Packet Transactions: High-Level Programming for Line-Rate Switches

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

ABSTRACT

Many algorithms for congestion control, scheduling, network measurement, active queue management, and traffic engineering require custom processing of packets in the data plane of a network switch. To run at line rate, these data-plane algorithms must be implemented in hardware. With today's switch hardware, algorithms cannot be changed, nor new algorithms installed, after a switch has been built.

This paper shows how to program data-plane algorithms in a high-level language and compile those programs into low-level microcode that can run on emerging programmable line-rate switching chips. The key challenge is that many data-plane algorithms create and modify algorithmic state. To achieve line-rate programmability for stateful algorithms, we introduce the notion of a *packet transaction*: a sequential packet-processing code block that is atomic and isolated from other such code blocks.

We have developed this idea in Domino, a C-like imperative language to express data-plane algorithms. We show with many examples that Domino provides a convenient way to express sophisticated data-plane algorithms, and show that these algorithms can be run at line rate with modest estimated chip-area overhead.

CCS Concepts

•**Networks** → **Programmable networks;**

Keywords

Programmable switches; stateful data-plane algorithms

1. INTRODUCTION

Network switches in modern datacenters, enterprises, and service-provider networks perform many tasks in addition to standard packet forwarding. The set of requirements for switches has only increased with time as network operators seek greater control over performance. Performance may be improved using both data-plane and control-plane mechanisms. This paper focuses on data-plane algorithms. These algorithms process and transform packets, reading and writing state in the switch. Examples include active queue management [38, 47, 51], scheduling [58], congestion control with switch feedback [45, 60], network measurement [63, 37], and data-plane traffic engineering [21].

An important requirement for data-plane algorithms is the ability to process packets at the switch's line rate: 10–100 Gbit/s on 10–100 ports. Therefore, these algorithms are typically implemented using dedicated hardware. Hardware designs are rigid and not reconfigurable in the field. Thus, implementing and deploying a new algorithm, or even modifying a deployed one, requires an investment in new hardware—a time-consuming and expensive proposition.

This rigidity affects many stakeholders adversely: vendors [2, 4, 6] building network switches with merchant-silicon chips [10, 11, 16], network operators deploying these switches [41, 54, 56], and researchers developing new data-plane algorithms [21, 37, 45, 60].

To run new data-plane algorithms after a switch has been built, researchers and companies have attempted to build programmable switches for many years, starting from efforts on active networks [61] to network processors [14] to software routers [8, 46]. All these efforts sacrificed performance for 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.

Programmable switching chips [3, 13, 19] competitive in performance with state-of-the-art fixed-function chips [10, 11, 16] are now becoming available. These chips implement a few low-level hardware primitives that can be configured by software into a processing pipeline, and are field-reconfigurable. Building a switch with such a chip is attractive because it does not compromise on data rates [28].

In terms of programmability, these chips today allow the network operator to program packet parsing and forwarding,

[*] Work done when employed at Barefoot Networks.

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

SIGCOMM '16, August 22 - 26, 2016, Florianopolis , Brazil

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4193-6/16/08…$15.00

DOI: http://dx.doi.org/10.1145/2934872.2934900

i.e., a programmer can program the set of protocol formats to be matched and the set of actions to be executed when matching packet headers in a match-action table. Languages such as P4 [27] are emerging as a way to express such match-action processing in a hardware-independent manner.

There is a gap between this form of programmability and the needs of data-plane algorithms. By contrast to packet forwarding, which doesn't modify state in the data plane, many data-plane algorithms create and modify *algorithmic state* in the switch as part of packet processing.

For such algorithms, programmability must directly capture the algorithm's intent without requiring the algorithm to be shoehorned into hardware primitives like a sequence of match-action tables. Indeed, the ability to directly capture an algorithm's intent pervades programming models for many other networking devices, e.g., software routers [46], network processors [36], and network endpoints [5].

By studying the requirements of data-plane algorithms and the constraints of line-rate hardware, we introduce a new abstraction to program and implement data-plane algorithms: a *packet transaction* (§3). A packet transaction is a sequential code block that is atomic and isolated from other such code blocks, with the semantics that any visible state is equivalent to a serial execution of packet transactions across packets in the order of packet arrival. Packet transactions let the programmer focus on the operations needed for each packet without worrying about other concurrent packets.

Packet transactions have an *all-or-nothing* guarantee: all packet transactions accepted by the packet transactions compiler will run at line rate, or be rejected. There is no "slippery slope" of running network algorithms at lower speeds as with network processors or software routers: when compiled, a packet transaction runs at line rate, or not at all. Performance is not just predictable, but guaranteed.

In realizing packet transactions, we make three new contributions. First, *Banzai*, a machine model for programmable line-rate switches (§2). Banzai models two important constraints (§2.4) for stateful line-rate operations: the inability to share state between different packet-processing units, and the requirement that any switch state modifications be visible to the next packet entering the switch. Based on these constraints, we introduce *atoms* to represent a programmable switch's packet-processing units.

Second, *Domino*, a new domain-specific language (DSL) for data-plane algorithms, with packet transactions at its core (§3). Domino is an imperative language with C-like syntax, to our knowledge the first to offer such a high-level programming abstraction for line-rate switches.

Third, *a compiler from Domino packet transactions to a Banzai target* (§4). The Domino compiler extracts *codelets* from transactions: code fragments, which if executed atomically, guarantee a packet transaction's semantics. It then uses program synthesis [59] to map codelets to atoms, rejecting the transaction if the atom cannot execute the codelet.

We evaluate expressiveness by programming a variety of data-plane algorithms (Table 3) in Domino and compare with P4. We find that Domino provides a more concise and natural programming model for stateful data-plane algorithms. Next, because no existing programmable switch supports the set of atoms required for our data-plane algorithms, we design a set of compiler targets for these algorithms based on Banzai (§5.2). We show that these targets are feasible in a 32-nm standard-cell library with $< 2\%$ cost in area relative to a 200 mm^2 baseline switching chip [40]. Finally, we compile data-plane algorithms written in Domino to these targets (§5.3) to show how a target's atoms determine the algorithms it can support. We conclude with several lessons for programmable switch design (§5.4).

Code for the Domino compiler, the Banzai machine model, and the code examples listed in Table 3 is available at http://web.mit.edu/domino.

2. A MACHINE MODEL FOR LINE-RATE SWITCHES

Banzai is a machine model for programmable line-rate switches that serves as the compiler target for Domino. Banzai is inspired by recent programmable switch architectures such as Barefoot Networks' Tofino [3], Intel's FlexPipe [13], and Cavium's XPliant Packet Architecture [19]. Banzai abstracts these architectures and extends them with stateful processing units to implement data-plane algorithms. These processing units, called *atoms*, model atomic operations that are natively supported by a programmable line-rate switch.

2.1 Background: Programmable switches

Packets arriving at a switch (top half of Figure 1) 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. Processing a packet at a stage may modify its header fields, through match-action rules, as well as some persistent state at that stage, e.g., packet counters. After the ingress pipeline, the packet is queued. Once the scheduler dequeues the packet, it is processed by a similar egress pipeline before it is transmitted.

To reduce chip area, there is only one ingress and one egress pipeline. This single pipeline is shared across all switch ports and handles aggregate traffic belonging to all ports, at all packet sizes. For instance, a 64-port switch with a line rate of 10 Gbit/s per port and a minimum packet size of 64 bytes needs to process around a billion packets per second, after accounting for minimum inter-packet gaps [28]. Equivalently, the pipeline runs at 1 GHz, and pipeline stages process a packet every clock cycle (1 ns). We assume one packet per clock cycle throughout the paper, and for concreteness, a 1 GHz clock frequency.

Having to process a packet every clock cycle in each stage constrains the operations that can be performed on each packet. In particular, any packet operation that modifies state visible to the next packet *must* finish execution in a single clock cycle (§2.3 shows why). Because of this restriction, programmable switching chips provide a small set of processing units or primitives for manipulating packets and state in a stage, unlike software routers. These primitives determine which algorithms run on the switch at line rate.

