Switch Code Generation Using Program Synthesis

Xiangyu Gao[1], Taegyun Kim[1], Michael D. Wong[1], Divya Raghunathan[2], Aatish Kishan Varma[1], Pravein Govindan Kannan[3], Anirudh Sivaraman[1], Srinivas Narayana[4], Aarti Gupta[2]
[1]*New York University* [2]*Princeton University* [3]*National University of Singapore* [4]*Rutgers University*

ABSTRACT

Writing packet-processing programs for programmable switch pipelines is challenging because of their *all-or-nothing* nature: a program either runs at line rate if it can fit within pipeline resources, or does not run at all. It is the compiler's responsibility to fit programs into pipeline resources. However, switch compilers, which use rewrite rules to generate switch machine code, often reject programs because the rules fail to transform programs into a form that can be mapped to a pipeline's limited resources—even if a mapping actually exists.

This paper presents a compiler, Chipmunk, which formulates code generation as a program synthesis problem. Chipmunk uses a program synthesis engine, SKETCH, to transform high-level programs down to switch machine code. However, naively formulating code generation as program synthesis can lead to long compile times. Hence, we develop a new domain-specific synthesis technique, *slicing*, which reduces compile times by 1–387× and 51× on average.

Using a switch hardware simulator, we show that Chipmunk compiles many programs that a previous rule-based compiler, Domino, rejects. Chipmunk also produces machine code with fewer pipeline stages than Domino. A Chipmunk backend for the Tofino programmable switch shows that program synthesis can produce machine code for high-speed switches.

CCS CONCEPTS

• **Networks → Programmable networks**;

KEYWORDS

Programmable switches; program synthesis; code generation; slicing; packet processing pipelines

ACM Reference Format:
Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, Aarti Gupta. 2020. Switch Code Generation Using Program Synthesis. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, NY, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3387514.3405852

SIGCOMM '20, August 10–14, 2020, Virtual Event, NY, USA

ACM ISBN 978-1-4503-7955-7/20/08…$15.00
https://doi.org/10.1145/3387514.3405852

1 INTRODUCTION

There has been a recent flurry of research on programming languages and hardware designs for high-speed programmable switch pipelines [8, 15, 23, 38–40, 73]. Today, it is possible to specify packet processing for line-rate switches at a high level of abstraction using languages like P4, making it easy for researchers, network operators, and equipment vendors to start programming switches.

However, writing *optimized* programs for line-rate switches is much more challenging than sample programs and tutorials [20] might suggest. A realistic switch program [22, 54, 55] must fit within highly constrained switch resource budgets to run successfully. Examples of resources include pipeline stages, arithmetic logic units (ALUs), SRAM memory for control plane rules, and containers for packet headers. To make things worse, packet-processing pipelines have an *all-or-nothing* characteristic: programs that can be accommodated within the switch's resources run at the line rate of the switch pipeline; otherwise they cannot run at all. Unlike processors, there is no middle ground where complex programs can run with slightly degraded performance. This forces developers to grapple with low-level details of the hardware such as the configurations of the available ALUs, sequencing of stages, and the usage of the available stage memory (both SRAM and TCAM), to squeeze their programs into the pipeline's resources.

The difficulty of writing optimized programs can be addressed using compilers. Today's switch compilers [17, 69] are structured around rewrite rules [31] that operate on small program fragments at a time. These rules repeatedly transform the program into simpler forms until it can be easily mapped to machine code. However, rule-based compilers can spuriously *reject* many programs as they are unable to rewrite them to a form that can fit within the limited switch resources, even when there exist ways to fit those programs into the switch. §3 provides an example.

Motivated by these drawbacks of rule-based compilers and inspired by the success of program synthesis in other domains [43, 48, 62, 63, 67], a recent workshop paper [46] observed that we can leverage *program synthesis,* i.e., automatically generating program implementations that satisfy a specification, to produce fast packet-processing code that fits within resource limits. The workshop paper observed that synthesis can be used to transform a high-level program (e.g., in C, P4-16 [16], or Domino [69]) into low-level machine code (e.g., ALU opcodes in a switch pipeline) by treating the machine code as the program to be synthesized and the high-level program as the specification. The current paper builds on the vision in that workshop paper and makes two main research contributions in designing a switch compiler, Chipmunk.

Domain-specific synthesis techniques (§4). Synthesis is a combinatorial search problem over a space of implementations that may satisfy a specification. Hence, a synthesis-based compiler can take

much longer to generate code than a rule-based compiler. We developed a new technique, *slicing*, to speed up synthesis-based compilation for pipelines. In slicing, we decompose the synthesis problem for switch pipeline code generation into a collection of independently synthesizable sub-problems called slices. In each slice, Chipmunk synthesizes a sub-implementation that has the same behavior as the specification, but just on a *single* packet field or state variable from the specification. These sub-implementations can be directly "stacked" on top of each other to form the full pipelined implementation. Each slice presents a simpler synthesis problem—since the implementation must only respect the specification for one variable—and hence can be synthesized with fewer pipeline stages and ALUs than the original specification. The reduction in stages and ALUs leads to a much smaller search space and time for synthesis. Beyond slicing, we also adapted several techniques from program synthesis to the context of packet processing (§4.5).

Retargetable code generation using a pipeline description language (§5). We designed the Chipmunk compiler to target two backends, a switch pipeline simulator and Barefoot's Tofino switch [23], as well as subsets of their full capabilities. Our experience made it clear to us that it would be useful to share the same underlying program synthesis technology across several backend targets. That is, our compiler should be *retargetable*: it should be able to generate machine code for different switch pipelines with different instruction sets driven solely by a declarative specification of the hardware's capabilities [10, 41]. To enable this, we developed a declarative domain-specific language (DSL) to specify the capabilities of a pipeline's ALUs and the interconnect between them. We call this DSL the *pipeline description language*. Chipmunk takes a description of the hardware written in this language, automatically formulates a synthesis problem for an off-the-shelf synthesis engine, SKETCH [72], solves it, and translates the results of synthesis into backend-specific machine code. For the Tofino backend, which doesn't support direct programming in assembly language, we used compiler pragmas to gain the low-level control over hardware resources required for code generation (§6).

We evaluated Chipmunk on both the simulator and Tofino backends using 14 benchmarks drawn from a variety of sources [58, 59, 69]. Our primary findings are that:

(1) Chipmunk successfully compiles many programs that a rule-based compiler, Domino [69], rejects.
(2) Chipmunk produces machine code with fewer pipeline stages—a highly constrained switch resource—relative to Domino.
(3) Slicing speeds up synthesis by 1–387× (average: 51×).
(4) Although Chipmunk is slower than Domino, Chipmunk's compile times are within 5 minutes on 12 out of 14 benchmarks, and within 2 hours for the rest.
(5) Chipmunk generates Tofino machine code for 10 out of 14 benchmarks. We believe the other 4 benchmarks are beyond the capabilities of Tofino ALUs (§7.2).

We have open sourced Chipmunk along with instructions to replicate this paper's results at https://chipmunk-project.github.io/. This work does not raise any ethical issues.

2 BACKGROUND

Programming languages for packet processing. Several languages now exist for packet processing, e.g., P4-14 [27], P4-16 [16], POF [73], and Domino [69]. This paper uses Domino as the language in which the input program is specified by the programmer. Domino is well-suited to expressing packet processing with an algorithmic flavor, e.g., maintaining sketches for measurement or implementing the RCP [76] protocol. Figure 1 shows an example Domino program that samples every 11^{th} packet going through a pipeline. Domino provides transactional semantics: operations in a Domino program execute from start to finish atomically, as though packets are being processed by the target exactly one packet at a time. This frees the programmer from having to deal with concurrency issues, delegating that to the compiler instead. The same transactional semantics are also supported by P4-16's `@atomic` construct [21]. P4-16's `@atomic` construct was influenced by Domino [4]; hence, we expect to also be able to support P4-16 `@atomic` in the frontend.

Packet-processing pipelines. A programmable switch consists of a programmable parser to parse packet headers, one or more programmable match-action ingress pipelines to manipulate headers, a packet scheduler, and one or more programmable match-action egress pipelines for additional header manipulations. We focus on the pipelines because that is where packet manipulations primarily occur. We consider a pipeline architecture for packet-processing based on RMT [39] and Banzai [1], which extends RMT with stateful computation. This architecture is now commonly known as the Protocol Independent Switch Architecture (PISA) [24] and is seen in many high-speed programmable switches [7, 8, 14, 23].

In PISA, an incoming packet first enters the parser. After parsing, the parsed packet headers are deposited in a *packet header vector (PHV):* a vector of containers each of which stores a single header field (e.g., IP TTL). This PHV is passed through the ingress and egress pipelines. Each pipeline stage contains multiple match-action tables that operate concurrently on PHV containers. Each match-action table identifies the rule of interest for the current packet using the *match unit* (e.g., SSH packets can be matched using a rule specifying TCP port 22) and modifies the packet using the *action unit* (e.g., adding 1 to a packet field) tied to that rule. The pipeline can maintain a small amount of action-unit-local *state* to perform the action, e.g., maintain a count of all SSH packets.

The PISA pipeline is assumed to be *feed-forward:* packets can only flow from an earlier stage to a later one, but not in reverse. This means that computations in a later stage can depend on computations in earlier ones, but not vice versa. In particular, a piece of state stored in a pipeline stage can be read, modified, and written *only once* by a packet as it passes through the pipeline. Switches can recirculate packets back into the pipeline to enable backward flow, but recirculation greatly degrades packet-processing throughput and we do not consider it here.

We refer to the action units in packet-processing pipelines as **Arithmetic Logic Units (ALUs)**. In this paper, we only focus on the pipeline's ALUs (not the match units) because the ALUs are where per-packet computation occurs—and hence the target of code generation. We further assume that the ALUs execute on all packets going through the pipeline. It is straightforward to implement use

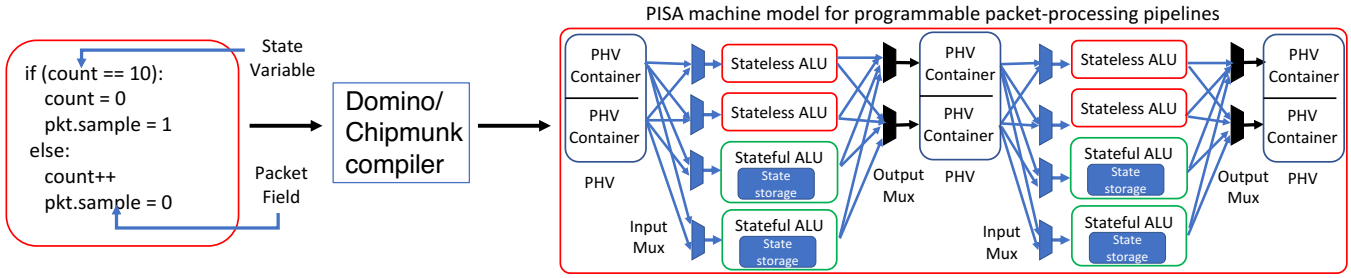

Figure 1: Program as a packet transaction in Domino along with a 2-by-2 pipelined grid (i.e., 2 stages, 2 stateful + 2 stateless ALUs per stage) showing the PISA machine model. Input muxes are used to determine which PHV container to use as an ALU operand. Output muxes are used to determine which ALU to use to update a container.

cases where the ALUs only execute on a subset of the packets because match rules can be added to the corresponding match-action tables to support such use cases. Hence, the entire pipeline can be abstracted out as a 2D grid of ALUs (Figure 1).

