Lucid: A Language for Control in the Data Plane

John Sonchack
Princeton University
jsonch@princeton.edu

Devon Loehr
Princeton University
dloehr@princeton.edu

Jennifer Rexford
Princeton University
jrex@cs.princeton.edu

David Walker
Princeton University
dpw@cs.princeton.edu

ABSTRACT

Programmable switch hardware makes it possible to move fine-grained control logic inside the network data plane, improving performance for a wide range of applications. However, applications with integrated control are inherently hard to write in existing data-plane programming languages such as P4. In this paper we present Lucid, a language that raises the level of abstraction for putting control functionality in the data plane. Lucid introduces abstractions that make it easy to write sophisticated data-plane applications with interleaved packet-handling and control logic, a specialized type system and syntactic constraints that prevent programmer bugs related to data-plane state, and an open-sourced compiler that translates Lucid programs into P4 optimized for the Intel Tofino. These features make Lucid general and easy to use, as we demonstrate by writing a suite of ten different data-plane applications in Lucid. Working prototypes take well under an hour to write, even for a programmer without prior Tofino experience, have around 10x fewer lines of code compared to P4, and compile efficiently to real hardware. In a case study of a stateful firewall written in Lucid, we find that moving control from a server onto a programmable switch using Lucid reduces the latency of performance-sensitive operations by over 300X.

CCS CONCEPTS

• **Networks** → **Programmable networks**; **Network dynamics**; • **Computer systems organization** → *Reconfigurable computing*; *Pipeline computing*; • **Software and its engineering** → *Distributed programming languages*; *Concurrent programming languages*; **Abstraction, modeling and modularity**.

KEYWORDS

network control, data plane programming abstractions, syntactic constraints, ordered type-and-effect system

ACM Reference Format:
John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. 2021. Lucid: A Language for Control in the Data Plane. In *ACM SIGCOMM 2021 Conference (SIGCOMM '21), August 23–28, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3452296.3472903

SIGCOMM '21, August 23–28, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8383-7/21/08…$15.00
https://doi.org/10.1145/3452296.3472903

1 INTRODUCTION

In the early days of Software-Defined Networking (SDN), controller applications changed network behavior by updating the match-action rules that switches use to forward packets. Unfortunately, many interesting applications needed the switches to direct packets to the controller (*e.g.*, to learn about new flows and install new rules in response), causing latency, overhead, and security vulnerabilities. In addition, writing these applications was tricky, since programmers had to reason about possible inconsistencies between the switches and the controller due to delays in installing new rules. As a result, few of these dynamic controller applications saw any significant deployment in practice.

The emergence of programmable data-plane hardware has the potential to change all that, by making it possible to move fine-grained control logic into the forwarding engines of individual switches. Modern hardware, *i.e.*, a PISA pipeline [5], supports not only flexible parsing and manipulation of packets, but also updates to persistent state (such as register memory) that can be used to affect the handling of future traffic.

Consider, for example, a stateful firewall that protects an enterprise from unsolicited traffic. By default, packets sent by external hosts are dropped. Upon receiving outbound packets, the switch state is updated to permit return traffic from the destination; after a period of inactivity, the switch returns to dropping such packets. Having a controller handle these "events"—the arrival of the first packet and the timeout after inactivity—introduces latency and overhead, and the subtle risk that return traffic starts arriving before the data plane is updated to permit it. Implementing this logic directly in the data plane reduces reaction time, and avoids the need for synchronization between controller and data plane.

The stateful firewall is just one of many examples. Figure 1 presents several other applications, along with their state and their data-plane and control-plane components. Past researchers have demonstrated that many of these applications, including load balancers [1, 15, 18], routers [15], and telemetry systems [30] benefit substantially from data-plane implementations.

Despite this, writing applications that do network control inside of PISA data planes is incredibly difficult. Languages like P4 offer the abstraction of single-threaded packet processing that does little (if any) state management. However, applications like the stateful firewall require multiple threads, one for packet handling (*e.g.*,

Lucid is available at: https://github.com/princetonUniversity/lucid

Application	Data Plane Operations	Control Plane Operations	Shared State
Router	get next hop	update route	forwarding database
Fault-Tolerant Router	get online path	probe link, reroute	link and route statistics
MAC Learner	get output port	learn, age, STP	MAC table
Flow Load Balancer	count packets, get path	set path	flow stats, paths
Flowlet Load Balancer	set, get path	age flowlets	flowlet hash table
NAT	translate address	allocate, pin, free	address map and pool
TCP Migration	adjust ack number	migrate flow	ack offset table
Stateful Firewall	check flow	add flow	allowed flow set
Probabilistic Firewall	check, add flow	rotate (age)	Bloom filters
Distributed Probabilistic FW	check flow	add, age, sync	replicated Bloom filters
Sketch-based Query	update sketch	reset, decode	approximate data structure
Telemetry Cache	append packet record	evict, free	per-flow log

Figure 1: Example applications in groups: (1) Routing, (2) Connection-oriented services, (3) Security, and (4) Telemetry.

forwarding permitted packets and dropping the rest) and several more for control operations that manage local state (*e.g.*, updating the rules in response to both flow arrivals and timeouts).

A programmer's first challenge is simply expressing control with packet processing abstractions, which requires using disjoint and low-level primitives such as parsers, match-action units, and packet recirculation. For more sophisticated control, such as distributed route computation or a delayed scan for inactive flows, programmers must also carefully orchestrate PISA components that are outside the scope of data-plane languages, such as programmable queues and packet generators.

Beyond this, there is a second equally daunting challenge: operating on persistent state (registers) in the data plane. The demands of line-rate constrain the Arithmetic Logic Units (ALUs) that operate on registers in all sorts of nuanced ways. Control operations can be complex and run up against these limits. Further, a PISA processor is a pipeline where each register is associated with a single stage. Multiple threads of control that share state must always access the underlying registers in a consistent order.

None of the above constraints are enforced in the stateful primitives of data-plane languages. So when compilation fails because of inevitable programmer error, the failure often occurs in a target-specific backend that is ill-equiped to provide meaningful source-level programmer feedback. Faced with this, the programmer must resort to a painful trial-and-error process of rewriting the program and grappling with cryptic compiler errors, never sure when the compiler will finally yield to the next tweak of the program.

Lucid: Simple, Event-driven Data-Plane Programming. This paper introduces Lucid, a high-level programming language for implementing control applications in PISA data planes.

Lucid programs are organized as multiple collaborating components located either on a single switch or distributed across many switches in a network. In Lucid, programmers program with high-level, abstract *events* and *handlers*. Each event is named and carries user-specified data, while its associated handler defines the atomic stateful computation to perform when an event occurs. An event could be a packet to process, a request to install a firewall entry, or a probe from a neighboring switch to report a link's status.

Events are also associated with *times* and *locations* to facilitate coordination between control operations among different switches, possibly with a delay. Programmers can write sophisticated control logic without having to worry about the low-level details of custom packet formats, parsers and deparsers, or having to "roll their own" mechanisms for buffering and delayed information processing.

Lucid's event-based abstractions for structuring applications and coordinating control are complemented by a careful "correct-by-construction" approach to stateful operations. Rather than allow programmers to write event handler code that operates on state in arbitrary ways, but may produce arbitrary failures in a PISA compiler's backend, Lucid introduces a persistent array abstraction that is carefully designed to rule out illegal constructions. This abstraction is supported by domain-specific syntactic constraints and a novel type system:

- **Syntactic constraints**: We design a sublanguage of *memops*, stateful operations that can execute in a single ALU of a PISA switch. Memop definitions that cannot fit in a single ALU are rejected, and source-level error messages point out exactly where any such mistakes occur, making it easier for programmers to understand how and why they must change the processing of an individual control operation.
- **Types and effects**: We develop a novel *ordered type-and-effect system* that limits the way programs interact with persistent memory. Our type system tracks the order in which handlers access registers and provides actionable source-level feedback when there are inconsistencies. This feedback helps programmers quickly identify control operations that must be changed (*e.g.*, decomposed into multiple simpler operations).

Together, Lucid's event-based abstractions and carefully-designed stateful interface give programmers a natural and modular way to express data-plane applications that interleave packet processing with ongoing control operations *and* help them navigate the difficulty of programming complex switch hardware.

The Lucid compiler shows how to map the above ideas to real hardware—the Intel Tofino. Our compiler analyzes Lucid programs and translates valid programs into Tofino-compatible `P4_16`. It also optimizes them to reduce pipeline resource requirements.

We evaluate Lucid by implementing a diverse set of applications including a stateful firewall, fault tolerant router, self-driving DNS protection service, telemetry cache, and more. All programs had 5-10X fewer lines of code than their P4 equivalents and, due to the Lucid compiler's optimizations, utilize the Tofino's limited pipeline stages efficiently. In writing these applications, we find that Lucid enables high developer productivity and presents a low barrier of entry to high-speed data-plane programming. Several applications were written by a PhD student who had never worked with the Tofino before (one of the authors, who worked only on the language front end). They used Lucid to implement a variety of interesting prototype applications, all in well under an hour. In our experience,

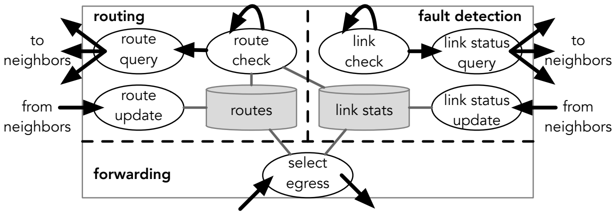

Figure 2: fast rerouter architecture. Circles represent operations, with arrows for (possibly delayed) control flow. Shaded objects are data structures.

it often takes students days or even weeks of debugging to get equivalent programs written in P4 to compile and use resources efficiently.

Finally, a case study of a stateful firewall written in Lucid demonstrates the performance benefit of integrating latency-sensitive network control in the data plane. We find that flow installation time is 300X lower for data-plane integrated control, compared to remote control from the switch CPU.

Summary. The main contribution of this paper is the design of Lucid, a high-level language for stateful, distributed data-plane programming. More specifically, this paper:

- motivates data-plane integrated control and identifies the enabling mechanisms in PISA processors (Section 2);
- introduces event-based abstractions that naturally generalize both packet and control processing (Section 3);
- designs an interface to data-plane state that uses syntactic constraints and a novel type system to identify programs ill-suited for the underlying hardware (Sections 4 and 5);
- describes an open-source, optimizing compiler targeting the Intel Tofino (Section 6); and
- evaluates Lucid, demonstrating that it is general, easy to use, and compiles efficiently to real hardware (Section 7).

Ethics. All user data in this work are from paper authors.

2 INTEGRATED DATA-PLANE CONTROL

Many network services benefit from *integrated data-plane control*, *i.e.*, the placement of control operations in the data plane's packet processing hardware rather than in servers or switch management CPUs. In this section, we illustrate the benefits of integrated data-plane control, and the enabling hardware mechanisms, with a driving example: the *fast rerouter*, a fault-tolerant forwarder that detects and dynamically routes around failed links.

