Autogenerating Fast Packet-Processing Code Using Program Synthesis

Xiangyu Gao
New York University

Taegyun Kim
New York University

Aatish Kishan Varma
New York University

Anirudh Sivaraman
New York University

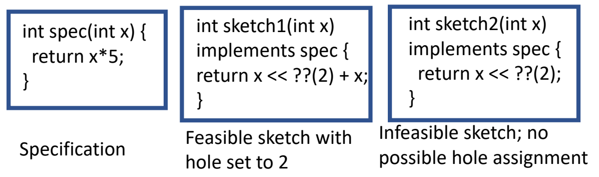
Srinivas Narayana
Rutgers University

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 hardware resources. We also outline future directions in applying program synthesis to code generation for packet processing.

ACM Reference Format:
Xiangyu Gao, Taegyun Kim, Aatish Kishan Varma, Anirudh Sivaraman, and Srinivas Narayana. 2019. Autogenerating Fast Packet-Processing Code Using Program Synthesis. In *ACM Workshop on Hot Topics in Networks (HotNets '19), November 13–15, 2019, Princeton, NJ, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3365609.3365858



```
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);
}
```

Specification
Feasible sketch with hole set to 2
Infeasible sketch; no possible hole assignment

Figure 1: Syntax-guided synthesis in SKETCH. ??(b) **is a hole whose value is 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. These experts must be aware of the cache and memory hierarchy for CPUs and SoC-based NICs; of ALU, TCAM, and SRAM limits for programmable switches; and 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 [36]. Second, while optimizing compilers might alleviate the difficulties of generating fast code, building optimizing compilers for a target requires significant engineering effort spanning decades [48]—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)* [25, 57, 64] that constrains the search space of programs using syntactic restrictions. As a concrete example of syntax-guided synthesis, in SKETCH [17, 59], 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 can be incorporated by limiting the number of sketch holes and using assertions over the holes. For instance, a switch pipeline with a fixed set of ALUs in each stage can be viewed as a sketch with holes representing hardware configurations such as choices of ALU opcodes and immediate operands (Appendix A). Further, the number of holes in the sketch are limited to reflect physical resource constraints on the number of ALUs and stages in the pipeline.

For fast packet processing, there is value to generating machine code that is *near-optimal* on some metric (e.g., throughput or latency). Synthesis-based code generators are much better at finding such near-optimal code relative to traditional optimizing compilers. This is because an optimizing compiler's algorithms are designed to generate consistently good code for all programs within a reasonable compilation time budget; however, synthesis can discover much better code by performing exhaustive search for a longer time. For instance, synthesis techniques have produced x86 and ARM binaries that outperform `gcc -O3` on programs that are a few hundred instructions long [47, 51]. The price of near-optimal code is increased compile time—a price worth paying 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 bits in the hardware configurations (i.e., the number of bits in the SKETCH holes). 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 [59, 64] and promising real-world applications of synthesis [23, 38, 46, 51, 53]. Second, several fast packet-processing programs are naturally small (e.g., BLUE [35], RED [37], or RCP [63]) and simple (e.g., no pointers or loops in P4 [31]). 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; Figure 4).

We present a case study of applying program synthesis to code generation for packet-processing switch pipelines based on the protocol-independent switch architecture (PISA) [15] (§2.2). The compilation of packet-processing programs to such switch pipelines has an *all or nothing* flavor [56]: programs that are compiled successfully run at line rate; all other programs fail to compile. Unlike x86 software, there is no graceful degradation of program performance with complexity. Hence, in practice, it can be a complex process to write a program to "fit" into the underlying switch hardware. Compounding this, existing switch compilers [11, 56] routinely reject 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 design Chipmunk (§3), a synthesis-aided code generator, to generate low-level code for a PISA hardware simulator. We compare Chipmunk with Domino [56] (§4). Chipmunk generates pipelined implementations of many programs that Domino rejects. This is because Domino incorrectly decides that the programs are too expressive to fit into the switch's computational capabilities. For programs that both Domino and Chipmunk can generate code for, code generated by Chipmunk 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 [19] P4-16 [12], POF [60], NPL [10], and

