Packet Transactions: A Programming Model for Data-Plane Algorithms at Hardware Speed

Anirudh Sivaraman[*], Mihai Budiu[†], Alvin Cheung[‡], Changhoon Kim[†], Steve Licking[†],
George Varghese[++], Hari Balakrishnan[*], Mohammad Alizadeh[*], Nick McKeown[+]
[*]*MIT CSAIL,* [†]*Barefoot Networks,* [‡]*University of Washington,* [++]*Microsoft Research,* [+]*Stanford University*

Abstract

Data-plane algorithms execute on every packet traversing a network switch; they encompass many schemes for congestion control, network measurement, active-queue management, and load balancing. Because these algorithms are implemented in hardware today, they cannot be changed after being built. To address this problem, recent work has proposed designs for programmable line-rate switches. However, the languages to program them closely resemble the underlying hardware, rendering them inconvenient for this purpose.

This paper presents Domino, a C-like imperative language to express data-plane algorithms. Domino introduces the notion of a *packet transaction*, defined as a sequential code block that is atomic and isolated from other such code blocks. The Domino compiler compiles Domino code to PISA, a family of abstract machines based on emerging programmable switch chipsets. We show how Domino enables several data-plane algorithms written in C syntax to run at hardware line rates.

1 Introduction

Network switches and routers in modern datacenters, enterprises, and service-provider networks perform a variety of tasks in addition to standard packet forwarding. The set of requirements for routers has only increased with time as network operators have sought to exercise greater control over performance and security, and to support an evolving set of network protocols.

Performance and security may be improved using both data-plane and control-plane mechanisms. This paper focuses on data-plane algorithms. These algorithms process every data packet, transforming the packet and often also some state maintained in the switch. Examples of such algorithms include congestion control with switch feedback [63, 89, 61, 34], active-queue management [56, 55, 66, 71, 74], network measurement [93, 54, 53], and load-balanced routing in the data plane [33].

Because data-plane algorithms process every packet, an important requirement is the ability to process packets at the switch's line rate. As a result, these algorithms are typically implemented using dedicated hardware. However, hardware designs are rigid and prevent reconfigurability in the field. They make it difficult to implement and deploy new algorithms without investing in new hardware—a time-consuming and expensive proposition.

This rigidity affects vendors [6, 10, 3] building network switches with merchant-silicon chips [12, 13, 22], network operators deploying switches [81, 77, 60], and researchers developing new switch algorithms [63, 71, 92, 94, 61]. To run data-plane algorithms after a switch has been built, researchers and companies have attempted to build programmable routers for many years, starting from efforts on active networks [90] to network processors [79] to software routers [65, 49]. All these efforts have compromised on speed to provide programmability, typically running an order of magnitude (or worse) slower than hardware line rates. Unfortunately, this reduction in performance has meant that these systems are rarely deployed in production networks, if at all.

Programmable switching chips [16, 32, 39], which are competitive with state-of-the-art fixed-function chipsets [12, 13, 22], are a recent alternative. These chips implement a few low-level hardware primitives that can be configured by software into a processing pipeline [5, 4, 87]. This approach is attractive because it does not compromise on data rates and the area overhead of programmability is small [39].

P4 [38, 26] is an emerging packet-processing language for such chips. P4 allows the programmer to specify packet parsing and processing without restricting the set of protocol formats or the set of actions that can be executed when matching packet headers in a match-action table. Data-plane tasks that require header rewriting can be expressed naturally in P4 [83].

By contrast, many data-plane algorithms don't rewrite headers, but instead manipulate internal switch state in a manner unique to each algorithm. We believe that network programmers would prefer the convenience of an imperative language such as C that directly captures the algorithm's intent without shoehorning algorithms into hardware constructs such as a sequence of match-action tables like P4 requires them to. Furthermore, this is predominantly how such algorithms are expressed in pseudocode [56, 88, 2, 66, 55], and implemented in sofware routers [65, 11, 49], network processors [50, 59], and at network endpoints [8].

This paper presents Domino, a new domain-specific language (DSL) for data-plane algorithms. Domino is an imperative language with C-like syntax. The key abstraction in Domino is a *packet transaction* (§3): a sequential code block that is atomic and isolated from other such code blocks. Packet transactions provide a convenient programming model, because they allow the programmer to focus on the operations needed for each packet without worrying about other packets that are concurrently being processed.

The Domino compiler (§4) compiles packet transactions to a family of abstract machines called PISA (§2) (for Protocol-Independent Switch Architecture). PISA generalizes the Reconfigurable Match-Action Table (RMT) [39] model and captures essential features of line-rate programmable switches [39, 32, 16]. In addition, PISA introduces *atoms* to represent atomic computations provided natively by a PISA machine, much like load-link/store-conditional, and packed-multiply-and-add on x86 machines today [14]. Atoms provide hardware support for packet transactions, similar to how an atomic test-and-set can implement an atomic increment.

To evaluate Domino, we express algorithms such as RCP [89], CoDel [71], heavy-hitter detection [93], and CONGA [33], as packet transactions in Domino (§5). Expressing these algorithms involved a straightforward translation of each algorithm's reference code to Domino syntax. The Domino compiler guarantees deterministic performance for these algorithms: all packet transactions that compile run at line rate on a PISA machine, or will be rejected by the compiler. We use the Domino compiler to determine if each algorithm can run at line rate (Table 4) on several different PISA machines that differ in the atoms they provide.

Our results indicate that it is possible to provide a familiar programming model, resembling DSLs for software routers and NPUs, and also achieve line-rate performance. These findings help resolve the concerns raised in recent work [38] that expressive languages are unsuitable for line-rate packet processing.

2 An abstract machine for switches

This section describes PISA (Protocol-Independent Switch Architecture [28]), a family of abstract machines for programmable switches that differ in the computational capabilities they provide. PISA machines serve as compiler targets for Domino programs. PISA's design is inspired by recent line-rate programmable switch architectures, such as RMT [39], Intel's FlexPipe [16], and Cavium's XPliant Packet Architecture [32], which we outline briefly first.

2.1 Programmable switch architectures

Programmable switches follow the switch model shown at the top of Figure 1. Packets arriving to the switch are parsed by a programmable parser that turns packets into header fields. These header fields are first processed by an ingress pipeline consisting of match-action tables arranged in stages. Following the ingress pipeline, the packet is queued. Once the packet is dequeued by the switch scheduler, it is processed by a similar egress pipeline before being transmitted from the switch.

2.2 The PISA abstract machine

PISA (the bottom half of Figure 1) models a switch pipeline such as the ingress or egress pipeline. A pipeline in PISA consists of a number of pipeline stages that execute synchronously on every time step. An incoming packet is processed by each stage and handed off to the next, until it exits the pipeline. Each stage has one time step of latency, where the time step is a physical quantity determined by the hardware. The inverse of this time step is the line rate supported by the pipeline. For instance, the RMT architecture has a line rate of 960 Million pkts / sec [39].

As an abstract machine, PISA only models components pertinent to data-plane algorithms. In particular, it models the computation within a match-action table in a stage (i.e., the action half of the match-action table), but not the match semantics (e.g., direct, ternary, or longest prefix). PISA also does not model packet parsing and assumes that packets arriving to it are already parsed.

2.3 Atoms: PISA's processing units

In PISA, each pipeline stage contains a vector of *atoms*. All atoms in this vector execute in parallel on every time step. Informally, an atom is an atomic unit of packet processing, which the PISA machine supports natively in hardware. We represent an atom as a body of sequential code. An atom completes execution of this body of code and modifies a packet before processing the next

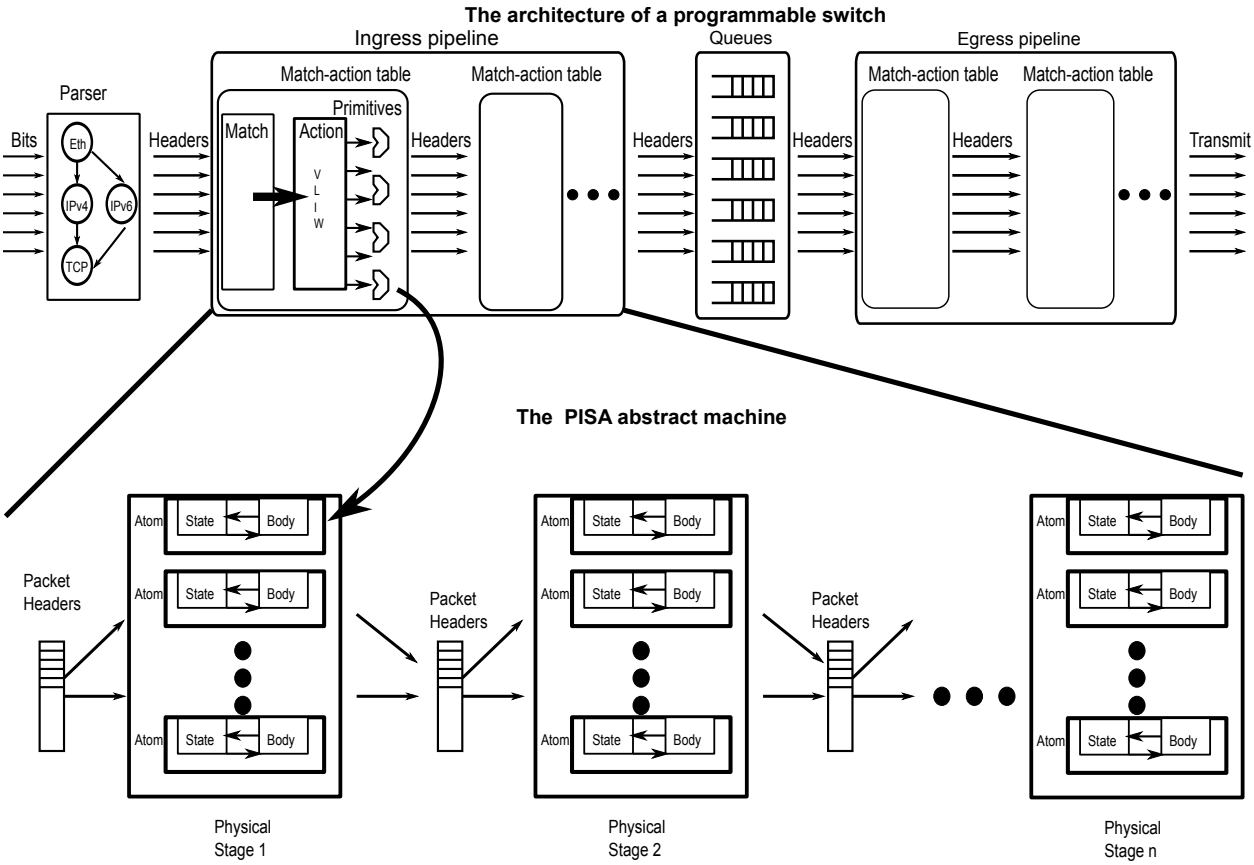


Figure 1: The PISA abstract machine and its relationship to programmable switch architectures.