The fast rerouter (see Figure 2) consists of three key components.
Forwarding. The fast rerouter looks up a next hop for each packet in an associative array based on its destination. Before forwarding, the program checks a second data structure to determine if the next hop is still reachable. If not, it may trigger rerouting.
Fault detection. Concurrent with forwarding, a fast rerouter node also regularly pings all of its directly connected neighbors to determine if they are still reachable.
Rerouting. Interleaved with the above, a reroute operation in the fast rerouter queries all of its neighbors to find the next hop with the lowest route length. This can be triggered by a packet with no next hop or a periodic route table scan.

2.1 Motivation: Low Latency Control

Low latency is a primary motivator for data-plane integrated control in many applications.

- **Fault tolerance services**, such as the fast rerouter and F10 [20], detect and mitigate failures in the data plane to minimize reaction time and therefore disruption to traffic.
- In **5G mobile cores** [27], integrating signal handling operations into the data plane reduces latency by up to 98%, enabling faster connection setup and migration for users.
- **Load balancers** [1, 18] with control loops in the data plane can react faster to congestion events. This, in turn, improves end-to-end application performance.
- **Security services** that operate on flows, such as DDoS defense systems [21] and stateful firewalls (Section 7.4), integrate control into the data plane to block threats or authorize trusted flows with less latency.

Root causes. In most cases, data-plane integration reduces latency by *eliminating communication overheads*. Consider the fast rerouter as an example. Detecting and routing around a link failure requires at least two rounds of messages between a switch and its neighbors: One round to determine that a next hop has failed and a second round to identify an alternate next hop. If the fast rerouter's control operations ran as a Linux application on the switch's CPU, the operating system itself would add around 400 µs of latency because unidirectional messaging between Linux socket endpoints takes around 100 µs end-to-end [33]. However, a version of the fast rerouter with control in the data plane would completely avoid this overhead, along with others due to control-related middleware [4, 11]. For example, sending a message (i.e., a packet) from a switch's data-plane processor to its neighbor takes around 1 µs, and is bound only by the propagation and queueing delays of the physical hardware. Data-plane integration also eliminates the communication overheads between a switch's data-plane and management processors (*e.g.*, PCIe latency [24]). These overheads dominate in single-node applications with simpler control operations, such as in a stateful firewall (Section 7.4).

2.2 PISA Programmable Packet Processing

This paper focuses on PISA (Protocol Independent Switch Architecture) processors. PISA is a compelling data-plane architecture for three reasons: First, it is programmable; second, it is a generalization of real-world chips, primarily the Intel Tofino, and third; it processes packets at a high and guaranteed line rate (one packet per clock). Given a platform-specific minimum packet size, a PISA processor can sustain a workload that saturates *all* ports simultaneously.

The core of a PISA processor, illustrated in Figure 3, is a programmable line-rate match-action pipeline. Line rate demands a tightly synchronized, *feed-forward design*: Each pipeline stage has a throughput of one packet per clock, and packets only ever move forward through the pipeline. Instruction-level parallelism is also critical for line rate. A packet's header moves through each stage in parallel, as a vector. When the packet header enters a stage, ternary (TCAM) and exact (hash + SRAM) match-action tables evaluate it to feed ALU vectors with instructions to modify header fields. The "programmable" aspect of the pipeline is the capability to set table

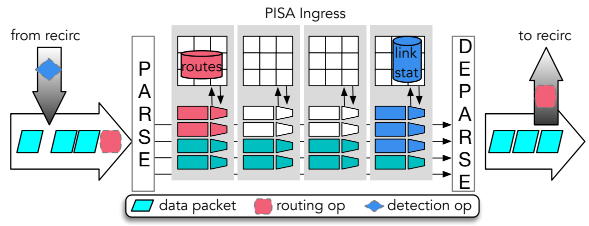

Figure 3: Interleaving the fast rerouter's control operations and packet processing in a PISA switch.

layouts and instructions at compile time, and set table entries from a management CPU at run time.

Stages also have stateful ALUs (sALUs) for updating local SRAM *register arrays*. Each stateful ALU can read from a single address in SRAM, perform limited computation, and write back to SRAM or modify metadata associated with the packet.

The ingress pipeline can direct packets to an egress port or, optionally, a recirculation port that brings the packet back to the start of the pipeline for additional processing. A recirculation port typically has the same bandwidth as a single front-panel port and shares the pipeline's packet-processing bandwidth, so only a fraction of packets can be recirculated without limiting throughput.

Finally, real-world PISA processors also include platform-specific and semi-programmable "support engines" outside of the core pipeline. The Intel Tofino, which this paper focuses on, includes four such engines: (1) a multicast engine to copy packets, (2) a queue manager to shape flows, (3) a packet generator for spawning packets, and (4) configurable MAC blocks that can dynamically pause queues based on Priority Flow Control (PFC) frames.

2.3 Packet-driven Control Operations

We know that packet processing maps well to a PISA chip [5], but how do we use it for more general control tasks? The key is to break control tasks down into atomic operations driven by the arrival of packets—when an ordinary data packet arrives, its presence and path through the switch can trigger execution of control operations that occur alongside regular forwarding operations. This can work well if control operations align with data packet arrival and are simple enough to fit in a single pass through a PISA switch.

But what if control operations are complex and their execution depends on one another rather than on the arrival of ordinary data packets? For example, Figure 2 sketches the control structure of the fast rerouter. Complex control operations like routing or fault detection can be decomposed into simpler units—route updates, route checks and route queries in the case of the routing component, for instance. And each of those operations can be performed in a single pass through a PISA pipeline. To ensure these operations occur at the appropriate cadence, it is possible to design, generate and parse new *synthetic* control packets to initiate execution of these control operations at the appropriate time.

2.4 Persistent State

Typically, the goal of control operations is to update state that affects the processing of subsequent packets. For example, reroute operations set entries in an array that determines where future packets are forwarded. A PISA pipeline stores this state in its stage-local SRAM banks. Control (and packet) operations read and write the state atomically using stateful ALUs. Atomicity means that a stateful operation can only update a single word of memory and perform computation that is simple enough to execute in a single instruction.

For more complex stateful operations, we can use multiple SRAM banks or stages. For example, when the fast rerouter forwards a packet, it looks up a next hop from an array in one stage, then uses an array in a subsequent stage to determine if the next hop is still active. Conveniently, multi-stage stateful operations are still atomic and line rate because of the feed-forward architecture of a PISA pipeline [28]. But this comes with a programmer challenge because it forces all packet and control operations to access state in the same order.

2.5 Control Threads via Recirculation

Some control operations far exceed the resource limits of a PISA pipeline. For instance, in the fast rerouter, checking the status of one link is a simple computation. But how do we check the status of an entire table of links? In general, *maintenance tasks* that require iteration over large tables or sets of values to identify stale or erroneous entries may appear impossible to support. However, we can use recirculation for *serial* processing, by recirculating a control packet multiple times to perform one part of its task in each pass, or we can use it for *parallel* processing, by recirculating multiple control packets back-to-back, each operating on a different entry.

A potential concern is that recirculation for control consumes bandwidth that could be used for packet processing. This is possible, but for many applications, overhead is low because control operations, even low-latency and fine-grained ones, are infrequent compared to data-plane packets. For example, consider the fast rerouter on a 1 Ghz PISA with 128 ports. It detects failed links by serially scanning the link status table with a control packet that recirculates once per µs. The recirculation throughput, 1 million packets/s, is only 0.1% of the pipeline's bandwidth. Even though overhead is low, it still checks each port often, once per 128 µs.

2.6 Scheduled Control via Support Engines

Of course, we may not always *want* control threads to operate at the highest possible rate, or, for that matter, at the same switch. This brings us to the last piece of the puzzle: how can a PISA processor schedule the *place* and *time* where control operations execute?

Place. Changing *where* an operation executes is straightforward, assuming that switches have addresses. Since control operations are processed like packets, a switch can schedule an operation at another location by encapsulating the corresponding control packet in an appropriately addressed frame and forwarding it just like any other packet. With line-rate multicast engines, we can even schedule an operation at *multiple* locations (such as the fast rerouter pinging all neighboring switches) in a single step.

Time. Changing *when* an operation executes is harder. Essentially, we need to buffer a control packet for some amount of time. A design for a generic PISA processor could buffer it in a register array along with the time at which it should be executed, and then scan the array periodically to find operations ready to execute. However, this approach could consume a large number of stateful ALUs. An alternative is to simply recirculate the control packet repeatedly until it is ready to execute, but this consumes recirculation bandwidth. A more efficient design, specialized for the Intel Tofino, is to use a dedicated queue for delayed control operations, which is paused and periodically released using PFC pause frames from the Tofino's packet generator.

2.7 Masters of Complexity

To many a hacker, our discussion of control in the data plane may sound glorious. All one needs to do is:

- break complex control operations into simpler ones;
- create formats, parsers, and deparsers for new synthetic packets to drive control operations where necessary;
- mind the constraints on stateful ALUs and per-stage computation;
- ensure that control operations access state in a consistent order;
- manually recirculate packets for multi-step operations, carefully interleaving the processing of recirculated and data packets;
- learn how to program the support engines outside the language of your programmable switch; and
- implement primitives for delay and distribution of control.

What's not to like? Everyone loves rolling multiple low-level resource constraints and a couple of different programming paradigms around in their head, before they even get to considering the high-level logic of their application! And yet, in our experience, expert programmers can spend weeks and hundreds or thousands of lines of code developing network control applications with relatively simple high-level ambitions.

Hence, rather than embracing this challenge to master complexity, we propose a better way: higher-level abstractions, static analysis, and automatic generation of low-level code. As the coming sections will show, these mechanisms together reduce programmer time from days to hours and lines of code by a factor of 10.

3 EVENT-DRIVEN PROGRAMMING

To create control applications for PISA switches, programmers currently must implement many low-level mechanisms by hand. In many ways, it is reminiscent of writing a distributed system without basic operating system services. Consider the challenge of adding the fast rerouter's *route query* control operation to a basic forwarding program written in P4. We must define a route query header along with parsers and deparsers; adjust the control flow to branch on that header's presence (in addition to existing branches); serialize generated queries into event packets; and finally configure the multicast engine to broadcast these packets to all neighbors. All of this effort only gets us to the point where we can *begin* to implement the interesting logic, e.g., the P4 that generates and responds to route queries.

3.1 Event-based Lucid Abstractions

The main idea behind Lucid's core abstractions is to unify packet processing with control operations through intuitive primitives for coordinating *when* and *where* events execute.

Events. Lucid abstracts both control operations and data packets as *events*. Every event consists of a four-tuple containing (1) a name, (2) carried data, (3) a time, and (4) a place. Events give programmers a way to structure multi-threaded programs that is missing from P4 and other existing data-plane languages. For example, the routing component of the fast rerouter has three events, corresponding to its three operations in Figure 2.

```
event route_query(int sender_id, int dst);
event route_reply(int sender_id, int dst, int pathlen);
event check_route(int dst);
```

Events are also a high-level abstraction for application-layer messages. The `route_query` event is a request that switch `sender_id` sends to its neighbor, asking for the length of its path to `dst`. A `route_reply` is a response to a query. Finally, `check_route` is an instruction *that a switch sends to itself* to check whether the route to `dst` has failed.

