ABSTRACT
Packet-processing code should be fast. But, it is hard to write fast code for programmable substrates such as high-speed switches, multicore SoC SmartNICs, FPGAs, middleboxes, and the end-host stack. Today, expert developers with deep familiarity with the underlying hardware handcraft such code. Making things worse, building optimizing compilers for these substrates requires significant development effort, which may not be available for these new, niche, and evolving substrates.

We propose an alternative: to automatically generate fast packet-processing code using program synthesis. For the domain of packet processing, using synthesis can generate faster code than an optimizing compiler at the cost of increased compile time. As a case study, we apply program synthesis to build a code generator, Chipmunk, for a simulator of the protocol-independent switch architecture (PISA). Chipmunk generates code for many programs that a previous code generator based on classical compiler optimizations rejects, and code generated by Chipmunk uses much fewer pipeline resources. We also outline future directions in applying program synthesis to code generation for packet processing.

ACM Reference Format:

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

HotNets ’19, November 13–15, 2019, Princeton, NJ. USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-7020-2/19/11...
https://doi.org/10.1145/3365069.3365858

Figure 1: Syntax-guided synthesis in SKETCH. ??(b) is a hole with value in [0, 2^b − 1]. << is the left-shift operator.

1 INTRODUCTION
There has been a proliferation of programmable network substrates recently. Examples include high-speed programmable switches, multicore SoC SmartNICs, FPGAs, software middleboxes, and the networking stack within servers. With growing link speeds, there is a need to run ever faster packet-processing code on these substrates. For example, SmartNICs run at line rates of 40–100 Gbit/s. Switches run at 100 Gbit/s per port and a few Tbit/s in aggregate.

Although these substrates are programmable, developing fast programs for them is hard for two reasons. First, developing such fast programs requires manual optimization by experts who are familiar with each underlying hardware architecture. Developers must be aware of the cache and memory hierarchy for CPUs and SoC-based NICs; aware of ALU, TCAM, and SRAM limits for programmable switches; and have an understanding of lookup tables, placement, and routing for an FPGA. For instance, Microsoft hired a dedicated team of hardware engineers to program its FPGA-based SmartNIC [9]. Second, while optimizing compilers might alleviate the difficulties of generating fast code, building optimizing compilers for a target requires significant engineering effort spanning decades [50]—effort that may not be available for new, niche, and evolving network hardware.

In response to the above concerns, we propose the use of program synthesis to develop code generators for emerging network substrates. Program synthesis is the process of automatically generating a program that meets a given specification. We focus on a recent variant of program synthesis called syntax-guided synthesis (SyGuS) [26, 59, 66] that constrains the search space of

```c
int spec(int x) { return x*5; }
int sketch1(int x) implements spec { return x << ??(2) + x; }
int sketch2(int x) implements spec { return x << ??(2); }
```
programs using syntactic restrictions. As a concrete example of syntax-guided synthesis, in SKETCH [19, 61], a programmer provides a program synthesizer the specification along with a sketch (Figure 1): a partial program with holes representing values within a finite range of integers. The partial program constrains the search space syntactically and encodes the programmer’s insight into the structure of the implementation. The synthesizer completes the sketch by filling in all holes with concrete values so that the completed sketch meets the specification—or says that synthesis is infeasible.

Syntax-guided synthesis can be applied to code generation by (1) using the developer’s program, say in C or P4, as the specification, (2) using the sketch to represent the structure of the substrate, and (3) using holes to represent a large but finite number of low-level hardware configurations such as assembly opcodes, operand choices for instructions, and contents of look-up tables. Resource constraints and performance objectives can be incorporated by using an appropriate sketch (e.g., limiting the total number of expressions used in a sketch). For instance, a switch pipeline with a fixed set of instructions in each stage can be viewed as a sketch with holes representing hardware configurations such as choices of opcodes and immediate operands (Appendix A).

Synthesis-based code generators have the potential to generate faster packet-processing code relative to optimizing compilers. This is because optimizing compilers are designed to generate consistently good code for all programs within a reasonable compilation time budget. For fast packet-processing, however, there is immense value to generating near-optimal machine code. Synthesis can discover such code by performing exhaustive search. For instance, synthesis techniques have produced x86 and ARM binaries with better performance than gcc -O3 on programs that are a few hundred instructions long [49, 53]. The cost of this near-optimal code is increased compile time—a tradeoff worth making for fast packet processing. Additionally, synthesis might permit rapid prototyping of compilers. This is because synthesis allows us to declaratively specify code generation for different substrates as synthesis problems, e.g., using sketches. This could allow us to reuse synthesis technology for performing code generation across many different packet-processing substrates.