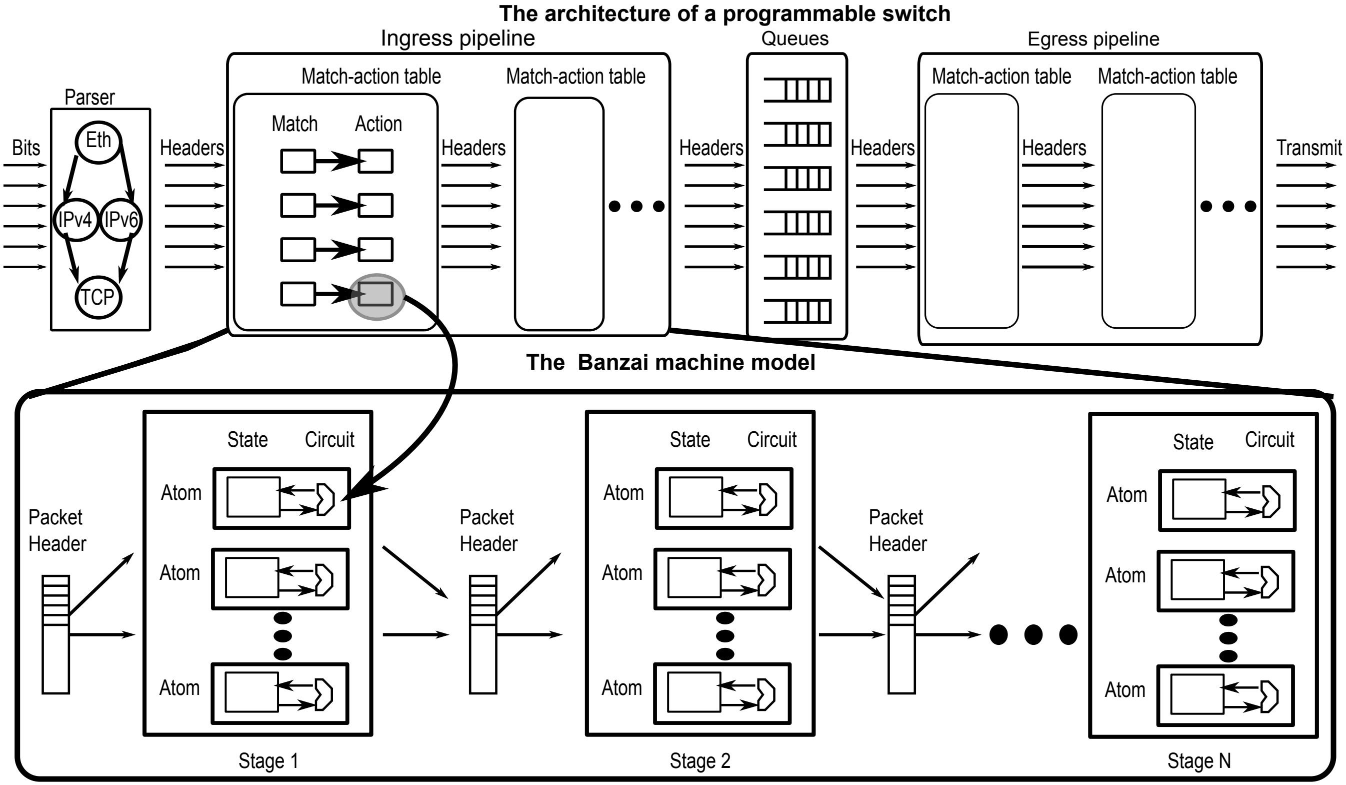


Figure 1: Banzai models the ingress or egress pipeline of a programmable switch. An atom corresponds to an action in a match-action table. Internally, an atom contains local state and a digital circuit modifying this state. Figure 2 details an atom.

The challenge for us is to develop primitives that allow a broad range of data-plane algorithms to be implemented, and to build a compiler to map a user-friendly description of an algorithm to the primitives provided by a switch.

2.2 The Banzai machine model

Banzai (the bottom half of Figure 1) models the ingress or egress switch pipeline. It models the computation within a match-action table in a stage (i.e., the action half of the match-action table), but not how packets are matched (e.g., direct or ternary). Banzai does not model packet parsing and assumes that packets arriving to Banzai are already parsed.

Concretely, Banzai is a feed-forward pipeline[1] consisting of a number of stages executing synchronously on every clock cycle. Each stage processes one packet every clock cycle and hands it off to the next. Unlike a CPU pipeline, which occasionally experiences pipeline stalls, Banzai's pipeline is deterministic, never stalls, and always sustains line rate. However, relative to a CPU pipeline, Banzai is restricted in the operations it supports (§2.4).

2.3 Atoms: Banzai's processing units

An *atom* is an atomic unit of packet processing supported natively by a Banzai machine, and the atoms within a Banzai machine form its instruction set. Each pipeline stage in Banzai contains a vector of atoms. Atoms in this vector modify mutually exclusive sections of the same packet header in parallel in every clock cycle, and process a new packet header every clock cycle.

In addition to packet headers, atoms may modify persistent state on the switch to implement stateful data-plane algorithms. To support such algorithms at line-rate, the atoms for a Banzai machine need to be substantially richer (Table 4) than the simple RISC-like stateless instruction sets for programmable switches today [28]. We explain why below.

Suppose we need to atomically increment a switch counter to count packets. One approach is hardware support for three simple single-cycle operations: *read* the counter from memory in the first clock cycle, *add* one in the next, and *write* it to memory in the third. This approach, however, does not provide atomicity. To see why, suppose packet A increments the counter from 0 to 1 by executing its read, add, and write at clock cycles 1, 2, and 3 respectively. If packet B issues its read at time 2, it will increment the counter again from 0 to 1, when it should be incremented to 2.

Locks over the shared counter are a potential solution. However, locking causes packet B to wait during packet A's increment, and the switch no longer sustains the line rate of one packet every clock cycle. CPUs employ microarchitectural techniques such as operand forwarding for this problem. But these techniques still suffer pipeline stalls, which prevents line-rate performance from being achieved.

Banzai provides an atomic increment operation at line rate with an *atom* to read a counter, increment it, and write it back in a single stage within one clock cycle. It uses the

[1] It is hard to physically route backward-flowing wires that would be required for feedback.

same approach of reading, modifying, and writing back to implement other stateful atomic operations at line rate.

Unlike stateful atomic operations, stateless atomic operations are easier to support with simple packet-field arithmetic. Consider, for instance, the operation `pkt.f1 = pkt.f2 + pkt.f3 - pkt.f4`. This operation does not modify any persistent switch state and only accesses packet fields. It can be implemented atomically by using two atoms: one atom to add fields f2 and f3 in one pipeline stage, and another to subtract f4 from the result in the next. An instruction set designer can provide *simple* stateless instructions operating on a pair of packet fields. These instructions can then be composed into larger stateless operations, without designing atoms specifically for each stateless operation.

Representing atoms. An atom is represented by a body of sequential code that captures the atom's behavior. It may also contain internal state local to the atom. An atom completes execution of this entire body of code, modifying a packet and any internal state before processing the next packet. The designer of a programmable switch would develop these atoms, and expose them to a switch compiler as the programmable switch's instruction set, e.g., Table 4.

Using this representation, a switch counter that wraps around at a value of 100 can be written as the atom:[2]

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

Similarly, a stateless operation like setting a packet field to a constant value can be written as the atom:

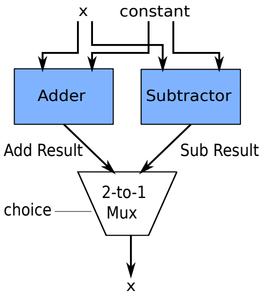
```
  pkt.field = value;
```

2.4 Constraints for line-rate operation

Memory limits. State in Banzai is local to each atom. It can neither be shared by atoms within a stage, nor atoms across stages. This is because building multi-ported memories accessible to multiple atoms is technically challenging and consumes additional chip area. However, state can be read into a packet header in one stage, for subsequent use by a downstream stage[3]. But, the Banzai pipeline is feed-forward, so state can only be carried forward, not backward.