ALUs must process packets at line rate. Hence, an ALU should be able to process a new packet every clock cycle (~1 ns). ALUs can be *stateless,* i.e., operating only on PHV containers; or *stateful,* i.e., operating both on ALU-local state and PHV containers. For stateless ALUs, the ALU should be able to update a new PHV container every cycle. For stateful ALUs, the entire read-modify-write operation on the state that the ALU operates on must complete within a cycle. This guarantees state consistency even if packets in consecutive cycles access the same state. Each ALU has a set of input multiplexers (*muxes*), one per operand. These muxes are used to determine which PHV containers are used as ALU operands. Each ALU provides certain operations (e.g., addition) and may take immediate operands. Each PHV container is fed by an output mux to determine which stateful/stateless ALU's output updates it.

Compiling programs to pipelines. A compiler for a pipeline (e.g., Domino [69]) takes a packet-processing program, written in a high-level language, and turns it into low-level machine code representing pipeline configurations, e.g., ALU opcodes, allocation of packet fields to PHV containers, and configurations of muxes (Figure 1).

Compiling programs to pipelines is *all-or-nothing*: successfully compiled programs can run at the pipeline's line rate, but a program that is rejected by the compiler can't run at all. Programs can be rejected for two reasons. The first reason is violating resource limits: the machine code generated by the compiler might consume more resources (e.g., stages, ALUs, rule/table memory) than available. The second reason is violating computational limits: the compiler might be unable to find a way to map computations in the program to the hardware's ALUs, even with infinite ALUs.

This places a significant responsibility on the compiler, which should ideally be able to find *some* machine code corresponding to the given high-level program—provided the program is within the resource and computational limits of the pipeline: the program's computations belong to the finite space of computations possible using a single pass through the pipeline's ALUs without recirculation. As an example of a program that exceeds these limits, if the pipeline only supports increment operations on state (but no multiply), and the program requires an exponentially weighted moving average filter over queueing delays, it is impossible to run the program using the pipeline.

3 THE CASE FOR PROGRAM SYNTHESIS

Drawbacks of rule-based compilers. Compilers for packet-processing pipelines often reject programs spuriously: a semantically-equivalent version of the same program will be accepted by the same compiler. We have observed this problem with both commercial [19] and academic compilers [6].

We illustrate the problem of spurious program rejections in the context of the Domino compiler [69] using a simple example. While this example is simplified for illustration, we have observed similar spurious program rejections with more complex examples as well. Figure 2 shows two Domino programs written to target a PISA pipeline. The two programs are semantically equivalent, i.e., given the same input packets and the same initial state variables, they will both produce the same trace of output packets and state values at run time. However, the Domino compiler exhibits a butterfly effect or a false positive compilation result: it successfully compiles the first program, but rejects the second one even though both of them have the same semantics.

To understand why, we look at the intermediate representation constructed by the Domino compiler. This representation is akin to a directed acyclic graph (DAG) of computations, but additionally groups together stateful computations that must finish atomically within one clock cycle. Figure 3 shows the DAGs for both versions of the program. The shaded nodes show stateful computations, while the unshaded nodes show stateless computations [69]. The DAGs differ in the complexity of stateful computations: the circled stateful computation of version 2 is more involved, and cannot be executed atomically by the available stateful ALU, while the stateful computations of version 1 can be. This is because the ALU considered here (the Read/Write ALU [69]) can only handle the atomic update of one state variable, but not the atomic update of two variables as required by version 2. Essentially, the compiler is running into a computational limit.

This difference in DAGs for 2 semantically equivalent programs occurs because Domino's compiler passes are program rewrite rules that repeatedly transform the program in an attempt to find a simpler version of it (i.e., the DAG representation) that readily maps to switch ALUs. However, these rewrite rules are *incomplete*: they do not find a semantically-equivalent version of the DAG that *can* map all nodes to the available ALU type (i.e., version 1's DAG) when given the version 2 program. In effect, the compiler's rules do not fully explore the search space of machine code programs that could implement the high-level program.

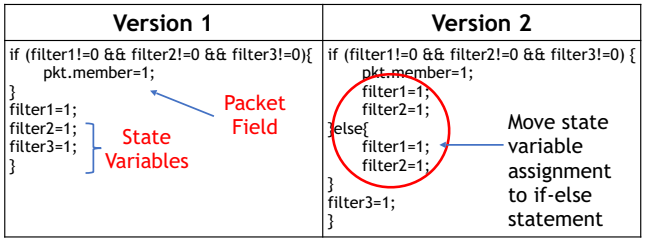

Figure 2: Two semantically equivalent versions of a Domino program [69]. Version 1 compiles; version 2 fails to.

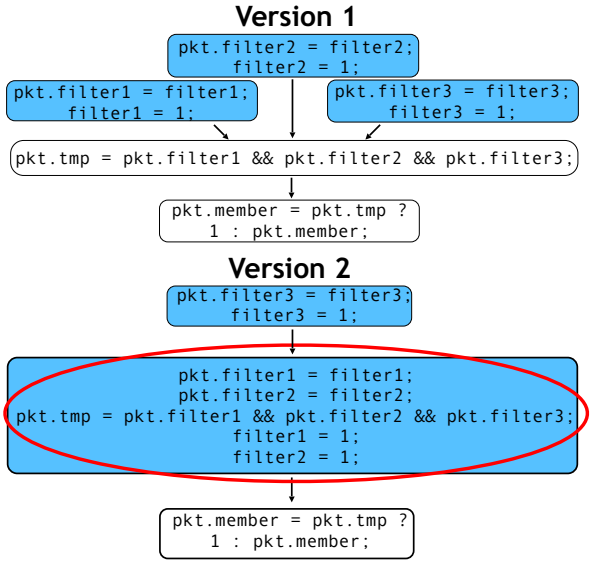

Figure 3: Simplified Domino DAGs for both versions. Stateless computations are unshaded; stateful are shaded. The circled node shows the large amount of atomic stateful computation in version 2.

Although the specific situation in Figure 3 could be fixed by a compiler developer through the addition of another rewrite rule, similar situations will continue to emerge in the future. In fact, rule-based compilers and programs can be thought of as two sides of an arms race, with compilers getting more complex over time to incorporate more rewrite rules that simplify more complex program patterns. By contrast, as we next discuss, by exhaustively searching the space of machine code, program synthesis has the potential to provide a simpler and more future-proof compiler design.

The case for program synthesis. To address the incompleteness of rule-based compilation, we first observe that the search for machine code for a given high-level program can be thought of as a combinatorial search problem called *program synthesis*. In program synthesis, we are looking for a program *implementation* that is semantically-equivalent to a program *specification:* on all legal inputs, the outputs of the specification and implementation agree. In our context, implementations are drawn from the set of all machine code programs that can be implemented on a 2D ALU grid of bounded size. The specification is the high-level language program that must be compiled. The benefit of the program synthesis approach is that synthesis engines can search a family of implementations using efficient algorithms [53, 72] (see Appendix B). If

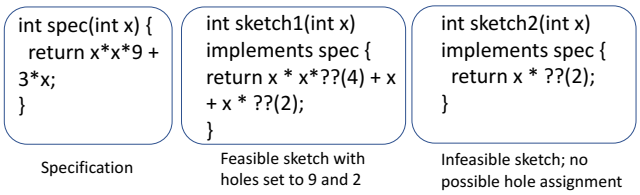

Figure 4: Synthesis in SKETCH. `??(b)` is a hole with a value in $[0, 2^b - 1]$

at least some implementations meet the specification, the synthesis engine is far more likely to find it than a rule-based compiler. Hence, synthesis greatly reduces butterfly effects similar to Figure 2.

However, without domain-specific techniques to prune the search space of implementations, synthesis can quickly become intractable in practice. Thus our goal is to leverage synthesis for compilation because of its ability to search the machine code space, while keeping compilation times reasonable using domain-specific techniques to prune the search space. In this paper, we aim for compilation times of an hour, although most of our programs compile in a few minutes (§7.1). While an hour seems excessive, we believe that it is better than the alternative of having developers tweak programs manually, which requires low-level hardware expertise, is error prone, and may take even longer. Further, there must be some price to pay for higher quality code; other compiler techniques (e.g., link-time optimization [11]) exhibit a similar tradeoff.

4 CODE GENERATION AS SYNTHESIS

Chipmunk takes as inputs: (1) a packet transaction and (2) a specification of the pipeline's capabilities. It produces machine code for that pipeline, which consumes a small number of resources (ALUs and pipeline stages). We first describe how we produce machine code given a fixed pipeline depth (stages) and width (ALUs per stage) (§4.1–§4.5). We then show how to use this to find code with small depth and width (§4.6).

4.1 Code Generation Using SKETCH

We briefly describe SKETCH, the program synthesis engine we use in this paper. SKETCH takes two inputs: a specification and a *sketch*, a partial program with *holes* representing unknown values in a finite range of integers. Sketches constrain the synthesis search space by only considering for synthesis those programs in which each sketch hole is filled with an integer belonging to the hole's range. Sketches encode human insight into the shape of synthesized programs. SKETCH then fills in all holes with integers so that the completed sketch meets the specification, assuming it is possible to meet the specification (Figure 4), or says that it is impossible to do so. Appendix B describes SKETCH's internals. To build a code generator using SKETCH, we need to determine an appropriate set of holes, the sketch, and the specification.

Holes. The code generator needs to ultimately choose the right value of low-level programmable hardware knobs (Table 1), e.g., which inputs to wire to each ALU (the input muxes), what operation each performs (ALU opcodes), which PHV container is used for the outputs (output muxes), which packet field is allocated to each

Programmable knob / SKETCH hole	Hole bit width
ALU opcode (e.g., +,-, *, /)	$log_2(\|opcodes\|)$
Input mux control: which PHV feeds an ALU	$log_2(\|PHVs\|)$
Output mux control: which ALU feeds a PHV	$log_2(\|ALUs\|)$
Indicator bit to track if a field is allocated to a PHV container	$\|fields\| * \|PHV\|$
Indicator bit to track if a state variable is allocated to a stateful ALU	$\|state\_vars\| * \|num\_stages\| * \|alus\_per\_stage\|$
Immediates/constants in instructions	constant bit width

Table 1: Programmable knobs and their hole bit width

container, etc. Our goal is to get SKETCH to make these choices. Hence, we encode these programmable knobs as SKETCH holes.

Sketch. We use a sketch to represent the space of valid pipeline configurations to be considered for implementing packet-processing programs. This sketch captures all the possible computations supported by the pipeline as a function of the holes. First, the sketch models the effect of each stateful and stateless ALU on PHV containers and ALU-local state, as a function of holes corresponding to the ALU opcode, input mux controls, immediate operands, and local state variable(s). Second, the sketch models the pipeline: the flow of PHVs from stage to stage, as a function of holes corresponding to the input/output mux controls and field to PHV allocations. In effect, our pipeline sketch is a programmatic representation of the 2D grid of ALUs in Figure 1; see Appendix A for an example.

4.2 Packet Transactions as Specifications

We now show how we encode specifications. We use the terms packet (state) vector to refer to a vector, every dimension of which corresponds to a single packet field (state variable) from the specification, i.e., the packet transaction in Figure 1. We use the term state+packet vector to represent the concatenation of the state and packet vectors. Our goal is to synthesize a pipeline implementation that respects the packet transaction specification on an arbitrary input *packet trace*: on any sequence of packet vectors and arbitrary initial state vector, the output sequence of packet vectors and final state vector must be the same for the specification and the implementation. We call this problem *trace-based synthesis*.

However, trace-based synthesis appears daunting since packet traces can be infinitely long, while SKETCH is designed to work with finite inputs. But we can reduce trace-based synthesis to a simpler finite problem we call *packet-based synthesis,* where our goal is to synthesize a pipeline implementation such that on any arbitrary *single* packet vector and arbitrary *single* previous state vector, the updated state+packet vector after processing that packet must be the same for the specification and the implementation.