Despite these potential benefits over optimizing compilers, syntax-guided synthesis faces a key challenge: it is a search problem over a large combinatorial search space of programs. The space grows exponentially with the number of holes (i.e., hardware configurations). However, we believe that our vision is feasible for three reasons. First, after over a decade of research, there are now mature open-source synthesis tools [61, 66] and promising real-world applications of synthesis [24, 39, 47, 53, 55]. Second, several fast packet-processing programs are naturally small (e.g., BLUE [36], RED [37], or RCP [65]) and simple (e.g., no pointers or loops in P4 [32]). This makes synthesis more tractable. Third, many hardware substrates exhibit significant symmetry. This allows us to prune the search space for synthesis by considering only one exemplar hardware configuration out of many equivalent configurations (§3; Fig. 4).

We present a case study of applying program synthesis to code generation for packet-processing pipelines (§3). We design Chipmunk, a synthesis-aided code generator, to generate low-level code for a hardware simulator of a pipeline based on the protocol-independent switch architecture (PISA) [17]. We compare Chipmunk with Domino [58] (§4). Chipmunk generates pipelined implementations of many programs that Domino rejects. This is because Domino incorrectly decides that the programs cannot satisfy the switch’s resource constraints. For programs that both Domino and Chipmunk can generate code for, Chipmunk’s generated code has much smaller pipeline depth.

We also outline directions for future work in applying program synthesis to generating fast packet processing code (§5). We describe three applications: optimizing packet processing on processors, automatically approximating programs to run faster, and providing performance troubleshooting hints to developers.

2 BACKGROUND

We now overview the programming language we use to program packet-processing pipelines, the hardware architecture of these pipelines, and the program synthesis technology that we use as a building block.

2.1 Programming language

Several languages now exist for packet processing, e.g., P4-14 [13], P4-16 [12], POF [62], NPL [11], and Domino [58]. Of these, we picked Domino as the language for this paper, meaning that the input program for which code generation is performed is expressed by the programmer in Domino. Figure 2 shows an example Domino program that samples every 10th packet going through a switch. We picked Domino because it is a (relatively) high-level language. Further, it provides transactional semantics: operations in a Domino program execute from start to finish atomically, as though packets are being processed by the pipeline exactly one packet at a time. This frees the programmer from having to deal with concurrency issues, delegating those to the compiler instead. The same transactional semantics are also supported by P4-16’s @atomic construct [15], P4-16’s @atomic construct was influenced by Domino [3];
hence, we expect to be able to support P4-16 as a frontend using the `@atomic` construct in the future.

### 2.2 Hardware architecture

We consider a packet-processing pipeline based on the hardware architecture described in RMT [33] and Banzai [58]. RMT provides a hardware architecture for programmable match-action processing, while Banzai extends RMT with stateful computation. The hardware architecture described in RMT and Banzai is commonly known as the Protocol Independent Switch Architecture (PISA) [17] and is the dominant architecture for high-speed programmable switches today [6, 16].

We simulate a simplified model of PISA by abstracting out switch computation into a 2D grid of ALUs (Figure 2). The x axis of this grid represents pipeline stages; the y axis represents parallel ALUs within a pipeline stage. Packets enter the grid from the left and exit from the right, and the grid is assumed to support a throughput of 1 packet per clock cycle (i.e., the line rate of the switching ASIC). A program’s packet fields are stored in the packet header vector (PHV); the PHV is a set of containers and each container holds a packet field as it is passed and transformed between stages. Similarly, a program’s state variables are stored within stateful ALUs.

ALUs are PISA’s computation units and can modify either packet fields alone (stateless ALUs) or both fields and switch state (stateful ALUs). Their computations are atomic in that any update to state within an ALU is visible to the next packet arriving at that ALU a clock cycle later. Our simulator allows us to experiment with a variety of simulated switch hardware by specifying different stateful and stateless ALUs with different sets of operations, represented by ALU opcodes. Operands to stateless and stateful ALUs can be PHV containers, immediate operands, or switch state; this is determined by the input mux. A stateless ALU’s output is written into the PHV container designated for that stateless ALU and a stateful ALU’s output can be routed into any container; this is determined by the output mux. Table 1 summarizes hardware configurations in our simulator. These are the configurations that need to be populated by any code generator, whether based on classical compiler optimizations or program synthesis.