Computational limits. Atoms need to execute atomically from one packet to the next, so any state internal to the atom must be updated before the next packet arrives. Because packets may be separated by as little as one clock cycle, we mandate that atom bodies finish execution within one clock cycle, and constrain atom bodies to do so.

We constrain atom bodies by defining *atom templates* (§4.3). An atom template is a program with configurable parameters that terminates within a clock cycle and specifies the atom's behavior. An example is an ALU with a restricted set of primitive operations (Figure 2a).



(a) Circuit for the atom

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

(b) Atom template

Figure 2: An atom and its template. The atom above can add or subtract a constant from a state variable `x` based on two configurable parameters, `constant` and `choice`.

Resource limits. We also limit the number of atoms in each stage (*pipeline width*) and the number of stages in the pipeline (*pipeline depth*). This is similar to limits on the number of stages, tables per stage, and memory per stage in programmable switch architectures [43].

2.5 What can Banzai not do?

Banzai is a good fit for data-plane algorithms that modify a small set of packet headers and carry out small amounts of computation per packet. Data-plane algorithms like deep packet inspection and WAN optimization require a switch to parse and process the packet payload as well—effectively parsing a large "header" consisting of each byte in the payload. This is challenging at line rates of 1 GHz, and such algorithms are best left to CPUs [52]. Some algorithms require complex computations, but not on every packet, e.g., a measurement algorithm that periodically scans a large table to perform garbage collection. Banzai's atoms model small operations that occur on every packet, and are unsuitable for such operations that span many clock cycles.

3. PACKET TRANSACTIONS

A programmer programs a data-plane algorithm by writing it as a packet transaction in Domino (Figure 3a). The Domino compiler then compiles this transaction to an atom pipeline for a Banzai machine (Figure 3b). We first describe packet transactions in greater detail by walking through an example (§3.1). Next, we discuss language constraints in Domino (§3.2) informed by line-rate switches. We then discuss triggering packet transactions (§3.3) and handling multiple transactions (§3.4).

3.1 Domino by example

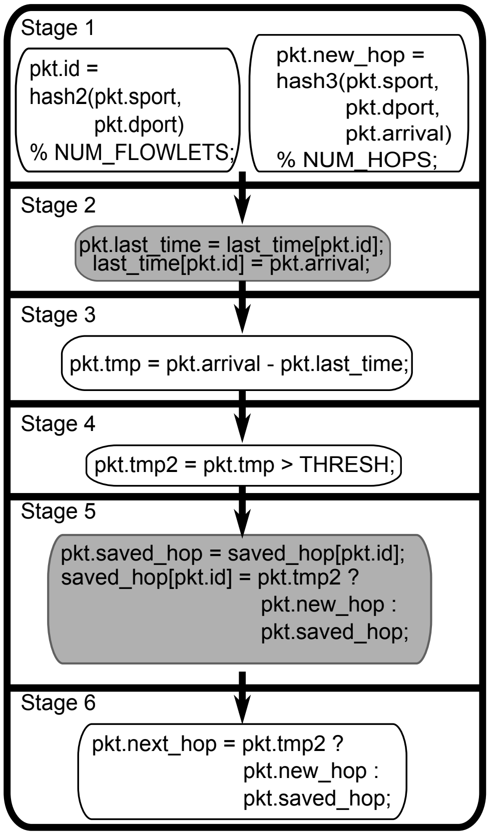
We use flowlet switching [57] as our running example. Flowlet switching is a load-balancing algorithm that sends bursts of packets, called flowlets, from a TCP flow on a randomly chosen next hop, provided the bursts are separated by a large enough time interval to ensure packets do not arrive out of order at a TCP receiver. For ease of exposition, we

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

[3] Figure 3b shows an example. `last_time` is read into `pkt.last_time` in stage 2, for subsequent use by stage 3.

```
 1  #define NUM_FLOWLETS     8000
 2  #define THRESH           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        > THRESH)
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  }
```

(a) Flowlet switching written in Domino


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

(b) 6-stage Banzai pipeline for flowlet switching. Control flows from top to bottom. Stateful atoms are in grey.

Figure 3: Programming flowlet switching in Domino

use only the source and destination ports in the hash function that randomly computes the next hop for flowlet switching.

Figure 3a shows flowlet switching in Domino and demonstrates its core language constructs. All packet processing happens in the context of a packet transaction (the function `flowlet` starting at line 17). The function's argument type `Packet` declares the fields in a packet (lines 5–12)[4] that can be referenced by the function body (lines 18–32). The function body can also modify persistent switch state using global variables (e.g., `last_time` and `saved_hop` on lines 14 and 15, respectively). The function body may use *intrinsics* such as `hash2` on line 23 to directly access hardware accelerators on the switch such as hash generators. The Domino compiler uses an intrinsic's signature to analyze read/write dependencies (§4.2), but otherwise considers it a blackbox.

Packet transaction semantics. Semantically, the programmer views the switch as invoking the packet transaction serially in the order in which packets arrive, with no concurrent packet processing. Put differently, the packet transaction modifies the passed-in packet argument and runs to completion, before starting on the next packet. These semantics allow the programmer to program under the illusion that a single, extremely fast, processor is serially executing the packet processing code for all packets. The programmer doesn't worry about parallelizing the code within and across pipeline stages to run at line rate.

3.2 The Domino language

Domino's syntax (Figure 4) is similar to C, but with several constraints (Table 1). These constraints are required for deterministic performance. Memory allocation, unbounded iteration counts, and unstructured control flow cause variable performance, which may prevent an algorithm from achieving line rate. Additionally, within a Domino transaction, each array can only be accessed using a single packet field, and repeated accesses to the same array are allowed only if that packet field is unmodified between accesses.

For example, all read and write accesses to `last_time` use the index `pkt.id`. `pkt.id` is not modified during the course of a single transaction execution (single packet); it only changes between executions (packets). This restriction on arrays mirrors restrictions on the stateful memories attached to atoms (§2.4), which require multiple ports to support distinct read and write addresses every clock cycle.

[4] A field is either a packet header, e.g., source port (`sport`) and destination port (`dport`), or packet metadata (`id`).

No iteration (while, for, do-while).
No unstructured control flow (goto, break, continue).
No heap, dynamic memory allocation, or pointers.
At most one location in each array is accessed by a single execution of a transaction.
No access to unparsed portions of the packet (payload).

Table 1: Restrictions in Domino

$$\begin{array}{rcl}
l \in \text{literals} \quad v \in \text{variables} & \mathit{bop} & \in \text{binary ops} \quad \mathit{uop} \in \text{unary ops} \\
e \in \text{expressions} & ::= & e.f \mid l \mid v \mid e \; \mathit{bop} \; e \mid \mathit{uop} \; e \mid e[d.f] \mid \\
 & & f(e_1, e_2, \ldots) \\
s \in \text{statements} & ::= & e = e \mid \texttt{if } (e) \; \{s\} \texttt{ else } \{s\} \mid s \; ; \; s \\
t \in \text{packet txns} & ::= & \mathit{name}(v)\{s\} \\
d \in \text{packet decls} & ::= & \{v_1, v_2, \ldots\} \\
sv \in \text{state var inits} & ::= & v = e \mid sv \; ; \; sv \\
p \in \text{Domino programs} & ::= & \{d; sv; t\}
\end{array}$$

Figure 4: Domino grammar. Type annotations (void, struct, int, and Packet) are elided for simplicity.

3.3 Triggering packet transactions

Packet transactions specify *how* to process packet headers and state. To specify *when* to run packet transactions, programmers use *guards*: predicates on packet fields that trigger a transaction if a packet matches the guard. For example, (`pkt.tcp_dst_port == 80`) would trigger heavy-hitter detection [63] on packets with TCP destination port 80.