To perform this reduction, we need to establish that a solution to packet-based synthesis is also a solution to trace-based synthesis, which follows by induction on the length of the packet sequence. We note that packet-based synthesis is a stronger requirement than trace-based synthesis because it requires agreement on any possible previous state vector, even if such a state vector might never occur in a state sequence starting with an arbitrary initial state vector. This reduction is also similar to a classic technique used in hardware verification [49], where a sequential equivalence-checking problem of matching a sequence of outputs between a specification and an implementation is reduced to a combinational equivalence-checking problem, which requires both the outputs and the states to match at every time step. This reduction is practically significant because synthesis tools can deal with the finite input space of a single state vector and a single packet vector.

We convert the packet-processing program written as a packet transaction in Domino into a SKETCH specification that takes as input a state+packet vector and outputs an updated state+packet vector. To convert the Domino program into a SKETCH specification, we added a pass to the Domino compiler [6]. This is relatively straightforward because both Domino and SKETCH have a very similar C-like syntax. Replacing Domino with P4-16 @atomic as the input language for Chipmunk would similarly need a p4c [17] compiler pass.

4.3 The Slicing Technique

While packet-based synthesis is simpler than trace-based synthesis, it is still too slow on several benchmarks (Table 3). To speed up synthesis further, we developed a technique called *slicing*. We start with a simplified version of slicing that only works for stateless and deterministic packet transactions. We then refine it to handle state and randomness.

To motivate slicing, observe that in packet-based synthesis, we require the specification and implementation to agree on the *entire* updated state+packet vector. Instead of requiring agreement on the entire vector, we factorize the requirement into a collection of simpler requirements or *slices*. Each slice is a simpler synthesis problem that synthesizes a pipelined implementation such that the implementation and the specification agree only on a single vector dimension of the updated state+packet vector (e.g., only `pkt.sample` or only `count` in Figure 1). Once we have successfully synthesized each slice, we merge together the slice implementations by stacking the resulting pipeline implementations on top of each other to form the final hardware implementation.

Slicing has two main advantages over packet-based synthesis. First, each slice can be synthesized in parallel because each slice implementation runs on an independent sub-grid of the pipeline with no overlap between the sub-grids. Second, because each slice's implementation only satisfies a subset of the specification, i.e., one dimension instead of all dimensions of the state+packet vector, it can be synthesized using a sub-grid with smaller size than would be needed for the original specification. A smaller grid requires fewer holes to be synthesized (Table 1), reducing the synthesis time (Table 3) for any one slice relative to the original specification.

Handling state modifications. When the packet transaction modifies state, the above slicing algorithm is no longer correct. To see why, consider the specification "`count++; pkt.f = count;`". This specification sets a packet field, `pkt.f`, to the most recent value of a switch counter, `count`, which increments on every packet. This problem will be factorized into two slices: one each for `pkt.f` and `count`. In the first slice, we require an implementation that sets `pkt.f` to the previous `count` + 1. In the second slice, we require an implementation that sets `count` to the previous `count` + 1.

Note, however, the first slice *does not* require `count` itself to be updated to its correct final value. This means that the first slice can produce implementations that simply set `pkt.f` to the previous

count + 1, without actually ever updating count! The result is that when a trace of multiple packets has passed through the pipeline, pkt.f will always be set to 1, i.e., the initial value of count (0) + 1. This is because count is never updated in the first slice. However, a correct implementation should set pkt.f to 1, 2, 3, ..., over successive packets.

More generally, for any slice S, suppose there is a state variable i such that S's packet field or state variable depends on i (e.g., pkt.f depends on count). Then we say i influences S. Then as part of the slice S, we should also require that i be updated in the implementation to match up with how the specification updates i. Otherwise, i may not be updated and S's packet field or state variable cannot make use of the updated i for its own computation.

Hence, for each slice, we additionally assert that the specification and implementation also agree on any state variables that influence that slice's packet field or state variable. §4.4 proves that this additional assert produces the correct behavior of all slices in the presence of state—as long as state modifications complete within a clock cycle as described in §2. In our example above, this additional assert ensures that count is also set to previous count + 1 in the first slice, in addition to pkt.f being set to previous count + 1.

Non-determinism. The use of randomness (e.g., hashing) within a packet transaction can result in the merged implementation (after slicing) differing in behavior from the packet transaction. This is because when a random-number-generating computation is duplicated in two or more slices, we cannot guarantee that the random numbers generated in each slice will be identical. This can be fixed by either seeding the random number generators in all slices to the same value from the control plane or precomputing such random numbers and storing them in packet fields before executing the packet transaction. We follow the second approach in this paper.

The cost of slicing. Slicing does not exploit opportunities to share computations between different slices. For instance, in our example, the update to count is duplicated across the two slices. Thus slicing requires additional ALUs and PHV containers for these redundant computations. In our evaluations, we find that programs do not use too many containers/ALUs in the first place (< 10) and using slicing adds at most 3 containers and ALUs per stage (Table 3). For context, RMT has about ~200 ALUs/PHV containers per stage [39]. This is a reasonable trade-off for faster compilation.

4.4 Correctness of slicing

We now prove the correctness of the slicing technique. We first introduce some notation before proving correctness.

Definition 4.1. **Spec** denotes the packet transaction specification for a single packet. It is a function with inputs comprising a packet vector $\vec{p}$ and a state vector $\vec{s}$; and with outputs comprising an *updated* packet vector and an *updated* state vector.

Individual fields in the output vector of the function can be accessed by member name or member index. For example, $Spec(\vec{p},\vec{s}).m$ denotes the member m of the output of the function **Spec** on inputs $\vec{p}$ and $\vec{s}$ and $Spec(\vec{p},\vec{s})[i]$ denotes the i^{th} member in the output vector of the function.

Definition 4.2. **Spec*** denotes the packet transaction specification for a sequence of packets. It is a function with inputs comprising a sequence of n packets $\{\vec{p_n}\}$, and an initial state vector $\vec{s}_0$; and with outputs comprising a final packet vector and a final state vector after the n^{th} packet is processed by the **Spec** function. It is defined inductively as follows:

$$\text{Spec}^*(\{\vec{p}_1\},\vec{s}_0) = \text{Spec}(\vec{p}_1,\vec{s}_0)$$
$$\text{Spec}^*(\{\vec{p}_n\},\vec{s}_0) = \text{Spec}(\vec{p}_n,\text{Spec}^*(\{\vec{p}_{n-1}\},\vec{s}_0).\vec{s})$$

where $\{\vec{p}_i\}$ denotes the first i packets of the input packet sequence.

Note that in the inductive step, Spec* applies the Spec function to the n^{th} packet in the sequence and the output state resulting from applying Spec* inductively.

Definition 4.3. **Impl** is a function with inputs comprising a packet vector $\vec{p}$ and a state vector $\vec{s}$, and with outputs comprising an *updated* packet vector and an *updated* state vector, as reflected by the functionality of the programmable switch.

Definition 4.4. **Impl*** is a function with inputs comprising a sequence of n packets $\{\vec{p_n}\}$ and an initial state vector $\vec{s}_0$, and with outputs comprising a final packet vector and a final state vector after the n^{th} packet is processed by the programmable switch. It is defined inductively as follows:

$$\text{Impl}^*(\{\vec{p}_1\},\vec{s}_0) = \text{Impl}(\vec{p}_1,\vec{s}_0)$$
$$\text{Impl}^*(\{\vec{p}_n\},\vec{s}_0) = \text{Impl}(\vec{p}_n,\text{Impl}^*(\{\vec{p}_{n-1}\},\vec{s}_0).\vec{s})$$

Definition 4.5. **Influence:** Consider the set $S = \{\vec{p},\vec{s}\}$. We say that $u \in S$ influences $v \in S$, if there exist c_1 and c_2 such that $Spec(\{\vec{p},\vec{s}\}/\{u\}, u = c_1).v \neq Spec(\{\vec{p},\vec{s}\}/\{u\}, u = c_2).v$ (or) if there exists $inter \in S$ such that u influences $inter$ and $inter$ influences v.

Let **I(i)** denote a vector of *indices* of state variables that influence the output of the i^{th} packet field, and let **NI(i)** denote a vector of *indices* of state variables that do *not* influence the output of the i^{th} packet field.

Definition 4.6. **Slicing-based synthesis**

$$\forall i\, \forall \vec{p}\, \forall \vec{s},\ \ \text{Spec}(\vec{p},\vec{s})[i,I(i)] = \text{Impl}_i(\vec{p},\vec{s})[i,I(i)]$$

where i represents the i^{th} packet field and $I(i)$ is as defined above. We refer to Impl_i as a slicing-based implementation function for the i^{th} packet field. The Impl_i are "stacked" on top of each other to form Impl.

Definition 4.7. **Trace-based synthesis**

$$\forall i\, \forall \{\vec{p}_n\}\, \forall \vec{s_0},\ \ \text{Spec}^*(\{\vec{p}_n\},\vec{s_0})[i,I(i)] = \text{Impl}_i^*(\{\vec{p}_n\},\vec{s_0})[i,I(i)]$$

Theorem 4.8. ***Slicing-based synthesis*** $\implies$ ***Trace-based synthesis.***

Proof. We will prove this for any i (i.e., for all i^{th} packet fields), and by induction on n, the number of packets in the sequence of packets processed by the switch.
For the base step, $n = 1$, we have:

$$\text{Spec}^*(\{\vec{p}_1\},\vec{s_0})[i,I(i)] = \text{Spec}(\vec{p}_1,\vec{s_0})[i,I(i)] \quad \ldots \text{(def of Spec}^*\text{)}$$
$$\text{Impl}_i^*(\{\vec{p}_1\},\vec{s_0})[i,I(i)] = \text{Impl}_i(\vec{p}_1,\vec{s_0})[i,I(i)] \quad \ldots \text{(def of Impl}_i^*\text{)}$$

By the definition of slicing-based synthesis,

$$\text{Spec}^*(\{\vec{p}_1\}, \vec{s}_0)[i, I(i)] = \text{Impl}_i^*(\{\vec{p}_1\}, \vec{s}_0)[i, I(i)]$$

Induction hypothesis: Assume that the claim holds for $n = k$:

$$\forall i \, \forall \{\vec{p}_k\} \forall \vec{s}_0, \quad \text{Spec}^*(\{\vec{p}_k\}, \vec{s}_0)[i, I(i)] = \text{Impl}_i^*(\{\vec{p}_k\}, \vec{s}_0)[i, I(i)] \quad (1)$$

Induction step: Now we prove the claim for $n = k + 1$.