Handlers. In a Lucid program, all computation happens in a handler. A handler specifies what happens, such as a control operation or a packet-processing function, when a switch gets an event. Each handler compiles to a slice of parallel tables, ALUs, and stateful ALUs, and executes in a single pass through the match-action pipeline. Although the low-level implementation is complex, the high-level language for writing handlers is simple and expressive. For example, here is the `route_query` handler from the fast rerouter.

```
handle route_query(int sender_id, int dst) {
    int pathlen = get_pathlen(dst);
    event reply = route_reply(SELF, dst, pathlen);
    generate Event.locate(reply, sender_id);
}
```

The handler runs on a switch when its neighbor `sender_id` schedules a `route_query` event to execute on it. The handler looks up the length of the path to `dst` from a persistent array (`pathlens`), and communicates the result to `sender_id` by scheduling a `route_reply` event to execute there. The programmer can implement all the logic for a route query (including the `get_pathlen` function) while only writing *roughly the number of lines it would take merely to declare a route query header in P4.*

Event generation. As the `route_query` example also shows, handlers not only perform some computation in response to an event/packet, but can also generate events to trigger additional future computation. This abstraction of time is powerful because it lets us break complex control operations up into a thread of events that executes over a period of time.

For example, in the fast rerouter, we implement the thread that periodically scans the status of every route request as a recursive event handler. The handler checks the status of a route at a certain position in the routing table, and then (recursively) generates another event to check the next position. As another example, the stateful firewall application (Section 7.4) uses an event that recurses a bounded number of times to implement the insert operation of a cuckoo hash table in the data plane.

Event combinators. Event combinators let a handler change *when* and *where* an event that it generates will execute.

The **locate** combinator, which the `route_query` example uses, lets a handler specify where an event will be generated. Similar to a

```
event a(); event b(); event c(); const group GRP = {2, 3};
handle a() {
 generate b();
 mgenerate Event.delay (Event.locate (c(), GRP), 10ms);
}
```

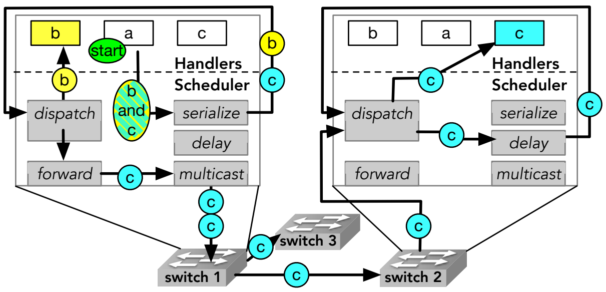

Figure 4: Event scheduling for a simple program.

`send` system call in Linux, the locate combinator provides a simple abstraction for unicast communication. Lucid also provides a multicast locate combinator for group communication.

The **delay** combinator changes *when* an event is executed. This combinator makes it easy to pause persistent computations, and hence resembles the Linux `sleep` system call. The fast rerouter uses the delay combinator to control the rate at which it pings its neighbors and also the rate at which it scans its routing table.

3.2 Data-Plane Event Scheduler

Lucid realizes the abstractions for event-based distribution and communication using an *event scheduling library* that is inlined into a Lucid program. As Figure 4 shows, the library sits logically between a Lucid application and the underlying network, filling the role of a lightweight operating system. The library is mostly comprised of hand-written code that is Tofino specific.

We describe the main components by following Figure 4, beginning with the execution of handler `a` at switch 1. This generates two events, `b` and `c`, by removing the `a` event header from the packet and attaching event headers for both `b` and `c`.

Event serialization. After the ingress pipeline finishes, Lucid's serializer transforms the single packet with headers for `b` and `c` into *serialized event packets*, one for each event.

First, the serializer uses the switch's multicast engine to create one copy of the packet for each event. When a copy arrives at the egress pipeline, it has headers for both `b` and `c`. The event serializer deletes one header from each copy, using a clone ID field that the Tofino provides as metadata.

Event dispatching. The event serializer sends the event packets for `b` and `c` to the switch's recirculation port. When these packets re-enter the ingress pipeline, the event data is extracted by a Lucid-generated parser and passed to an *event dispatcher*, an ingress match-action table that performs one of three actions based on an event's *location* and *delay*.

Non-local events: If the event's location is not the current switch, the dispatcher calls a user-configured forwarding table to select an output port or multicast group for the event. In the example program, the dispatcher at switch 1 sets a multicast group for event packet `c`, sending it to switches 2 and 3.

Delayed local events: For events that are destined to the local switch, but with a delay > 0, the dispatcher calls a *delay* function. In the example, switch 2 initially delays event `c`.

Processable events: When an event's delay is 0 and its location is the current switch, the dispatcher applies the compiler-generated tables that implement the event handlers. In the example program, the dispatcher at switch 1 will do this for `b` as soon as it arrives.

Implementing delay. Delay is the most sophisticated function in the scheduler. Lucid implements this with *pausable egress queues*. Events to delay are placed in a special "delay queue" of the recirculation port. The queue is paused most of the time and unpaused at a regular interval to release packets, e.g., once every 100 µs. When events exit the queue, a table in egress updates their delay parameter based on their queue time. The packets recirculate and repeat until their delay is 0. PFC (Priority Flow Control) packets let the event scheduler time the queue. We send a stream of packets into the pipeline that consists of pairs of PFC packets at a low, constant rate. The first PFC packet in a pair unpauses the queue to let event packets out, while the second one repauses it. The PFC stream can be generated by either the pipeline's packet generator or, if none is available, the switch's CPU.

4 OPERATING ON PERSISTENT STATE

There are two kinds of state in a data-plane program: local state that lives for the processing of a single packet and global (or persistent) state that remains across packets. In prior data-plane languages, persistent state makes development unreasonably challenging. The problem is not that the hardware has constraints, but rather that the constraints are left *implicit* by the language.

For example, P4 programs store persistent state in `RegisterArrays` and use `RegisterActions` with arbitrary blocks of C-like code to operate on those arrays. It is easy to write a `RegisterAction` that is too complex for the underlying hardware to support, but often difficult to figure out why. The "decision" that a particular `RegisterAction` is too complex is made by part of the compiler's back-end that is far removed from the source code, for example, a target-specific assembler. When a problem occurs at this late stage, neither the programmer nor the compiler have any direct way of figuring out what went wrong. For example, here is a `RegisterAction` body that is too complex for the Tofino and results in an assembler error related to operand referencing.

```
void apply(inout bit<32> memCell) {
    if (memCell > y) {
        memCell = memCell + y;
    } else {
        memCell = x + y;
    }
}
```

Lucid's solution is a carefully designed interface to persistent state that enables syntactic checks on untransformed source code. These checks occur at the very beginning of compilation, quickly identify invalid programs, and return *source-level* error message that tell us exactly what is wrong.

4.1 The Array Module

Lucid programs store persistent state in arrays, such as `pathlens` in this example from the fast rerouter.

```
global pathlens = new Array<<32>>(tbl_sz);
memop incr(int stored, int added) { return stored + added; }
fun int get_pathlen(int dst) {
    return Array.get(pathlens, dst, incr, 1);
}
```

All computation on arrays is done through Lucid's `Array` module, whose methods abstract the stateful operations that are possible within a single ALU. In the above example, the `Array.get` method retrieves `pathlens[dst]`. `Array` also includes `set` and `update` (i.e., set and get in parallel) methods.

Of course, the stateful ALUs of a PISA switch can do more than read or write values from persistent memory. They can read the state, perform a small amount of computation, and then write the state back to memory and/or packet metadata [28]. Lucid's `Array` module has a functional interface to these capabilities. In the above example, the third argument of the call to `Array.get` is `incr`, a function. `Array.get` will read the value from `pathlens[dst]` and return `incr(pathlens[dst], 1)`, with `1` coming from the fourth argument to `get`.

Lucid's interface to state is flexible and allows programmers write modular and re-usable code. Functions like `incr` can be re-used in any call to `Array.get`, `set`, or `update`. Further, arrays themselves can be arguments to functions or handlers.

4.2 Memop Functions

Function arguments to `Array` methods, such as `incr` in the above example, are *memops*: a special kind of function that is syntactically restricted to ensure that it does not do more computation than what a single stateful ALU can support. The syntax of a *memop* is limited by the instruction set of the targeted PISA processor because every `Array` method that uses a *memop* must be able to compile to a valid instruction. At the same time, the syntax must carefully balance *expressiveness* and *regularity*. On the one hand, we would like *memops* to be flexible enough to implement any program that the underlying hardware can support. On the other hand, we would like *memops* to be as simple and regular as possible, to decrease the Lucid learning curve make it easier to use.

With those design criteria in mind, we defined a *memop* to be a function of two arguments that satisfies the following constraints:

- the body is either a single return statement or an if statement containing one return statement in each branch;
- each variable is used at most once per expression; and
- only ALU-supported operators are used.

When a user declares a *memop*, we automatically check that its body satisfies these requirements. If this check passes, the programmer is guaranteed that the operation is compilable; if not, our compiler can explain exactly what is wrong.

We believe Lucid's current *memop* syntax strikes a good balance between regularity and expressiveness, favoring regularity over expressiveness ever so slightly. While we can construct expressions that can be implemented on the Tofino, but not in Lucid's memops, we have yet to encounter such a situation in any of the applications we have implemented so far (see Figure 9).

While our current memop definition is geared specifically for the Tofino, the principle the design suggests is quite general: Use static, source-level constraints to limit the expressions programmers write, as they write them. Doing so makes it possible to provide targeted programmer feedback that pinpoints the exact line and character where an error occured and ultimately saves one of the most important resources, programmer time.

```
const int SIZE = 16;

global arr1 = new Array<<32>>(SIZE);
global arr2 = new Array<<32>>(SIZE);

handle setArr1(int idx, int data) {
    int x = Array.get(arr2, idx);
    Array.set(arr1, idx, x);
}
handle setArr2(int idx, int data) {
    int x = Array.get(arr1, idx);
    Array.set(arr2, idx, x);
}
```

Figure 5: A disordered program.

5 ORDERED DATA ACCESS

Most data-plane programs with integrated control use *multiple* persistent variables. For example, the fast rerouter has a next hop array and link status array. A general constraint of any PISA processor is that such persistent data must be partitioned across the stages of a feed-forward pipeline. This leads to a natural order in which a program can access the data. Current languages force programmers to manually track the order of data access in their programs, which compounds the state-related development challenges described in Section 4.

To illustrate this issue, Figure 5 presents a simple but invalid Lucid program. The program declares two arrays `arr1` and `arr2`, and two handlers `setArr1` and `setArr2` which access those arrays in different orders. In general, programs of this form cannot be compiled to a PISA pipeline—one handler demands that data for `arr1` appear earlier in the pipeline than data for `arr2` and vice versa, creating irresolvable constraints.