Guards can be realized using an exact match in a match-action table, with the actions being the atoms compiled from a packet transaction. Guards can take various forms, e.g., exact, ternary, longest-prefix, and range-based matches, depending on the matches supported by the match-action pipeline. Because guards map straightforwardly to the match key in a match-action table, we focus only on compiling packet transactions in this paper.

3.4 Handling multiple transactions

So far, we have discussed a single packet transaction corresponding to a single data-plane algorithm. In practice, a switch would run multiple data-plane algorithms, each processing its own subset of packets. To address this, we envision a policy language that specifies pairs of guards and transactions. Realizing a policy is straightforward when all guards are disjoint. When guards overlap, multiple transactions need to execute on the same subset of packets, requiring a mechanism to compose transactions.

One composition semantics is to run the two transactions one after another sequentially in a user-specified order. This can be achieved by concatenating the two transaction bodies to create a larger transaction. We leave a detailed exploration of multiple transactions to future work, and focus only on compiling a single packet transaction here.

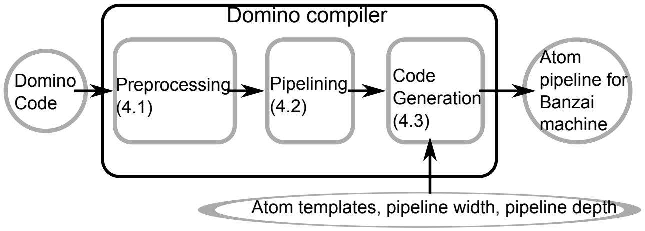


Figure 5: Passes in the Domino compiler

4. THE DOMINO COMPILER

The Domino compiler translates Domino programs to Banzai targets. The compiler provides an *all-or-nothing model*: if compilation succeeds, the program will run at line rate on the target with packet transaction semantics. Otherwise, if the program cannot run at line rate, it will not compile. This all-or-nothing model trades off diminished programmability for guaranteed line-rate performance, in contrast to software routers that provide greater flexibility, but lower and unpredictable run-time performance [34].

The Domino compiler has three passes (Figure 5), which we illustrate using the flowlet switching example. *Preprocessing* (§4.1) simplifies packet transactions into a simpler three-address code form [18]. *Pipelining* (§4.2) transforms preprocessed code into code for a *Pipelined Virtual Switch Machine (PVSM)*, an intermediate representation that models a switch pipeline with no computational or resource limits. *Code generation* (§4.3) transforms this intermediate representation into configuration for a Banzai machine, given the machine's computational and resource limits (§2.4), and rejects the program if it can not run at line rate. The Domino compiler uses many existing compilation techniques, but adapts them in important ways for line-rate switches (§4.4).

4.1 Preprocessing

Branch removal. A packet transaction's body can contain (potentially nested) branches (e.g., Lines 27 to 29 in Figure 3a). Branches alter control flow and complicate dependency analysis, i.e., whether a statement should precede another. We transform branches into conditional assignments, starting from the innermost `if` and proceeding outwards (Figure 6). This turns the transaction body into straight-line code with no branches, which simplifies dependency analysis during pipelining (§4.2).

Rewriting state variable operations. We now identify state variables in a packet transaction, e.g., `last_time` and `saved_hop` in Figure 3a. For each state variable, we create a *read flank* to read the variable into a temporary packet field. For an array, we also move the index expression into the read flank using the fact that only one array index is accessed per packet (§3.2). Within the packet transaction, we replace the state variable with the temporary packet field, and create a *write flank* to write this temporary packet field back to the state variable (Figure 7). After this, the only operations on state variables are reads and writes; all arithmetic happens on packet fields. Restricting stateful operations simplifies handling of state during pipelining (§4.2).

Converting to static single-assignment form. We next convert the code to static single-assignment form (SSA) [32], where every packet field is assigned exactly once. We do this by replacing every assignment to a packet field with a new packet field and propagating this until the next assignment to the same field (Figure 8) . Because fields are assigned once, SSA removes Write-After-Read and Write-After-Write dependencies. Only Read-After-Write dependencies remain during pipelining (§4.2).

Flattening to three-address code. Three-address code is a representation where all instructions are either reads/writes into state variables or operations on packet fields of the form `pkt.f1 = pkt.f2 op pkt.f3`, where `op` can be an arithmetic, logical, relational, or conditional [5] operator. We also allow either one of `pkt.f2` or `pkt.f3` to be an intrinsic function call. To convert to three-address code, we flatten expressions that are not in three-address code using temporary packet fields, e.g., `pkt.tmp2` in Figure 9.

Flattening to three-address code breaks down statements in the packet transaction into a much simpler form that is closer to the atoms available in the Banzai machine. For instance, there are no nested expressions. The simpler form of three-address code statements makes it easier to map them one-to-one to atoms during code generation (§4.3).

4.2 Pipelining

At this point, the preprocessed code is still one sequential code block. Pipelining turns this sequential code block into a pipeline of *codelets*, where each codelet is a sequential block of three-address code statements. This codelet pipeline corresponds to an intermediate representation we call the *Pipelined Virtual Switch Machine (PVSM)*. PVSM has no computational or resource limits, analogous to intermediate representations for CPUs [15] that have infinite virtual registers. Later, during code generation, we map these codelets to atoms available in a Banzai machine while respecting its constraints.

We create PVSM's codelet pipeline using the steps below.

1. Create a graph with one node for each statement in the preprocessed code.
2. Now, add *stateful dependencies* by adding a pair of edges between the read and write flanks of the same state variable, e.g., in Figure 10a, the node pair `pkt.last_time = last_time[pkt.id]` and `last_time[pkt.id] = pkt.arrival`. Because of preprocessing, all stateful operations are paired up as read and write flanks. Hence, there is no risk of a "stranded" stateful operation.
3. Now, add *stateless dependencies* by adding an edge from any node that writes a packet variable to any node that reads the same packet variable, e.g., from `pkt.tmp = pkt.arrival - pkt.last_time` to `pkt.tmp2 = pkt.tmp > THRESH` in Figure 10a. We only check read-after-write dependencies because write-after-read and write-after-write dependencies don't exist after SSA, and we eliminate control dependencies [32] through branch removal.
4. Generate strongly connected components (SCCs) of this dependency graph and condense them into a directed acyclic graph (DAG). This captures the notion that all operations on a state variable must be confined to one codelet/atom because state cannot be shared between atoms. Figure 10b shows the DAG produced by condensing Figure 10a.
5. Schedule the resulting DAG by creating a new pipeline stage when one node depends on another. This results in the codelet pipeline shown in Figure 3b.[6]

4.3 Code generation

To determine if a codelet pipeline can be compiled to a Banzai machine, we consider two constraints specified by any Banzai machine (§2.4). Resource limits specify the number of atoms in a stage (pipeline width) and number of stages (pipeline depth), while computational limits specify the atom templates provided by a Banzai machine.

Resource limits. To handle resource limits, we scan each pipeline stage in the codelet pipeline starting from the first to check for pipeline width violations. If we violate the pipeline width, we insert as many new stages as required and spread codelets evenly across these stages. We continue until the number of codelets in all stages is under the pipeline width, rejecting the program if we exceed the pipeline depth.

Computational limits. Next, we determine if each codelet in the pipeline can be mapped to atoms provided by the Banzai machine. In general, codelets have multiple three-address code statements that need to execute atomically. For instance, updating the state variable `saved_hop` in Figure 3b 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 a codelet, given an atom template.

Each atom template has a set of configuration parameters, where the parameters determine the atom's behavior. For instance, Figure 2a shows an atom that can perform stateful addition or subtraction, depending on the configuration parameters `choice` and `constant`. Each codelet can be viewed as a functional specification of the atom. With that in mind, the mapping problem is equivalent to searching for values of the atom's configuration parameters that result in the atom implementing the codelet.