$\text{Spec}^*(\{\vec{p}_{k+1}\}, \vec{s}_0)[i, I(i)]$

$= \text{Spec}(\vec{p}_{k+1}, \text{Spec}^*(\{\vec{p}_k\}, \vec{s}_0).\vec{s})[i, I(i)]$ . . . (def of Spec^*)

$= \text{Impl}_i(\vec{p}_{k+1}, \text{Spec}^*(\{\vec{p}_k\}, \vec{s}_0).\vec{s})[i, I(i)]$

. . . (def of **Slicing-based synthesis**)

$= \text{Impl}_i(\vec{p}_{k+1}, \text{Spec}^*(\{\vec{p}_k\}, \vec{s}_0).\vec{s}[I(i)], \text{Spec}^*(\{\vec{p}_k\}, \vec{s}_0).\vec{s}[NI(i)])[i, I(i)]$

. . . (split state variables into influence and non-influence parts)

$= \text{Impl}_i(\vec{p}_{k+1}, \text{Impl}_i^*(\{\vec{p}_k\}, \vec{s}_0).\vec{s}[I(i)], \text{Spec}^*(\{\vec{p}_k\}, \vec{s}_0).\vec{s}[NI(i)])[i, I(i)]$

. . . (by equation (1))

$= \text{Impl}_i(\vec{p}_{k+1}, \text{Impl}_i^*(\{\vec{p}_k\}, \vec{s}_0).\vec{s}[I(i)], \text{Impl}_i^*(\{\vec{p}_k\}, \vec{s}_0).\vec{s}[NI(i)])[i, I(i)]$

. . . (the variables in NI(i) cannot influence the variables in I(i) or i, so we can set them to any value without affecting the final output)

$= \text{Impl}_i(\vec{p}_{k+1}, \text{Impl}_i^*(\{\vec{p}_k\}, \vec{s}_0).\vec{s})[i, I(i)]$

. . . (merge state variables with indices in I(i) and NI(i))

$= \text{Impl}_i^*(\{\vec{p}_{k+1}\}, \vec{s}_0)[i, I(i)]$ . . . (def of Impl_i^*)

□

4.5 Other Optimizations

Scaling up synthesis to larger input ranges. SKETCH is designed to synthesize implementations that meet the specification on a small range of inputs for each input variable (e.g., all x values between 0 and 31 in Figure 4). Scaling SKETCH to synthesize implementations that meet the specification on larger ranges of inputs (e.g., all 32-bit integers) is challenging. This is because the SAT solver within SKETCH isn't optimized for rapid verification on large input ranges for two reasons. First, SKETCH's unary encoding is particularly inefficient in its use of memory [26, 70]. Second, a SAT solver does not contain many of the theories available in a full-blown SMT solver such as Z3 [29]. Z3 is better suited for verifying spec-implementation equivalence over large input ranges because it contains specialized decision procedures for integers and bit vectors that scale to larger input ranges.

To address this problem, we decouple the input ranges for synthesis and verification, adapting an idea proposed in prior work [57]. We use SKETCH to synthesize a solution for a small input bit width of 2, i.e., all packet fields and state variables are assumed to take values between 0 and $2^2 - 1$. Then, we take the resulting completed sketch (i.e., with all holes filled in with integers) and use Z3 to verify it on a larger input bit width (currently all 10-bit integers). If Z3 finds a counterexample, we rerun SKETCH again, using asserts to rule out the previously obtained hole values (hole elimination) or using the newly found counterexample to create an additional concrete input on which the specification and the sketch must agree (counterexample assertion). Between hole elimination and counterexample assertion, we find that counterexample assertion performs much better because it can rule out not only the hole assignment that led to the current Z3 verification counterexample, but also all other hole assignments that could have potentially led to that counterexample.

Constant synthesis. One challenge for program synthesis tools is synthesizing holes with large bit ranges. In our context, these are immediate operands for ALUs, which can be up to 32 bits wide. The difficult of synthesizing such large holes has been documented before [30, 65]. With SKETCH, we also observed a steep increase in synthesis time when using holes with bit widths exceeding 15 bits [26]. To synthesize large holes for immediate operands, we developed an algorithm based on using a dynamic pool of constants from which SKETCH picks immediate operands. Our algorithm initializes this pool to all constants that appear in the input packet transaction. In addition, we augment the pool with all numbers in a small range of integers (0–3). If Z3 verification fails, we *update* the pool with packet field and state variable values appearing in the counterexample produced by Z3. This algorithm is very efficient but incomplete—the main source of incompleteness in Chipmunk—because it only samples a few integer values; hence, it can fail to find a large hole when one actually exists. However, empirically, we find that it performs well because it adapts to the supplied program and learns from counterexamples.

Canonicalization. SKETCH needs to allocate packet fields to PHV containers and state variables to stateful ALUs, while respecting the hardware constraint that no PHV container or stateful ALU is oversubscribed. There are many feasible allocations that satisfy this constraint. However, in a symmetric grid, where the same type of ALU is tiled out over the entire grid and each ALU can use any PHV container as an operand, many of these allocations are equivalent to each other. We can use this symmetry to speed up synthesis.

In a symmetric grid, for PHV allocation, we rename packet fields so that they have canonical names $f_1, f_2, \ldots$ following [35]. Then, we allocate f_1 to container 1, f_2 to container 2, and so on. This allocation is as good as any other because in a symmetric grid all containers are equivalent in their abilities (i.e., an ALU can use any of these containers as an operand and can output to any of these containers). In other words, any allocation can be canonicalized by renaming variables.

For state variables, the situation is similar, but with one important difference. It doesn't matter which stateful ALU within a stage a state variable is allocated to due to symmetry. However, it *does* matter which stage's ALU a state variable is allocated to. This is because of dependencies within the feed-forward pipeline: an update to a state variable in a later stage can depend on the value of a state variable in an earlier stage, but the reverse is disallowed in a feed-forward pipeline. Thus state allocation exhibits symmetry within ALUs in one stage, but not across ALUs of different stages. Hence, we still use SKETCH to determine which stage a state variable should go into, but assign the state variable to a canonical stateful ALU within that stage.

4.6 Reducing Grid Size by Parallel Search

So far, we have focused on code generation for an ALU grid of a certain depth and width. To reduce resource usage, we need to find a small grid to implement each slice of the packet transaction. To

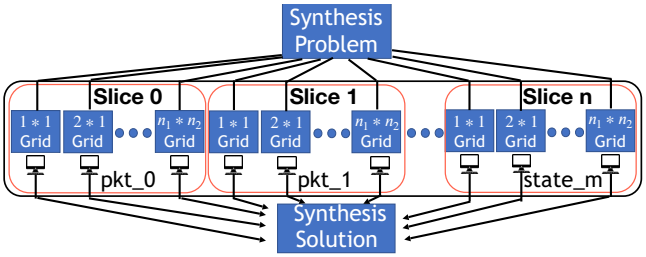

Figure 5: Parallel search to reduce grid size

do so, we independently solve a synthesis problem for each combination of slice and grid size in parallel (Figure 5). For a packet transaction to be successfully synthesized, all its slices must be successfully synthesized at some grid size (which may be different for each slice). Thus, if a transaction can be successfully synthesized, the time to successfully synthesize that transaction is the maximum of the times to successfully synthesize each of its slices. For each slice, the time to successfully synthesize that slice is the minimum of the times to successfully synthesize that slice across all grid sizes searched. We set an upper bound on the grid size and a timeout on the synthesis time for any one of the parallel synthesis problems. For each parallel synthesis problem, we internally use SKETCH's parallel mode [53]; thus, there are two levels of parallelism. Our parallel search strategy over grid sizes does not guarantee the smallest possible grid size for a problem, but builds on our observation that smaller grid sizes generally lead to faster synthesis times, if the slice can actually fit into the smaller grid. We have not currently implemented a full system to run synthesis problems in parallel and emulate it using sequential execution in our evaluations. However, we believe it will be straightforward to implement such a system given the embarrassingly parallel nature of our search.

5 RETARGETABLE CODE GENERATION

The techniques in the last section can generate hole-value assignments for a packet-processing program written in a high-level language, given a sketch of the pipeline. Two problems remain: (1) a sketch must be developed for the pipeline; and (2) the hole-value assignments must be mapped to a format that the hardware/backend understands.

Unfortunately, we found that developing a sketch of the pipeline manually is error-prone for several reasons (§5.1). Hence, we developed a *pipeline description language*, a declarative specification of a pipeline ALU's compute capabilities and the interconnection between these ALUs (§5.1). We designed a *pipeline sketch generator* (§5.2) that takes specifications in this language and automatically produces a sketch of the pipeline. Thus, this pipeline description language enables *retargetable code generation* [10, 41]: Chipmunk can generate code for a number of distinct packet-processing pipelines, given a description of each pipeline.

Pipeline descriptions were also directly useful in targeting the two backends that we support, a simulator for the Banzai machine model [1] and the Tofino ASIC [23]. In particular, we were able to use declarative pipeline specifications to automatically generate executable Banzai *behavioral models* (§5.3), which were helpful in debugging Chipmunk itself. We also leveraged the pipeline specification language to produce a Tofino-specific code generator that

$l \in$ literals $v \in$ variables bin_op $\in$ binary ops $un\_op \in$ unary ops

$t \in$ ALU type declaration	::=	`stateful` | `stateless`
$d \in$ packet field declarations	::=	list of variables v
$h \in$ hole declarations	::=	list of variables v
$sv \in$ state variable declarations	::=	list of variables v
$e \in$ expressions	::=	$l \mid v \mid e\ bin\_op\ e \mid un\_op\ e$
		| `Mux`$(e, \ldots)$
$s \in$ statements	::=	$e = e \mid s\ ;\ s \mid$ `return` (e)
		| `if` $(e)\ \{s\} \mid$ `if` $(e)\ \{s\}$ `else` $\{s\}$
$p \in$ alu specification	::=	$t; d; h; sv; s$

Figure 6: ALU DSL. Mux is an input/output mux.

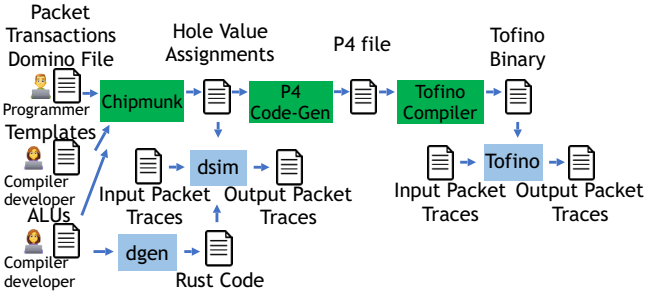

Figure 7: Workflow of Chipmunk

"lifts" the low-level hole-value assignments from Chipmunk to a surface language (i.e., P4-14) accepted by Tofino's compiler (§5.4).

5.1 Pipeline Description Language

Writing a pipeline sketch manually is hard for three reasons.

(1) **Large sketches:** Even for modest grid sizes (e.g., a 3-by-3 pipelined grid), the total number of hole bits and the number of lines in the sketch file often exceeds a few hundred (see Appendix A for an example sketch).

(2) **Different ALU capabilities on different targets:** There are significant differences in the capabilities of ALUs in the different pipelines we were experimenting with: a simulator of a switch pipeline (Banzai [69]), the Tofino switch [23], and subsets of the ALU functionalities provided by either. We found that constructing a pipeline sketch manually for each of these three different cases—and each ALU within each case—was highly error-prone.

(3) **Diverse grid interconnects among ALUs:** The specific connectivity among ALUs and PHVs differs from one pipeline to another. For example, Banzai provides all-to-all connectivity between all PHVs and ALUs: an ALU operand can come from any PHV. However, for wiring efficiency, some switching chips only provide all-to-all connectivity within clusters of PHVs and ALUs: an ALU operand can only come from a PHV in the same cluster as that ALU.

For these reasons, we developed two domain-specfic languages (DSLs) to write switch pipeline specifications, one each for (1) *computation* (e.g., opcodes of an ALU) and (2) *communication* (e.g., the interconnection between these ALUs to create a pipeline through input and output muxes). Thus, the computation DSL specifies *local* intra-ALU features of a switch pipeline, while the communication DSL specifies *global* inter-ALU features of a switch pipeline. We expect these specifications to be written once at switch design time by the backend compiler developer, not repeatedly by the programmer who writes Domino programs (Figure 7).

DSL to describe ALU computation. We developed an ALU DSL (Figure 6) to specify the computational capabilities of a single switch ALU, i.e., the programmable knobs available in the ALU (Table 1). Figure 6 shows the grammar of the ALU DSL. The DSL allows a developer to specify the number of ALU operands, where each operand comes from (i.e., packet field, state, immediate operand), the ALU's operation over its operands in the form of simplified C-like code, and what value(s) the ALU must return. Our DSL is expressive in its ability to express diverse target ALUs. It can express all the stateful and stateless instructions proposed as Banzai atoms [69], the stateless and stateful ALUs in the Tofino ASIC [23], as well as subsets of the functionality of each of these ALUs.