packet. An atom may also contain internal state that persists across packets and influences the atom's behavior from one packet to the next. For instance, a switch counter that wraps around at 100 can be written as the atom below.[1]

```
if (counter < 99)
  counter++;
else
  counter = 0;
```

Similarly, a stateless operation that sets a packet field (such as P4's `modify_field` primitive [26]) can be written as the atom below:

```
p.field = value;
```

PISA generalizes several aspects of existing programmable switch architectures. The vector of atoms in each stage generalizes RMT's very-large instruction-word (VLIW) [39] that executes primitive actions on packet fields in parallel. Internal state within an atom models persistent switch state such as meters, counters, and P4's register abstraction [26] in a unified manner. We assume all state is initialized by the switch control plane, which we don't explicitly model in PISA.

2.4 Constraining atoms

Atoms in PISA execute on every time step, reading all packet fields at the beginning and writing all packet fields at the end of a time step. To prevent data races, PISA forbids two atoms in a stage from writing to the same packet field. To provide deterministic performance at line rate, atoms must be suitably constrained. We impose two such constraints that distinguish PISA from software routers [65] and network processors [19] that sacrifice determinism for programmability.

First, PISA machines are *shared-nothing*: each atom maintains a certain number of state variables that are local to that atom alone. Their values can be communicated to atoms in subsequent stages only when the values are copied into packet fields. This restriction reflects the capabilities of line-rate switches: accessing shared memory from multiple switch stages is technically challenging because it requires multi-ported RAMs and routing long wires on the chip.

Second, we constrain the complexity of atoms by

[1] We use `p.x` to represent field `x` within a packet `p` and `x` to represent a state variable `x` that persists across packets.

defining *atom templates* (§4.6). An atom template is a program that always terminates and specifies how the atom is executed. One example is an ALU with a restricted set of primitive operations to choose from (Figure 12). Atom templates allow us to create different PISA machines that support different atoms natively. In practice, we expect such atom templates to be designed by an ASIC engineer and exposed as part of a PISA machine's instruction set. As programmable switches evolve, the capabilities of atoms will evolve as well. However, atoms cannot be arbitrarily complex: the line rate is inversely proportional to an atom's execution latency (§5.3).

3 Programming using packet transactions

We now illustrate programming using packet transactions in Domino, using flowlet switching [82] as an example. Flowlet switching is a load-balancing algorithm that sends bursts of packets (called flowlets) from a TCP flow on different paths, provided the bursts are separated by a large enough interval in time to ensure packets do not arrive out of order at a TCP receiver. Figure 2 shows flowlet switching as expressed in Domino. For simplicity, the example hashes only the source and destination ports; it is easy to extend it to the full 5-tuple.

This example demonstrates the core language constructs in Domino. All packet processing happens in the context of a packet transaction (the function `flowlet` starting at line 17). The function's argument `pkt` declares the fields in a packet (lines 5–12)[2] that can be referenced by the function body (lines 18–32). In addition, the function body can reference state variables that represent persistent state stored on the switch. These are declared as global variables (e.g. `last_time` and `saved_hop` declared on lines 14 and 15, respectively).

Conceptually, the switch invokes the packet transaction function on each incoming packet sequentially. The function modifies the passed-in packet argument until the end of the function body, before processing the next packet. Domino forbids return statements, and hence execution will always end at the end of the function body. The function may invoke *intrinsics* such as `hash2` on lines 23 and `hash3` on line 18. Intrinsics are hardware primitives provided by the abstract machine that are not interpreted by Domino. The Domino compiler uses an intrinsic's signature to infer dependencies and supplies a canned run-time implementation, but otherwise does not interpret or analyze intrinsics. The overall language is a constrained subset of C (Table 1).

As an illustration of Domino's constraints, arrays can be used as state variables alone but not as packet fields. Furthermore, all accesses to a given array within one execution of a transaction, i.e. one packet, must use the same array index. For instance, accesses to the array `last_time` use the index `pkt.id`, which is constant for each packet, but changes from one packet to the next. This restriction simplifies the treatment of arrays in the compiler, while still allowing us to express data-plane algorithms of practical interest. The restrictions in Table 1 seem severe, but are required for deterministic performance. Memory allocation, unbounded iteration counts, and unstructured control flow cause variable performance. These are precisely the restrictions in Domino relative to software routers like Click [65] with greater flexibility and variable performance.

No iteration (while, for, do-while).
No switch, goto, return, break, or continue.
No pointers.
No dynamic memory allocation / heap.
Array index must be constant for every execution of the transaction.
No access to packet data i.e. unparsed portion of the packet.
No arrays in packet fields.

Table 1: Restrictions in Domino

When compiled to a PISA machine (§2), the Domino compiler converts the code in Figure 2 into the atom pipeline shown in Figure 3. The next section describes the steps involved in this compilation.

4 The Domino compiler

The Domino compiler borrows many techniques from the compiler literature. The PISA architecture, however, poses unique challenges for compilation requiring a synthesis of techniques that, to the best of our knowledge, is novel. Further, as we illustrate throughout this section, constraining Domino for deterministic performance simplifies the Domino compiler relative to mainstream compilers for imperative languages.

Because Domino's syntax is a subset of C, we use Clang's library interface [20] to parse and implement the passes in the compiler. The overall architecture of the compiler is shown in Figure 4. Throughout this section, we use flowlet switching from Figure 2 as a running example to demonstrate compiler passes. While we have simplified the code for readability, the code output by the Domino compiler after each pass isn't materially different from the version presented here.

[2] We use fields to refer to both packet headers such as source port (`sport`) and destination port (`dport`) and packet metadata (`id`).

```
1  #define NUM_FLOWLETS     8000
2  #define THRESHOLD        5
3  #define NUM_HOPS         10
4
5  struct Packet {
6    int sport;
7    int dport;
8    int new_hop;
9    int arrival;
10   int next_hop;
11   int id; // array index
12 };
13
14 int last_time [NUM_FLOWLETS] = {0};
15 int saved_hop [NUM_FLOWLETS] = {0};
16
17 void flowlet(struct Packet pkt) {
18   pkt.new_hop = hash3(pkt.sport,
19                       pkt.dport,
20                       pkt.arrival)
21                 % NUM_HOPS;
22
23   pkt.id  = hash2(pkt.sport,
24                   pkt.dport)
25             % NUM_FLOWLETS;
26
27   if (pkt.arrival - last_time[pkt.id]
28       > THRESHOLD)
29   { saved_hop[pkt.id] = pkt.new_hop; }
30
31   last_time[pkt.id] = pkt.arrival;
32   pkt.next_hop = saved_hop[pkt.id];
33 }
```

Figure 2: Flowlet switching written in Domino

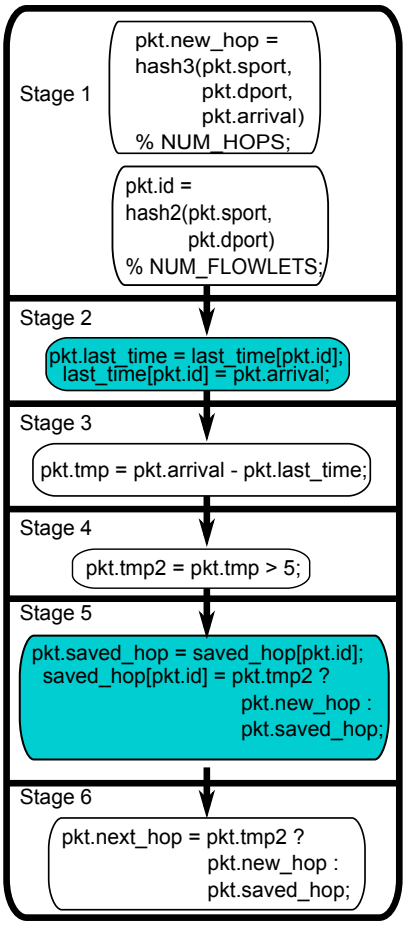


Figure 3: Compiled 6-stage PISA pipeline implementing flowlet switching. Control flows from top to bottom. Atoms manipulating state are shaded in blue.