### 2.3 Program synthesis

We use the SKETCH program synthesizer for developing Chipmunk. We briefly describe SKETCH’s internals here; [59] has more details. SKETCH is given as inputs a specification to a satisfy and a partial program (the sketch) (Figure 1). 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. For example, in Figure 1, $m = 2$ and $n = 5$, the default width of integer inputs in SKETCH. Then, the program synthesis problem solves for $c$ in the following formula in first-order logic [61]:

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

Equation 1 is an instance of the quantified boolean formula problem (QBF) [18]. QBF is a generalization of SAT that allows multiple $\forall$ and $\exists$ quantifiers; SAT implicitly supports a single $\forall$ or $\exists$. While QBF solvers exist [4, 8], they are not optimized for the QBF instances found in program synthesis [59]. Hence, SKETCH uses an algorithm called counterexample-guided inductive synthesis.
(CEGIS) [60, 61], designed to work efficiently for the QBF instances found in program synthesis.

CEGIS (Figure 3) 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:

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

(2)

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

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

(3)

Both the synthesis and verification phases of CEGIS are simpler than solving Equation 1 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.

Figure 3: The CEGIS algorithm for synthesis.

### 3.1 The Chipmunk code generator

In addition to a Domino program, Chipmunk takes as arguments the number of stages in the pipeline, the pipeline width, and a specification of the capabilities of the stateful and stateless ALUs. Given these arguments, Chipmunk generates a sketch corresponding to the functionality of the switch data path (Figure 2) and then invokes SKETCH to solve it; Appendix A contains a simplified version of this generated sketch. The specification for the sketch is the packet transaction; the holes are Banzai configurations (Table 1). We now describe how Chipmunk sets up and then solves the synthesis problem for SKETCH to generate the set of Banzai configurations.

**Allocating packet fields to PHV containers.** Packet fields in the program need to be allocated to PHV containers in the hardware. When a program fails to compile, the developer requires intricate knowledge of the hardware to interpret a compilation error. Compounding this, existing switch compilers [14, 38, 58] fail to compile a program even when a semantically-equivalent rewrite of the program can be compiled, pushing the burden of finding such a rewrite onto the developer.

We use SKETCH to build Chipmunk, a PISA code generator that is much more resilient to how a developer writes packet-processing programs (§4). We now describe how Chipmunk synthesizes the hardware configurations of the simulator (Table 1) to implement the Domino program supplied to Chipmunk. While we developed Chipmunk for switch pipelines, we believe Chipmunk’s techniques are also applicable to similar NIC pipelines [41].

### 3.1 The Chipmunk code generator

In addition to a Domino program, Chipmunk takes as arguments the number of stages in the pipeline, the pipeline width, and a specification of the capabilities of the stateful and stateless ALUs. Given these arguments, Chipmunk generates a sketch corresponding to the functionality of the switch data path (Figure 2) and then invokes SKETCH to solve it; Appendix A contains a simplified version of this generated sketch. The specification for the sketch is the packet transaction; the holes are Banzai configurations (Table 1). We now describe how Chipmunk sets up and then solves the synthesis problem for SKETCH to generate the set of Banzai configurations.

**Allocating packet fields to PHV containers.** Packet fields in the program need to be allocated to PHV containers in the hardware. There are natural constraints on this allocation: each packet field is assigned to exactly one container and each container is assigned to at most one packet field. We use indicator variable holes to represent these allocations, e.g., $I[f, c]$ tracks if field $f$ is stored in container $c$ over the entire pipeline. SKETCH solves for the indicator variable holes while respecting the allocation constraints, which are expressed as SKETCH assertions.

We can reduce the number of indicator variables and speed up synthesis by exploiting symmetry in the common case of homogeneous grids, where the same stateful and stateless ALU types are repeated across the 2D grid.
and each ALU can access the same set of operands using its input muxes. To exploit symmetry, we apply the idea of canonicalization [28] and rename program fields to a canonical set \( f_1, f_2, \ldots, f_m \). We then map \( f_1 \) to container 1, \( f_2 \) to container 2, etc. Intuitively, any allocation can be changed into a canonicalized one by renumbing containers (Fig. 4); hence there is no loss of expressiveness by forcing a canonical allocation.