We use the SKETCH program synthesizer [59] for this purpose, as the atom templates can be easily expressed using SKETCH. SKETCH also provides efficient search algorithms and has been used for similar purposes in other domains [29, 30]. As an illustration, assume we want to map the codelet `x=x+1` to the atom template shown in Figure 2b. SKETCH will search for possible parameter values so that the resulting atom is functionally identical to the codelet, for all possible input values of `x` up to a certain bound. In this case, SKETCH finds the solution with `choice=0` and

[5] Conditional operations alone have four arguments.

[6] We refer to this both as a codelet and an atom pipeline because codelets map one-to-one atoms (§4.3).

```
if (pkt.arrival - last_time[pkt.id] > THRESH) {         pkt.tmp = pkt.arrival - last_time[pkt.id] > THRESH;
  saved_hop[pkt.id] = pkt.new_hop;               ⟹      saved_hop[pkt.id] = pkt.tmp  // Rewritten
}                                                                          ? pkt.new_hop
                                                                           : saved_hop[pkt.id];
```

Figure 6: Branch removal

```
                                           pkt.id = hash2(pkt.sport,            // Read flank
pkt.id = hash2(pkt.sport,                                 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
```

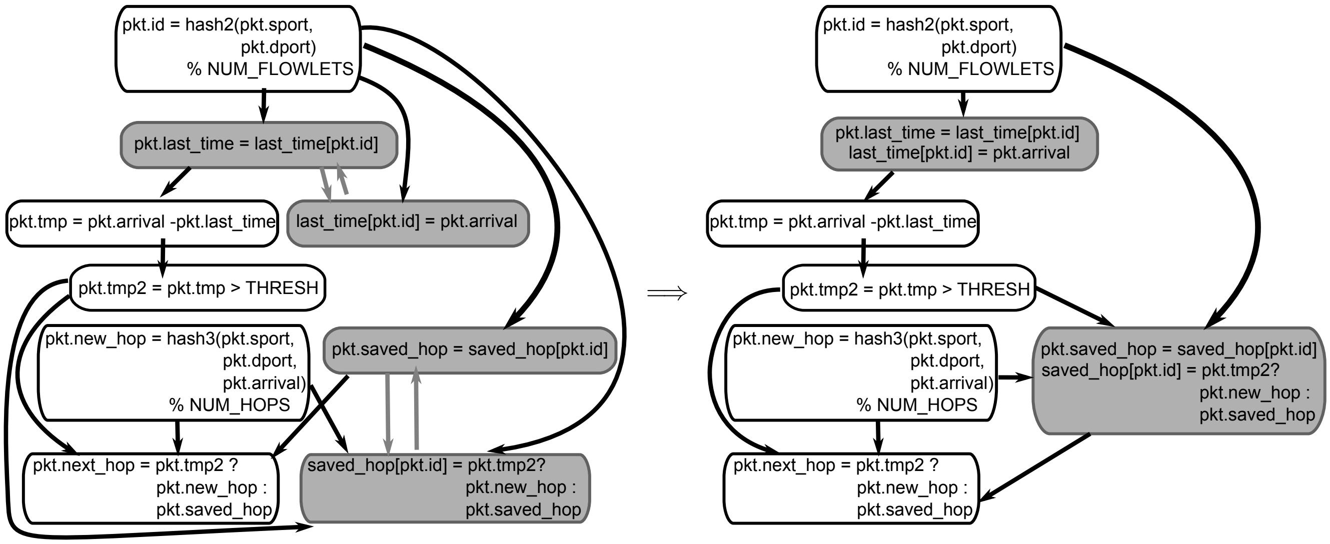
Figure 7: Rewriting state variable operations

```
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 8: Converting to static single-assignment form

```
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 > THRESH;
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 9: Flowlet switching in three-address code. Lines 1 and 4 are flipped relative to Figure 3a because pkt.id is an array index expression and is moved into the read flank.


pkt.id = hash2(pkt.sport, pkt.dport) % NUM_FLOWLETS
pkt.last_time = last_time[pkt.id]
pkt.tmp = pkt.arrival -pkt.last_time
last_time[pkt.id] = pkt.arrival
pkt.tmp2 = pkt.tmp > THRESH
pkt.new_hop = hash3(pkt.sport, pkt.dport, pkt.arrival) % NUM_HOPS
pkt.saved_hop = saved_hop[pkt.id]
pkt.next_hop = pkt.tmp2 ? pkt.new_hop : pkt.saved_hop
saved_hop[pkt.id] = pkt.tmp2? pkt.new_hop : pkt.saved_hop
⟹
pkt.id = hash2(pkt.sport, pkt.dport) % NUM_FLOWLETS
pkt.last_time = last_time[pkt.id] last_time[pkt.id] = pkt.arrival
pkt.tmp = pkt.arrival -pkt.last_time
pkt.tmp2 = pkt.tmp > THRESH
pkt.new_hop = hash3(pkt.sport, pkt.dport, pkt.arrival) % NUM_HOPS
pkt.saved_hop = saved_hop[pkt.id] saved_hop[pkt.id] = pkt.tmp2? pkt.new_hop : pkt.saved_hop
pkt.next_hop = pkt.tmp2 ? pkt.new_hop : pkt.saved_hop

(a) Stateless dependencies in black, stateful in gray.

(b) DAG after condensing SCCs.

Figure 10: Dependency graphs before and after condensing strongly connected components

Technique	Prior Work	Differences
Conversion to straight-line code	If-Conversion [23]	No backward control flow (gotos, break, continue)
SSA	Cytron et al. [32]	SSA runs on straight-line code with no branches
Strongly Connected Components	Lam [48]	Scheduling in space vs. time
Code generation using program synthesis	Chlorophyll [53], technology mapping [49], instruction selection [20]	Optimal vs. best-effort mapping, One-to-one mapping vs. one-to-many mapping

Table 2: Domino's compiler in relation to prior work

`constant=1`. In contrast, if the specification is the codelet `x=x*x`, SKETCH will return an error as no parameters exist.

Using program synthesis for code generation frees the compiler developer from implementing custom code generators for different Banzai machines. Instead, the compiler developer only has to express the Banzai machine's atom templates using SKETCH, and the SKETCH synthesizer automatically maps codelets to atoms.

4.4 Related compiler techniques

Table 2 shows the relationship between Domino's compilation techniques and prior work. The use of Strongly Connected Components (SCCs) is inspired by software pipelining for VLIW architectures [48]. The size of the largest SCC affects the *maximum throughput* of the pipelined loop in software pipelining. For Domino, it affects the *circuit area* of the atom required to run a program at line rate. Domino trades off an increase in space for line-rate performance.

Program synthesis was used for code generation in Chlorophyll [53]. Code generation for Domino also shares similar goals to technology mapping [49] and instruction selection [20]. However, prior work maps a code sequence to *multiple* instructions/tiles, using heuristics to minimize instruction count. Domino's problem is simpler: we map each codelet to a single atom using SKETCH. The simpler problem allows a non-heuristic solution: if there is any way to map the codelet to an atom, SKETCH will find it.

Branch removal resembles If-Conversion [23], a technique used in vectorizing compilers. This procedure is easier in Domino because there is no backward control transfer (`goto`, `break`, `continue`).

5. EVALUATION

We evaluate Domino's expressiveness by using it to program several data-plane algorithms (Table 3), and comparing it to writing them in P4 (§5.1). To validate that these algorithms can run at line rate, we design a concrete set of Banzai machines (Table 4) as compiler targets for Domino (§5.2). We estimate that these machines are feasible in hardware because their atoms incur modest chip area overhead. We use the Domino compiler to compile the algorithms in Table 3 to the targets in Table 4 (§5.3). We conclude with some lessons for programmable switch design (§5.4).

5.1 Expressiveness

We program several data-plane algorithms (Table 3) using Domino. These algorithms encompass data-plane traffic engineering, in-network congestion control, active queue management, network security, and measurement. We also used Domino to express the priority computation for programming scheduling using push-in first-out queues [58].