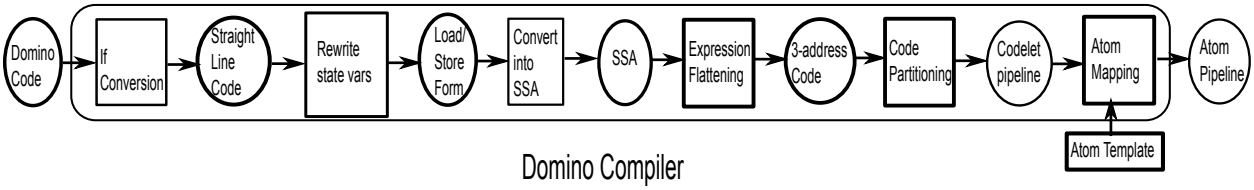


Figure 4: Passes in the Domino compiler

```
if (pkt.arrival - last_time[pkt.id] >
    THRESHOLD) {
  saved_hop[pkt.id] = pkt.new_hop;
}
```

$\Longrightarrow$

```
pkt.tmp = pkt.arrival - last_time[pkt.id] > THRESHOLD;
saved_hop[pkt.id] = // Rewritten
  pkt.tmp ? pkt.new_hop : saved_hop[pkt.id];
```

Figure 5: Conversion to straight-line code

4.1 If-conversion to straight-line code

A packet transaction's body can contain (potentially nested) conditional statements (e.g., Lines 27 to 29 in Figure 2, or CoDel [2]). These statements alter control flow and complicate dependence analysis between statements i.e. whether a statement should follow or precede another. We eliminate such statements by transforming

```
pkt.id = hash2(pkt.sport,                    pkt.id = hash2(pkt.sport,            // Read flank
               pkt.dport)                                 pkt.dport)
         % NUM_FLOWLETS;                               % NUM_FLOWLETS;
...                                  ⟹      pkt.last_time = last_time[pkt.id];   // Read flank
last_time[pkt.id] = pkt.arrival;             ...
...                                          pkt.last_time = pkt.arrival;         // Rewritten
                                             ...
                                             last_time[pkt.id] = pkt.last_time;   // Write flank
```

Figure 6: Adding read and write flanks

```
pkt.id = hash2(pkt.sport,                    pkt.id0 = hash2(pkt.sport,           // Rewritten
               pkt.dport)                                  pkt.dport)
               % NUM_FLOWLETS;                             % NUM_FLOWLETS;
pkt.last_time = last_time[pkt.id]; ⟹         pkt.last_time0 = last_time[pkt.id0]; // Rewritten
...                                          ...
pkt.last_time = pkt.arrival;                 pkt.last_time1 = pkt.arrival;        // Rewritten
last_time[pkt.id] = pkt.last_time;           last_time[pkt.id0] = pkt.last_time1;  // Rewritten
```

Figure 7: SSA transformation. Note that all assignments are renamed as they can be preceded by reads.

```
1 pkt.id             = hash2(pkt.sport, pkt.dport) % NUM_FLOWLETS;
2 pkt.saved_hop      = saved_hop[pkt.id];
3 pkt.last_time      = last_time[pkt.id];
4 pkt.new_hop        = hash3(pkt.sport, pkt.dport, pkt.arrival) % NUM_HOPS;
5 pkt.tmp            = pkt.arrival - pkt.last_time;
6 pkt.tmp2           = pkt.tmp > THRESHOLD;
7 pkt.next_hop       = pkt.tmp2 ? pkt.new_hop : pkt.saved_hop;
8 saved_hop[pkt.id] = pkt.tmp2 ? pkt.new_hop : pkt.saved_hop;
9 last_time[pkt.id] = pkt.arrival;
```

Figure 8: Flowlet switching in three-address code. Lines 1 and 4 are flipped relative to Figure 2 because `pkt.id` is an array index expression and is moved into the read flank.

them into the ternary conditional operator, starting from the innermost if statements and recursing outwards (Figure 5). This procedure is known as if-conversion [36]. Unlike traditional languages, performing if-conversion in Domino is easy as there is no unstructured control flow. If conversion turns the body of packet transactions into straight-line code, where control passes sequentially without branching. Straight-line code simplifies the rest of the compiler, like computing the static single-assignment form(§4.3).

4.2 Rewriting state variable operations

We next identify both array and scalar state variables used in a packet transaction, such as `last_time` and `saved_hop` in Figure 2. State variables are easy to identify syntactically, since all variables are either packet fields or state variables. For each state variable, we create a *read flank* to read the state variable into a temporary packet field. For an array, we also move the index expression into the read flank, exploiting the fact that only one array index is accessed by each packet in valid Domino programs. Then, throughout the packet transaction, we replace the state variable with the packet temporary, and create a *write flank* to write the packet temporary back into the state variable. Figure 6 illustrates this transformation. After this pass, the code resembles code for a load-store architecture [21]: all state variables must be loaded into packet variables before arithmetic can be performed on them. Restricting operations on state variables simplifies subsequent code partitioning (§4.5).

4.3 Converting to single-assignment form

We next convert the code to static single-assignment form (SSA), as shown in Figure 7). In SSA form, every variable is assigned exactly once. To compute the SSA, we replace every definition of a packet variable with a new packet variable and propagate this new variable until the next definition of the same variable. State

variables are already in SSA: after their flanks have been added, every state variable is written exactly once in the write flank. While general algorithms for computing the SSA are fairly involved [47], Domino's SSA computation is simpler because it runs after if conversion and hence operates on straight-line code. Because every variable is assigned exactly once, SSA removes Write-After-Read and Write-After-Write dependencies. Only Read-After-Write dependencies remain, simplifying dependency analysis during code partitioning (§4.5). We execute copy propagation [9] after SSA to reduce the number of temporary packet variables.

4.4 Flattening to three-address code

The input is next converted to three-address code [31], where all instructions are either reads / writes into state variables or operations on packet variables of the form `pkt.f1 = pkt.f2 op pkt.f3;` where op can be a conditional,[3] arithmetic, logical, or relational operator. We also allow either one of `pkt.f2` or `pkt.f3` to be an intrinsic function call. All expressions that are not already in three-address code are flattened by introducing temporaries as illustrated in Figure 8. While flattening expressions may result in redundant temporaries that compute the same subexpression, we remove such temporaries using common subexpression elimination [7].

4.5 Code partitioning to codelets

At this point, the code is still sequential. Code partitioning turns sequential code into a pipeline of *codelets*, where each codelet is a small sequential block of three-address code statements. Each codelet is then mapped to an atom provide by a particular PISA machine (§4.6), or an error is returned if it cannot be mapped.

To partition code into codelets, we carry out the following steps:

1. Create a node for each statement (Figure 8) in the packet transaction after expression flattening.
2. Create a bidirectional edge between nodes N1 and N2, where N1 is a read from a state scalar / state array and N2 is a write into the same variable. This step captures the constraint that state is internal to an atom in PISA.
3. Create an edge (N1, N2) for every pair of nodes N1, N2 where N2 reads a variable written by N1. We only check read-after-write dependencies because control dependencies turn into data dependencies after if conversion, and write-after-read and write-after-write dependencies don't exist in SSA.
4. Generate strongly connected components (SCCs) of the resulting graph (Figure 9) and condense the SCCs to create a directed acyclic graph (DAG) (Figure 10). This step captures the constraint that all operations on a state variable must reside within the same atom because state is internal to an atom.
5. Schedule the resulting DAG using critical path scheduling [64] by creating a new pipeline stage every time one operation needs to follow another according to the precedence relationship established by the DAG (Figure 10).

The resulting codelet pipeline[4] shown in Figure 3 implements the flowlet packet transaction. Further, the codelets have a stylized form. Codelets that don't manipulate state contain exactly one three-address code statement after expression flattening. Codelets that manipulate state contain at least two statements: a read from a state variable and a write to a state variable, with optionally one or more updates to the state variable through packet temporaries in between.

4.6 Mapping codelets to atoms

Next, we determine how codelets map one-to-one to atoms provided by the PISA machine. We consider codelets that do and don't manipulate state separately.

Stateless codelets As mentioned, stateless codelets have only one statement in three-address code form (e.g., any of the unshaded boxes in Figure 3). We assume that all PISA machines support stateless atoms that correspond to a single statement in three-address code. P4's primitives [26] and RMT's VLIW action set [39] both resemble this form. Under this assumption, each such codelet is mapped to an atom provided by the PISA machine. If the PISA machine supports stateless atoms beyond a single three-address code statement, this is still correct, although suboptimal. For instance, if the PISA machine supports a multiply-and-accumulate atom [23], our approach generates two atoms (one each for the multiply and accumulate), although one suffices.

Stateful codelets Stateful codelets have multi-line bodies that need to execute atomically. For instance, updating the state variable `saved_hop` in Figure 3 requires a read followed by a conditional write. It is not apparent whether such codelets can be mapped to an available atom. We develop a new technique to determine the implementability of such stateful codelets, on a PISA machine that provides a specific stateful atom template.

First, each atom template is parameterized, where the parameters determine the actual functionality provided by the atom. For instance, Figure 12 shows a hard-

[3] Ternary/Conditional operators take in 4 addresses instead of 3.

[4] We refer to this both as a codelet and an atom pipeline because codelets map one-to-one atoms (§4.6).