**Allocating state variables to stateful ALUs.** State variables from the input specification should be assigned to specific stateful ALUs in the Banzai grid. The indicator variable holes \( [s, x, y] \) track if state variable \( s \) is assigned to stateful ALU \( x \) in stage \( y \). Similar to allocating packet fields, we exploit symmetry in homogeneous grids, and canonicalize the state variables in the program to \( s_1, s_2, \ldots, s_n \). Hence, variable \( s_j \) is allocated to stateful ALU \( i \) within a stage. However, there is an important wrinkle: \( \text{SKETCH} \) still needs to determine which stage a state variable \( s_j \) must be allocated to, due to dependencies between state variables, i.e., if an update to state variable \( s_i \) depends on the value of \( s_j \), \( s_j \) must be allocated to a stage that is earlier than that of \( s_i \).

**Allocating opcodes and mux controls for ALUs.** We use \( \text{SKETCH} \) holes to represent the opcode used by each ALU and the mux controls. The size of opcode holes depends on the number of operations supported by an ALU. The size of mux holes depends on the number of PHV containers (the pipeline width). In experiments, we find that constraining opcode holes to take on fewer values than the hardware allows can sometimes speed up synthesis (e.g., by only considering arithmetic ALU opcodes), provided the program can be fully expressed using those opcodes. However, at other times, such constraints increase synthesis time if the program requires the full expressiveness of the hardware, causing synthesis to fail. We are designing heuristics to balance both possibilities.

**Scaling Chipmunk to a large number of input bits.** Once Chipmunk sets up the sketch to synthesize Banzai configurations, \( \text{SKETCH} \)’s problem can be stated as: find an assignment of values to all holes so that the sketch and the specification have the same output for all possible input packets and initial values of state.

To solve this problem, \( \text{SKETCH} \) uses the CEGIS algorithm described earlier. However, \( \text{SKETCH} \) limits the range of inputs to speed up synthesis. By default, \( \text{SKETCH} \) only searches over all 5-bit integers for each scalar input. Hence, it is possible that the hole assignment returned by \( \text{SKETCH} \) fails to work over larger input ranges, say 32-bit packet fields. To scale \( \text{SKETCH} \) to larger input sizes, we decouple the input ranges for synthesis and verification (an idea proposed in prior work [25, 40]) and use a theorem prover to scale verification to much larger input ranges. Specifically, we implement our own “outer-loop” version of CEGIS with \( \text{SKETCH} \) as the inner synthesis component: we first use \( \text{SKETCH} \) to find hole assignments over a small input range, and then use the Z3 theorem prover [21] to verify that these assignments are correct for all inputs over a larger range (currently 10-bit integers). If Z3 finds a counterexample, we rerun \( \text{SKETCH} \) by using the counterexample as an additional concrete input on which the specification and sketch must agree, in addition to \( \text{SKETCH} \)’s own small input range.

**Limitations.** First, Chipmunk’s support for immediate ALU operands is preliminary because \( \text{SKETCH} \) cannot synthesize large constants quickly. Hence, we restrict the size of immediate operands; we plan to fix this by leveraging theory-based constant synthesis proposed in recent work [23]. Second, Chipmunk currently assigns one packet field to one fixed container over the entire pipeline, limiting the total number of packet fields in the program to at most the number of containers. However, it is possible to support more packet fields than containers by reusing the same container to store different packet fields in different pipeline stages. Third, running Chipmunk on a real switch such as Tofino [16] requires translating Chipmunk’s holes to low-level switch configurations that can be accepted by the Tofino compiler. We are currently designing such a translator.

## 4 EVALUATION
We compare Chipmunk against the current Domino code generator [58]. The Domino code generator is based largely on classical compiler techniques that use rewrite rules on the abstract syntax tree of the program, e.g., branch elimination and data flow analysis. For benchmarks, we pick a set of test programs drawn from several sources [46, 57, 58]. We then generate code using both Domino and Chipmunk. For a given program and code generator, we measured code generation quality using two metrics: (1) whether the code generator can actually generate code for the program and (2) if it can, the number of stages and the maximum number of ALUs per stage used by the generated code.

**Test programs and ALUs.** We started with 8 programs drawn from multiple sources [46, 57, 58]. Because these programs were previously compiled with Domino, they were written to ensure successful code generation with Domino. Hence, to compare Domino and Chipmunk, we mutated these programs in semantic-preserving ways to generate 10 mutations of each of the 8 programs. In theory, both Domino and Chipmunk should be able to successfully generate code for all these mutations because Domino could generate code for the original 8 programs. For each of the mutations, we used the Domino does use synthesis in a limited form after much preprocessing, but for engineering expediency rather than for code optimization.
stateful ALU that was used to generate code for the original program. We determined this stateful ALU from the sources of the original program [46, 57, 58]. For the stateless ALU, we developed a stateless ALU based on Banzai’s stateless ALU [58], which supports arithmetic, boolean, relational, and conditional operators, similar to RMT.