In all these cases, the algorithms are already available as blocks of imperative code from online sources; translating them to Domino syntax was straightforward. In contrast, expressing any of them in P4 requires manually teasing out portions of the algorithm that can reside in independent match-action tables and then chaining these tables together.

Of the algorithms in Table 3, only flowlet switching has a publicly available P4 implementation [9] that we can compare against. This implementation requires 231 lines of uncommented P4, compared to only 37 lines of Domino code in Figure 3a. Not only that, using P4 also requires the programmer to manually specify tables, the actions within tables, and how tables are chained—all to implement a single data-plane algorithm. The Domino compiler automates this process; to demonstrate this, we developed a backend for Domino that generates the equivalent P4 code. We list the number of lines of code for these auto-generated P4 programs in Table 3.

5.2 Compiler targets

We design a set of compiler targets for Domino based on the Banzai machine model (§2). First, we describe how to assess the feasibility of atoms: whether they can run at a 1 GHz clock frequency, and what area overhead they incur in silicon. Next, we discuss the design of stateless and stateful atoms separately. Finally, we discuss how these stateless and stateful atoms are combined together in our compiler targets.

Atom feasibility. We synthesize a digital circuit corresponding to an atom template by writing the atom template in Verilog, and using the Synopsys Design Compiler [7] to compile the Verilog code. The Design Compiler checks if the resulting circuit meets timing at 1 GHz in a 32-nm standard-cell library, and outputs its gate area. We use this gate area, along with the area of a 200 mm^2 baseline switching chip [40], to estimate the area overhead for provisioning a Banzai machine with multiple instances of this atom.

Designing stateless atoms. Stateless atoms are easier to design because arbitrary stateless operations can be broken up into multiple pipeline stages without violating atomicity (§2.3). We design a stateless atom that can support simple arithmetic (add, subtract, left shift, right shift), logical (and, or, xor), relational (>=, <=, ==, !=), and conditional instructions (C's "?" operator) on a pair of packet fields. Any packet field can also be substituted with a constant operand. This stateless atom meets timing at 1 GHz and occupies an area of 1384 μm^2 (Table 4).

Designing stateful atoms. The choice of stateful atoms determines the algorithms a line-rate switch can support. A more complex stateful atom can support more data-plane al-

Algorithm	Stateful operations	Most expressive atom	# of stages, max. atoms/stage	Ingress or Egress Pipeline?	Domino LOC	P4 LOC
Bloom filter (3 hash functions)	Test/Set membership bit on every packet.	Write	4, 3	Either	29	104
Heavy Hitters [63] (3 hash functions)	Increment Count-Min Sketch [31] on every packet.	RAW	10, 9	Either	35	192
Flowlets [57]	Update saved next hop if flowlet threshold is exceeded.	PRAW	6, 2	Ingress	37	107
RCP [60]	Accumulate RTT sum if RTT is under maximum allowable RTT.	PRAW	3, 3	Egress	23	75
Sampled NetFlow [17]	Sample a packet if packet count reaches N; Reset count to 0 when it reaches N.	IfElseRAW	4, 2	Either	18	70
HULL [22]	Update counter for virtual queue.	Sub	7, 1	Egress	26	95
Adaptive Virtual Queue [47]	Update virtual queue size and virtual capacity	Nested	7, 3	Ingress	36	147
Priority computation for weighted fair queueing [58]	Compute packet's virtual start time using finish time of last packet in that flow.	Nested	4, 2	Ingress	29	87
DNS TTL change tracking [26]	Track number of changes in announced TTL for each domain	Nested	6,3	Ingress	27	119
CONGA [21]	Update best path's utilization/id if we see a better path. Update best path utilization alone if it changes.	Pairs	4, 2	Ingress	32	89
CoDel [51]	Update: Whether we are marking or not. Time for next mark. Number of marks so far. Time at which min. queueing delay will exceed target.	Doesn't map	15, 3	Egress	57	271

Table 3: Data-plane algorithms

gorithms, but may not meet timing and occupies more area. To illustrate this effect, we design a containment hierarchy (Table 4) of stateful atoms, where each atom can express all stateful operations that its predecessor can. These atoms start out with the simplest stateful capability: the ability to read or write state alone. They then move on to the ability to read, add, and write back state atomically (RAW), a predicated version of the same (PRAW), and so on. When synthesized to a 32-nm standard-cell library, all our stateful atoms meet timing at 1 GHz. However, the atom's area and minimum end-to-end propagation delay increases with the atom's complexity (Table 4).

The compiler targets. We design seven Banzai machines as compiler targets. A single Banzai machine has 600 atoms.

1. 300 are stateless atoms of the single stateless atom type from Table 4.
2. 300 are stateful atoms of one of the seven stateful atom types from Table 4 (Read/Write through Pairs).

These 300 stateless and stateful atoms are laid out physically as 10 stateless and stateful atoms per pipeline stage and 30 pipeline stages. While the number 300 and the pipeline layout are arbitrary, they are sufficient for all examples in Table 3, and incur modest area overhead as we show next.

We estimate the area overhead of these seven targets relative to a 200 mm^2 chip [40], which is at the lower end of chip sizes today. For this, we multiply the individual atom areas from Table 4 by 300 for both the stateless and stateful atoms. For 300 atoms, the area overhead is 0.2 % for the stateless atom and 0.9 % for the Pairs atom, the largest among our stateful atoms. The area overhead combining both stateless and stateful atoms for all our targets is at most 1.1%—a modest price for the programmability it provides.

5.3 Compiling Domino programs to Banzai machines

We now consider every target from Table 4[7], and every data-plane algorithm from Table 3 to determine if the algorithm can run at line rate on a particular Banzai machine.

We say an algorithm can run at line rate on a Banzai machine if every codelet within the data-plane algorithm can be mapped (§4.3) to either the stateful or stateless atoms provided by the Banzai machine. Because our stateful atoms

[7] Because every target is uniquely identified by its stateful atom type, we use the two interchangeably.

Atom	Description	Area (μm^2) at 1 GHz	Min. delay (ps)
Stateless	Arithmetic, logic, relational, and conditional operations on packet/constant operands	1384	387
Read/Write	Read/Write packet field/constant into single state variable.	250	176
ReadAddWrite (RAW)	Add packet field/constant to state variable (OR) Write packet field/constant into state variable.	431	316
Predicated ReadAddWrite (PRAW)	Execute RAW on state variable only if a predicate is true, else leave unchanged.	791	393
IfElse ReadAddWrite (IfElseRAW)	Two separate RAWs: one each for when a predicate is true or false.	985	392
Subtract (Sub)	Same as IfElseRAW, but also allow subtracting a packet field/constant.	1522	409
Nested Ifs (Nested)	Same as Sub, but with an additional level of nesting that provides 4-way predication.	3597	580
Paired updates (Pairs)	Same as Nested, but allow updates to a pair of state variables, where predicates can use both state variables.	5997	606

Table 4: Atom areas and minimum critical-path delays in a 32-nm standard-cell library. All atoms meet timing at 1 GHz. Each of the seven compiler targets contains 300 instances of one of the seven stateful atoms (Read/Write to Pairs) and 300 instances of the single stateless atom.

are arranged in a containment hierarchy, we list the *most expressive* stateful atom/target required for each data-plane algorithm in Table 3.

As Table 3 shows, the choice of stateful atom determines what algorithms can run on a switch. For instance, with only the ability to read or write state, only the Bloom Filter algorithm can run at line rate, because it only requires the ability to test and set membership bits. Adding the ability to increment state (the RAW atom) permits Heavy Hitters to run at line rate, because it employs a count-min sketch that is incremented on each packet.

5.4 Lessons for programmable switches

Atoms with a single state variable support many algorithms. The algorithms from Bloom Filter through DNS TTL Change Tracking in Table 3 can run at line rate using the Nested Ifs atom that modifies a single state variable.