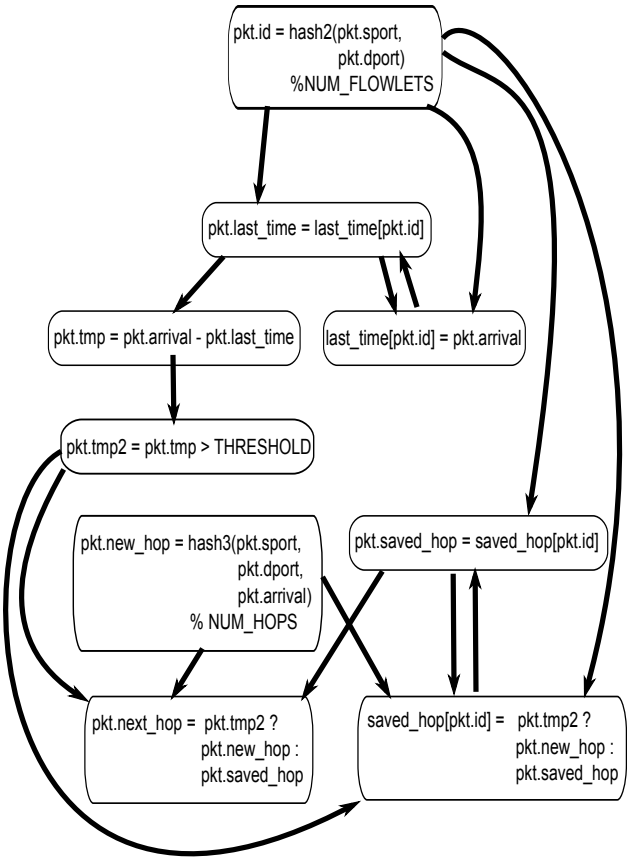


Figure 9: Dependency before condensing SCCs

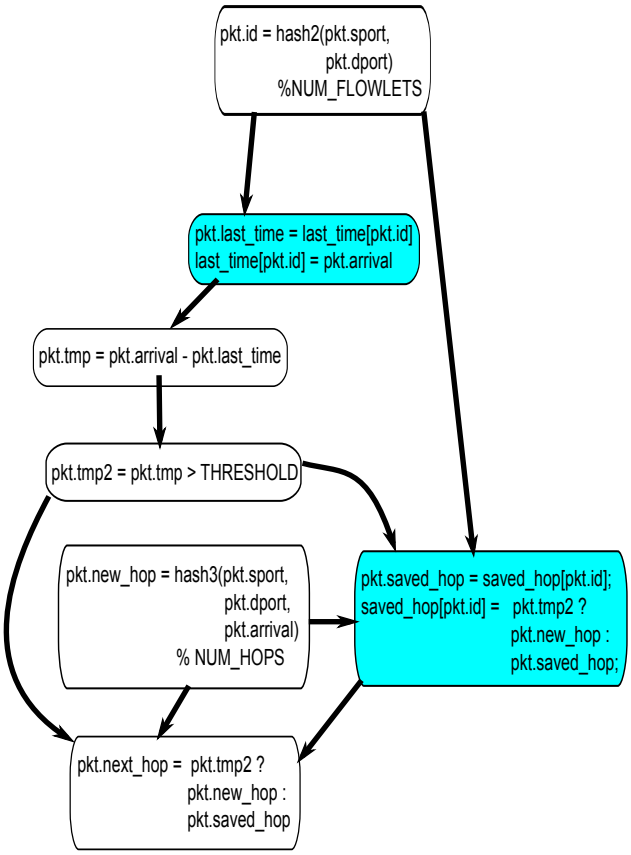


Figure 10: Dependency graph after condensing SCCs

ware circuit that is capable of performing stateful addition or subtraction, depending on the value of the constant and which output is selected from the multiplexer. Its atom template is shown in Figure 13, where `choice` and `constant` represent the tunable parameters. Each codelet can be viewed as a functional specification of the atom. With that in mind, the mapping problem is equivalent to searching for the value of the parameters to configure the atom such that it implements the provided specification.

While many algorithms can be used to perform the search, in the Domino compiler we use the SKETCH program synthesizer [86] for this purpose, as the atom templates can be easily expressed using SKETCH, while SKETCH also provides efficient search algorithms and has been used for similar purposes across different domains [85, 42, 43, 76].

As an illustration, assume we want to map the codelet `x=x+1` to the atom template shown in Figure 13. The Domino compiler feeds in the codelet and the atom template into SKETCH (Figure 11), and SKETCH will search for possible values of the parameters such that the resulting atom performs the same functionality as the codelet, for all possible input values of `x`. In this case this is done by setting `choice=0` and `constant=1`. In contrast, if the codelet `x=x*x` was supplied as the specification, SKETCH will return an error as no such parameter value exists. To minimize search time, the range of possible inputs and parameter values need to be specified in the template (e.g., all 8 bit integers), and our experiments show that the search finishes quickly, taking 10 secs at most.

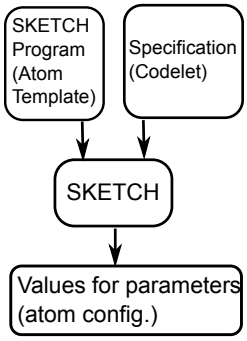


Figure 11: Overview of SKETCH and its application to atom configuration

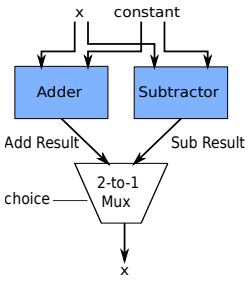


Figure 12: Circuit for an atom that can add or subtract a constant from a state variable.

```
bit choice = ??;
int constant = ??;
if (choice) {
  x = x + constant;
} else {
  x = x - constant;
}
```

Figure 13: Circuit representation as an atom template.

4.7 Verifying compilations

Our testing infrastructure verifies that the compilation is correct. By this, we mean that the externally visible behavior of the packet transaction (Figure 2) is indistinguishable from its pipelined implementation (Figure 3). We verify correctness by feeding in the same set of test packets to both the packet transaction and its implementation and comparing the outputs from both on the set of externally visible fields.

To create test packets, we scan the packet transaction and generate the set of all fields read from or written to by the transaction. We initialize each field by sampling independently and uniformly from the space of all 32-bit signed integers. To compare outputs from the packet transaction and its implementation, we track renames that occur after SSA and compare each output field in the transactional form with its last rename in the implementation. This lets us quickly "spot check" our compilations and helped us discover a few compiler bugs during development.

4.8 Targeting real switches

Domino doesn't yet generate code for actual hardware. We considered compiling Domino to P4, which would then allow Domino to target programmable switches by leveraging ongoing work [24, 25, 62] in P4 compilation. However, P4 currently doesn't support sequential execution within a pipeline stage or the conditional operator as a primitive action. Both are required to correctly execute the codelets/atoms produced by Domino (Figure 3). We have submitted a proposal [27] for both to the P4 language consortium. The proposal has been tentatively accepted and is likely to be included in P4 v1.1.

Atom	Description
Write	Write packet field/constant into single state variable.
ReadAddWrite (RAW)	Add packet field/constant to state variable (OR) Write packet field/constant into state variable.
Predicated ReadAddWrite (RAW)	Execute RAW on state variable only if a predicate is true, else leave unchanged.
IfElse ReadAddWrite (IfElseRAW)	Execute two separate RAWs: one each for when a predicate is true or false.
Subtract (Sub)	Same as IfElseRAW, but also allow subtracting a packet field/constant.
Nested Ifs (Nested)	Same as Sub, but with an additional level of nesting that provides 4-way predication.
Paired updates (Pairs)	Same as Nested, but allow updates to a pair of state variables, where predicates can use both state variables.

Table 2: Atoms used in evaluation. Appendix A provides the SKETCH code and circuit diagrams for these atoms.

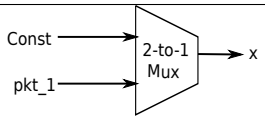
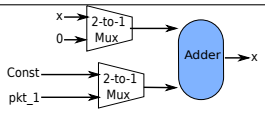
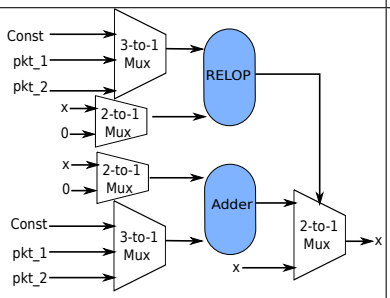
Atom	Circuit	Element depth
Write		1
ReadAddWrite (RAW)		2
Predicated ReadAddWrite (PRAW)		3

Table 3: Element depth and propagation delay increases with complexity of atoms.

5 Evaluation

To evaluate Domino, we expressed several data-plane algorithms (Table 4) in Domino to determine if they can be implemented on different PISA machines that provide different types of stateful atoms (Table 2). Expressing most of these algorithms in Domino involved little effort beyond simply translating their imperative code/pseudocode to Domino—a sign that Domino's abstractions are convenient to use.

5.1 Experimental procedure

As mentioned in §4.6, we focus only on stateful atoms and assume all stateless codelets are supported by a PISA machine because they are single three-address

Algorithm	Stateful computation	Least expressive atom	Pipeline depth, width	Ingress or Egress Pipeline?
Bloom filter [40] (3 hash functions)	Set membership bit on every packet.	Write	4, 3	Either
Heavy Hitters [93] (3 hash functions)	Increment Count-Min Sketch [46] on every packet.	RAW	10, 9	Either
Flowlets [82]	Update saved next hop if flowlet threshold is exceeded.	PRAW	6, 2	Ingress
RCP [89]	Accumulate RTT sum if RTT is under maximum allowable RTT.	PRAW	3, 3	Egress
Sampled NetFlow [30]	Sample a packet if packet count reaches N; Reset count to 0 when it reaches N.	IfElseRAW	4, 2	Either
HULL [35]	Update counter for virtual queue.	Sub	7, 1	Egress
Adaptive Virtual Queue [66]	Update virtual queue size and virtual capacity	Nested	7, 3	Ingress
CONGA [33]	Update best path's utilization/id if we see a better path. Update best path utilization alone if it changes.	Pairs	4, 2	Ingress
trTCM [29]	Update token counts for each token bucket	Doesn't map	7, 3	Either
CoDel [71]	Update: Whether we are marking or not. Time for next mark. Number of marks so far. Time at which min. queuing delay will exceed target.	Doesn't map	15, 3	Egress

Table 4: Data-plane algorithms

code statements. For simplicity, the stateful atoms we consider only permit state updates and forbid packet field updates intermixed with state updates. Assuming the ability to read a state variable[5], such field updates can be treated as stateless operations in subsequent pipeline stages.

We also assume every PISA machine provides only one stateful atom template, though we don't restrict the number of instances of this template. This is because ASIC engineers prefer to design, implement, verify, and physically layout one circuit, thereby amortizing design and layout effort over multiple instances of the same circuit. Table 2 gradually increases the capability of this single atom template. We designed the atoms in Table 2, and hence the PISA machines providing them, to form a containment hierarchy: each atom can express all data-plane algorithms that its predecessor can.

We now consider every atom/PISA machine from Table 2, and every data-plane algorithm from Table 4 to determine if the algorithm is *implementable* on a particular PISA machine. We say an algorithm is implementable on a PISA machine, if every stateful codelet within the data-plane algorithm can be mapped (§4.6) to the single stateful atom provided by the PISA machine. Because atoms are arranged in a containment hierarchy, we list the *least expressive* atom that can be used to implement a data-plane algorithm in Table 4.

[5]The inability to read a state variable renders it powerless!

5.2 Interpreting the results

Table 4 tells a network programmer the "minimal atom" required to run a data-plane algorithm at line rate. For an ASIC engineer, the same table describes the algorithms that are implementable on a PISA machine with a specific stateful atom. For instance, a PISA machine with the Pairs atom can implement the first eight algorithms, while a machine with a simpler RAW atom can implement only the first two.