DSL to describe ALU interconnect. We specify the interconnections between ALUs in a grid using a *grid template* written in the Jinja2 template language [9]. A grid template is a skeleton of a sketch for a pipeline grid, representing the scaffolding or glue required to connect together ALUs, such as wires and muxes connecting PHVs to ALU inputs, and ALU outputs to PHVs (Figure 1). The grid template has placeholders to hold SKETCH code for ALUs (generated from the ALU DSL) and can repeat ALUs for a given width and depth.

5.2 Pipeline Sketch Generation

Chipmunk's pipeline sketch generator takes an ALU DSL program and a grid template as input and generates a sketch (e.g., Appendix A) corresponding to the ALU, repeating ALU code as necessary to fill out the 2D grid specified in the grid template. Chipmunk then feeds the generated sketch to SKETCH, which returns a value for each hole or says that the sketch is infeasible. If it is infeasible, we return a compile error. If it is feasible, we use the hole-value assignments to program the backends (§5.3, §5.4, and §6).

5.3 Producing Behavioral Models

A declarative pipeline description language enables the *automatic generation of pipeline behavioral models* for the Banzai machine model from pipeline DSL specifications, akin to P4-bmv2 [28]. We designed a behavioral-model-generator, `dgen` [78], which takes as input a switch pipeline specification in our DSLs (§5.1) and generates Rust code that simulates the action of that pipeline on packets. The Rust code when built produces an executable version of the pipeline `dsim` [78], which can consume and output packets, manipulating both the packets and internal pipeline state, serving as a behavioral model for the pipeline. Thus, `dsim` allows us to observe the input-output behavior of machine code (e.g., ALU opcodes, mux settings, etc.) produced by Chipmunk.

We used `dsim` to fuzz-test Chipmunk using random test packets for 100+ packet transactions drawn from our programs (§7), i.e., test whether Chipmunk generates correct pipelined machine code from packet transactions. We did this in three steps. First, we created random packet vectors as test inputs. Second, we directly executed the packet transaction on these test packets to record its input-output behavior without pipelining. We performed direct execution by writing each packet transaction as a Rust function and running the Rust function repeatedly on the test packets. Third, we compare the behavior from direct execution with the behavior `dsim` produces on the hole-value assignments generated by Chipmunk for the same packet transaction. If the two behaviors are different, it points to a bug in Chipmunk's code generation. Our fuzz testing has not yet revealed any bugs.

5.4 Lifting to Switch Surface Languages

Leveraging a DSL for expressing ALUs also enabled us to translate the outputs from Chipmunk's synthesis into an input language supported by the Tofino switch compiler [19], P4-14. We did this in two steps. First, we developed a P4-14 template program in Jinja2 [9], with placeholders for P4 registers and tables, which have a one-to-one correspondence with the 2D grid of ALUs from the pipeline sketch. For instance, each stateless ALU corresponds to a P4-14 primitive action, each stateful ALU corresponds to a P4 extern [18], and each state variable corresponds to a P4-14 register.

Second, we filled in the placeholders in the P4-14 template by translating the low-level integer-valued hole-value assignments outputted by synthesis into higher level P4-14 code accepted by the Tofino compiler. In particular, we translated holes from stateless ALUs (opcodes, operands) and stateful ALUs (opcodes, operands, and choice of output PHVs) into P4-14 code by leveraging a traversal of the abstract syntax tree (AST) of the ALU expressed in our DSL. For instance, let's say the ALU DSL file contains a function that takes three parameters: two operands (A and B) and an opcode. The hole-value assignments provide the value of the opcode, say 3, which stands for the + operation. Then, we can traverse the AST corresponding to the function and simplify the function to A+B, by treating the opcode as a constant (3) and applying constant propagation on the AST.

6 EXPERIENCES WITH TOFINO

After filling in the placeholders, the P4 program is given to the Tofino compiler to generate a Tofino binary. However, this can sometimes result in an incorrect implementation due to a subtle interplay between P4's sequential semantics and the Tofino hardware's parallel operation. To see why, consider a "swap" packet transaction that swaps the value of two packet fields (top and bottom) without using any temporary fields. This transaction can be implemented in Tofino using a single pipeline stage with 2 ALUs (top and bottom). The top stateless ALU transfers the bottom PHV to the top PHV and the bottom stateless ALU does the reverse, using an add opcode in each ALU with 0 as one operand. Chipmunk can also generate hole-value assignments in our behavioral model for this one-stage implementation of the swap transaction.

Now, how do we realize this swap in P4? We can create two P4 tables, one each for the top and bottom ALU, and use P4-14 `apply` statements to execute each table's ALU on incoming packets. However, the `apply` statement has sequential semantics. Because each table reads a field (top/bottom) that the other writes (bottom/top), sequential semantics will force the Tofino compiler to infer a read-after-write dependency between the two tables. Hence, the Tofino compiler will place the tables in two consecutive stages, not one. This is not just wasteful in stages, it is incorrect relative to what we want: it results in both top and bottom taking the same value instead of swapping values. Here, the Tofino compiler is correctly respecting P4-14's sequential semantics for `apply` statements, but the behavior is different from what Chipmunk expects from parallel

execution of the ALUs. In other words, it is hard to express the intra-stage parallelism required for operations like swap directly in P4, despite the hardware supporting it.[1]

To address the gap in the abstraction levels between Chipmunk's needs and P4-14, we use a compiler pragma to instruct the Tofino P4 compiler to ignore all table dependencies that it finds on its own. We instead enforce all dependencies ourselves. Chipmunk already handles dependencies because dependencies need to be respected to generate machine code that agrees with the specification. To enforce dependencies that Chipmunk finds, we use a second compiler pragma. This pragma instructs the Tofino compiler to place a table containing a stateful/stateless ALU in the same stage that the ALU belonged to in Chipmunk's output.

Reflections. Considerable research has looked at raising the level of abstraction of languages for networking. On the other hand, when building the Tofino backend, we needed to *lower* the level of abstraction to enforce low-level control over the hardware using pragmas. Pragmas are effectively a mechanism to get the Tofino compiler out of the way—to perform fewer program analyses and make fewer modifications. We could have avoided pragmas and created a simpler backend if Tofino supported direct assembly programming. We hope our results make a case for switching chip vendors to support such low-level interfaces to their chips.

7 EVALUATION

Our evaluation answers the questions listed below.

(1) How does synthesis-based compilation compare to rule-based compilation? (§7.1) We investigate this using the Chipmunk and Domino compilers for the Banzai target, on the metrics of (i) ability to compile programs successfully, (ii) resource usage (i.e., pipeline stages and ALUs per stage) of successfully compiled programs, and (iii) compile time.
(2) Can synthesis effectively target a switching ASIC? (§7.2) We show experimental results using Chipmunk to target Tofino.
(3) How beneficial are slicing and the other optimizations in synthesis-based compilation? (§7.3)

Choice of baseline. For our baseline compiler, we used the Domino rule-based compiler because it can also take as input a high-level specification in transactional style, similar to Chipmunk. Domino also has a few classical compiler optimizations baked into it (e.g., common subexpression elimination, strength reduction, fusing code segments, etc.) [6]. We note that Domino also uses SKETCH for the final step of code generation after much preprocessing using classical rewrite rules [69, §4.3]. As our results show, using SKETCH so late in code generation doesn't help significantly with compiler quality. By contrast, Chipmunk treats the entire code generation problem as a program synthesis problem, with little preprocessing.

Comparing with a commercial compiler like the Tofino compiler [19] would have been preferable to comparing with a research prototype like Domino. However, we can not directly compare with the Tofino compiler because it does not support a transactional style of programming in either of its frontend languages (P4-14 and P4-16). Instead, with the Tofino compiler, the programmer has to manually partition code into different tables and then chain together the tables to implement their desired feature—in other words, program at a lower level than the P4-16 `@atomic` or packet transactional style. We note, however, that we have observed the same butterfly effects that motivated our work (§2) with the Tofino compiler as well. Our results make a case for including program synthesis within commercial compilers, allowing them to support a similar transactional programming style.

Benchmarks. We collected 14 benchmarks (Table 2) from multiple sources [58, 59, 69]. These were previously written as Domino programs using the packet transactions abstraction [69]. All of these benchmarks are known to successfully compile with Domino using one of the 7 stateful atoms combined with the stateless atom proposed by Banzai [69]. We model each of these 7+1 atoms as an ALU using our ALU DSL. We verified that these benchmarks can indeed be successfully compiled with Chipmunk using the same ALU that was used for Domino.

We further create semantic-preserving rewrites of the benchmarks, which we call *mutations*, e.g., Figure 2. Comparing Chipmunk and Domino using mutations allows us to measure whether the two compilers can still compile mutations of an original program that was itself successfully compiled. The mutations can also be used to compare Domino and Chipmunk's resource consumption on a larger set of programs than we started out with.

To create mutations, we added a compiler pass to the Domino compiler to modify Domino programs in semantic-preserving ways. This pass repeatedly transforms programs by randomly picking one of three transformations. The pass modifies (1) `if(x) B else A` into `if(!x) A else B`, (2) `if(A and B)` into `if (B and A)`, and (3) `if(x)` into `if(x and 1==1)`. These mutations are simplistic and are not fully representative of the diverse ways in which developers craft programs that are semantically equivalent. Yet, even with these mutations, we demonstrate (§7.1) that Domino fails to compile many mutations, while Chipmunk successfully compiles all of them.

Experimental setup. We used a single stateless ALU type for all experiments regardless of which stateful ALU we used. This stateless ALU is modeled after the stateless atom proposed in Banzai [69]. For the stateful ALU, we used the same stateful Banzai atom for each benchmark as reported in the benchmark's source [58, 59, 69]. Unless stated otherwise, we run Chipmunk with slicing (§4.3) and all other optimizations (§4.5) enabled. We use SKETCH's parallel mode, which takes advantage of multi-core parallelism [53].

Recall that Chipmunk uses parallel search over different slices and grid sizes (§4.6) by running different synthesis problems on different machines. Additionally, each machine needs multiple cores for SKETCH's parallel mode. We do not have a cluster of physical machines readily available and the cost of running unoptimized Chipmunk on EC2 is prohibitive. Instead, we used a single 32-core 64-hyperthread 256-GB RAM machine (Intel Xeon Gold 6132) to run Chipmunk with the Banzai ALUs (§7.1 and §7.3) and used a single 28-core 56-hyperthread 64-GB RAM machine (AMD Opteron 6272) to run Chipmunk with the Tofino ALUs (§7.2), and report compilation times emulating the parallel search strategy from §4.6. That is, the compile times we present in §7.1, §7.2, and §7.3 were obtained by sequential experiments, with the compile time for a

[1] We note that Tofino does support a swap primitive that can be directly invoked by the programmer as an intrinsic function without writing a swap program in P4. However, the broader point illustrated by our programmer-written swap still holds: code generation requires us to express intra-stage parallelism, which is challenging.

given ALU grid size being the maximum across all per-slice compile times, and the per-slice compile time being the minimum compile time across all ALU grid sizes for that slice. When we don't use slicing, the compile time is simply the minimum compile time across all ALU grid slices. We also estimate the monetary cost of running Chipmunk with slicing on Amazon EC2.

7.1 Synthesis *vs.* Rule-Based Compilation

For each of the 14 benchmarks, we created 10 mutations using our mutating compiler pass. This gives us 140 programs to compare Domino and Chipmunk on, using the following metrics: (i) what % of the mutations of an original program can the compiler successfully compile? (we call this the *compile rate*), (ii) how many pipeline stages and ALUs per stage are needed to fit the program?, (iii) how long does it take for successful compilation? We average across 10 mutations for each benchmark.