Results. We supply the $8 \times 10 = 80$ mutations to both Domino and Chipmunk. We report the fraction of the mutations of each of the 8 original programs that Domino and Chipmunk can successfully generate code for (Table 2). On this metric, Chipmunk is significantly better. Domino fails to generate code for most mutations of the original programs because it is unable to fit them within the constraints of the pipeline’s ALUs. Chipmunk’s code generation rate is close to 100%. This is expected because Domino can generate code for the original programs; hence, in theory we should be able to generate code for any semantic-preserving mutations of the originals. In one case (flowlet switching), Chipmunk does not successfully generate code for all mutations. This was because Chipmunk’s code generation times are variable and sometimes exceeded our timeout (Table 2). Increasing our timeout causes Chipmunk to successfully generate code for several failed programs.

When both Domino and Chipmunk are successful, we find that Chipmunk significantly reduces the number of pipeline stages required to fit the program (Figure 5). Such improvements are significant because programmable hardware pipelines are severely constrained on the number of pipeline stages, e.g., Tofino has 12 stages [2]. Chipmunk’s output is comparable to and sometimes worse than Domino on maximum number of ALUs per stage, which is more abundant (e.g., RMT has around 200 per stage [33]). Domino’s resource usage also has more variability across mutations (shown by the error bars) than Chipmunk, which has no variability. Chipmunk’s main drawback relative to Domino (which generates code in a few seconds) is higher and more variable code generation time (Table 2).

## 5 FUTURE WORK

### 5.1 Synthesizing Fast Processor Code

Several processor-based packet-processing substrates have emerged just in the last few years, such as the eXpress Data Path (XDP [22]), DPDK [5], and SoC-based SmartNICs [7, 10]. On these substrates, it is desirable for programs to run with the highest throughput (i.e., NIC line rate) and least packet-processing latency possible. However, it is challenging to tune performance, since it depends on complex factors such as the layout of data structures, memory access patterns, and low-level assembly instructions emitted by compilers.

Program synthesis in the form of superoptimizing compilation [28, 42, 47, 49, 53] has the potential to better this situation. Unlike a standard optimizing compiler that performs local program transformations to improve performance, a superoptimizing compiler searches over the space of instruction sequences to attempt to find an optimal sequence of instructions (according to a stated objective function such as minimum instruction count) implementing the entire input program. The main caveat is that input programs today are restricted to at most a few hundred instructions; on such programs, superoptimizing compilers have been reported to produce code with performance that beats gcc -O3 output [49, 53].

We believe that superoptimizing compilers are a good way to optimize performance of small packet-processing programs on SmartNICs and end hosts, for two reasons. First, superoptimizing compilers can handle multiple instruction set architectures (ISAs) with relatively small engineering effort [48]. Second, the stringent performance requirements for high-speed packet processing at 100 Gbit/s and beyond makes it desirable to squeeze out every last bit of performance, rather than relying on standard compiler optimizations.

Research Questions. Existing superoptimizing compilers use very simple performance models (e.g., number of instructions) to optimize programs running on one processor core. Yet, they take significant amounts of time to emit code. Can we enhance superoptimizers to generate code that runs on multi-core SmartNIC platforms? Can we effectively incorporate memory access

---

We include both multicore SoC SmartNICs and end hosts.
costs, patterns, and data layouts? Can compilation scale to reasonable-size network programs and run within a reasonable time? Can we formulate an intermediate representation like LLVM to support multiple SmartNIC ISAs for superoptimization?

5.2 Approximate Program Synthesis

There are many situations where data plane resources are heavily constrained, necessitating approximate—but fast—packet processing. Examples include sampled statistics, measurement sketches that trade off counter accuracy for line-rate performance and reduced memory, and multi-tenant scenarios where it is essential to pack as many network programs as possible into the switch or NIC [45]. Each such situation today requires developing a custom approach to trade accuracy for resource savings or high performance.

Program synthesis can provide a general method to reduce program resource usage through approximation. Approximate compilers [35, 44, 52, 54] already exist to target hardware with instructions that reduce energy. Recently, a more general approximate program synthesis framework [30, 31] has emerged. This framework has been used to improve the performance of some programs by an order of magnitude [30] while producing approximate results with bounded errors.