We also discuss broader lessons for designing programmable switching chips. First, atoms supporting stateful operations on a single state variable are sufficient for several data-plane algorithms (Bloom Filters through Adaptive Virtual Queue in Table 4). However, there are algorithms that need the ability to update a pair of state variables atomically. One example is CONGA, whose code we reproduce below:

```
if (p.util < best_path_util[p.src]) {
  best_path_util[p.src] = p.util;
  best_path[p.src] = p.path_id;
} else if (p.path_id == best_path[p.src]) {
  best_path_util[p.src] = p.util;
}
```

Here, `best_path` (the path id of the best path for

a particular destination) is updated conditioned on `best_path_util` (the utilization of the best path to that destination)[6] and vice versa. There is no way to separate the two state variables into separate stages and still guarantee correctness.

The Pairs atom, where the update to a state variable is conditioned on a predicate of a pair of state variables, allows us to implement CONGA at line rate. However, it is *still* insufficient for algorithms such as CoDel [71] and the two-rate three-color meter (trTCM) [29]. On a positive note, however, we observed that the codelets in both trTCM and CoDel are still restricted to a pair of state variables. We haven't yet encountered a triplet of state variables all falling in the same strongly connected component/codelet, requiring a three-way state update.

5.3 Performance vs. programmability

While powerful atoms like Pairs can implement more data-plane algorithms, they come at a cost. A more expressive atom needs more gates in hardware and incurs longer propagation delays. As an illustration, consider the circuits (Table 3) for the first three atoms from Table 2. We use the number of elements (muxes, adders, subtractors, and relational operators) on the longest path between input and output, the element depth, as a crude proxy for propagation delay, and observe that it increases with atom complexity. At some point, the propagation delay may prevent the circuit from meeting a particular line rate. We plan to synthesize these atoms to circuits in a standard cell library to study this more rigorously.

The larger takeaway is that we can begin to quantify the programmability-performance tradeoff that switch designers intuitively understand. From a corpus of data-plane algorithms, we can rigorously—modulo compiler inefficiencies—determine which of them can run at a given line rate, given the stateful atom supported at that line rate. A lower line rate would permit larger propagation delays, more expressive atoms, and more data-plane algorithms. The end result is a programmability-performance curve where the number of implementable algorithms (programmability) increases as the line rate decreases (performance).

5.4 Compilation times

The data-plane algorithms we consider are all under 100 lines of code. Time spent in the front-end is negligible; instead, compilation time is dominated by SKETCH. To speed up the search, we limit SKETCH to search for constants (e.g., for addition) of size up to 5 bits, given that the constants we observe within stateful codelets in our algorithms are small.

Quantitatively, our longest compilation time is 10 seconds when CoDel doesn't map to a PISA machine with the Pairs atom. This time will increase if we increase the bit width of constants that SKETCH has to search; however, because data-plane algorithms are small, we don't expect compilation times to be a concern.

[6] `p.src` is the address of the host originating this message, and hence the destination for the host receiving it and executing CONGA.

6 Related work

Data-Plane Algorithms Several data-plane algorithms are now commonplace, e.g., lookup algorithms based on longest-prefix, exact, or ternary matches. Domino focuses on data-plane algorithms that aren't widely available because of the engineering effort required for hardware implementations. Further, the growing list of new algorithms [61, 92, 94, 71, 33] makes it hard to commit to a hardware implementation. Domino allows network programmers to modify these algorithms more rapidly.

Abstract machines for line-rate switches Relative to P4's abstract switch model [38], PISA contributes the notions of atoms, sequential execution within atoms, and state encapsulated by atoms. Closest to PISA is NetASM [80], an intermediate representation and abstract machine for programmable data planes. PISA differs from NetASM by explicitly targeting line-rate switches in two ways. First, all state in PISA is internal to an atom (and hence a stage), while NetASM's ATM construct allows access to shared state from multiple pipeline stages. Second, PISA uses atom templates to limit the amount of useful work that can be performed in an atom.

Programmable Data Planes Software-based data planes such as Click [65], RouteBricks [49], and Fastpass [75] are flexible but lack the performance required for deployments. Network Processors [18, 19] (NPUs) were an attempt to bridge the gap. While NPUs are faster than software routers, they remain an order of magnitude slower than merchant silicon chips [39].

An alternative is to use FPGAs to improve performance relative to software routers and NPUs; examples include NetFPGA [68], Switchblade [37], Chimpp [78], and [84]. These designs are slower than switching ASICs, and are rarely used in production network equipment. The Arista 7124 FX [1] is a commercial switch with an on-board FPGA, but its capacity is limited to 160 Gbits/sec when using the on-board FPGA— 10x less than the multi-terabit capacities of programmable switch chips [32]. In addition, FPGAs are hard to program. Relative to FPGAs, Domino seeks to provide both the line-rate performance of switching ASICs and a familiar programming model.

Packet-processing languages Many programming languages target the network control plane. Examples

include Frenetic [58], Pyretic [70], and Maple [91]. [57] is a survey of such approaches. In contrast, Domino focuses on the data plane. Several DSLs explicitly target data-plane packet processing. Click [65] uses C++ for packet processing on software routers. Imperative languages such as NOVA [59], packetC [50], Intel's auto-partitioning C compiler [48], PacLang [51, 52], and Microengine C [15, 17] target network processors [18, 19].

Domino's C-like syntax and sequential semantics are inspired by these languages. However Domino is constrained relative to its predecessors: for instance, it forbids loops and includes no synchronization constructs because all state is internal to an atom. These constraints were chosen to target line-rate switching chips with shared-nothing memory architectures. As a result, Domino's compiler presents a different programming model: all Domino programs that compile run at line-rate, while those that can't run at line rate are rejected outright. Unlike an NPU or software router, there is no slippery slope of degrading performance with increasing program complexity.

P4 [38] is an emerging packet-processing language that explicitly targets line-rate programmable switching chips. However, while P4 is a natural model for many header-manipulation tasks such as switching, ACLs, routing, and tunnelling [83], it is ill-suited to programming data-plane algorithms that rely on intricate state manipulation.

Compiler Techniques Domino's compiler uses three-address codes [31], static-single assignment form [47], and if conversion [36] from the compiler literature. However, Domino's constrained design allows us to simplify these techniques relative to their mainstream uses. The use of strongly connected components is based on similar uses in software pipelining [67] for VLIW architectures. However, dependence analysis for loop-carried dependencies in software pipelining (equivalent to state in Domino) is more involved than the simple syntactic checks used by Domino. The use of synthesis to map codelets to atoms is based on Chlorophyll's [76] use of program synthesis for compilation to unconventional targets.

Hardware compilation Prior work [73, 41, 72] has focused on deriving digital circuits from imperative programs. These approaches simplify hardware development, but the performance of each program depends on its complexity. Domino has an all-or-nothing guarantee: all code that compiles can run at line rate or is rejected by the compiler. Mapping from codelets to atoms is similar to technology mapping [69, 45, 44], where a target hardware circuit (represented as a graph) is implemented by tiling the circuit graph with primitive circuits from a technology library. Domino's problem is simpler. We only need to verify if a codelet maps to an atom—not implement a codelet using multiple atoms. Recent work [62] focuses on compiling P4 programs to hardware targets such as the RMT and FlexPipe architectures. However, their work focuses on compiling stateless data-plane tasks such as forwarding and routing, while Domino focuses on stateful data-plane algorithms.

7 Conclusion

This paper presented Domino, a C-like imperative language that allows programmers to write packet-processing code using packet transactions: sequential code blocks that are atomic and isolated from other such code blocks. The Domino compiler compiles packet transactions to PISA, a family of abstract machines based on programmable line-rate switch architectures [16, 32, 39]. Our results suggest that it is possible to have both a familiar programming model and line-rate performance—i.e. if the algorithm can indeed run at line rate.

These results suggest that, unlike a claim posited in recent work [38], it is possible to express data-plane algorithms in a C-like language and achieve line-rate performance. Domino shows that it is possible with careful design to find a sweet spot between expressiveness and performance, somewhere between P4 and Click. Further, as §4 shows, careful language design simplifies the compiler.

Our conclusion is that packet transactions provide a familiar programming abstraction, can achieve line-rate performance, and result in a simple compiler. Packet-processing languages are still in their infancy; we hope these results prompt further work on programming abstractions for packet-processing hardware.

References

[1] 7124fx application switch. https://www.arista.com/assets/data/pdf/7124FX/7124FX_Data_Sheet.pdf.

[2] Appendix: Codel pseudocode. http://queue.acm.org/appendices/codel.html.

[3] Arista - arista 7050 series. https://www.arista.com/en/products/7050-series.

[4] Cavium and XPliant introduce a fully programmable switch silicon family scaling to 3.2 terabits per second. http://www.cavium.com/newsevents-Cavium-and-XPliant-Introduce-a-Fully-Programmable-Switch-Silicon-Family.html.

[5] Cavium XPliant switches and Microsoft azure networking achieve SAI routing interoperability. http://www.cavium.com/newsevents-Cavium-XPliant-Switches-and-Microsoft-Azure-Networking-Achieve-SAI-Routing-Interoperability.html.

[6] Cisco nexus family. http://www.cisco.com/c/en/us/products/switches/cisco_nexus_family.html.

[7] Common subexpression elimination. https://en.wikipedia.org/wiki/Common_subexpression_elimination.

[8] Components of Linux Traffic Control. http://tldp.org/HOWTO/Traffic-Control-HOWTO/components.html.

[9] Copy propagation. https://en.wikipedia.org/wiki/Copy_propagation.

[10] Dell force10. http://www.force10networks.com/.

[11] DPDK: Data plane development kit. http://dpdk.org/.

[12] High Capacity StrataXGS®Trident II Ethernet Switch Series. http://www.broadcom.com/products/Switching/Data-Center/BCM56850-Series.

[13] High-density 25/100 gigabit ethernet StrataXGS tomahawk ethernet switch series. http://www.broadcom.com/products/Switching/Data-Center/BCM56960-Series.