We report our results in Table 2. We find that (i) Chipmunk can compile all the mutations we produced, while Domino fails in many cases, (ii) Chipmunk's average compilation times are longer than Domino's, but largely fit into our time budget of ∼1 hour (§3), and (iii) when both Chipmunk and Domino can both compile a mutation, Chipmunk requires fewer pipeline stages and slightly higher ALUs per stage than Domino. However, stages are far more constrained resources (e.g., 32) compared to ALUs in each stage (e.g., 224) [39].

The quality benefits of synthesis also come with a monetary cost: Chipmunk requires more compute resources than Domino (§4.6). However, this cost is reasonable. We use some typical numbers to estimate the cost of a Chipmunk compilation when implementing the strategy from §4.6 in the cloud. To estimate the degree of parallelism with slicing, we use some typical numbers from our benchmarks. Assuming 5 slices per program across both packet fields and state variables, and 10 grid sizes to be searched for each slice (i.e., up to a 3*3 grid, which is the largest grid size we search), we require around 50 VMs. We pick the m5.16xlarge spot EC2 instance [2] with 64 vCPUs and 256 GB RAM because it is closest to our local machine. With per-second billing, a one-minute minimum billing time [3], and a typical synthesis time of 5 minutes (Table 2), the compilation cost is roughly $2.66, with the fairly pessimal assumption that all 50 VMs are occupied the entire 5 minutes. We note that the alternative of rule-based compilers will cost much more in hourly developer wages [60] due to compilation failures.

7.2 Compilation to Tofino Switching ASIC

After modeling the Tofino stateful and stateless ALUs using the ALU DSL (§5.1), we were able to compile and run 10 out of our 14 original benchmarks (without mutations) on Tofino. We report the resource consumption and the compilation times in Table 4. Compilation times are well within an hour for all benchmarks. The times in Tables 4 and 2 are different because (1) the ALUs are different (Banzai vs. Tofino) and (2) different machines were used.

However, we were unable to compile 4 benchmarks to Tofino. The mutations in Table 2 were theoretically guaranteed to compile to *some* Banzai atom because the original programs compiled to that atom; we do not have any such guarantees with Tofino as these benchmarks have not been compiled to Tofino before. In fact, for all 4 of these benchmarks, the resulting sketch file for at least one slice was infeasible up to a grid size of 2*3. Looking closer, we noticed that these 4 benchmarks need a more complicated condition for conditional state updates than supported by Tofino—a computational limit (§3). Hence, we suspect these benchmarks may not be able to compile to *any* grid size given our Tofino ALU model. However, we cannot yet prove that these programs cannot be compiled to even an infinite grid of a certain ALU type. Proving such "unrealizability" is an area of research in synthesis [50].

7.3 Benefits of Optimizations

We now compare Chipmunk's performance with and without slicing on two metrics: pipeline resource usage and compilation time. We keep all other optimizations enabled. Table 3 shows the benefits of slicing for the Banzai backend. We observe that slicing provides a few orders of magnitude speedup on several benchmarks; this is primarily because slicing a program allows us to fit the program within a smaller grid, which translates into a smaller search space/time for SKETCH. Slicing causes a small increase in ALUs per stage because slicing does not share computations between slices. However, Chipmunk with slicing still consumes fewer stages than Domino (Table 2). Beyond performance, slicing also helped us debug the generated P4 programs on Tofino. Each slice could be tested independently on a small grid—instead of testing the whole program on a larger grid. This enabled us to localize and fix bugs in P4 code generation faster.

Even with slicing, some benchmarks (BLUE (decrease) and Stateful firewall) still incur a long synthesis time. To speed these up, we can involve the programmer in synthesis and have them set some holes in the generated sketch (e.g., ALU opcodes, output muxes) based on their insight into the program. For BLUE (decrease) and Stateful firewall, we observed speedups of 4.18 × and 37.14 × relative to the Chipmunk times in Table 2 by intelligently setting the value of only 3% of all the holes, hinting at the promise of an interactive approach to further reduce compile time.

We also measured the benefits of using counterexample assertion vs. hole elimination when integrating SKETCH with Z3 and the benefits of canonicalization (§4.5). Overall, with all other optimizations and slicing enabled, we observed an average 6.21 × speedup with counterexample assertion vs. hole elimination and an average 11.02 × speedup with canonicalization (Appendix C).

8 LIMITATIONS AND FUTURE WORK

We now briefly discuss Chipmunk's main limitations along with avenues for future work. First, while Chipmunk rejects far fewer programs than Domino (Table 2), it is still incomplete and can reject feasible programs. In particular, because slicing has an additional cost PHV/ALU cost, a sliced packet transaction may no longer fit into a small grid, while the original packet transaction may have. Additionally, our constant synthesis algorithm is also incomplete. Second, even after slicing, Chipmunk's compile times are still much longer than Domino. We believe there is still room to improve Chipmunk by exploiting dependencies between parts of the packet transaction, similar to Domino's use of computation DAGs (Figure 3). As shown in §7.3, interactively involving the programmer can also substantially speed up synthesis-based compilation; this is another area that we plan to explore. Third, Chipmunk uses

Program	Chipmunk compile rate	Domino compile rate	Chipmunk depth, width	Domino depth, width	Chipmunk compile time (s)	Domino compile time (s)	Banzai ALU [69]
BLUE (increase) [44]	100%	0%	4,6	N/A	213	N/A	**pred raw**
BLUE (decrease) [44]	100%	0%	4,6	N/A	1134	N/A	**sub**
CONGA [32]	100%	0%	1,7	N/A	16	N/A	**pair**
Flowlet switching [68]	100%	100%	3,8	8.3,4	280	1.5	**pred raw**
Learn filter [69]	100%	100%	3,8	17.5,4	291	2.1	**raw**
Marple new flow [59]	100%	0%	2,3	N/A	12	N/A	**pred raw**
Marple TCP NMO [59]	100%	0%	3,5	N/A	15	N/A	**pred raw**
RCP [76]	100%	100%	2,9	5.6,5	34	2	**pred raw**
Sampling [69]	100%	0%	2,2	N/A	33	N/A	**if else**
SNAP heavy hitter [34]	100%	100%	1,3	3.3,3	70	1.2	**pair**
Spam Detection [34]	100%	80%	1,3	3.1,3	51	7	**pair**
Stateful firewall [34]	100%	100%	4,8	15.5,4.1	7020	1	**pred raw**
STFQ [47]	100%	0%	2,7	N/A	36	N/A	**nested if**
DNS TTL change [36]	100%	0%	3,10	N/A	223	N/A	**nested if**

Table 2: Compile rate, time, and resources averaged over 10 mutations; ALU names refer to Banzai's atoms.

Program	Compile time (s)			Depth, width	
	Slicing	Orig.	speedup	Slicing	Orig.
BLUE (increase) [44]	213	2792	13.11×	4,6	4,4
BLUE (decrease) [44]	1134	32400	28.57×	4,6	4,4
CONGA [32]	16	16	1×	1,7	1,6
Flowlet switching [68]	280	61035	217.98×	3,8	4,7
Learn filter [69]	291	291	1×	3,8	5,6
Marple new flow [59]	12	12	1×	2,3	2,3
Marple TCP NMO [59]	15	16	1.07×	3,5	3,4
RCP [76]	34	96	2.82×	2,9	3,6
Sampling [69]	33	33	1×	2,2	2,2
SNAP heavy hitter [34]	70	75	1.07×	1,3	1,2
Spam Detection [34]	51	62	1.22×	1,3	1,2
Stateful firewall [34]	7020	>86400	>12.31×	4,8	N/A
STFQ [47]	36	1795	49.86×	2,7	4,9
DNS TTL change [36]	223	>86400	>387.44×	3,10	N/A

Table 3: Resource usage, compile time with and without slicing, averaged over 10 mutations. 1 day timeout.

Program	Depth, width	Chipmunk compile time (s)
BLUE(increase) [44]	2, 5	112
BLUE(decrease) [44]	2, 6	113
CONGA [32]	1, 7	6
Flowlet switching [68]	2, 7	95
Marple new flow [59]	1, 2	5
Marple TCP NMO [59]	2, 4	8
RCP [76]	1, 8	26
Sampling [69]	1, 2	24
SNAP heavy hitter [34]	1, 3	40
DNS TTL change [36]	2, 8	34

Table 4: Compiling original benchmarks to Tofino.

significant compute resources for its parallel grid search (§4.6) to reduce compile time; reducing this resource usage by packing synthesis runs into fewer machines is another area for future work. Fourth, currently all our benchmarks are relatively small programs. For future work, we hope to scale Chipmunk sufficiently to perform code generation for much larger production-quality programs such as switch.p4 [22]. Fifth, Chipmunk is currently restricted to code generation and does not concern itself with the problem of allocating memory for large stateful arrays or match-action tables. Combining ILP [56] or SMT [45] techniques for memory allocation with synthesis for code generation is another area for future work.

9 RELATED WORK

Synthesis has been applied to synthesize network updates [59, 66], routing table configurations from high-level policies [42, 74], policies from configurations [37], and control planes [75]. These efforts target network configurations and policies pertaining to access control, reachability, and isolation. In contrast, Chipmunk uses program synthesis to generate packet-processing code for programmable switches. Program slicing [77] computes a subset of program statements that can influence a variable's value at some program location. It has been applied to debugging [77], network verification [61, 64], and query optimization [33]. Chipmunk applies slicing in the new context of machine code generation for switches, requiring considerable adaptation of the basic slicing idea (§4.3).

10 CONCLUSION

We presented Chipmunk, a program-synthesis-based compiler for switches. Chipmunk fits programs into limited switch pipeline resources—programs which might otherwise be rejected by rule-based compilers. To do so, it leverages domain-specific synthesis techniques to expedite compilation and uses a pipeline description language to target multiple backends. We hope that the techniques and results we have presented will stimulate follow-on research in designing program synthesis algorithms that compile programs faster and with fewer compute resources, and produce machine code with lower switch resource consumption. We also believe the ideas here are more generally applicable to FPGA [13, 51] and ASIC-based [52] SmartNIC pipelines.

ACKNOWLEDGEMENTS

We are grateful to our shepherd, Rob Sherwood, the anonymous SIGCOMM reviewers, Amy Ousterhout, Cheng Tan, Keith Winstein, Mina Tahmasbi Arashloo, Michael Walfish, and Aurojit Panda for their many comments on previous drafts of this paper. We thank Armando Solar-Lezama for his detailed responses to our questions on the SKETCH mailing list and Alvin Cheung for many insightful discussions on program synthesis. This work was supported in part by the National Science Foundation award FMitF 1837030.