Research Questions. Network programs are typically written as functions over a single packet, e.g., in P4. How should one synthesize network programs with bounded inaccuracy over a packet trace, rather than just a single packet? How should we incorporate resource constraints such as memory usage, which depends on the workload (e.g., number of flows) and not just the program? Given a library of high-performance primitives such as counting, hashing, etc., is it possible to synthesize measurement sketches (e.g., count-min sketch) that capture a statistic with guaranteed memory-accuracy tradeoffs?

5.3 Synthesizing Program Repairs

Developers of packet processing programs frequently need to troubleshoot correctness, security, and performance issues with their software. While it is impossible to remove the need for human insight from troubleshooting, we believe it is beneficial to generate human-interpretable repair hints automatically. Yet, there are few avenues today to provide such hints to ease the troubleshooting process.

Small, localized rewrites of the program source code can serve as useful hints to fix many issues. Examples include suggesting edits to a program to fit it into a switch pipeline, rewrites for offending eBPF program code to move past eBPF verification errors [1, 20], and hints to rewrite “hot” code regions of a DPDK program to improve its performance.

Research Questions. Is it possible to generate local rewrites to fit a problematic network program into a packet-processing pipeline? Can we speed up a slow network program by replacing hot code regions with fast implementations using a database of localized code rewrites [28]? Can program synthesis replace unsafe data flows in an eBPF program with safe ones without drastically changing the whole program? It may often be necessary to change the semantics of a program to fix an issue. Can we develop a domain-specific measure of the semantic distance between the rewritten program and the original one? How should a synthesizer use such a measure when suggesting rewrites?

6 RELATED WORK

Program synthesis has been applied to several areas of networking: synthesis of network updates [43, 51], synthesis of routing table configurations from policies [34, 63], the inverse problem of synthesis of policies from configuration [29], and synthesis of control planes [64]. These efforts target synthesis of network-wide policies and configurations, where the policies and configurations pertain to reachability, isolation, and access control. We apply program synthesis to the problem of generating per-device low-level hardware-specific code (e.g., assembly, Verilog, microcode, or FPGA bitstreams) from higher level imperative specifications of packet-processing algorithms.

7 CONCLUSION

Writing fast packet-processing code for programmable network substrates is challenging, and today is best left to experts who deeply understand the underlying hardware. Instead, we propose the use of program synthesis to automatically generate fast packet-processing code. Our initial results are very encouraging. We hope they prompt further research on synthesis-based code generators for programmable network substrates.

ACKNOWLEDGMENTS

We thank Alvin Cheung for many insightful discussions on program synthesis. Armando Solar-Lezama very generously answered several questions on the SKETCH mailing list. Amy Ousterhout, Mina Tahmasbi Arashloo, Vibhaalakshmi Sivaraman, Jennifer Rexford, Michael Walsh, Aurojit Panda, Ravi Netravali, Suvinay Subramanian, Deep Ghosh, and Sharad Chole provided several useful comments on drafts of this paper. The anonymous HotNets reviewers provided many constructive comments that improved the paper’s presentation.
General Low-power Computation. In PLDI, 2011.


A APPENDIX

This appendix presents a simplified version of the sketch generated by Chipmunk for a 2-by-2 grid (Figure 2) and a simple spec. We use ... wherever appropriate to signify that the code is similar to code presented before. The full sketch is available here: https://gist.github.com/XiangyuG/1f009d812151f966b93c1fbf65bc0a09

// num_pipeline_stages = 2
// num_alus_per_stage = 2 (2 stateless ALUs + 2 stateful ALUs)
// num_phv_containers = 2

// Definitions of muxes and ALUs of the switch pipeline
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 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_0_1_mode(int input0,int input1,int mode) {...}
int stateful_alu_0_1_mode(int input0,int input1,int mode) {...}
int stateful_alu_0_1_mode(int input0,int input1,int mode) {...}

// 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 + 0) <= 2);
    assert((salu_active_1_0 + salu_active_1_1 + 0) <= 2);
    assert((salu_active_0_0 + salu_active_1_0 + 0) <= 1);
    assert((salu_active_0_1 + salu_active_1_1 + 0) <= 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 indicator variables.
    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_1_0,omux_phv_0_1_ctrl);
    // Write state
    if (salu_active_0_0 == 1) { state_and_packet.state_0 = state_operand_salu_0_0; }
    if (salu_active_0_1 == 1) { state_and_packet.state_1 = state_operand_salu_0_1; }
    /*********** Stage 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);
}