But, some algorithms modify a pair of state variables atomically. An example is CONGA, whose code is given 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)[8] and vice versa. These two state variables cannot be separated into different stages and still guarantee a packet transaction's semantics. The Pairs atom, where the update to a state variable is conditioned on a predicate of a pair of state variables, allows CONGA to run at line rate.

There will always be algorithms that cannot sustain line rate. While the targets and their atoms in Table 4 are sufficient for several data-plane algorithms, there are algorithms that they can't run at line rate. An example is CoDel, which cannot be implemented because it requires a square root operation that isn't provided by any of our targets. One possibility is a look-up table abstraction that allows us to approximate such mathematical functions. However, regardless of what set of atoms we design for a particular target, there will always be algorithms that cannot run at line rate.

Atom design is constrained by timing, not area. Atoms are affected by two factors: their area and their timing, i.e., the minimum delay on the critical path of the atom's combinational circuit. For the few hundred atoms that we require, atom area is insignificant (< 2%) relative to chip area. Further, even for future atoms that are larger, area may be controlled by provisioning fewer atom instances.

However, atom timing is critical. Table 4 shows a 3.4x range in minimum critical-path delay between the simplest and the most complex atoms. This increase can be explained by looking at the simplified circuit diagrams for the first three atoms (Table 5), which show an increase in circuit depth with atom complexity.

Because the clock frequency of a circuit is at least as small as the reciprocal of this minimum critical-path delay, a more complex atom results in a lower clock frequency and a lower line rate. Although all our atoms have a minimum critical-path delay under 1 ns (1 GHz), it is easy to extend them with functionality that violates timing at 1 GHz.

In summary, for a switch designer, the minimum delay on the critical path of atoms is the most important metric to optimize. The most programmable line-rate switches will have the highest density of useful stateful functionality squeezed into a critical path budget of 1 clock cycle.

Compilers can be used to design instruction sets. Designing an instruction set for a programmable substrate is a chicken-and-egg problem: the choice of instructions determines which algorithms can execute on that target, while the choice of algorithms dictates what instructions are required in the target. Indeed, other programmable substrates (GPUs, CPUs, DSPs) go through an iterative process to design a good instruction set.

A compiler can aid this process. To show how, we describe how we designed the stateful atoms in Table 4. We pick a data-plane algorithm, partially execute the Domino

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

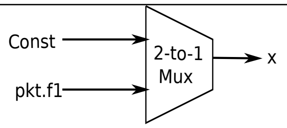
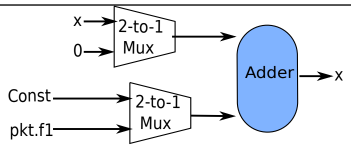
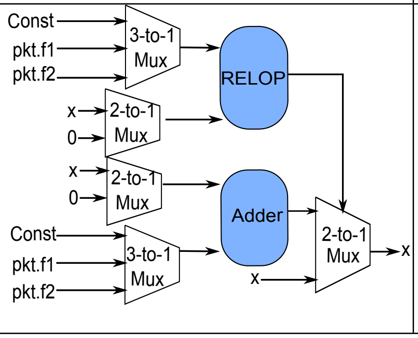
Atom	Circuit	Min. delay (ps)
Read/Write		176
ReadAddWrite (RAW)	 x 0 2-to-1 Mux Const pkt.f1 2-to-1 Mux Adder x	316
Predicated ReadAddWrite (PRAW)	 Const pkt.f1 pkt.f2 3-to-1 Mux RELOP x 0 2-to-1 Mux x 0 2-to-1 Mux Adder Const pkt.f1 pkt.f2 3-to-1 Mux x 2-to-1 Mux x	393

Table 5: An atom's minimum critical-path delay increases with circuit depth. Mux is a multiplexer. RELOP is a relational operation (>, <, ==, !=). `x` is a state variable. `pkt.f1` and `pkt.f2` are packet fields. `Const` is a constant operand.

compiler to generate a codelet pipeline, inspect the stateful codelets, and create an atom that expresses all the computations required by the stateful codelets. We check that an atom can express all these computations by fully executing the compiler on the data-plane algorithm with that atom as the target. We then move on to the next algorithm, extending our atom through a process of trial-and-error to capture more computations, and using the compiler to verify our intuitions on extending atoms. In the process, we generate a hierarchy of atoms, each of which works for a subset of algorithms.

Our atom design process is manual and ad hoc at this point, but it already shows how a compiler can aid in instruction-set design for programmable switches. Using the same iterative approach involving a compiler, we anticipate the atoms in Banzai machines evolving as data-plane algorithms demand more of the hardware.

6. RELATED WORK

Abstract machines for line-rate switches. NetASM [55] is an abstract machine and intermediate representation (IR) for programmable data planes that is portable across network devices: FPGAs, virtual switches, and line-rate switches. Banzai is a low-level machine model for line-rate switches alone and can be used as a NetASM target. Because of its role as a low-level machine model, Banzai models practical constraints required for line-rate operation (§2.4) that an IR like NetASM doesn't have to. For instance, Banzai machines don't permit sharing state between atoms and use atom templates to limit computations that can happen at line rate.

Programmable data planes. Eden [25] provides a programmable data plane using commodity switches by programming end hosts alone. Domino targets programmable switches that provide more flexibility relative to an end-host-only solution. For instance, Domino allows us to program in-network congestion control and AQM schemes, which are beyond Eden's capabilities. Tiny Packet Programs (TPP) [42] allow end hosts to embed small programs in packet headers, which are then executed by the switch. TPPs use a restricted instruction set to facilitate switch execution; we show that switch instructions must and can be substantially richer (Table 4) for stateful data-plane algorithms.

Software routers [35, 46] and network processors [14] are flexible, but at least 10×–100× slower than programmable switches [19, 3]. FPGA-based platforms like the Corsa DP 6440 [1], which supports an aggregate capacity of 640 Gbit/s, are faster, but still 5×–10× slower than programmable switches [3, 19].

Programming languages for networks. Many programming languages target the network control plane [39, 62]. Domino focuses on the data plane, which requires different programming constructs and compilation techniques.

Several DSLs target the data plane. Click [46] uses C++ for packet processing on software routers. packetC [36], Intel's auto-partitioning C compiler [33], and Microengine C [12] target network processors. Domino's C-like syntax and sequential semantics are inspired by these DSLs. However, because it targets line-rate switches, Domino is more constrained. For instance, because compiled programs run at line rate, Domino forbids loops, and because Banzai has no shared state, Domino has no synchronization constructs.

Jose et al. [43] focus on compiling P4 programs to programmable data planes such as the RMT and FlexPipe architectures. Their work focuses only on compiling stateless data-plane tasks such as forwarding and routing, while Domino handles stateful data-plane algorithms.

Abstractions for stateful packet processing. SNAP [24] programs stateful data-plane algorithms using a network transaction: an atomic block of code that treats the entire network as one switch [44]. It then uses a compiler to translate network transactions into rules on each switch. SNAP needs a compiler to compile these switch-local rules to a switch's pipeline, and can use Domino for this purpose.

FAST [50] provides switch support and software abstractions for state machines. Banzai's atoms support more general stateful processing beyond state machines that enable a much wider class of data-plane algorithms.

7. DISCUSSION

Packet transactions provide a pathway to take algorithms that were hitherto meant only for software routers and run them on emerging programmable line-rate switching chips. However, more work must be done before packet transactions are ready for production use.

1. Packet transactions provide the first transactional semantics for line-rate packet processing. These semantics make it easier to reason about correctness and performance, but they exclude algorithms that cannot run at line rate while respecting these semantics. Are weaker semantics sensible? One possibility is approximating transactional semantics by only processing a sampled packet stream. This provides an increased time budget for each packet in the sampled stream, potentially allowing the packet to be *recirculated* through the pipeline multiple times for packet processing.
2. Our compiler doesn't aggressively optimize. For instance, it is possible to fuse two stateful codelets incrementing two independent counters into the same instance of the Pairs atom. However, by carrying out a one-to-one mapping from codelets to the atoms implementing them, our compiler precludes these optimizations. Developing an *optimizing* compiler for packet transactions is an area for future work.
3. Supporting multiple packet transactions in Domino also requires further work. When a switch executes multiple transactions, there may be opportunities for inter-procedural analysis [20], which goes beyond compiling individual transactions and looks at multiple transactions together. For instance, the compiler could detect computations common to multiple transactions and execute them only once.
4. Finally, we have a manual design process for atoms. Formalizing this design process and automating it into an atom-design tool would be useful for switch designers. For instance, given a corpus of data-plane algorithms, can we automatically mine this corpus for stateful and stateless codelets, and design an atom (or atoms) that captures the computations required by some (or all) of them?