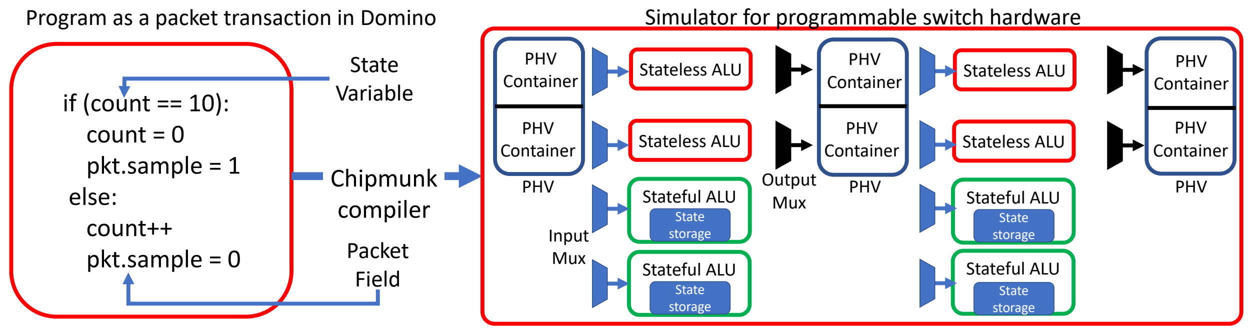


Figure 2: Chipmunk compilation to a 2-by-2 PISA grid. Wires into input, output muxes elided for clarity.

Domino [56]. This paper uses Domino as the language in which the input program is specified by the developer. Domino is well-suited to expressing packet processing with an algorithmic flavor (e.g., implementing RCP over all packets) where the task is compute-heavy, in contrast to forwarding tasks (e.g., tunneling, ACLs) where the task is memory-heavy. Figure 2 shows an example Domino program that samples every 11th packet going through a switch. Domino provides transactional semantics: operations in a Domino program execute from start to finish atomically, as though packets are being processed by the pipeline serially 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 supported by P4-16's `@atomic` construct [3, 13]. Hence, we believe it should be straightforward to extend our work to generate code for algorithmic tasks expressed in P4-16 using the `@atomic` construct.

2.2 Hardware architecture

We consider a packet-processing pipeline based on the hardware architecture described in RMT [32] and Banzai [56]. 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) [15] and is the dominant architecture for high-speed programmable switches today [6, 14].

We simulate a simplified form of PISA by abstracting out all switch computation into a 2D grid of ALUs that process all packets (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*. Each container can be thought of as memory that holds a packet field (e.g., `pkt.sample` in Figure 2) as it is passed and transformed between stages. Similarly, a program's state variables (e.g., `count` in Figure 2) are stored within stateful ALUs.

Configuration	Description
ALU opcode	ALU's operation (e.g., add, sub)
Input mux control	Choice of ALU's input
Output mux control	Where a container's value comes from
Packet field allocation	The container a packet field occupies
State variable allocation	The ALU a state variable occupies
Immediate operands	Constant operands for instructions

Table 1: PISA hardware configurations

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 choice is determined by an input mux (Figure 2). A stateless ALU's output is written into the PHV container designated for that stateless ALU. A stateful ALU's output can be routed into any container; this choice is determined by an output mux. Table 1 summarizes the 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 using SKETCH

We use the SKETCH program synthesizer for developing Chipmunk. We briefly describe SKETCH's internals here; [57] has more details. SKETCH is given as inputs a specification to satisfy and a partial program (the sketch) (Figure 1). We consider both the specification and the

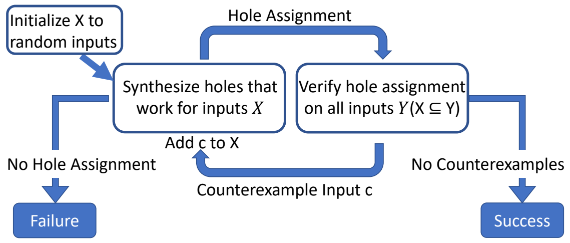

Initialize X to random inputs
Hole Assignment
Synthesize holes that work for inputs X
Verify hole assignment on all inputs Y (X ⊆ Y)
Add c to X
No Hole Assignment
Counterexample Input c
No Counterexamples
Failure
Success