These constraints are fundamental to any PISA pipeline, but they are not enforced by P4. If we write the program from Figure 5 in P4 for the Tofino, compilation does not fail until the Tofino backend. When the backend cannot solve the program's layout constraints to allocate stateful data to particular stages of the pipeline, it fails with an error that states "Table placement cannot make any more progress", but gives no indication of what is wrong with the program.

Lucid resolves this issue by interpreting a program's data declarations as an implicit, high-level specification of the programmer's data layout intentions. The Lucid type system then verifies that the order of data accesses in the rest of the program is consistent with the specification, guaranteeing that compilation is possible (if enough pipeline stages are available). When an access ordering error arises, a useful *source-level* error message indicates the specific lines of code in conflict.

5.1 Well-ordered Programs

We say a Lucid program is *well-ordered* if the data *accesses* in every handler follow the same order as the global data *declarations*. In other words, we treat the order of data declarations as a *specification* that clearly documents requirements for all handlers. It is an easy specification for programmers to write, as they must declare and initialize their data anyway.

The specification is high level—it does not refer to specific hardware stages; in fact, programs such as this are portable across hardware platforms with different numbers of stages. The compiler has the flexibility to place any object in any stage so long as it faithfully implements the program's semantics. The specification merely ensures that, provided a program adheres to its requirements, the compiler can find *some* solution to the data-allocation problem.

If programmers do not adhere to the specification, a simple error message directs them to the disordered portion of their program. For the program in Figure 5, Lucid would issue an error for the `setArr1` handler saying that it accesses `arr2` and `arr1` in the opposite order of their declarations. While verifying a toy example like Figure 5 is straightforward, the verification problem becomes increasingly difficult as programs grow and use auxiliary functions to encapsulate common idioms and user-defined abstractions. Such functions may access global variables (directly or via arguments), which constrains the order in which they may be called from other functions or handlers.

5.2 Types for Ordered Data Access

We use a *type-and-effect* system to check that a Lucid program is well-ordered while also performing regular type checking. The full details of this type system appear in Appendix A.

While past work [7, 17] explored the use of ordering constraints to ensure correct access to volatile state in other contexts (*e.g.*, "no-use-after-free" properties for memory managers or "no data access without first acquiring a lock"), we are unaware of prior uses of ordered type systems for data layout along pipelines, or more generally in the context of networking. On the one hand, systems such as ordered logic [26] appear too restrictive for our purposes—functions that refer to an ordered variable cannot be declared until prior variables are used. On the other hand, prior systems [7, 17] designed for enforcing protocols such as open-read/write-close sequences over OS resources are unnecessarily complex, requiring additional type annotations or sophisticated inference mechanisms. In Lucid, the key difference is that we know all the ordered variables in advance, allowing us to create a simpler system that can still define and verify functions separately from where they are used.

At a high level, our strategy is to use a system in which effects are integers representing abstract *stages*. Each ordered variable (either an array or a counter) is associated with a stage based on the order in which variables are declared. During typechecking, we keep track of a *current stage*, which tracks the most recently-accessed ordered variable. The typechecking fails if the program attempts to access an ordered variable whose stage is less than the current one, indicating that during execution the packet would have already passed by that data in the pipeline.

Formally, our typing judgement has the following form:

$$\Gamma, \epsilon_1 \vdash e : \tau, \epsilon_2.$$

In English, this can be interpreted as the statement "Starting with environment Γ and at stage ϵ_1, the expression e has type τ and will finish evaluating in stage ϵ_2". We use this to prove the following soundness theorem:

Theorem: If $\emptyset, \epsilon_1 \vdash e : \tau, \epsilon_2$, then either e is a value, or $e \longrightarrow e'$ and there is some ϵ_1' such that $\emptyset, \epsilon_1' \vdash e' : \tau, \epsilon_2$.

```
Array nexthops = new Array<<32>>(NUM_HOSTS);
Array pcts = new Array<<32>>(NUM_PORTS_X3);
Array hcts = new Array<<32>>(NUM_HOSTS);
memop plus(int cur, int x){return cur + x;}

event count_pkt(int dst, int proto);
handle count_pkt (int dst, int proto) {
   int idx = Array.get(nexthops, dst);
   if (proto != TCP) {
      if (proto == UDP)
         idx = idx + NUM_PORTS;
      else
         idx = idx + NUM_PORTS_X2;
   }
   Array.set(pcts, idx, plus, 1);
   if (proto == TCP)
      Array.set(hcts, dst, plus, 1);
}
```

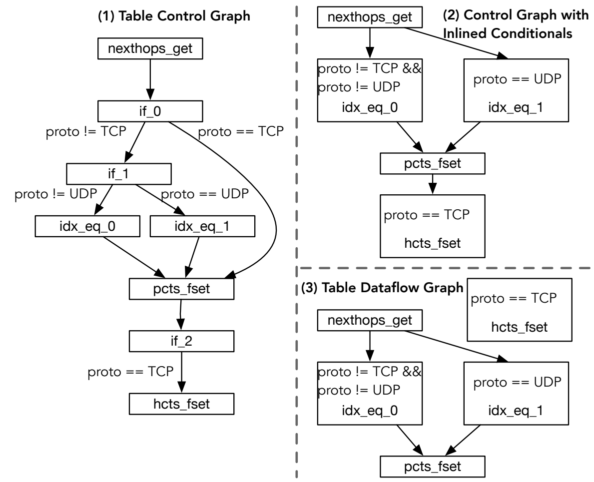

Figure 6: Top: a Lucid handler using only atomic statements. Bottom: the handler represented as an atomic table graph (1) and optimized to require fewer pipeline stages (2 and 3).

This theorem implies that any program which typechecks will never "get stuck" trying to access unavailable data. The proof of the theorem appears in Appendix A.

6 COMPILING TO THE TOFINO

After syntax and type checking, the Lucid compiler translates a program into P4_16 optimized for the Intel Tofino. Most of the backend's complexity lies in compiling handler bodies, since events map directly to packet headers and the event scheduler (Section 3.2) is mostly static code. The main steps of handler compilation are translating to atomic P4 tables and optimizing control flow.

6.1 Translating to Atomic P4 Tables

The compiler first uses function inlining and subexpression elimination to reduce a handler's body into a graph of statements that are each simple enough to execute with at most one Tofino ALU. The top of Figure 6 is an example of a handler where all statements are already in this atomic form. The compiler then translates each atomic statement directly into a P4 table, to produce an atomic table graph like the one shown in Figure 6(1). There are three kinds of atomic tables, demonstrated in Figure 7.

Lucid

```
idx = idx + NUM_PORTS;
```

```
Array.setm(tcp_cts, port, plus, 1);
```

```
if(proto != TCP)
```

P4

Operation Table

```
action do_idx_add {idx = idx + NUM_PORTS;}
table tbl_idx_add {
  actions = {do_idx_eq;}
  const default_action = {do_idx_eq;}
}
```

Memory Operation Table

```
RegisterAction<...>(tcp_cts) setm_1 = {
  void apply(inout bit<32>m, out bit<32>r){
    mem = mem + 1;
  }
};
action do_setm_1() { setm_1.execute(port);}
table tbl_setm_1 {
  actions = {do_setm_1;}
  const default_action = {do_setm_1;}
}
```

Branch Table

```
action if_true(); action if_false();
table tbl_if {
  keys = {proto : ternary;}
  actions = {if_true; if_false;}
  entries = {
    (TCP) : if_false;
    (_)   : if_true;
  }
}
```

Figure 7: Examples from Figure 6 of the three kinds of Atomic P4 tables that the Lucid compiler generates.

Operation table. An operation table uses a single ALU to evaluate a binary expression over two local variables (metadata in P4) and assign its result to a third local variable.
Memory operation table. A memory operation table uses a single stateful ALU to update one element in a P4 register array. It is a direct translation of a call to an Array method.
Branch table. A branch table uses two match-action rules to compare a local variable against a constant to determine which table will execute next.

6.2 Optimizing Control Flow

The Lucid compiler optimizes the atomic table graph in three steps to reduce the number of pipeline stages it requires.

Inlining branch operations. Branch tables are wasteful in the table representation of a Lucid program because, in a PISA pipeline, a branch table's successors must be placed in a subsequent stage.

The compiler eliminates branch tables by first transforming each non-branch table to check the conditions necessary for its own execution using static match-action rules. The table executes its single action if there is a match, else a no-op. For example, by following the path from the root of Figure 6(1) to the table `idx_eq_0`, we see that it only executes if `(proto != TCP)&& (proto != UDP)`, thus in Figure 6(2), the table `idx_eq_0` tests these conditions before executing.

The compiler applies this transformation to all tables then deletes the branch tables. As Figure 6(1) and (2) show this saves three pipeline stages in the example program.

Rearranging tables. Next, the compiler rearranges tables based on data flow to further reduce the number of stages used by a program. For example, in Figure 6(2), the table that implements `Array.set(hcts, dst, plus, 1);` (`hcts_fset`) does not have any data flow dependencies on previous tables. None of the variables that it reads are modified by any other tables in the program, so it can be executed in parallel with the first table, as shown in Figure 6(3).

Merging tables and actions. For complicated programs, the atomic table representation of a Lucid program would still require many stages because it uses many tables (one per operation) and PISA processors can only support a limited number of tables per stage. Lucid's final optimization reduces table overhead by merging atomic tables into multiple-operation tables. This is possible because Lucid-generated tables use only static rules. Figure 8 shows how tables from the example in Figure 6 get merged.

The compiler uses a simple greedy algorithm that produces a pipeline with M stages and N merged tables per stage by walking the atomic table graph topologically. For each table t, it finds the earliest merged table that t can be merged into. This decision is based on data flow constraints, a simple model of the free resources in each stage, and a small number of Tofino-specific constraints.

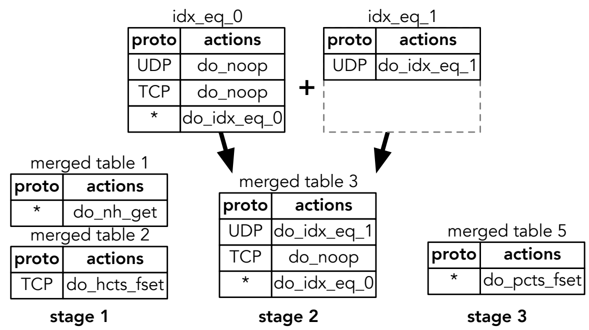

Figure 8: Merged tables for the program in Figure 6.

When a merged table m that can fit t is found, m is replaced with m'—the cross product of m and t. The algorithm ends when either all atomic tables have been merged, or it reaches an atomic table that cannot be placed.

7 EVALUATION

We evaluate Lucid by implementing the applications described in Figure 9 and compiling them to the Tofino with the Lucid compiler [1] and P4-studio verision 9.2. We analyze the design of Lucid, the effectiveness of its optimizations and the runtime overhead of recirculation. Finally, a case study with the stateful firewall evaluates the potential performance benefit of data-plane integrated control.

7.1 Language Design

We first compare the lines of code required to program in Lucid versus P4. *Flow [30] is a complex application that gives us a point of comparison to hand-written P4. The Lucid program is a complete implementation of *Flow in 149 lines of code. The published implementation, in P4_14, is 1559 lines of code—over 10X longer.