REFERENCES

[1] A machine model for line-rate programmable switches. https://github.com/packet-transactions/banzai.
[2] Amazon EC2 Spot Instances Pricing. https://aws.amazon.com/ec2/spot/pricing/.
[3] Announcing Amazon EC2 per second billing. https://aws.amazon.com/about-aws/whats-new/2017/10/announcing-amazon-ec2-per-second-billing/.
[4] Concurrency Model for P4. https://github.com/p4lang/p4-spec/issues/48.
[5] DepQBF Solver. http://lonsing.github.io/depqbf/.
[6] Domino Compiler. https://github.com/chipmunk-project/domino-compiler.
[7] High-Capacity StrataXGS Trident 4 Ethernet Switch Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series.
[8] Intel FlexPipe. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.
[9] Jinja - Jinja Documentation (2.10.x). https://jinja.palletsprojects.com/en/2.10.x/.
[10] lcc, A Retargetable Compiler for ANSI C. https://sites.google.com/site/lccretargetablecompiler/.
[11] LLVM Link Time Optimization: Design and Implementation. https://llvm.org/docs/LinkTimeOptimization.html.
[12] MarkusRabe/cadet: A fast and certifying solver for quantified Boolean formulas. https://github.com/MarkusRabe/cadet.
[13] Mellanox Innova-2 Flex Open Programmable SmartNIC. https://www.mellanox.com/page/products_dyn?product_family=276&mtag=programmable_adapter_cards_innova2flex&ssn=7j4vr3u5elh91qnkb9ubjsdlo4.
[14] Mellanox Products: Spectrum 2 Ethernet Switch ASIC. https://www.mellanox.com/page/products_dyn?product_family=277&mtag=spectrum2_ic.
[15] NPL Specification. https://github.com/nplang/NPL-Spec.
[16] P4-16 language specification. https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html.
[17] P4 Compiler. https://github.com/p4lang/p4c.
[18] P4 Extern Types. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html#sec_extern.
[19] P4 Studio | Barefoot. https://www.barefootnetworks.com/products/brief-p4-studio/.
[20] P4 Tutorial. https://github.com/p4lang/tutorials.
[21] P4_16 language specification. https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html.
[22] p4lang/switch: Consolidated switch repo (API, SAI and Nettlink). https://github.com/p4lang/switch.
[23] Product Brief Tofino Page | Barefoot. https://barefootnetworks.com/products/brief-tofino/.
[24] Programming the Forwarding Plane - Nick McKeown. https://forum.stanford.edu/events/2016/slides/plenary/Nick.pdf.
[25] Quantified Boolean Formula. https://en.wikipedia.org/wiki/True_quantified_Boolean_formula.
[26] [Sketchusers] Strange error for large integer constants. https://lists.csail.mit.edu/pipermail/sketchusers/2019-July/000094.html.
[27] The P4 Language Specification. https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf.
[28] The reference P4 software switch. https://github.com/p4lang/behavioral-model.
[29] The Z3 Theorem Prover. https://github.com/Z3Prover/z3.
[30] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample Guided Inductive Synthesis Modulo Theories. In *Computer Aided Verification*, 2018.
[31] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.
[32] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *SIGCOMM*, 2014.
[33] Jesus M. Almendros-Jimenez, Josep Silva, and Salvador Tamarit. XQuery Optimization Based on Program Slicing. In *CIKM*, 2011.
[34] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful network-wide abstractions for packet processing. In *SIGCOMM*, 2016.
[35] Sorav Bansal and Alex Aiken. Automatic Generation of Peephole Superoptimizers. In *ASPLOS*, 2006.
[36] Leyla Bilge, Engin Kirda, Christopher Kruegel, and Marco Balduzzi. EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis. In *NDSS*, 2011.
[37] Rudiger Birkner, Dana Drachsler Cohen, Laurent Vanbever, and Martin Vechev. Config2Spec: Mining Network Specifications from Network Configurations. In *NSDI*, 2020.
[38] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM CCR*, 2014.
[39] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *SIGCOMM*, 2013.
[40] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. dRMT: Disaggregated Programmable Switching. In *SIGCOMM*, 2017.
[41] Jack W. Davidson and Christopher W. Fraser. The Design and Application of a Retargetable Peephole Optimizer. *TOPLAS*, 1980.
[42] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *NSDI*, 2018.
[43] Grigory Fedyukovich, Maaz Bin Safeer Ahmad, and Rastislav Bodik. Gradual Synthesis for Static Parallelization of Single-pass Array-processing Programs. In *PLDI*, 2017.
[44] Wu-chang Feng, Kang G. Shin, Dilip D. Kandlur, and Debanjan Saha. The BLUE Active Queue Management Algorithms. *IEEE/ACM Transactions on Networking*, 2002.
[45] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. In *SIGCOMM*, 2020.
[46] Xiangyu Gao, Taegyun Kim, Aatish Kishan Varma, Anirudh Sivaraman, and Srinivas Narayana. Autogenerating Fast Packet-Processing Code Using Program Synthesis. In *HotNets*, 2019.
[47] Pawan Goyal, Harrick M. Vin, and Haichen Chen. Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *SIGCOMM*, 1996.
[48] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of Loop-Free Programs. In *PLDI*, 2011.
[49] Gary D. Hachtel and Fabio Somenzi. *Logic synthesis and verification algorithms*. Springer, 2006.
[50] Qinheping Hu, Jason Breck, John Cyphert, Loris D'Antoni, and Thomas Reps. Proving Unrealizability for Syntax-Guided Synthesis. In *CAV*, 2019.
[51] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. The P4->NetFPGA Workflow for Line-Rate Packet Processing. In *FPGA*, 2019.
[52] Stephen Ibanez, Muhammad Shahbaz, and Nick McKeown. The Case for a Network Fast Path to the CPU. In *HotNets*, 2019.
[53] Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama, and Jeffrey S. Foster. Adaptive Concretization for Parallel Program Synthesis. In *CAV*, 2015.
[54] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *NSDI*, 2018.
[55] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *SOSP*, 2017.
[56] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling Packet Programs to Reconfigurable Switches. In *NSDI*, 2015.
[57] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified Lifting of Stencil Computations. In *PLDI*, 2016.
[58] Anirudh Sivaraman Kaushalram. *Designing Fast and Programmable Routers*. PhD thesis, EECS Department, Massachusetts Institute of Technology, September 2017.
[59] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-Directed Hardware Design for Network Performance Monitoring. In *SIGCOMM*, 2017.
[60] U.S. Bureau of Labor Statistics. Occupational Employment and Wages, May 2018, Software developers, Systems software. https://www.bls.gov/oes/current/oes151133.htm.
[61] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. Verifying Reachability in Networks with Mutable Datapaths. In *NSDI*, 2017.
[62] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. Swizzle Inventor: Data Movement Synthesis for GPU Kernels. In *ASPLOS*, 2019.
[63] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling Up Superoptimization. In *ASPLOS*, 2016.
[64] Gordon D. Plotkin, Nikolaj Bjørner, Nuno P. Lopes, Andrey Rybalchenko, and George Varghese. Scaling Network Verification Using Symmetry and Surgery. In *POPL*, 2016.
[65] Regehr, John. Synthesizing Constants. https://blog.regehr.org/archives/1636.
[66] Shambwaditya Saha, Santhosh Prabhu, and P. Madhusudan. NetGen: Synthesizing Data-plane Configurations for Network Policies. In *SOSR*, 2015.
[67] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic Superoptimization. In *ASPLOS*, 2013.
[68] Shan Sinha, Srikanth Kandula, and Dina Katabi. Harnessing TCPs Burstiness using Flowlet Switching. In *HotNets*, 2004.

[69] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *SIGCOMM*, 2016.
[70] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.
[71] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching Concurrent Data Structures. In *PLDI*, 2008.
[72] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial Sketching for Finite Programs. In *ASPLOS*, 2006.
[73] Haoyu Song. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In *HotSDN*, 2013.
[74] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Genesis: Synthesizing Forwarding Tables in Multi-tenant Networks. In *POPL*, 2017.
[75] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Synthesis of Fault-Tolerant Distributed Router Configurations. In *SIGMETRICS*, 2018.
[76] C.H. Tai, J. Zhu, and N. Dukkipati. Making Large Scale Deployment of RCP Practical for Real Networks. In *INFOCOM*, 2008.
[77] Mark Weiser. Program Slicing. In *ICSE*, 1981.
[78] Michael D. Wong, Aatish Kishan Varma, and Anirudh Sivaraman. Testing compilers for programmable switches through switch hardware simulation. *ArXiv*, abs/2005.02310, 2020.