Figure 3: The CEGIS algorithm for synthesis.

sketch to be functions from a switch state and a packet to a new switch state and a packet [26, 56]. 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 [59]:

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

Equation 1 is an instance of the quantified boolean formula problem (QBF) [16]. QBF is a generalization of boolean satisfiability (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 [57]. Hence, SKETCH uses an algorithm called *counterexample-guided inductive synthesis (CEGIS) [58, 59]*, 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:

$$\exists c \in \{0,1\}^m : S(x_1) = P(x_1,c) \land \ldots S(x_k) = P(x_k,c) \quad (2)$$

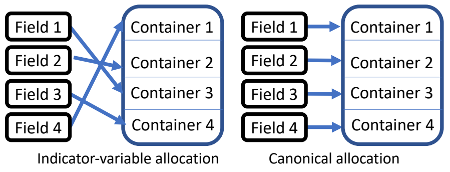


Figure 4: An indicator-variable allocation can be transformed into a canonical allocation.

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

$$\forall x \in \{0,1\}^n : S(x) = P(x,c^*) \quad (3)$$

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.

3 CODE GENERATION FOR PIPELINES

We use SKETCH to build Chipmunk, a PISA code generator that robustly fits packet-processing programs to switch pipelines regardless of how a developer might express her specific program (§1). We 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 [40].

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 input-output behavior 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 various hardware configurations (Table 1). We now describe how Chipmunk sets up and then solves the synthesis problem for SKETCH to generate the set of hardware 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. Currently, Chipmunk 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, and excluding the possibility of reusing the same container to store different packet fields in different pipeline stages. We plan to address this restriction in future work.

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 [27] 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 renumbering containers (Figure 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 hardware. The indicator variable holes $I[s, x, y]$ 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_i is allocated to stateful ALU i within a stage. However, there is an important wrinkle: SKETCH still needs to determine which stage a state variable s_i 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 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 hardware configurations (Table 1), 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 (new switch state and new packet) for all possible inputs (initial values of switch state and the old packet).

To solve this problem, SKETCH uses the CEGIS algorithm described earlier. However, SKETCH limits the range of inputs to speed up synthesis. By default, SKETCH only searches over all 5-bit integers for each scalar input. Hence, it is possible that the hole assignment returned by SKETCH fails to work over larger input ranges, say 32-bit packet fields. To scale SKETCH to larger input ranges, we decouple the input ranges for synthesis and verification (an idea proposed in prior work [24, 39]) and use a theorem prover to scale verification to much larger input ranges. Specifically, we implement our own "outer-loop" version of CEGIS with SKETCH as the inner synthesis component: we first use SKETCH to find hole assignments over a small input range, and then use the Z3 theorem prover [20] 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 SKETCH by using the counterexample as an additional concrete input on which the specification and sketch must agree, in addition to SKETCH's own small input range.

Limitations. First, Chipmunk's support for immediate ALU operands is preliminary because 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 [22]. Second, running Chipmunk on a real switch such as Tofino [14] 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 [56]. 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.[1] For benchmarks, we pick a set of test programs drawn from several sources [45, 55, 56]. 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 [45, 55, 56]. Because these programs were previously compiled with Domino, they

[1]Domino does use synthesis in a limited form after much preprocessing, but for engineering expediency rather than for code optimization.

Programs	Chipmunk	Domino	Chipmunk time (sec)
RCP [63]	100 %	100%	17.7
Stateful Firewall [26]	100 %	90 %	2295
Sampling [56]	100%	0%	7.3
BLUE (increase) [35]	100%	0%	10.7
BLUE (decrease) [35]	100%	0%	52.5
Flowlet switching [54]	70%	100%	3648
Detecting new flows [45]	100%	0%	7.7
Detecting flow reordering [45]	100%	0%	8.3

Table 2: Code generation rate and time for Chipmunk and Domino

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 stateful ALU that was used to generate code for the original program. For the stateless ALU, we developed a stateless ALU based on Banzai's stateless ALU [56], which supports arithmetic, boolean, relational, and conditional operators, similar to RMT.