We are unaware of hand-written implementations for the other Lucid applications and, due to the time required to program the Tofino in P4, we did not re-implement any ourselves. However, using *Flow as a calibration point suggests that the Lucid compiler produces P4 that is within 15% the length of hand-written P4. Thus, we conclude that Lucid reduces lines of code by around 10X for diverse applications.

[1] https://github.com/princetonUniversity/lucid

Application	Description	LoC Lucid	LoC P4	Tofino Stages
Stateful Firewall (SFW)	Blocks connections not initiated by trusted hosts. ***Control events update a Cuckoo hash table.***	189	2267	10
Fast Rerouter (RR)	Forwards packets, identifies failures, and routes. ***Control events perform fault detection and routing.***	115	899	8
Closed-loop DNS Defense (DNS)	Detects/blocks DNS reflection attack with sketches & Bloom filters. ***Control events age data structures.***	215	1874	10
*Flow [30]	Batches packet tuples by flow to accelerate analytics. ***Control events allocate memory.***	149	1927	12
Consistent Shared State (SRO)[35]	Strongly consistent distributed arrays. ***Control events synchronize writes.***	94	897	11
Distributed Prob. Firewall (DFW)	Distributed Bloom filter firewall. ***Control events sync. updates.***	66	1073	10
+Aging	***Adds control events for aging.***	119	1595	10
Single-dest. RIP	Routing with the classic Route Information Protocol (RIP). ***Control events distribute routes.***	81	764	8
Simple NAT	Basic network address translation. ***Control events buffer packets and install entries.***	41	707	11
Historical Prob. Queries (CM)	Measures flows with sketches for historical queries. ***Control events age and export state periodically.***	93	856	5

Figure 9: Applications with data plane-integrated control, implemented in Lucid and compiled to the Barefoot Tofino. The role of control events is bolded.

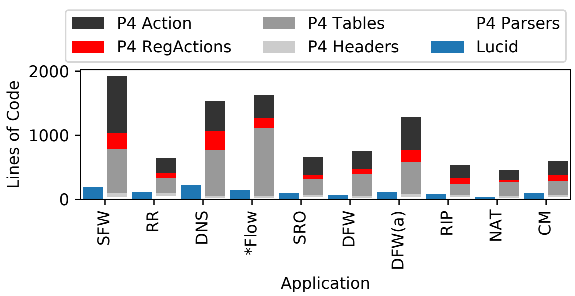

Figure 10: Breakdown of P4 code in Figure 9.

Application	NAT	RIP	Dist FW	Dist FW + Aging
Dev. Time	25m	40m	25m	25m + 30m

Figure 11: Time for a student without Tofino experience to write Tofino-compiling Lucid applications.

Figure 10 breaks down the lines of P4. Actions and tables take the most lines. An interesting observation is that for most applications, the entire Lucid program was fewer lines of code than just the register actions in P4. This is partially because P4 register actions are not reusable like Lucid's memops—the programmer must manually copy the code every time they want to repeat the same operation on a different array. In Lucid programs, and even across programs, we re-use the same generic memops multiple times.

The applications in Figure 11 were implemented by a PhD student who has never programmed the Tofino before. As it shows, it took less than an hour to write each of these non-trivial prototypes that successfully compiled to the Tofino. With P4, this level of productivity is hard to imagine, even for an experienced Tofino programmer. For PhD students new to the architecture, it can take weeks to do anything non-trivial. Needless to say, we are excited by the potential for Lucid to save future students' time.

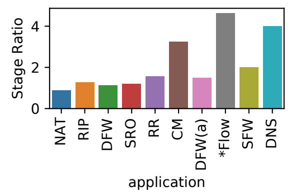

Figure 12: Optimized stage count vs unoptimized.

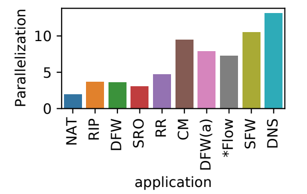

Figure 13: ALU instrs. per stage in optimized code.

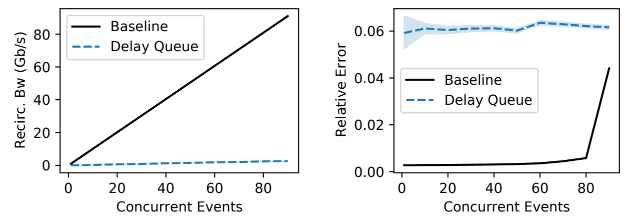

Figure 14: Pausable queue overhead and accuracy.

7.2 Optimization Benchmarks

Compiler optimizations. Next, we evaluate Lucid's compiler optimizations by comparing the number of required stages with and without optimizations. Figure 12 shows the ratio for each application. For unoptimized stages, we report the number of atomic P4 tables in the longest code path, as many programs did not fit into the Tofino's pipeline without optimization. Optimizations reduced stage requirements by a factor of 1.5-4 for most applications. The benefit was greater for complex applications, such as *Flow and the closed-loop DNS defense system, which originally had critical paths nearly 4X too long for the Tofino's pipeline.

Figure 13 shows the number of Lucid statements that the compiler mapped to each stage. It ranged from 2 - 13, demonstrating that the compiler was able to find and exploit a significant amount of parallelism in the programs.

Event scheduler optimizations. A key optimization in Lucid's event scheduler is the pauseable queue mechanisms for reducing the overhead of delaying events via recirculation. We measured the bandwidth overhead and timing accuracy of delaying 64B event packets on one of the Tofino's 100 Gb/s recirculation ports, with and without the pausable queues.

As Figure 14 shows, the pausable queues make the recirculation bandwidth cost of delayed events negligible. The bandwidth cost for delaying 90 concurrent events indefinitely was 5.5 Gb/s. In comparison, delaying 90 concurrent events *without* Lucid's pausable queues consumed over 95 Gb/s—the port was effectively saturated.

This nearly 20X reduction in overhead has two costs: increased packet buffer utilization and timing variance. The increase in packet buffer utilization is small compared to the amount of recirculation throughput saved. For example, storing 90 64B events in a queue uses around 7KB of packet buffer (depending on memory cell size). The Tofino has 22MB of shared packet-buffer memory, or a bit more than 320KB per port. So, with 90 concurrent events we trade around

Recirc. use	Recirc. rate	Applications
array scanning *(periodic event)*	$O(\frac{\text{num. entries}}{\text{scan interval}})$	timeouts (SFW,RR), aging (DFW,CM,DNS), synchronization (RIP)
per-flow operations *(external event)*	E[O(flow rate)]	setup (SFW,DFW,NAT), mem. allocation (*Flow), fault recovery (RR)
distributed operations *(external event)*	O(op. rate)	sync. write (SRO), flow setup (DFW)

Figure 15: Recirculation uses in Figure 9 applications.

flow rate (f)	10K flows/s	100K flows/s	1M flows/s
recirc. rate	815K pkts/s	2M pkts/s	16M pkts/s
pipeline utilization	0.08%	0.22%	1.66%
min. pkt. size	125.26	125.55	127.67

Figure 16: Modeled worst-case recirculation overhead for the stateful firewall with $N = 2^{16}$ and i =100 ms.

2% of the recirculation port's fair share of buffer space for a 20X reduction in bandwidth utilization.

Pausable queues also increases the variance of event execution times. As Figure 14 shows, event delay was off by up to approximately 50 µs when using pausable queues that release every 100 µs.

7.3 Recirculation Overhead

Figure 15 characterizes recirculation in our applications. Recirculation has two purposes: periodic operations, whose rate is governed by user-configurable parameters; and externally-triggered operations, whose rate is goverend by the workload.

We analyze the stateful firewall to understand recirculation overhead more concretely. This application requires more recirculation than the others in Figure 9 because it features both periodic operations (array-scanning to find and time out flows) and external operations (recirculation to install per-flow entries into a cuckoo hash table). We base analysis on an idealized PISA processor with a throughput of 1B packets per second that services 10 100Gb/s front-panel ports plus a 100Gb/s recirculation port. This processor supports line rate on all front-panel ports simultaneously when packets are larger than 125B and the recirculation port has no load.

Given this idealized platform, we derive a simple explanatory model of the stateful firewall's recirculation rate. Model parameters are: N, the size of the firewall's table; i, the per-flow timeout check interval; and f, the flow-arrival rate. The worst-case recirculation rate, r, is: $r = \frac{\mathrm{N}}{\mathrm{i}} + f \cdot \log(N)$. The first term is recirculation for timeout scanning and the second term is *worst-case* recirculation for flow installation, as an installation in a Cuckoo table may require $\log(N)$ Cuckoo operations, each of which requires a recirculation.

Figure 16 shows that the recirculation rate is high in absolute numbers, but only a small percentage of the pipeline's packet-processing bandwidth—a workload with 1M new flows per second has less than a 2% bandwidth overhead. At this point, the pipeline could still support line rate on all front-panel ports if all packets were larger than 128B (versus 125B with no recirculation load).

7.4 Stateful Firewall Case Study

Finally, to evaluate the latency reduction that data-plane integrated control can provide, we benchmark a stateful firewall in Lucid.

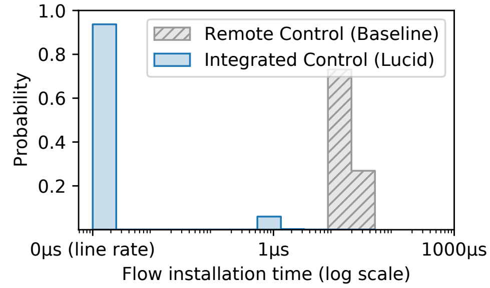

Figure 17: Stateful firewall flow installation times. 1000 trials using a 2048-element table with a load factor of .3125.

Implementation. The core of the Lucid stateful firewall is a Cuckoo hash table [25]. Each flow maps to one of two possible locations, addressed by a different hash of its key. A lookup operation simply checks both locations in sequence. An install operation for flow f attempts to place it in either of f's locations. If both are occupied, a "Cuckoo" operation replaces a colliding victim v with f, then generates an event to re-install v. This process repeats until an install operation has no victim, or the install handler detects that it has attempted to re-install f more than twice, indicating failure [19]. While a flow v is being re-installed, its entry is stored in a stash at the end of the pipeline, so that the (re-)install operations are transparent to concurrent lookups.

Flow installation time. Our benchmark metric is flow installation time—the difference between when the first packet of a flow arrives and when the corresponding installation operation completes. This metric is critical in a stateful firewall because flow installation must complete in between the arrival of the first packet from an outbound flow and the arrival of the first packet from the return flow, *i.e.*, one RTT. If not, the firewall will disrupt traffic by either dropping return packets or queuing the first packet of every new flow until the installation is complete.

Baseline. We compare against a baseline that represents flow installation with efficient remote control, using Mantis [34]. Mantis is a driver-level framework for low-latency control in the management CPU of a Tofino switch. We measure the time required for a Mantis control thread to install a new entry into a P4 match-action table in the Tofino. This is a lower bound because it ignores the time required for the CPU to detect that a new flow has arrived, for example by polling a P4 register in the Tofino that stores a ring buffer of new flow keys.