[14] Intel 64 and ia-32 architectures software developer's manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf.

[15] Intel enhances network processor family with new software tools and expanded performance. http://www.intel.com/pressroom/archive/releases/2001/20010220net.htm.

[16] Intel FlexPipe. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.

[17] Intel internet exchange architecture. http://www.intel.com/design/network/papers/intelixa.pdf.

[18] Intel IXP2800 network processor. http://www.ic72.com/pdf_file/i/587106.pdf.

[19] IXP4XX Product Line of Network Processors. http://www.intel.com/content/www/us/en/intelligent-systems/previous-generation/intel-ixp4xx-intel-network-processor-product-line.html.

[20] Libtooling. http://clang.llvm.org/docs/LibTooling.html.

[21] Load/store architecture. https://en.wikipedia.org/wiki/Load/store_architecture.

[22] Mellanox Products: SwitchX-2 Ethernet Optimized for SDN. http://www.mellanox.com/page/products_dyn?product_family=146&mtag=switchx_2_en.

[23] Multiply-accumulate operation. https://en.wikipedia.org/wiki/Multiply-accumulate_operation.

[24] Netronome showcases next-gen intelligent server adapter delivering 20x ovs performance at open networking summit 2015. https://netronome.com/netronome-showcases-next-gen-intelligent-server-adapter\-delivering-20x-ovs-performance-at-open-networking\-summit-2015/.

[25] P4 for an FPGA target. http://schd.ws/hosted_files/p4workshop2015/33/GordonB-P4-Workshop-June-04-2015.pdf.

[26] P4 Specification. http://p4.org/spec/p4-latest.pdf.

[27] P4's action-execution semantics and conditional operators. https://github.com/anirudhSK/p4-semantics/raw/master/p4-semantics.pdf.

[28] Protocol-independent switch architecture. http://schd.ws/hosted_files/p4workshop2015/c9/NickM-P4-Workshop-June-04-2015.pdf.

[29] RFC 2698 - a two rate three color meter. https://tools.ietf.org/html/rfc2698.

[30] Sampled netflow. http://www.cisco.com/c/en/us/td/docs/ios/12_0s/feature/guide/12s_sanf.html.

[31] Three-address code. https://en.wikipedia.org/wiki/Three-address_code.

[32] XPliant™Ethernet Switch Product Family. http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html.

[33] ALIZADEH, M., EDSALL, T., DHARMAPURIKAR, S., VAIDYANATHAN, R., CHU, K., FINGERHUT, A., LAM, V. T., MATUS, F., PAN, R., YADAV, N., AND VARGHESE, G. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *SIGCOMM* (2014).

[34] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data Center TCP (DCTCP). In *SIGCOMM* (2010).

[35] ALIZADEH, M., KABBANI, A., EDSALL, T., PRABHAKAR, B., VAHDAT, A., AND YASUDA, M. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX, pp. 253–266.

[36] ALLEN, J. R., KENNEDY, K., PORTERFIELD, C., AND WARREN, J. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1983), POPL '83, ACM, pp. 177–189.

[37] ANWER, M. B., MOTIWALA, M., TARIQ, M. B., AND FEAMSTER, N. Switchblade: A platform for rapid deployment of network protocols on programmable hardware. In *SIGCOMM* (2011).

[38] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev. 44*, 3 (July 2014), 87–95.

[39] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *SIGCOMM* (2013).

[40] BRODER, A., MITZENMACHER, M., AND MITZENMACHER, A. B. I. M. Network applications of bloom filters: A survey. In *Internet Mathematics* (2002), pp. 636–646.

[41] BUDIU, M., AND GOLDSTEIN, S. C. Compiling application-specific hardware. In *Proceedings of the 12th International Conference on Field Programmable Logic and Applications* (Montpellier (La Grande-Motte), France, September 2002), pp. 853–863.

[42] CHEUNG, A., SOLAR-LEZAMA, A., AND MADDEN, S. Using program synthesis for social recommendations. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management* (New York, NY, USA, 2012), CIKM '12, ACM, pp. 1732–1736.

[43] CHEUNG, A., SOLAR-LEZAMA, A., AND MADDEN, S. Optimizing database-backed applications with query synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2013), PLDI '13, ACM, pp. 3–14.

[44] CLARKE, E. M., MCMILLAN, K. L., ZHAO, X., FUJITA, M., AND YANG, J. Spectral transforms for large boolean functions with applications to technology mapping. In *Design Automation, 1993. 30th Conference on* (1993), IEEE, pp. 54–60.

[45] CONG, J., AND DING, Y. Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based fpga designs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 13*, 1 (1994), 1–12.

[46] CORMODE, G., AND MUTHUKRISHNAN, S. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms 55*, 1 (Apr. 2005), 58–75.

[47] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Language Systems 13*, 4 (1991), 451–490.

[48] DAI, J., HUANG, B., LI, L., AND HARRISON, L. Automatically partitioning packet processing applications for pipelined architectures. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 237–248.

[49] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B.-G., FALL, K., IANNACCONE, G., KNIES, A., MANESH, M., AND RATNASAMY, S. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 15–28.

[50] DUNCAN, R., AND JUNGCK, P. packetC Language for High Performance Packet Processing. In *11th IEEE International Conference on High Performance Computing and Communications* (2009).

[51] ENNALS, R., SHARP, R., AND MYCROFT, A. Linear types for packet processing. In *Programming Languages and Systems*, D. Schmidt, Ed., vol. 2986 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004, pp. 204–218.

[52] ENNALS, R., SHARP, R., AND MYCROFT, A. Task partitioning for multi-core network processors. In *Compiler Construction*, R. Bodik, Ed., vol. 3443 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005, pp. 76–90.

[53] ESTAN, C., AND VARGHESE, G. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst. 21*, 3 (Aug. 2003), 270–313.

[54] ESTAN, C., VARGHESE, G., AND FISK, M. Bitmap algorithms for counting active flows on high-speed links. *IEEE/ACM Trans. Netw. 14*, 5 (Oct. 2006), 925–937.

[55] FENG, W.-C., SHIN, K. G., KANDLUR, D. D., AND SAHA, D. The blue active queue management algorithms. *IEEE/ACM Trans. Netw. 10*, 4 (Aug. 2002), 513–528.

[56] FLOYD, S., AND JACOBSON, V. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw. 1*, 4 (Aug. 1993), 397–413.

[57] FOSTER, N., GUHA, A., REITBLATT, M., STORY, A., FREEDMAN, M., KATTA, N., MONSANTO, C., REICH, J., REXFORD, J., SCHLESINGER, C., WALKER, D., AND HARRISON, R. Languages for software-defined networks. *Communications Magazine, IEEE 51*, 2 (February 2013), 128–134.

[58] FOSTER, N., HARRISON, R., FREEDMAN, M. J., MONSANTO, C., REXFORD, J., STORY, A., AND WALKER, D. Frenetic: A Network Programming Language. In *ICFP* (2011).

[59] GEORGE, L., AND BLUME, M. Taming the ixp network processor. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (New York, NY, USA, 2003), PLDI '03, ACM, pp. 26–37.

[60] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. Vl2: A scalable and flexible data center network. In *SIGCOMM* (2009).

[61] HONG, C.-Y., CAESAR, M., AND GODFREY, P. B. Finishing flows quickly with preemptive scheduling. In *SIGCOMM* (2012).

[62] JOSE, L., YAN, L., VARGHESE, G., AND MCKEOWN, N. Compiling Packet Programs to Reconfigurable Switches. In *NSDI* (2015).

[63] KATABI, D., HANDLEY, M., AND ROHRS, C. Congestion Control for High Bandwidth-Delay Product Networks. In *SIGCOMM* (2002).

[64] KELLEY JR, J. E., AND WALKER, M. R. Critical-path planning and scheduling. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference* (1959), ACM, pp. 160–173.

[65] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click Modular Router. *ACM Trans. Comput. Syst. 18*, 3 (Aug. 2000), 263–297.

[66] KUNNIYUR, S. S., AND SRIKANT, R. An adaptive virtual queue (avq) algorithm for active queue management. *IEEE/ACM Trans. Netw. 12*, 2 (Apr. 2004), 286–299.

[67] LAM, M. Software pipelining: An effective scheduling technique for vliw machines. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1988), PLDI '88, ACM, pp. 318–328.

[68] LOCKWOOD, J. W., MCKEOWN, N., WATSON, G., GIBB, G., HARTKE, P., NAOUS, J., RAGHURAMAN, R., AND LUO, J. NetFPGA–An Open Platform for Gigabit-Rate Network Switching and Routing. In *IEEE International Conf. on Microelectronic Systems Education* (2007).

[69] MICHELI, G. D. *Synthesis and Optimization of Digital Circuits*, 1st ed. McGraw-Hill Higher Education, 1994.

[70] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., AND WALKER, D. Composing Software-defined Networks. In *NSDI* (2013).

[71] NICHOLS, K., AND JACOBSON, V. Controlling Queue Delay. *ACM Queue 10*, 5 (May 2012).

[72] NIKHIL, R. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on* (June 2004), pp. 69–70.

[73] NURVITADHI, E., HOE, J., KAM, T., AND LU, S. Automatic pipelining from transactional datapath specifications. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 30*, 3 (March 2011), 441–454.

[74] PAN, R., NATARAJAN, P., PIGLIONE, C., PRABHU, M., SUBRAMANIAN, V., BAKER, F., AND VERSTEEG, B. Pie: A lightweight control scheme to address the bufferbloat problem. In *High Performance Switching and Routing (HPSR), 2013 IEEE 14th International Conference on* (July 2013), pp. 148–155.

[75] PERRY, J., OUSTERHOUT, A., BALAKRISHNAN, H., SHAH, D., AND FUGAL, H. Fastpass: A Centralized "Zero-queue" Datacenter Network. In *SIGCOMM* (2014).

[76] PHOTHILIMTHANA, P. M., JELVIS, T., SHAH, R., TOTLA, N., CHASINS, S., AND BODIK, R. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *PLDI* (2014), pp. 396–407.

[77] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 123–137.