Results. We supply the 8*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 incorrectly concludes that the programs are too expressive to be implemented using 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

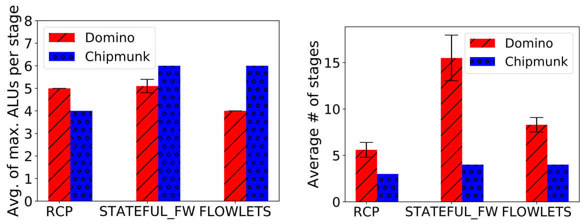


Figure 5: Resources used by Chipmunk, Domino.

sometimes worse than Domino on the maximum number of ALUs per stage, which is more abundant (e.g., RMT has around 200 per stage [32]). 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[2] packet-processing substrates have emerged just in the last few years, such as the eXpress Data Path (XDP [21]), DPDK [5], and SoC-based SmartNICs [7, 9]. 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* [27, 41, 46, 47, 51] 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 [47, 51].

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 costs, patterns, and data layouts? Can compilation scale

[2]We include both multicore SoC SmartNICs and end hosts.

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 [44]. 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 [34, 43, 50, 52] already exist to target hardware with instructions that reduce energy. Recently, a more general *approximate program synthesis* framework [29, 30] has emerged. This framework has been used to improve the performance of some programs by an order of magnitude [29] 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 address resource constraints like 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, 18], 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 [27]? 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 [42, 49], synthesis of routing table configurations from policies [33, 61], the inverse problem of synthesis of policies from configuration [28], and synthesis of control planes [62]. 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 Walfish, 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.

REFERENCES

[1] An eBPF overview, part 1: Introduction. https://www.collabora.com/news-and-blog/blog/2019/04/05/

an-ebpf-overview-part-1-introduction/.

[2] Arista 7170 Multi-function Programmable Networking. https://www.arista.com/assets/data/pdf/Whitepapers/7170_White_Paper.pdf.

[3] Concurrency Model for P4. https://github.com/p4lang/p4-spec/issues/48.

[4] DepQBF Solver. http://lonsing.github.io/depqbf/.

[5] DPDK: Data Plane Development Kit. http://dpdk.org/.

[6] High-Capacity StrataXGS Trident 4 Ethernet Switch Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series.

[7] LiquidIO II Smart NICs. https://www.marvell.com/ethernet-adapters-and-controllers/liquidio-smart-nics/index.jsp.

[8] MarkusRabe/cadet: A fast and certifying solver for quantified Boolean formulas. https://github.com/MarkusRabe/cadet.

[9] Netronome Agilio SmartNICs. https://www.netronome.com/products/smartnic/overview/.

[10] NPL Specification. https://github.com/nplang/NPL-Spec.

[11] P4 Studio | Barefoot. https://www.barefootnetworks.com/products/brief-p4-studio/.

[12] P4_16 Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html.

[13] P4_16 Language Specification Concurrency Model. https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html#sec-concurrency.

[14] Product Brief Tofino Page | Barefoot. https://barefootnetworks.com/products/brief-tofino/.

[15] Programming the Forwarding Plane - Nick McKeown. https://forum.stanford.edu/events/2016/slides/plenary/Nick.pdf.

[16] Quantified Boolean Formula. https://en.wikipedia.org/wiki/True_quantified_Boolean_formula.

[17] Sketch Source Code. https://people.csail.mit.edu/asolar/sketch-1.7.5.tar.gz.

[18] The art of writing eBPF programs: a primer.|Sysdig. https://sysdig.com/blog/the-art-of-writing-ebpf-programs-a-primer/.

[19] The P4 Language Specification. https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf.

[20] The Z3 Theorem Prover. https://github.com/Z3Prover/z3.

[21] XDP - IO Visor Project. https://www.iovisor.org/technology/xdp.

[22] A. Abate, C. David, P. Kesseli, D. Kroening, and E. Polgreen. Counterexample Guided Inductive Synthesis Modulo Theories. In *Computer Aided Verification*, 2018.