Benchmarks. Figure 17 shows the distribution of flow installation times. Average flow installation time for the data-plane integrated version was only 49 ns. For over 90% of flows, installation completed during the processing of the flow's first packet—an effective flow installation time of 0 ns. Most of the remaining flows installed in a single recirculation—about 600 ns. The worst case was 4 recirculations or around 2.4 µs. In comparison, the remote-controlled baseline took at least 12 µs to install a rule for a new flow into a P4 match-action table, with an average of 17.5 µs—*over 300X longer than the data-plane integrated version.* End-to-end flow installation time with remote control would be much higher in practice, because of the time required to inform the remote controller that a new flow has arrived and the queuing delays that would occur when multiple flows arrived simultaneously.

Memory efficiency. A drawback of the current Lucid firewall prototype is memory efficiency. It uses around 3X more memory than the remotely controlled baseline. The Lucid version does not include mechanisms to resolve or eliminate installation failures [3, 19], so the load factor must be kept low (0.3125 in our experiments) to keep the probability of flow installation failure low. The remotely-controlled baseline, on the other hand, uses the native match-action tables of the Tofino that are based on a more sophisticated Cuckoo hashing algorithm that supports a load factor near 1. Improved algorithms for Cuckoo hashing are possible in Lucid, for example we implemented a Cuckoo hash table with a stash [19]. We leave exploration of more advanced Lucid data structures for future work.

8 DISCUSSION

Lucid's abstractions, syntactic restrictions, and backend optimizations go a long way towards making data-plane programming feel less like arcane magic and more like software engineering. Still, Lucid is a work in progress. This section discusses the current limitations of Lucid and data-plane integrated control.

8.1 Language Limitations

Lucid's limitations arise from our design goal: a high-level language that lets us reliably write and compile self contained applications to a specific, widely-used PISA processor.

Read-only tables. Lucid does not provide a direct abstraction of a PISA processor's match-action tables because they are read-only from the PISA processor. Programs that wish to use match-action tables to classify packets can implement them outside of Lucid, in P4 that applies to packets before/after the Lucid event dispatcher is called. Updating these tables is slow because it must involve the switch's CPU. For higher performance, programmers can build "software" data structures in Lucid, like the Cuckoo hash table in the stateful firewall. These data structures can still operate at line rate and with reasonable recirculation overhead, but require more stages compared to lower-level P4 equivalents and cannot use TCAM. There are many efficient packet-classification data structures in software [12], it is likely that many of them can be adapted to Lucid.

Memop completeness. Not all instructions that a Tofino stateful ALU can execute are expressible as a *memop*. This is not a fundamental limitation, but rather an artifact arising from our desire to present a regular and easy-to-use abstraction. An industrial version of Lucid would likely include multiple kinds of memops that struck different balances, or targeted different types of ALUs.

Portability. Lucid currently only targets the Intel Tofino. Portable data-plane languages are a focus of recent research [9]. The challenge of portability arises from the diversity of data plane hardware. P4 programs, for example, are often not portable because many commonly-used primitives (such as those for stateful operations) are platform-specific externs. Although Lucid is not portable, its event-based abstractions and ordered type-and-effect system are relevant to any PISA-like packet processor. The concept of *memops* is also portable, though the syntax checks of a *memop* will likely have to change across targets [28]. Another portability-related concern is supporting multiple parallel pipelines. In Lucid, multi-pipeline processors can simply be abstracted as a set of directly connected switches that are programmed using the same core Lucid primitives.

Optimization. The optimizations in Lucid's compiler are heuristics based on our experiences with hand-coding efficient P4-Tofino programs. Recent work suggests that sophisticated optimizations based on synthesis [10] or ILP [14] algorithms may do better. However, in general, finding an optimal PISA layout is NP complete [32].

8.2 Integrated Control Limitations

While data-plane integrated control can have significant benefits, it is not always the best option. We identify three factors that can make remote control more appealing.

Compute-bound operations. Compute-bound operations [13] do not benefit as much from the reduced communication overhead that data-plane integration provides. Further, for compute-bound tasks, a server may be faster than the data-plane processor [6]. Of course, future packet processing architectures may shift the balance [31].

Centralized, network-wide control. As presented, data-plane integrated control makes the distributed architecture of a control application transparent to the programmer. An advantage of remote control is that it is easy to centralize, which reduces programmer effort, as demonstrated by prior control-plane languages like Flowlog [23], Frenetic [8], NetKAT [2], and McNetKAT [29]. An open question is whether we can present the abstraction of logically centralized control atop a data-plane integrated control layer, to provide the simplest programming model with minimal communication overhead.

Runtime overhead. Data-plane integrated control adds runtime overhead due to packet recirculation. This overhead is often low (Section 7.3), but is application dependent and should be considered by the programmer. Future architectures could eliminate this overhead with hardware to process control operations in parallel with packets [16] and periodically synchronize shared state.

9 CONCLUSION

Lucid makes it easy to write data-plane applications with high-performance integrated control. PISA switches *already* have all the necessary mechanisms; however, programmers today must use them at a very low level. Instead of writing packet-processing functions that always execute here and now, Lucid programmers can use intuitive event-based abstractions to distribute control in both time and space. Complementing this is a careful correct-by-construction approach to stateful operations in the data plane that uses syntactic constraints and a sound type system to improve compiler feedback and rule out programs that are unlikely to compile.

We realize these ideas for the Intel Tofino, with an optimizing compiler that generates efficient, Tofino-compatible P4. A diverse range of Lucid applications require require ~10X fewer lines of code, compared to P4 equivalents. Programmers without *any* prior Tofino experience are able to write compiling code in well under an hour. Finally, a Lucid stateful firewall outperforms a remotely-controlled baseline by over 300X.

Overall, Lucid is general, fast, and easy to use. It will save time, enable new applications, and perhaps change the way we think about what hardware data planes can do.

Acknowledgments. We thank our shepherd, Brent Stephens, and the anonymous reviewers for their feedback. We also thank Mihai Budiu, Ben Pfaff, Leonid Ryzhyk, and Muhammad Shahbaz for fruitful discussions and useful feedback on this project, and Dovid Braverman for his help with developing the compiler front-end. This work is supported by NSF grants CNS-1703493 and FMitF-1837030 and DARPA grants HR0011-17-C-0047 and HR0011-20-C-0160.

REFERENCES

[1] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, The Vinh Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM*. 503–514.

[2] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 113–126.

[3] Yuriy Arbitman, Moni Naor, and Gil Segev. 2009. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In *International Colloquium on Automata, Languages, and Programming*. Springer, 107–118.

[4] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. 2014. ONOS: Towards an open, distributed SDN OS. In *Workshop on Hot Topics in Software Defined Networking*. 1–6.

[5] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM*. 99–110.

[6] Xiaoqi Chen. 2020. Implementing AES encryption on programmable switches via scrambled lookup tables. In *ACM SIGCOMM Workshop on Secure Programmable Network Infrastructure*. 8–14.

[7] Rob DeLine and Manuel Fahndrich. 1999. Natural deduction for intuitionistic non-commutative linear logic. In *International Conference on Typed Lambda Calculi and Applications*.

[8] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A Network Programming Language. In *ACM International Conference on Functional Programming*. 279–291.

[9] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. 2020. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. In *ACM SIGCOMM*. 435–450.

[10] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 2020. Switch Code Generation Using Program Synthesis. In *ACM SIGCOMM*. 44–61.

[11] The P4.org API Working Group. [n.d.]. P4Runtime Specification. https://p4lang.github.io/p4runtime/spec/main/P4Runtime-Spec.html

[12] Pankaj Gupta and Nick McKeown. 2001. Algorithms for packet classification. *IEEE Network* 15, 2 (2001), 24–32.

[13] Brandon Heller, Srinivasan Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. 2010. ElasticTree: Saving energy in data center networks. In *USENIX Networked Systems Design and Implementation*, Vol. 10. 249–264.

[14] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, David Walker, and Rob Harrison. 2020. Elastic Switch Programming with P4All. In *ACM SIGCOMM HotNets Networks*. 168–174.

[15] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. 2020. Contra: A programmable system for performance-aware routing. In *USENIX Symposium on Networked Systems Design and Implementation*. 701–721.

[16] Stephen Ibanez, Gianni Antichi, Gordon Brebner, and Nick McKeown. 2019. Event-driven packet processing. In *ACM Workshop on Hot Topics in Networks*. 133–140.

[17] Atsushi Igarashi and Naoki Kobayashi. 2001. Enforcing high-level protocols in low-level software. *SIGPLAN Notices* 36 (May 2001). Issue 5.

[18] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. Hula: Scalable load balancing using programmable data planes. In *ACM SIGCOMM Symposium on SDN Research*. 1–12.

[19] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. 2008. More robust hashing: Cuckoo hashing with a stash. In *European Symposium on Algorithms*. Springer, 611–622.

[20] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. 2013. F10: A Fault-Tolerant Engineered Network. In *USENIX Symposium on Networked Systems Design and Implementation*. 399–412.

[21] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. 2021. Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches. In *USENIX Security Symposium*.

[22] Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

[23] Tim Nelson, Andrew D. Ferguson, Michael J.G. Scheer, and Shriram Krishnamurthi. 2014. Tierless Programming and Reasoning for Software-Defined Networks. In *USENIX Networked Systems Design and Implementation*. 519–531.

[24] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. 2018. Understanding PCIe performance for end host networking. In *ACM SIGCOMM*. ACM, 327–341.

[25] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.

[26] Jeff Polakow and Frank Pfenning. 1999. Natural deduction for intuitionistic non-commutative linear logic. In *International Conference on Typed Lambda Calculi and Applications*.

[27] Rinku Shah, Vikas Kumar, Mythili Vutukuru, and Purushottam Kulkarni. 2020. TurboEPC: Leveraging Dataplane Programmability to Accelerate the Mobile Packet Core. In *ACM Symposium on SDN Research*. 83–95.

[28] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM*. 15–28.

[29] Steffen Smolka, Praveen Kumar, David M. Kahn, Nate Foster, Justin Hsu, Dexter Kozen, and Alexandra Silva. 2019. Scalable Verification of Probabilistic Networks. In *ACM SIGPLAN Programming Language Design and Implementation*. 190–203.

[30] John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. 2018. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *Flow. In *USENIX Annual Technical Conference*. 823–835.

[31] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, and Kunle Olukotun. 2020. Taurus: An intelligent data plane. *arXiv preprint arXiv:2002.08987* (2020).

[32] Balázs Vass, Erika Bérczi-Kovács, Costin Raiciu, and Gábor Rétvári. 2020. Compiling Packet Programs to Reconfigurable Switches: Theory and Algorithms. In *P4 Workshop in Europe*. 28–35.

[33] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *USENIX Annual Technical Conference*. 43–56.

[34] Liangcheng Yu, John Sonchack, and Vincent Liu. 2020. Mantis: Reactive Programmable Switches. In *ACM SIGCOMM*. 296–309.