8. CONCLUSION

This paper presented Domino, a C-like imperative language that allows programmers to write packet-processing code using packet transactions, which are sequential code blocks that are atomic and isolated from other such code blocks. The Domino compiler compiles packet transactions to hardware configurations for Banzai, which is a machine model based on programmable line-rate switch architectures [13, 19, 3]. Our results suggest that it is possible to have both the convenience of a familiar programming model and the performance of a line-rate switch, provided that the algorithm can indeed run at line rate. Packet-processing languages are still in their infancy; we hope these results will prompt further work on programming abstractions for high-performance packet-processing hardware.

Acknowledgements

We thank our shepherd, Bruce Maggs, the anonymous SIGCOMM reviewers, Amy Ousterhout, and Pratiksha Thaker for their suggestions that improved the presentation of the paper. This work was partly supported by NSF grants CNS-1563826 and CNS-1563788. We thank the industrial partners of the MIT Center for Wireless Networks and Mobile Computing (Wireless@MIT) for their support.

9. REFERENCES

[1] 100G Data Planes, DP 6440, DP 6430 | Corsa Technology. http://www.corsa.com/products/dp6440/.
[2] Arista - Arista 7050 Series. https://www.arista.com/en/products/7050-series.
[3] Barefoot: The World's Fastest and Most Programmable Networks. https://barefootnetworks.com/media/white_papers/Barefoot-Worlds-Fastest-Most-Programmable-Networks.pdf.
[4] Cisco Nexus Family. http://www.cisco.com/c/en/us/products/switches/cisco_nexus_family.html.
[5] Components of Linux Traffic Control. http://tldp.org/HOWTO/Traffic-Control-HOWTO/components.html.
[6] Dell Force10. http://www.force10networks.com/.
[7] Design Compiler - Synopsys. http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages/default.aspx.
[8] DPDK: Data Plane Development Kit. http://dpdk.org/.
[9] Flowlet Switching in P4. https://github.com/p4lang/tutorials/tree/master/SIGCOMM_2015/flowlet_switching.
[10] High Capacity StrataXGS®Trident II Ethernet Switch Series. http://www.broadcom.com/products/Switching/Data-Center/BCM56850-Series.
[11] High-Density 25/100 Gigabit Ethernet StrataXGS Tomahawk Ethernet Switch Series. http://www.broadcom.com/products/Switching/Data-Center/BCM56960-Series.
[12] Intel Enhances Network Processor Family with New Software Tools and Expanded Performance. http://www.intel.com/pressroom/archive/releases/2001/20010220net.htm.
[13] Intel FlexPipe. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.
[14] 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.
[15] LLVM Language Reference Manual - LLVM 3.8 documentation. http://llvm.org/docs/LangRef.html#abstract.
[16] Mellanox Products: SwitchX-2 Ethernet Optimized for SDN. http://www.mellanox.com/page/products_dyn?product_family=146&mtag=switchx_2_en.
[17] Sampled NetFlow. http://www.cisco.com/c/en/us/td/docs/ios/12_0s/feature/guide/12s_sanf.html.
[18] Three-address code. https://en.wikipedia.org/wiki/Three-address_code.
[19] XPliant™ Ethernet Switch Product Family. http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html.
[20] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
[21] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed Congestion-Aware Load Balancing for Datacenters. In *SIGCOMM*, 2014.
[22] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *NSDI*, 2012.
[23] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In *POPL*, 1983.

[24] M. T. Arashloo, Y. Karol, M. Greenberg, J. Rexford, and D. Walker. SNAP: Stateful Network-Wide Abstractions for Packet Processing. *arXiv:1512.00822*.

[25] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea. Enabling End-Host Network Functions. In *SIGCOMM*, 2015.

[26] L. Bilge, E. Kirda, C. Kruegel, and M. Balduzzi. EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis. In *NDSS*, 2011.

[27] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM CCR*, July 2014.

[28] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *SIGCOMM*, 2013.

[29] A. Cheung, A. Solar-Lezama, and S. Madden. Using Program Synthesis for Social Recommendations. In *CIKM*, 2012.

[30] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing Database-backed Applications with Query Synthesis. In *PLDI*, 2013.

[31] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *Journal of Algorithms*, April 2005.

[32] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Language Systems*, 1991.

[33] J. Dai, B. Huang, L. Li, and L. Harrison. Automatically Partitioning Packet Processing Applications for Pipelined Architectures. In *PLDI*, 2005.

[34] M. Dobrescu, K. Argyraki, and S. Ratnasamy. Toward Predictable Performance in Software Packet-Processing Platforms. In *NSDI*, 2012.

[35] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *SOSP*, 2009.

[36] R. Duncan and P. Jungck. packetC Language for High Performance Packet Processing. In *11th IEEE International Conference on High Performance Computing and Communications*, 2009.

[37] C. Estan, G. Varghese, and M. Fisk. Bitmap Algorithms for Counting Active Flows on High-speed Links. *IEEE/ACM Trans. Netw.*, Oct. 2006.

[38] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. Netw.*, Aug. 1993.

[39] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ICFP*, 2011.

[40] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design Principles for Packet Parsers. In *ANCS*, 2013.

[41] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.

[42] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. In *SIGCOMM*, 2014.

[43] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling Packet Programs to Reconfigurable Switches. In *NSDI*, 2015.

[44] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "One Big Switch" Abstraction in Software-defined Networks. In *CoNEXT*, 2013.

[45] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *SIGCOMM*, 2002.

[46] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 2000.

[47] S. S. Kunniyur and R. Srikant. An Adaptive Virtual Queue (AVQ) Algorithm for Active Queue Management. *IEEE/ACM Trans. Netw.*, Apr. 2004.

[48] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *PLDI*, 1988.

[49] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1st edition, 1994.

[50] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan. Flow-level State Transition As a New Switch Primitive for SDN. In *SIGCOMM*, 2014.

[51] K. Nichols and V. Jacobson. Controlling Queue Delay. *ACM Queue*, 10(5), May 2012.

[52] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A Framework for NFV Applications. In *SOSP*, 2015.

[53] P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodik. Chlorophyll: Synthesis-aided Compiler for Low-power Spatial Architectures. In *PLDI*, pages 396–407, 2014.

[54] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the Social Network's (Datacenter) Network. In *SIGCOMM*, 2015.

[55] M. Shahbaz and N. Feamster. The Case for an Intermediate Representation for Programmable Data Planes. In *SOSR*, pages 3:1–3:6, 2015.

[56] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM*, 2015.

[57] S. Sinha, S. Kandula, and D. Katabi. Harnessing TCPs Burstiness using Flowlet Switching. In *HotNets*, 2004.

[58] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable Packet Scheduling at Line Rate. In *SIGCOMM*, 2016.

[59] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial Sketching for Finite Programs. In *ASPLOS*, 2006.

[60] C. Tai, J. Zhu, and N. Dukkipati. Making Large Scale Deployment of RCP Practical for Real Networks. In *INFOCOM*, 2008.

[61] D. L. Tennenhouse and D. J. Wetherall. Towards an Active Network Architecture. In *DARPA Active Networks Conference and Exposition*, 2002.

[62] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *SIGCOMM*, 2013.

[63] M. Yu, L. Jose, and R. Miao. Software Defined Traffic Measurement with OpenSketch. In *NSDI*, 2013.