF grants 1513797 and 1717554. We are grateful to Coşku Acay, Ryan Doenges, Rolph Recto, Rachit Nigam, Adrian Sampson, our reviewers, and our shepherd, James Bornholt, for their invaluable feedback on the paper. We also appreciate Kevin Zhang's contributions to the early development and testing of the PDL compiler. Finally we would like to thank Muhammad Umar and Coşku Acay for their assistance in evaluating the processor implementations and preparing the software artifact for evaluation, respectively.

References

[1] Sam Ainsworth. 2021. GhostMinion: A strictness-ordered cache system for Spectre mitigation. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*. Association for Computing Machinery, 592–606. https://doi.org/10.1145/3466752.3480074

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*. IEEE, 1212–1221.

[3] Gregor V Bochmann. 1982. Hardware specification with temporal logic: An example. *IEEE Transactions on Computers* 31, 03 (1982), 223–231.

[4] Thomas Bourgeat, Clément Pit-Claudel, and Adam Chlipala. 2020. The Essence of Bluespec: A core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 243–257.

[5] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, and Adam Chlipala. 2017. Kami: A platform for high-level parametric hardware specification and its modular verification. In *Int'l Conf on Functional Programming (ICFP)*. 1–30.

[6] Cong, Jason and Liu, Bin and Neuendorffer, Stephen and Noguera, Juanjo and Vissers, Kees and Zhang, Zhiru. 2011. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491.

[7] Embedded Microprocessor Benchmark Consortium. 2021. EEMBC. https://www.eembc.org/coremark/index.php. Accessed: 2021-08-01.

[8] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th Int'l*

Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Springer-Verlag, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[9] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-directed scheduling of streaming accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 408–422.

[10] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[11] Steven F Hoover. 2017. Timing-abstract circuit design in transaction-level Verilog. In *2017 IEEE Int'l Conf. on Computer Design (ICCD)*. IEEE, 525–532.

[12] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically scheduled high-level synthesis. In *2018 ACM/SIGDA Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*. 127–136.

[13] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, et al. 2009. Replacing testing with formal verification in Intel® CoreTM i7 processor execution engine validation. In *Int'l Conf. on Computer Aided Verification (CAV)*. 414–429.

[14] Daniel Kroening and Wolfgang J. Paul. 2001. Automated pipeline design. In *38th annual Design Automation Conf. (DAC)*. 810–815.

[15] Gai Liu, Joseph Primmer, and Zhiru Zhang. 2019. Rapid generation of high-quality RISC-V processors from functional instruction set specifications. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[16] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable accelerator design with time-sensitive affine types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 393–407.

[17] R. Nikhil. 2004. Bluespec System Verilog: Efficient, correct RTL from high level specifications. In *ACM and IEEE Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*. 69–70.

[18] Eriko Nurvitadhi, James C Hoe, Timothy Kam, and Shih-Lien L Lu. 2011. Automatic pipelining from transactional datapath specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 3 (2011), 441–454.

[19] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. MachSuite: Benchmarks for accelerator design and customized architectures. In *Proceedings of the IEEE International Symposium on Workload Characterization*. Raleigh, North Carolina.

[20] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. 2016. End-to-end verification of processors with ISA-Formal. In *Int'l Conf. on Computer Aided Verification (CAV)*. 42–58.

[21] Berkeley Architecture Research. 2021. Sodor Core. https://chipyard.readthedocs.io/en/dev/Generators/Sodor.html. Accessed: 2021-08-01.

[22] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. 1993. Jade: A high-level, machine-independent language for parallel programming. *Computer* 26, 6 (1993), 28–38.

[23] Simon Rokicki, Davide Pala, Joseph Paturel, and Olivier Sentieys. 2019. What you simulate is what you synthesize: Designing a processor core from C++ specifications. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[24] Mostafa Sayahkarajy, E Supriyanto, MH Satria, and Hasri Samion. 2017. Design of a microcontroller-based artificial pacemaker: An internal pacing device. In *2017 International Conference on Robotics, Automation and Sciences (ICORAS)*. IEEE, 1–5.

[25] James E Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W Rhett Davis, Paul D Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, et al. 2007. FreePDK: An open-source variation-aware design kit. In *2007 IEEE international conference on Microelectronic Systems Education (MSE'07)*. IEEE, 173–174.

[26] Robert E. Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering (TSE)* 12, 1 (Jan. 1986), 157–171.

[27] Donald Thomas and Philip Moorby. 2008. *The Verilog® hardware description language*. Springer Science & Business Media.

[28] Robert M Tomasulo. 1967. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development* 11, 1 (1967), 25–33.

[29] Andrew Waterman and Krste Asanović. 2017. The RISC-V instruction set manual, volume I: User-level ISA, Version 2.2. (2017).

[30] Steven JE Wilton and Norman P Jouppi. 1996. CACTI: An enhanced cache access and cycle time model. *IEEE Journal of solid-state circuits* 31, 5 (1996), 677–688.

[31] F. Zaruba and L. Benini. 2019. The Cost of application-class processing: Energy and performance analysis of a Linux-ready 1.7-GHz 64-Bit RISC-V core in 22-nm FDSOI technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 11 (Nov 2019), 2629–2640. https://doi.org/10.1109/TVLSI.2019.2926114

[32] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 503–516. http://www.cs.cornell.edu/andru/papers/asplos15

[33] Hongce Zhang, Weikun Yang, Grigory Fedyukovich, Aarti Gupta, and Sharad Malik. 2020. Synthesizing environment invariants for modular hardware verification. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 202–225.