[35] Lior Zeno, Dan R. K. Ports, Jacob Nelson, and Mark Silberstein. 2020. SwiShmem: Distributed Shared State Abstractions for Programmable Switches. In *ACM SIGCOMM HotNets Workshop*.

Appendices are supporting material that has not been peer-reviewed.

A LUCID'S TYPE SYSTEM

$\langle \epsilon\ (stages) \rangle$ ::= $0 \mid 1 \mid 2 \mid \dots$

$\langle T\ (base\ types) \rangle$ ::= $\text{Unit} \mid \text{Int}$

$\langle tau\ (types) \rangle$ ::= $T \mid \text{ref}\,(T, \epsilon) \mid (\tau, \epsilon) \rightarrow (\tau, \epsilon)$

$\langle x\ (variables) \rangle$ ::= alphanumeric

$\langle g\ (global\ variables) \rangle$ ::= $g_0 \mid \dots \mid g_{n-1}$

$\langle v\ (values) \rangle$ ::= $() \mid \mathbb{Z} \mid g \mid \texttt{fun}\ (x : \tau, \epsilon) \rightarrow e$

$\langle e\ (expressions) \rangle$ ::= $v \mid x \mid e + e \mid \text{let } x = e \text{ in } e \mid !e \mid e := e \mid e\ e$

Figure 18: A toy language and type system

INT
$$\frac{n \in \mathbb{Z}}{\Gamma, \epsilon \vdash n : \texttt{Int}, \epsilon}$$

UNIT
$$\frac{}{\Gamma, \epsilon \vdash () : \texttt{Unit}, \epsilon}$$

GLOBAL VARIABLE
$$\frac{}{\Gamma, \epsilon \vdash g_i : \texttt{ref}(T_i, i), \epsilon}$$

LOCAL VARIABLE
$$\frac{\Gamma[x] = \tau}{\Gamma, \epsilon \vdash x : \tau, \epsilon}$$

PLUS
$$\frac{\Gamma, \epsilon \vdash e_1 : \texttt{Int}, \epsilon_1 \qquad \Gamma, \epsilon_1 \vdash e_2 : \texttt{Int}, \epsilon_2}{\Gamma, \epsilon \vdash e_1 + e_2 : \texttt{Int}, \epsilon_2}$$

LET
$$\frac{\Gamma, \epsilon \vdash e_1 : \tau_1, \epsilon_1 \qquad \Gamma[x := \tau_1], \epsilon_1 \vdash e_2 : \tau_2, \epsilon_2}{\Gamma, \epsilon \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2, \epsilon_2}$$

DEREF
$$\frac{\Gamma, \epsilon \vdash e : \texttt{ref}(T, \epsilon_1), \epsilon_2 \qquad \epsilon_2 \leq \epsilon_1}{\Gamma, \epsilon \vdash !e : T, \epsilon_1 + 1}$$

UPDATE
$$\frac{\Gamma, \epsilon \vdash e_1 : T, \epsilon_1 \qquad \Gamma, \epsilon_1 \vdash e_2 : \texttt{ref}(T, \epsilon_2), \epsilon_3 \qquad \epsilon_3 \leq \epsilon_2}{\Gamma, \epsilon \vdash e_2 := e_1 : \texttt{Unit}, \epsilon_2 + 1}$$

ABS
$$\frac{\Gamma[x := \tau_{in}], \epsilon_{in} \vdash e : \tau_{out}, \epsilon_{out}}{\Gamma, \epsilon \vdash \texttt{fun}\ (x : \tau_{in}, \epsilon_{in}) \rightarrow e : (\tau_{in}, \epsilon_{in}) \rightarrow (\tau_{out}, \epsilon_{out}), \epsilon}$$

APP
$$\frac{\Gamma, \epsilon \vdash e_1 : (\tau_{in}, \epsilon_{in}) \rightarrow (\tau_{out}, \epsilon_{out}), \epsilon_1 \qquad \Gamma, \epsilon_1 \vdash e_2 : \tau_{in}, \epsilon_2 \qquad \epsilon_2 \leq \epsilon_{in}}{\Gamma, \epsilon \vdash e_1\ e_2 : \tau_{out}, \epsilon_{out}}$$

Figure 19: Typing rules

This appendix presents a simplified definition of Lucid's type system, as well as an operational semantics and soundness proof. To begin, figure 18 defines a grammar for a model ML-like language on which we will define our type-and-effect system. We present the system for this language purely for convenience; adapting the rules to Lucid's C-like syntax presents no theoretical challenge.

The system is defined with respect to some predefined, ordered set of n global variables g_0 through g_{n-1}, each of which has an associated *base type* T_i. Base types are simply types which do not reference stages – in this example, the only two base types are `Unit` and `Int`. Despite the name, global variables are treated as values in the language, not as variables. They can be thought of exactly like ref cells in OCaml, and this line of thinking inspires much of the syntax used for them in the language.

Effects in this system are called *stages*, and are used to track which global variables have been used so far. Intuitively, the stage i represents the pipeline stage containing g_i. We begin typechecking at stage 0, and increment the stage when global variables are used. We can then ensure that the global variables are used in order by only allowing variable g_i to be used if the current stage is at most i.

The types in this system are mostly straightforward, but note that functions now have starting and ending stages as well as input and output types, and we have a `ref` type representing the type of a global variable – in general, the type of g_i is `ref` (T_i, i).

Expressions in this language are also straightforward, except that we have two operators on global variables: dereference ($!e$), which returns the current value of the variable, and update $e_1 := e_2$, which updates global variable e_1 to hold the value of e_2. Both of these operators access the global variable, and hence should never be used out-of-order. Note that these operators do not exist in Lucid; instead, there are several builtin functions for performing these accesses.

A.0.1 The Typing Judgement. Our typing judgement has the form $\Gamma, \epsilon_1 \vdash e : \tau, \epsilon_2$, where Γ is an environment which maps local variables to values. In English, this judgement can be read as "starting with environment Γ at stage ϵ_1, the expression e has type τ and will finish evaluation in stage ϵ_2". The typing rules are presenting in 19.

The most interesting rules here are the DEREF and UPDATE rules, as they are the ones which interact with stages. Each first typechecks its subexpression(s) and expects to receive a global variable g_i (i.e. something with `ref` type) as its first argument. Crucially, neither rule can be applied unless the stage after evaluating the subexpression(s) is at most i. If this is satisfied, typechecking finishes in stage $i + 1$. There is a similar constraint on the function application rule (APP), which specifies that the current stage when beginning to evaluate a function be at most the function's starting effect.

A.0.2 Extensions in Practice. For clarity, we only present a minimal system here. In practice, the algorithm we implemented for Lucid programs differs in two ways beyond simple syntactic differences. First, it performs type and stage *inference*, rather than simply checking the user's annotations, using an imperative algorithm analogous to Algorithm J [22].

Our algorithm also allows for polymorphic functions, so that a single function definition can be re-used for different input types or at different starting stages. For example, a function which takes two global variables as arguments and accesses them in order should work for any two arguments where the first is less than the second. To express this, function types can be extended contain constraints on polymorphic stages which appear in the type. These constraints have the form $\epsilon \leq \epsilon$, and can be automatically inferred and checked by the type system.

$$\textsc{plus-1}\ \frac{(G, n, e_1) \rightarrow (G', n', e_1')}{(G, n, e_1 + e_2) \rightarrow (G', n', e_1' + e_2)} \qquad \textsc{plus-2}\ \frac{(G, n, e_2) \rightarrow (G', n', e_2')}{(G, n, v + e_2) \rightarrow (G', n', v + e_2')}$$

$$\textsc{plus-2}\ \frac{v_1, v_2 \in \mathbb{Z} \qquad v_3 \text{ is the integer sum of } v_1 \text{ and } v_2}{(G, n, v_1 + v_2) \rightarrow (G, n, v_3)}$$

$$\textsc{let-1}\ \frac{(G, n, e_1) \rightarrow (G', n', e_1')}{(G, n, \texttt{let } x = e_1 \texttt{ in } e_2) \rightarrow (G', n', \texttt{let } x = e_1' \texttt{ in } e_2)}$$

$$\textsc{let-2}\ \frac{}{(G, n, \texttt{let } x = v \texttt{ in } e_2) \rightarrow (G, n, e_2[v/x])} \qquad \textsc{deref-1}\ \frac{(G, n, e) \rightarrow (G', n', e')}{(G, n, !e) \rightarrow (G', n', !e')}$$

$$\textsc{deref-2}\ \frac{n \leq i}{(G, n, !g_i) \rightarrow (G, i + 1, G[i])} \qquad \textsc{update-1}\ \frac{(G, n, e_1) \rightarrow (G', n', e_1')}{(G, n, e_2 := e_1) \rightarrow (G', n', e_2 := e_1')}$$

$$\textsc{update-2}\ \frac{(G, n, e_2) \rightarrow (G', n', e_2')}{(G, n, e_2 := v) \rightarrow (G', n', e_2' := v)} \qquad \textsc{update-3}\ \frac{n \leq i}{(G, n, g_i := v) \rightarrow (G[i := v], i + 1, ())}$$

$$\textsc{app-1}\ \frac{(G, n, e_1) \rightarrow (G', n', e_1')}{(G, n, e_1\ e_2) \rightarrow (G', n', e_1'\ e_2)} \qquad \textsc{app-2}\ \frac{(G, n, e_2) \rightarrow (G', n', e_2')}{(G, n, v\ e_2) \rightarrow (G', n', v\ e_2')}$$

$$\textsc{app-3}\ \frac{v_1 = \texttt{fun } (x : \tau, \epsilon) \rightarrow e}{(G, n, v_1\ v_2) \rightarrow (G, n, e[v_2/x])}$$

Figure 20: Operational Semantics

B SOUNDNESS OF TYPE SYSTEM

In this section we define an operational semantics for the example language defined in Appendix A, and prove the soundness of our type system.

B.1 Operational Semantics

Our small-step operational semantics is defined on *states* of our program, which are three tuples (G, n, e). Here, G is an array of values such that $G[i]$ is the current value of global variable g_i. We write $G[i]$ for the value in G at index i, and $G[i := v]$ for the array with all entries the same as G, but where index i has value v instead. We say that G is well-typed if $G[i]$ has type T_i for all i; that is, if $\emptyset, \epsilon \vdash G[i] : T_i, \epsilon$ for all ϵ.

n is an index into the array indicating the next global variable to be used – global variables with index less than n are inaccessible. Finally, e is the expression we are evaluating.

Note the syntactic convention that the metavariable v will only be used to represent expressions which are values. We use a standard definition of variable substitution, where $e[v/x]$ means "e with the value v substituted for the variable x wherever it appears".

B.2 Important Lemmas

We first prove a number of useful lemmas. The Canonical Forms lemma says that typing a value does not change the stage, and that only integer values have type integer, and similarly for other types. The Substitution lemma states that uniformly replacing a variable with a value of the same type does not affect our ability to type an expression. Finally, the weakening lemma says that if an expression typechecks starting at some stage, then it also typechecks starting from any earlier stage.