[23] M. B. S. Ahmad and A. Cheung. Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications. In *SIGMOD*, 2018.

[24] M. B. S. Ahmad and A. Cheung. Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications. In *SIGMOD*, 2018.

[25] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, 2013.

[26] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *SIGCOMM*, 2016.

[27] S. Bansal and A. Aiken. Automatic Generation of Peephole Superoptimizers. In *ASPLOS*, 2006.

[28] R. Birkner, D. D. Cohen, L. Vanbever, and M. Vechev. Config2Spec: Mining Network Specifications from Network Configurations. In *NSDI*, 2020.

[29] J. Bornholt, E. Torlak, L. Ceze, and D. Grossman. Approximate Program Synthesis. In *Workshop on Approximate Computing Across the Stack*, 2015.

[30] J. Bornholt, E. Torlak, D. Grossman, and L. Ceze. Optimizing Synthesis with Metasketches. In *POPL*, 2016.

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

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

[33] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev. Net-Complete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *NSDI*, 2018.

[34] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural Acceleration for General-Purpose Approximate Programs. In *MICRO*, 2012.

[35] W.-c. Feng, K. G. Shin, D. D. Kandlur, and D. Saha. The BLUE Active Queue Management Algorithms. *IEEE/ACM Transactions on Networking*, 2002.

[36] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*, 2018.

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

[38] S. Kamil, A. Cheung, S. Itzhaky, and A. Solar-Lezama. Verified Lifting of Stencil Computations. In *PLDI*, 2016.

[39] S. Kamil, A. Cheung, S. Itzhaky, and A. Solar-Lezama. Verified Lifting of Stencil Computations. In *PLDI*, 2016.

[40] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy. High Performance Packet Processing with FlexNIC. In *ASPLOS*, 2016.

[41] H. Massalin. Superoptimizer: A Look at the Smallest Program. In *ASPLOS*, 1987.

[42] J. McClurg, H. Hojjat, P. Černý, and N. Foster. Efficient Synthesis of Network Updates. In *PLDI*, 2015.

[43] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard. Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels. In *OOPSLA*, 2014.

[44] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. DREAM: Dynamic Resource Allocation for Software-defined Measurement. In *SIGCOMM*, 2014.

[45] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-Directed Hardware Design for Network Performance Monitoring. In *SIGCOMM*, 2017.

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

[47] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati. Scaling Up Superoptimization. In *ASPLOS*, 2016.

[48] A. D. Robison. Impact of Economics on Compiler Optimization. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, 2001.

[49] S. Saha, S. Prabhu, and P. Madhusudan. NetGen: Synthesizing Data-plane Configurations for Network Policies. In *SOSR*, 2015.

[50] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate Data Types for Safe and General Low-power Computation. In *PLDI*, 2011.

[51] E. Schkufza, R. Sharma, and A. Aiken. Stochastic Superoptimization. In *ASPLOS*, 2013.

[52] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing Performance vs. Accuracy Trade-offs with Loop

Perforation. In *ESEC/FSE*, 2011.

[53] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated Feedback Generation for Introductory Programming Assignments. In *PLDI*, 2013.

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

[55] A. Sivaraman. *Designing Fast and Programmable Routers*. PhD thesis, EECS Department, Massachusetts Institute of Technology, September 2017.

[56] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *SIGCOMM*, 2016.

[57] A. Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[58] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching Concurrent Data Structures. In *PLDI*, 2008.

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

[60] H. Song. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2013.

[61] K. Subramanian, L. D'Antoni, and A. Akella. Genesis: Synthesizing Forwarding Tables in Multi-tenant Networks. In *POPL*, 2017.

[62] K. Subramanian, L. D'Antoni, and A. Akella. Synthesis of fault-tolerant distributed router configurations. *Proc. ACM Meas. Anal. Comput. Syst.*, Apr. 2018.

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

[64] E. Torlak and R. Bodik. Growing Solver-aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 2013.

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/1f009d812151f966b93c1fbf65bc0a69

```
// 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 + 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 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);
}
```