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

A EXAMPLE OF CODE GENERATION SKETCH

This appendix presents a simplified version of the sketch generated by Chipmunk for a 2-by-2 grid and a simple spec. We use ...wherever appropriate to signify that the code is similar to code presented before.

```
// num_pipeline_stages = 2
// num_alus_per_stage = 2 (2 stateless ALUs + 2 stateful ALUs)
// num_phv_containers = 2
// imux stands for input mux; omux for output mux
int stateless_alu_0_0_imux1_ctrl= ??(1);        int stateless_alu_0_1_imux1_ctrl= ??(1);
int stateless_alu_0_0_imux2_ctrl= ??(1);        int stateless_alu_0_1_imux2_ctrl= ??(1);
int stateless_alu_0_0_immediate= ??(2);         int stateless_alu_0_1_immediate= ??(2);
int stateless_alu_0_0_opcode= ??(2);            int stateless_alu_0_1_opcode= ??(2);
int stateful_alu_0_0_mode_global= ??(1);        int stateful_alu_0_1_mode_global= ??(1);
int stateful_alu_0_0_const_0_global= ??(2);     int stateful_alu_0_1_const_0_global= ??(2);
int stateless_alu_1_0_imux1_ctrl= ??(1);        int stateless_alu_1_1_imux1_ctrl= ??(1);
int stateless_alu_1_0_imux2_ctrl= ??(1);        int stateless_alu_1_1_imux2_ctrl= ??(1);
int stateless_alu_1_0_immediate= ??(2);         int stateless_alu_1_1_immediate= ??(2);
int stateless_alu_1_0_opcode= ??(2);            int stateless_alu_1_1_opcode= ??(2);
int stateful_alu_1_0_mode_global= ??(1);        int stateful_alu_1_1_mode_global= ??(1);
int stateful_alu_1_0_const_0_global= ??(2);     int stateful_alu_1_1_const_0_global= ??(2);
int stateful_alu_0_0_imux_ctrl= ??(1);          int stateful_alu_0_1_imux_ctrl= ??(1);
int stateful_alu_1_0_imux_ctrl= ??(1);          int stateful_alu_1_1_imux_ctrl= ??(1);
int omux_phv_0_0_ctrl= ??(2);                   int omux_phv_0_1_ctrl= ??(2);
int omux_phv_1_0_ctrl= ??(2);                   int omux_phv_1_1_ctrl= ??(2);
int salu_active_0_0= ??(1);                     int salu_active_0_1= ??(1);
int salu_active_1_0= ??(1);                     int salu_active_1_1= ??(1);

// Definitions of muxes and ALUs of the switch pipeline
// Input mux for each ALU
int stateful_alu_imux_0_0(int input0,int input1, int stateful_alu_0_0_imux_ctrl_local) {
    if (stateful_alu_0_0_imux_ctrl_local == 0) { return input0;}
    else { return input1; }
}
int stateful_alu_imux_0_1(int input0,int input1, int stateful_alu_0_1_imux_ctrl_local) {...}
int stateful_alu_imux_1_0(int input0,int input1, int stateful_alu_1_0_imux_ctrl_local) {...}
int stateful_alu_imux_1_1(int input0,int input1, int stateful_alu_1_1_imux_ctrl_local) {...}
// Output mux for each PHV container
int omux_phv_0_0(int input0,int input1,int input2,int omux_phv_0_0_ctrl_local) {
    if (omux_phv_0_0_ctrl_local == 0) {return input0;}
    else if (omux_phv_0_0_ctrl_local == 1) {return input1;}
    else {return input2;}
}
int omux_phv_0_1(int input0,int input1,int input2,int omux_phv_0_1_ctrl_local) {...}
int omux_phv_1_0(int input0,int input1,int input2,int omux_phv_1_0_ctrl_local) {...}
int omux_phv_1_1(int input0,int input1,int input2,int omux_phv_1_1_ctrl_local) {...}
// Definition of ALUs
int stateless_alu_0_0_mux1(int input0,int input1, int stateless_alu_0_0_imux1_ctrl_local) {
    if (stateless_alu_0_0_imux1_ctrl_local == 0) { return input0;}
    else { return input1; }
}
int stateless_alu_0_0_mux2(int input0,int input1, int stateless_alu_0_0_imux2_ctrl_local) {...}
int stateless_alu_0_0(int input0,int input1,int opcode,int immediate,int imux1_ctrl_hole_local,int imux2_ctrl_hole_local) {
    int pkt_0 = stateless_alu_0_0_mux1(input0,input1,imux1_ctrl_hole_local);
    int pkt_1 = stateless_alu_0_0_mux2(input0,input1,imux2_ctrl_hole_local);
    if (opcode==0) { return pkt_0+pkt_1;}
    else if (opcode==1) { return pkt_0-pkt_1;}
    else if (opcode==2) { return pkt_0+immediate;}
    else { return pkt_0-immediate;}
}
int stateless_alu_0_1_mux1(int input0,int input1, int stateless_alu_0_1_imux1_ctrl_local) {...}
int stateless_alu_0_1_mux2(int input0,int input1, int stateless_alu_0_1_imux2_ctrl_local) {...}
int stateless_alu_0_1(int input0,int input1,int opcode,int immediate,int imux1_ctrl_hole_local,int imux2_ctrl_hole_local) {...}
int stateful_alu_0_0_Mode(int input0,int input1,int mode) {
    if (mode == 0) {return input0;}
    else {return input1;}
}
int stateful_alu_0_0(ref int state_0, int pkt_0, int mode, int const_0) {
    int old_state_0 = state_0;
    state_0 = stateful_alu_0_0_Mode(state_0 + const_0, pkt_0, mode);
    return old_state_0;
}
int stateful_alu_0_1_Mode(int input0,int input1,int mode) {...}
int stateful_alu_0_1(ref int state_0, int pkt_0, int mode, int const_0) {...}
int stateful_alu_1_0_Mode(int input0,int input1,int mode) {...}
int stateful_alu_1_0(ref int state_0, int pkt_0, int mode, int const_0) {...}
int stateful_alu_1_1_Mode(int input0,int input1,int mode) {...}
int stateful_alu_1_1(ref int state_0, int pkt_0, int mode, int const_0) {...}


// Data type for holding result from spec and implementation
struct StateAndPacket {
    int pkt_0;
    int state_0;
    int state_1;
}
```

```
// Specification
|StateAndPacket| program(|StateAndPacket| state_and_packet) {
    state_and_packet.pkt_0 = 1 + state_and_packet.state_0;
    state_and_packet.state_1 = state_and_packet.state_0;
    return state_and_packet;
}

// Implementation
|StateAndPacket| pipeline (|StateAndPacket| state_and_packet) {
    // Constraints to allocate state variables to stateful ALUs
    assert((salu_active_0_0 + salu_active_0_1) <= 2);
    assert((salu_active_1_0 + salu_active_1_1) <= 2);
    assert((salu_active_0_0 + salu_active_1_0) <= 1);
    assert((salu_active_0_1 + salu_active_1_1) <= 1);
    // Container i will be allocated to packet field i from the spec (canonical allocation).
    int input_0_0 = 0;
    int input_0_1 = 0;
    // One variable for each stateful ALU's state operand
    // This will be allocated to a state variable from the program using the salu_active indicator variables above.
    int state_operand_salu_0_0 = 0;
    int state_operand_salu_0_1 = 0;
    int state_operand_salu_1_0 = 0;
    int state_operand_salu_1_1 = 0;
    /********** Stage 0 *********/
    // Read each PHV container from corresponding packet field.
    input_0_0 = state_and_packet.pkt_0;
    // Stateless ALUs
    int destination_0_0 = stateless_alu_0_0(input_0_0,input_0_1,stateless_alu_0_0_opcode,stateless_alu_0_0_immediate,
                                            stateless_alu_0_0_imux1_ctrl,stateless_alu_0_0_imux2_ctrl);
    int destination_0_1 = stateless_alu_0_1(...);
    // Stateful operands
    int packet_operand_salu0_0_0 = stateful_alu_imux_0_0(input_0_0,input_0_1,stateful_alu_0_0_imux_ctrl);
    int packet_operand_salu0_1_0 = stateful_alu_imux_0_1(...);
    // Read stateful ALU slots from allocated state vars.
    if (salu_active_0_0 == 1) {
        state_operand_salu_0_0 = state_and_packet.state_0;
    }
    if (salu_active_0_1 == 1) {...}
    // Stateful ALUs
    int state_alu_output_0_0 = stateful_alu_0_0(state_operand_salu_0_0,packet_operand_salu0_0_0,
                                                stateful_alu_0_0_mode_global,stateful_alu_0_0_const_0_global);
    int state_alu_output_0_1 = stateful_alu_0_1(...);
    // Outputs
    int output_0_0 = omux_phv_0_0(state_alu_output_0_0,state_alu_output_0_1,destination_0_0,omux_phv_0_0_ctrl);
    int output_0_1 = omux_phv_0_1(state_alu_output_0_0,state_alu_output_0_1,destination_0_1,omux_phv_0_1_ctrl);
    // Write state_0
    if (salu_active_0_0 == 1) { state_and_packet.state_0 = state_operand_salu_0_0;}
    // Write state_1
    if (salu_active_0_1 == 1) { state_and_packet.state_1 = state_operand_salu_0_1;}
    /********** Stage 1 *********/
    // Input of this stage is the output of the previous one.
    int input_1_0 = output_0_0;
    int input_1_1 = output_0_1;
    ...
    // Write pkt_0 at the end of the pipeline.
    state_and_packet.pkt_0 = output_1_0;
    // Return updated packet fields and state vars
    return state_and_packet;
}
// Main sketch routine that asserts equivalence of pipeline and spec
harness void main(int pkt_0, int state_0, int state_1) {
    |StateAndPacket| x = |StateAndPacket|(pkt_0 = pkt_0,state_0 = state_0,state_1 = state_1);
    |StateAndPacket| pipeline_result = pipeline(x);
    |StateAndPacket| program_result = program(x);
    assert(pipeline_result.state_0 == program_result.state_0);
    assert(pipeline_result.state_1 == program_result.state_1);
    assert(pipeline_result.pkt_0 == program_result.pkt_0);
}
```

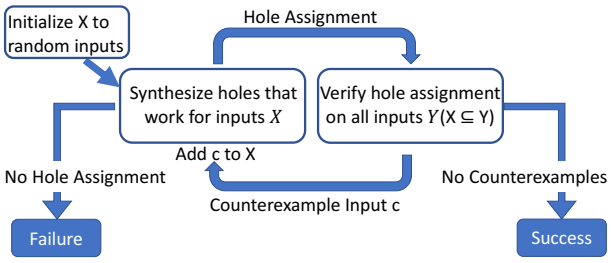

Figure 8: The CEGIS algorithm for synthesis.

Program	Cex assertion (sec)	Hole elimination (sec)
BLUE (increase) [44]	213	>2130
BLUE (decrease) [44]	1134	>11340
CONGA [32]	16	16
Flowlet switching [68]	280	>2800
Learn filter [69]	291	291
Marple new flow [59]	12	12
MARPLE TCP NMO [59]	15	15
RCP [76]	34	>340
SAMPLING [69]	33	>330
SNAP heavy hitter [34]	70	>700
Spam Detection [34]	51	>510
Stateful firewall [34]	7020	>70200
DNS TTL change [36]	223	438
STFQ [47]	36	36

Table 5: Compilation time for Hole elimination vs. counterexample assertion in SKETCH-Z3 loop

Program	Canonicalized (sec)	Synthesized (sec)
BLUE (increase) [44]	213	273
BLUE (decrease) [44]	1134	1056
CONGA [32]	16	18
Flowlet switching [68]	280	1112
Learn filter [69]	291	355
Marple new flow [59]	12	19
MARPLE TCP NMO [59]	15	23
RCP [76]	34	46
SAMPLING [69]	33	59
SNAP heavy hitter [34]	70	70
Spam Detection [34]	51	51
Stateful firewall [34]	7020	9810
DNS TTL change [36]	223	616
STFQ [47]	36	4803

Table 6: Compilation time in seconds for synthesized vs. canonical allocation

B PROGRAM SYNTHESIS USING SKETCH

We briefly describe SKETCH's internals here; [70] has more details. SKETCH is given as inputs a specification to satisfy and a partial program (the sketch) (Figure 4). Let x be an n-bit vector representing all inputs to both the specification S and the partial program P. The task of the synthesizer is to determine values of all the holes in P such that the results of executing the specification and the sketch on an input x, $S(x)$ and $P(x)$, are the same for all x. Let c be an m-bit vector representing all holes that need to be determined (or "filled in") by SKETCH to complete the sketch. Then, the program synthesis problem solves for c in the following formula in first-order logic [72]:

$$\exists c \in \{0,1\}^m, \forall x \in \{0,1\}^n : S(x) = P(x,c) \quad (2)$$

Equation 2 is an instance of the quantified boolean formula problem (QBF) [25]. QBF is a generalization of boolean satisfiability (SAT) that allows multiple ∀ and ∃ quantifiers; SAT implicitly supports a single ∀ or ∃. While QBF solvers exist [5, 12], they are not optimized for the QBF instances found in program synthesis [70]. Hence, SKETCH uses an algorithm called *counterexample-guided inductive synthesis (CEGIS) [71, 72]*, designed to work efficiently for the QBF instances found in program synthesis.

CEGIS (Figure 8) exploits the *bounded observation hypothesis*: for typical specifications, there are a small number of representative inputs that form a "perfect test suite," i.e., if the specification and the completed sketch agree on this test suite, then they agree on all inputs. To exploit this hypothesis, CEGIS repeatedly alternates between two phases: (1) synthesizing on a small set of concrete test inputs and (2) verifying that the completed sketch matches the specification on all possible inputs. A failed verification generates a counterexample that is added to the set of concrete test inputs, and a fresh iteration of synthesis+verification follows. CEGIS terminates when either the verification phase succeeds or the synthesis phase fails, i.e., there is no way to find values for the holes that allow P and S to match on the concrete test input set.

The synthesis phase of CEGIS is represented by the following formula. Here $x_1, x_2, \ldots x_k$ are the current set of concrete test inputs:

$$\exists c \in \{0,1\}^m : S(x_1) = P(x_1,c) \wedge \ldots S(x_k) = P(x_k,c) \quad (3)$$

The verification phase is represented by the following formula. Here, c^* is the hole solution being verified:

$$\forall x \in \{0,1\}^n : S(x) = P(x,c^*) \quad (4)$$

Both the synthesis and verification phases of CEGIS are simpler than solving Equation 2 directly as a QBF problem. This is because each phase fixes either the test inputs (synthesis) or holes (verification) to concrete values, which turns the resulting subproblem into a SAT problem, which can be fed to a more efficient SAT (instead of QBF) solver.

C DEEP DIVE INTO CHIPMUNK OPTIMIZATIONS

We now examine the impact of our optimizations on compile time. In these experiments, we keep slicing and all other optimizations enabled, only toggling on or off one optimization.

C.1 Hole elimination vs. counterexample assertion

We considered different modes in which Z3 and SKETCH can cooperate in code generation. Our intuition was that counterexample assertion would lead to faster compile time because hole elimination only eliminates the particular hole assignment that caused that Z3 failure, while a new counterexample would eliminate *all* the hole assignments that could have caused that failure. Our experiments (Table 5) confirm this intuition. We set a timeout for the hole elimination to 10× the time for counterexample assertion to limit run time of our experiments; without a timeout, the benefits of counterexamples would be even more pronounced. Hence, we use counterexample assertion.

C.2 Canonical vs. synthesized allocation

We study two ways in which state variables can be allocated to stateful ALUs and packet fields can be allocated to PHV containers. In canonical allocation, as discussed in §4.5, a packet field is always

allocated to its canonical container and a state variable is always allocated to its canonical ALU after SKETCH determines its stage. In synthesized allocation, on the other hand, we disregard symmetry, and ask SKETCH to find the allocation from scratch. Hence, in synthesized allocation, SKETCH determines which container to use for a field, and which stage and which stateful ALU in that stage to use for a state variable. Table 6 shows the results. We find that canonical allocation and synthesized allocation perform similarly on benchmarks that have a short compilation time, but that canonical allocation can significantly reduce time on the longer benchmarks.