Lemma (Canonical forms): If $\emptyset, \epsilon \vdash v : \tau, \epsilon'$, then $\epsilon = \epsilon'$ and:

- if $\tau = \texttt{Int}$ then $v \in \mathbb{Z}$
- if $\tau = \texttt{ref}(T, k)$, then $v = g_k$ and $T = T_k$.
- if $\tau = (\tau_{in}, \epsilon_{in}) \rightarrow (\tau_{out}, \epsilon_{out}$ then $v = \texttt{fun } (x : \tau_{in}, \epsilon_{in}) \rightarrow e$ and $\emptyset[x := \tau_{in}], \epsilon_{in} \vdash e : \tau_{out}, \epsilon_{out}$.

Proof: Inversion of the typing relation. ■

Definition: If Γ is a map, let $\Gamma \backslash x$ denote the same map without a binding for x.

Lemma (Substitution Lemma): If $\Gamma[x] = \tau$ and $\emptyset, i \vdash v : \tau, i$ and $\emptyset, \epsilon \vdash e : \tau', \epsilon'$, then $\Gamma \backslash x, \epsilon \vdash e[v/x] : \tau', \epsilon'$

Proof: This is a standard lemma and may be proved for our language in the standard way. ■

Lemma (Weakening): If $\emptyset, \epsilon \vdash e : \tau, \epsilon'$, and $\epsilon_1 \leq \epsilon'$, then there is some $\epsilon_1' \leq \epsilon'$ such that $\emptyset, \epsilon_1 \vdash e : \tau, \epsilon_1'$.

Proof: Straightforward induction on the typing derivation. ■

B.3 Progress

We prove our soundness theorem in the standard way: by combining progress and preservation lemmas.

Theorem (Progress): If $\emptyset, i \vdash e : \tau, j$ then either e is a value or for all well-typed G there exist some G', j', e' such that $(G, i, e) \rightarrow (G', j', e')$.

Proof: Structural induction on the typing derivation.

Case INT/UNIT/GLOBAL VARIABLE/ABS: In these cases the expression is already a value, so the result is trivial.

Case LOCAL VARIABLE: This case is impossible, as our typing judgement contains an empty environment.

Case PLUS: In this case, $e = e_1 + e_2$. By induction, either e_1 is a value or it steps to some (G', i', e_1'). In the latter case, we may apply rule PLUS-1 to show that $(G, i, e_1 + e_2) \rightarrow (G', i', e_1' + e_2$.

Similarly, either e_2 is a value or it steps to some other (G', i', e_2'). If e_1 is a value and e_2 steps, then we may apply rule PLUS-2 to show that $(G, i, e_1 + e_2) \rightarrow (G', i', e_1 + e_2')$

Finally, if both e_1 and e_2 are values, then note that by the premises of the PLUS rule, both have type Int. By our canonical forms lemma, $e_1, e_2 \in Z$, so we may apply the PLUS-3 rule to show that $(G, i, e_1 + e_2) \rightarrow (G, i, v)$ where v is the sum of e_1 and e_2.

Case LET: In this case, $e = \texttt{let } x = e_1 \texttt{ in } e_2$. By induction, either e_1 is a value or it steps to something. In the latter case we may apply rule LET-1; otherwise, we may apply rule LET-2.

Case DEREF: In this case, $e = !e_1$. By induction, either e_1 is a value or it steps to something. In the latter case we may apply rule DEREF-1; otherwise, note that we have the premise $\emptyset, i \vdash e_1 : \texttt{ref}(T, k), j'$ where $j' \leq k$. Since e_1 is a value, our canonical forms lemma tells us that $e_1 = G_k$, and $j' = i$. Hence $i = j' \leq k$, so we can apply rule DEREF-2 to show that $(G, i, !e_1) \rightarrow (G, k + 1, G[k])$.

Case UPDATE: In this case, $e = e_2 := e_1$. As in previous parts, the only interesting case is when e_1 and e_2 are both values. In that case, as in the DEREF rule, canonical forms tells us that $e_2 = g_k$ for some $k \geq i$, and thus we can apply rule UPDATE-3.

Case APP: In this case, $e = \texttt{let } x = e_1 \texttt{ in } e_2$. The reasoning is analogous to the PLUS case. ■

B.4 Preservation

Theorem (Preservation): If $\emptyset, i \vdash e : \tau, j$, G is well-typed, and $(G, i, e) \rightarrow (G', i', e')$, then G' is also well-typed, and there is some $j' \leq j$ such that $\emptyset, i' \vdash e' : \tau, j'$.

Proof: Structural induction on the typing derivation.

Case INT/UNIT/GLOBAL VARIABLE: This case cannot occur, because values do not evaluate to anything.

Case LOCAL VARIABLE: This case also cannot occur, because the typing judgement contains an empty environment.

Case PLUS: In this case $e = e_1 + e_2$ and $\tau = \texttt{Int}$. Thus the proof that e steps must have used either PLUS-1, PLUS-2, or PLUS-3. We also have the premises of the PLUS rule: $\emptyset, i \vdash e_1 : \texttt{Int}, k$ and $\emptyset, k \vdash e_2 : \texttt{Int}, j$.

- If we used PLUS-1, then we know that $(G, i, e_1) \rightarrow (G', i', e_1')$ and $e' = e_1' + e_2$. Thus by induction, G' is well-typed and $\emptyset, i' \vdash e_1' : \texttt{Int}, k'$ for some $k' \leq k$. We may use weakening on the second premise to obtain $\emptyset, k' \vdash e_2 : \texttt{Int}, j'$ for some $j' \leq j$, and combine these judgements to show that $\emptyset, i' \vdash e_1' + e_2 : \texttt{Int}, j'$ as required.
- If we used PLUS-2, then we know that $(G, i, e_2) \rightarrow (G', i', e_2')$ and $e' = e_1 + e_2'$. We also know that e_1 is a value, so by canonical forms $e_1 \in \mathbb{Z}$ and $k = i$. Thus we may combine this with the second premise and use induction to conclude that G' is well-typed, and that $\emptyset, i' \vdash e_2' : \texttt{Int}, j'$ for some $j' \leq j$. Since $e_1 \in \mathbb{Z}$ we may use the INT rule to show that $\emptyset, i' \vdash e_1 : \texttt{Int}, j$, and combine this with the previous judgement to show that $\emptyset, i' \vdash e_1 + e_2' \vdash \texttt{Int}, j'$ as required.
- If we used PLUS-3, we know that $G' = G$ and is hence well-typed, $j = i$, and both e_1 and e_2 are values, so $e_1, e_2 \in \mathbb{Z}$ and thus $e' \in \mathbb{Z}$ as well. Thus we may simply use the INT rule to show that $\emptyset, i \vdash e' : \texttt{Int}, j$ as required.

Case LET: In this case, $e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$, and we have the premises $\emptyset, i \vdash e_1 : \tau_1, k$ and $\emptyset[x := \tau_1], k \vdash e_2 : \tau, j$.

The proof that e steps must have used either LET-1 or LET-2. The LET-1 case is analogous to the PLUS-1 case. In the LET-2 case, we know that e_1 is a value, $G' = G$ and hence is well-typed, and $i = j$. By canonical forms, $k = i$. By the substitution lemma, the second premise becomes $\emptyset, i \vdash e_2[e_1/x] : \tau, j$, which is precisely what we wanted to show.

Case DEREF: In this case, $e = !e_1$, and we have the premises that $\emptyset, i \vdash e_1 : \texttt{ref}(T, k), k'$ where $k' \leq k$ and $\tau = T$. As usual, we must have used either the DEREF-1 or DEREF-2 rule to prove that e steps. The DEREF-1 case is analogous to the PLUS-1 case.

If we used DEREF-2, then we know that $G' = G$ is well-typed, $j = k + 1$, e_1 is a value, and $e' = G[k]$. By canonical forms, $e_1 = g_k$ and $\tau = T = T_k$. We need only show that $\emptyset, k + 1 \vdash G[k] : \tau, k + 1$, which follows immediately from G being well-typed.

Case UPDATE: In this case, $e = e_2 := e_1$, $\tau = \texttt{Unit}$, and we have the premises that $\emptyset, i \vdash e_1 : T, k_1$, $\emptyset, k_1 \vdash e_2 : \texttt{ref}(T, k_2), k_3$ where $k_3 \leq k_2$. As usual, we must have used either UPDATE-1, UPDATE-2, or UPDATE-3 to show that e steps, and the first two cases are analogous to PLUS-1 and PLUS-2, respectively.

In the UPDATE-3 case, we know that the output value is (), which can trivially be typed using the UNIT rule. So we need only show that $G' = G[k_2 := e_1$ is well-typed. But since e_1 is a value, we must have used the INT or UNIT rule to prove the first premise, and that rule works for all ϵ. Thus $G'[k_2]$ has the right type, and all other entries are unchanged, so G' is well-typed.

Case APP: In this case, $e = e_1\ e_2$, and we have the premises $\emptyset, i \vdash e_1 : (\tau_{in}, \epsilon_{in}) \rightarrow (\tau_{out}, \epsilon_{out}), k$ and $\emptyset, k \vdash e_2 : \tau_{in}, k_2$ where $k_2 \leq \epsilon_{in}$, $\tau = \tau_{out}$ and $j = \epsilon_{out}$. As usual, we must have used either the APP-1, APP-2, or APP-3 rules here, and the first two cases are again analogous to PLUS-1 and PLUS-2.

If we used the APP-3 rule, then we know that $G' = G$ is well-typed, that $i' = i$, that $e_1 = \texttt{fun}(x : \tau_1, \epsilon_1) \rightarrow e_{body}$ and e_2 are both values, and that $e' = e_{body}[e_2/x]$. Since both e_1 and e_2 are values, by canonical forms we know that $i = k = k_2$.

By the canonical forms lemma on e_1, we know that $\emptyset[x := \tau_{in}], \epsilon_{in} \vdash e_{body} : \tau_{out}, \epsilon_{out}$. Now, e_2 is a value, and by the second premise it has type τ_{in}; thus by the substitution lemma $\emptyset, \epsilon_{in} \vdash e_{body}[e_2/x] : \tau_{out}, \epsilon_{out}$. Since $i = k_2 \leq \epsilon_{in}$, by weakening there is some ϵ_{out}' such that $\emptyset, i \vdash e_{body}[e_2/x] : \tau_{out}, \epsilon_{out}'$ ■

B.5 Soundness

Finally, we combine the progress and preservation theorems to prove that expressions which typecheck always evaluate – that is, "Well-typed programs do not get stuck". We denote the transitive closure of the evaluation relation by $\rightarrow^*$.

Theorem (Soundness): If $\emptyset, \epsilon_1 \vdash e : \tau, \epsilon_2$, then either e is a value, or $e \rightarrow e'$ and there is some ϵ_1' such that $\emptyset, \epsilon_1' \vdash e' : \tau, \epsilon_2$.

Proof: Inductively apply preservation to show that G_2 and e_2 are well-typed, then apply progress to show that e_2 is either a value, or another step can be taken. ■