[78] RUBOW, E., MCGEER, R., MOGUL, J., AND VAHDAT, A. Chimpp: A Click-based programming and simulation environment for reconfigurable networking hardware. In *ANCS* (2010).

[79] SHAH, N. Understanding network processors.

[80] SHAHBAZ, M., AND FEAMSTER, N. The case for an intermediate representation for programmable data planes. In *SOSR* (2015), pp. 3:1–3:6.

[81] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., KANAGALA, A., PROVOST, J., SIMMONS, J., TANDA, E., WANDERER, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. Jupiter rising: A decade of Clos topologies and centralized control in google's datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 183–197.

[82] SINHA, S., KANDULA, S., AND KATABI, D. Harnessing TCPs Burstiness using Flowlet Switching. In *3rd ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)* (San Diego, CA, November 2004).

[83] SIVARAMAN, A., KIM, C., KRISHNAMOORTHY, R., DIXIT, A., AND BUDIU, M. Dc.p4: Programming the forwarding plane of a data-center switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (New York, NY, USA, 2015), SOSR '15, ACM, pp. 2:1–2:8.

[84] SIVARAMAN, A., WINSTEIN, K., SUBRAMANIAN, S., AND BALAKRISHNAN, H. No silver bullet: Extending sdn to the data plane. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2013), HotNets-XII, ACM, pp. 19:1–19:7.

[85] SOLAR-LEZAMA, A., RABBAH, R., BODÍK, R., AND EBCIOĞLU, K. Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 281–294.

[86] SOLAR-LEZAMA, A., TANCAU, L., BODIK, R., SESHIA, S., AND SARASWAT, V. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2006), ASPLOS XII, ACM, pp. 404–415.

[87] STANLEY, S. Roving reporter: Reference platforms for sdn and nfv. `https://embedded.communities.intel.com/community/en/hardware/blog/2013/06/03/roving-reporter-reference-platforms-for-sdn-and-nfv`.

[88] STOICA, I., SHENKER, S., AND ZHANG, H. Core-stateless fair queueing: A scalable architecture to approximate fair bandwidth allocations in high-speed networks. *IEEE/ACM Trans. Netw. 11*, 1 (Feb. 2003), 33–46.

[89] TAI, C., ZHU, J., AND DUKKIPATI, N. Making Large Scale Deployment of RCP Practical for Real Networks. In *INFOCOM* (2008).

[90] TENNENHOUSE, D. L., AND WETHERALL, D. J. Towards an active network architecture. In *DARPA Active NEtworks Conference and Exposition, 2002. Proceedings* (2002), IEEE, pp. 2–15.

[91] VOELLMY, A., WANG, J., YANG, Y. R., FORD, B., AND HUDAK, P. Maple: Simplifying sdn programming using algorithmic policies. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 87–98.

[92] WILSON, C., BALLANI, H., KARAGIANNIS, T., AND ROWTRON, A. Better never than late: Meeting deadlines in datacenter networks. In *SIGCOMM* (2011).

[93] YU, M., JOSE, L., AND MIAO, R. Software defined traffic measurement with opensketch. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 29–42.

[94] ZATS, D., DAS, T., MOHAN, P., BORTHAKUR, D., AND KATZ, R. Detail: Reducing the flow completion time tail in datacenter networks. In *SIGCOMM* (2012).

A Atom templates and circuit diagrams for atoms

The appendix provides circuit diagrams (Figures 14– 20) and templates (Table 6) for the atoms in Table 2. Table 5 summarizes the notation we use in this section.

Construct	Description
	N-to-1 multiplexer with enable bit.
MuxN(a1, a2, ..., aN)	If enabled, return one of a1, a2, ... aN.
	If disabled, return 0.
Opt(a)	Return a or 0. (Equivalent to Mux1(a)).
rel_op(x, y)	Return one of $x < y$, $x > y$, $x! = y$, $x == y$.
Const()	Return an integer constant in the range [0, 31].[a]
x, y	State variables
pkt_1, pkt_2	Packet fields

Table 5: Notation used in atom templates

[a] We restrict constants to 5 bits because all constants within stateful codelets in our data-plane algorithms are under 32. Larger ranges increase synthesis time.

Atom	Atom template	Element depth
Write Figure 14	`x = Mux2(pkt_1, Const());`	1
ReadAddWrite (RAW) Figure 15	`x = Opt(x) + Mux2(pkt_1, Const());`	2

Predicated ReadAddWrite (PRAW)
Figure 16
	`if (rel_op(Opt(x), Mux3(pkt_1, pkt_2, Const()))) {`
	`  x = Opt(x) + Mux3(pkt_1, pkt_2, Const());`
	`}`
		3

If-Else ReadAddWrite (IfElseRAW)
Figure 17
	`if (rel_op(Opt(x), Mux3(pkt_1, pkt_2, Const()))) {`
	`  x = Opt(x) + Mux3(pkt_1, pkt_2, Const());`
	`} else {`
	`  x = Opt(x) + Mux3(pkt_1, pkt_2, Const());`
	`}`
		3

Subtract (Sub)
Figure 18
	`if (rel_op(Opt(x), Mux3(pkt_1, pkt_2, Const()))) {`
	`  x = Opt(x) + Mux3(pkt_1, pkt_2, Const()) - Mux3(pkt_1, pkt_2, Const());`
	`} else {`
	`  x = Opt(x) + Mux3(pkt_1, pkt_2, Const()) - Mux3(pkt_1, pkt_2, Const());`
	`}`
		4

Nested Ifs (Nested)
Figure 19
	`if (rel_op(Opt(x) + Mux2(pkt_1, pkt_2) - Mux2(pkt_1, pkt_2), Const())) {`
	` if (rel_op(Opt(x) + Mux2(pkt_1, pkt_2) - Mux2(pkt_1, pkt_2), Const())) {`
	`  x = Opt(x) + Mux3(pkt_1, pkt_2, Const()) - Mux3(pkt_1, pkt_2, Const());`
	` } else {`
	`  x = Opt(x) + Mux3(pkt_1, pkt_2, Const()) - Mux3(pkt_1, pkt_2, Const());`
	` }`
	`} else {`
	` if (rel_op(Opt(x) + Mux2(pkt_1, pkt_2) - Mux2(pkt_1, pkt_2), Const())) {`
	`  x = Opt(x) + Mux3(pkt_1, pkt_2, Const()) - Mux3(pkt_1, pkt_2, Const());`
	` } else {`
	`  x = Opt(x) + Mux3(pkt_1, pkt_2, Const()) - Mux3(pkt_1, pkt_2, Const());`
	` }`
	`}`
		6

Paired Updates (Pairs)
Figure 20
	`if (rel_op(Mux2(x, y) + Mux2(pkt_1, pkt_2) - Mux2(pkt_1, pkt_2), Const())) {`
	` if (rel_op(Mux2(x, y) + Mux2(pkt_1, pkt_2) - Mux2(pkt_1, pkt_2), Const())) {`
	`  x = Opt(x) + Mux3(pkt_1, pkt_2, Const()) - Mux3(pkt_1, pkt_2, Const());`
	`  y = Opt(y) + Mux3(pkt_1, pkt_2, Const()) - Mux3(pkt_1, pkt_2, Const());`
	` } else {`
	`  x = Opt(x) + Mux3(pkt_1, pkt_2, Const()) - Mux3(pkt_1, pkt_2, Const());`
	`  y = Opt(y) + Mux3(pkt_1, pkt_2, Const()) - Mux3(pkt_1, pkt_2, Const());`
	` }`
	`} else if (rel_op(Mux2(x, y) + Mux2(pkt_1, pkt_2) - Mux2(pkt_1, pkt_2), Const())) {`
	` if (rel_op(Mux2(x, y) + Mux2(pkt_1, pkt_2) - Mux2(pkt_1, pkt_2), Const())) {`
	`  x = Opt(x) + Mux3(pkt_1, pkt_2, Const()) - Mux3(pkt_1, pkt_2, Const());`
	`  y = Opt(y) + Mux3(pkt_1, pkt_2, Const()) - Mux3(pkt_1, pkt_2, Const());`
	` } else {`
	`  x = Opt(x) + Mux3(pkt_1, pkt_2, Const()) - Mux3(pkt_1, pkt_2, Const());`
	`  y = Opt(y) + Mux3(pkt_1, pkt_2, Const()) - Mux3(pkt_1, pkt_2, Const());`
	` }`
	`}`
		6

Table 6: SKETCH code for atoms described in Table 2

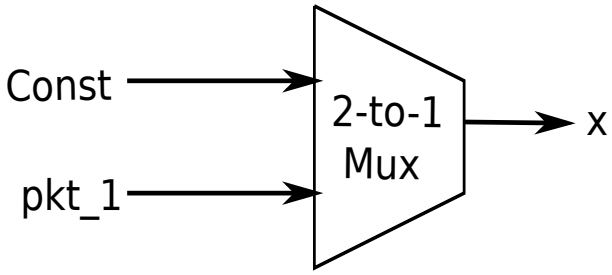


Figure 14: Circuit for Write atom with depth 1.

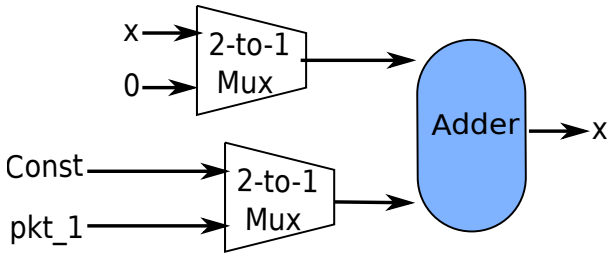


Figure 15: Circuit for RAW atom with depth 2.

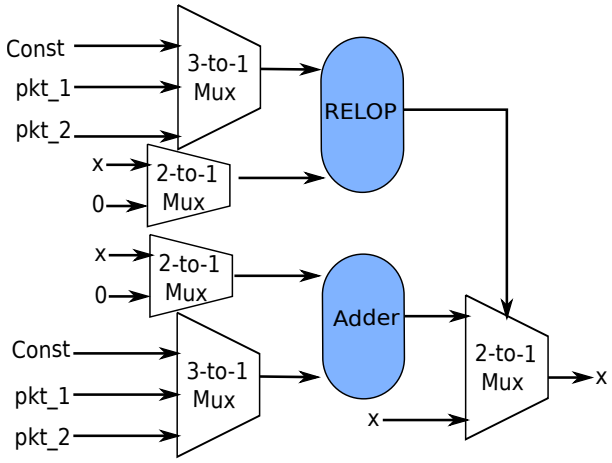


Figure 16: Circuit for PRAW atom with depth 3.

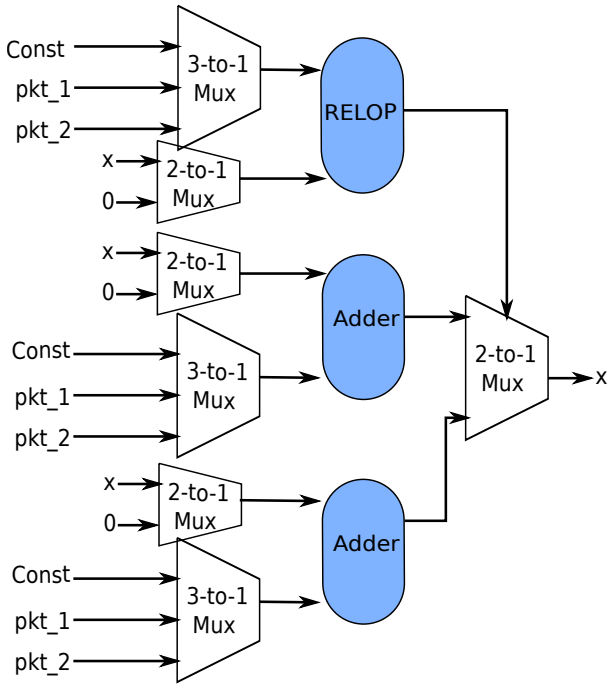


Figure 17: Circuit for IfElseRAW atom with depth 3.

Figure 18: Circuit for Sub atom with depth 4.

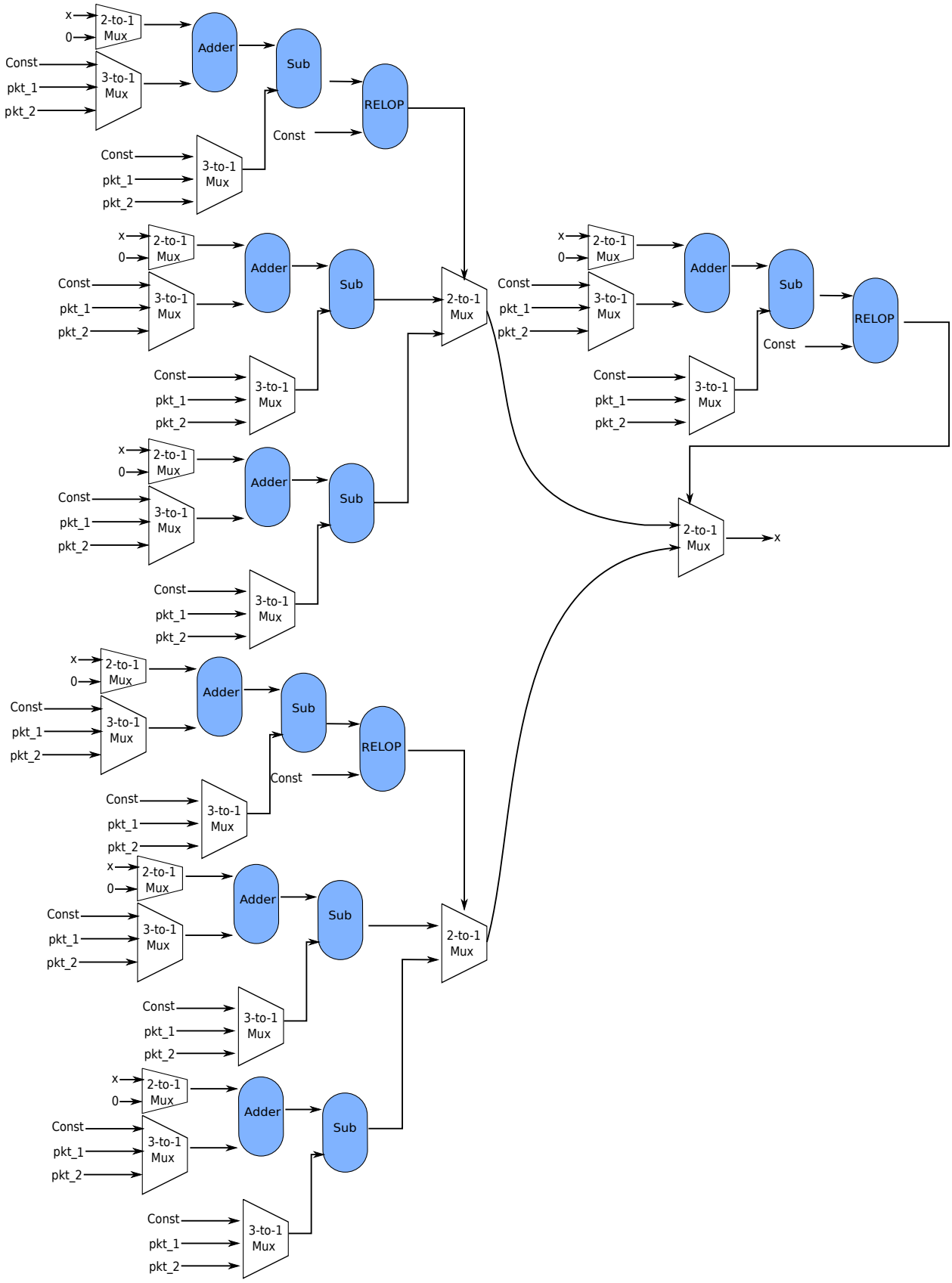


Figure 19: Circuit for Nested atom with depth 6.

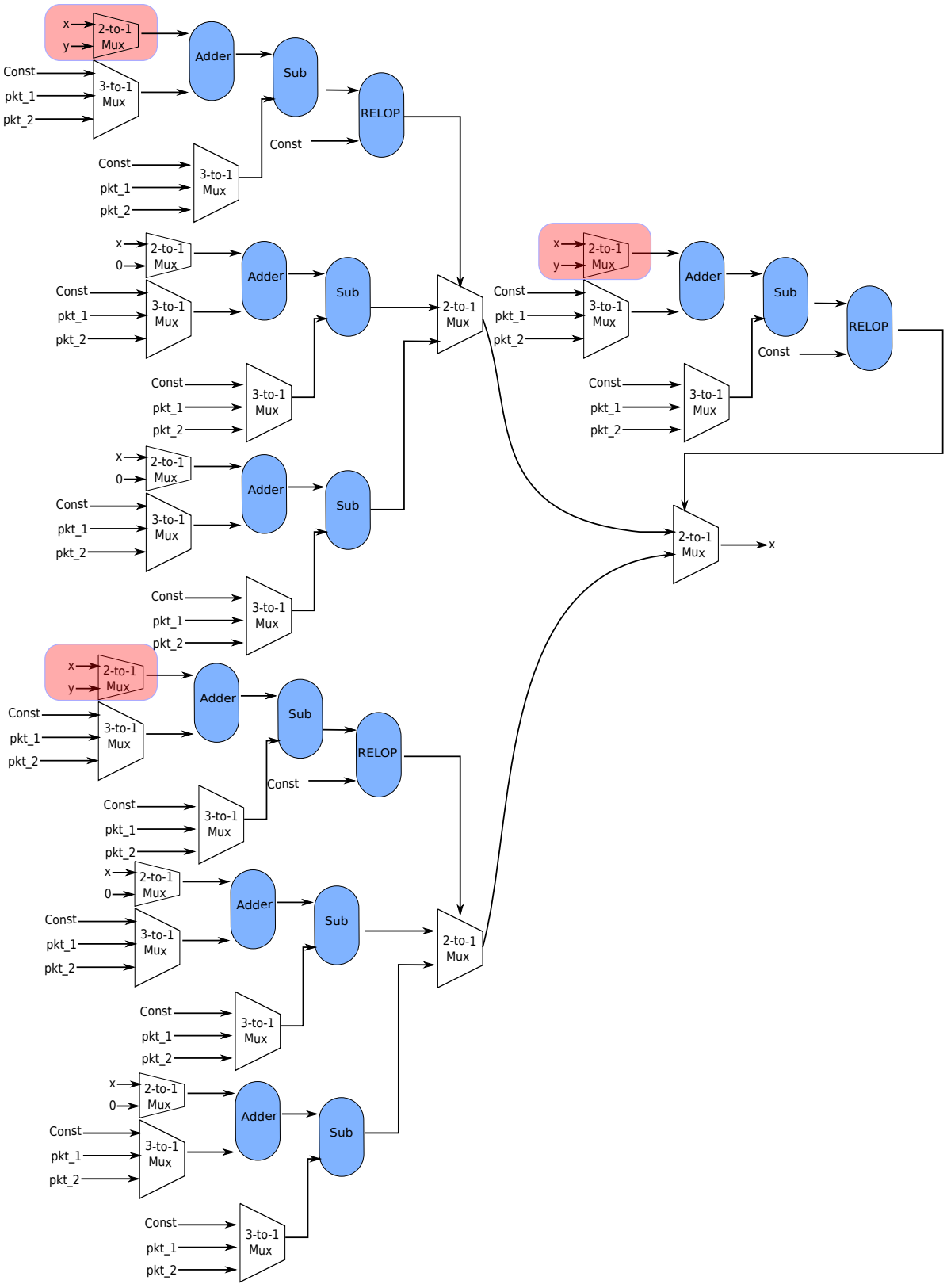


Figure 20: One-half of the circuit for the Pairs atom with depth 6. The other half is identical, except that it updates y instead of x, and isn't shown for simplicity. The shaded regions denote the differences in the Pairs atom relative to the Nested atom: the predicates can depend on both x and y in the Pairs atom.