# Autogenerating Fast Packet-Processing Code Using Program Synthesis
Paper **#153**

## ABSTRACT

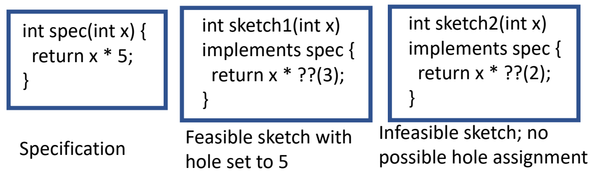
Packet-processing code should be fast. But, it is hard to write fast code for programmable substrates such as high-speed switches, multicore SoC SmartNICs, FPGAs, middleboxes, and the end-host stack. Today, expert developers with deep familiarity of the underlying hardware hand-craft such code. Making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

We propose an alternative: to automatically generate fast packet-processing code using *program synthesis.* For the domain of fast packet processing, synthesis-based code generation can generate faster code than an optimizing compiler at the cost of increased compile time. We outline several potential applications of synthesis to code generation for packet processing. As a case study, we apply program synthesis to build a compiler for a switch pipeline simulator. Our compiler compiles many programs that the previous compiler rejects and uses significantly fewer pipeline resources.

## 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 a server. With growing link speeds, there is a need to run ever-faster packet-processing code on these substrates. For example, code running on a SmartNIC must run at high line rates of 40–100 Gbit/s and beyond. Programmable 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 remains surprisingly hard, for two reasons. First, developing such fast code requires handcrafting and optimization by experts who are familiar with each underlying hardware architecture. This requires awareness of the cache and memory hierarchy for CPUs and SoC-based NICs; awareness of ALU, TCAM, and SRAM limits for programmable switches; and 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 [7]. Second, while optimizing compilers might alleviate programming difficulties, building optimizing compilers for a substrate requires significant engineering effort spanning decades [44]—effort that may not be available for new, niche, and evolving network hardware.



```
int spec(int x) {
  return x * 5;
}
```

```
int sketch1(int x)
implements spec {
  return x * ??(3);
}
```

```
int sketch2(int x)
implements spec {
  return x * ??(2);
}
```

Specification
Feasible sketch with hole set to 5
Infeasible sketch; no possible hole assignment

Figure 1: Syntax-guided synthesis in SKETCH. `??(b)` is a hole that can hold a value between 0 and $2^b - 1$.

This paper proposes 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)* [18, 52, 58] that constrains the search space of programs using syntactic restrictions. As a concrete example of syntax-guided synthesis, in SKETCH [54, 12], 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 either filling in all holes with concrete values so that the completed sketch meets the specification, or saying that the 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 appropriately constraining the sketch (e.g., a limit on the total number of assembly instructions). For example, 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.

Synthesis-based compilers 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 time budget. For fast packet-processing, however, there is great value to generating *near-optimal code*, which synthesis can discover 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 [43, 47]. 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: synthesis allows us to declaratively specify code generation for different substrates as synthesis problems, e.g., using sketches, allowing us to reuse synthesis technology for code generation across many diverse and fast-changing packet-processing substrates.

Despite these benefits over optimizing compilers, syntax-guided synthesis faces a key challenge: it is a search problem over a large combinatorial 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 [54, 58] and promising real-world applications of synthesis [49, 41, 16, 31, 47]. Second, several fast packet-processing programs are naturally small (e.g., BLUE [28], RED [29], or RCP [57]), making synthesis more manageable. Third, many hardware substrates exhibit significant symmetry, allowing us to prune the search space by considering only one exemplar hardware configuration out of a large set of equivalent configurations (§3; Fig. 3).

We describe four applications of program synthesis to generating fast packet processing code and the research questions they raise (§2): fitting programs to high-speed pipelines, optimizing program performance on processors, automatically approximating programs to run faster, and providing performance troubleshooting hints to developers. We report on a preliminary case study (§3) of the first application. We design Chipmunk, a synthesis-aided compiler to generate low-level code for a hardware simulator of a programmable switch pipeline called Banzai [3, 51]. We compare Chipmunk with Domino [51], and show that Chipmunk can generate high-speed pipelined implementations of programs that Domino rejects because Domino incorrectly decides that the programs cannot satisfy the constraints imposed by the programmable switch. For programs that both Domino and Chipmunk successfully compile, Chipmunk produces code with much smaller pipeline depth.

## 2. APPLICATIONS OF SYNTHESIS TO CODE GENERATION

### 2.1 Synthesizing for High-Speed Pipelines

The compilation of expressive network programs to high-speed switch pipelines [25, 4] has an *all or nothing* flavor: programs that are compiled successfully run at line rate, and all other programs fail compilation entirely. Unlike x86 software, there is no option to run with degraded performance. In our experience developing programs for switch-like substrates, namely the Banzai machine model [3, 51] and Barefoot's Tofino chip [11], it can be a complex process to 'fit' a network program to these substrates. When a program fails to compile, the developer requires intricate knowledge of the hardware to interpret a compilation error. Compounding this, existing switch compilers [30, 10, 51] may unduly fail to compile a program even when a semantically-equivalent rewritten program can fit the pipeline, pushing the burden of finding such a rewrite onto the developer.

***Research Questions.*** Can we build a switch compiler that is *complete*, i.e., if there is any way to compile a program to hardware, the compiler should do so, regardless of how the program is written? As we will show in our case study (§3), our synthesis-aided compiler, Chipmunk, is able to compile many semantic-preserving mutations of an original program that can itself be compiled. Hence, a user is relieved of the burden of manually tweaking their program to fit the hardware, even when programming in a high-level language. Moving beyond switch pipelines, can we use synthesis techniques to generate code for other pipelines, such as FPGAs [6, 7] and RMT pipelines within NICs [33]?

### 2.2 Synthesizing Fast Processor Code

Several processor-based[1] packet-processing substrates have emerged just in the last few years, such as the eXpress Data Path (XDP [14]), DPDK [2], and SoC-based SmartNICs [8, 5]. 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 instructions emitted by compilers.

Program synthesis in the form of *superoptimizing compilation* [36, 47, 43, 41, 20] has the potential to better this situation. Unlike a standard optimizing compiler that performs *local* program transformations to improve performance, a superoptimizing compiler seeks to find the *optimal* sequence of instructions (according to a

[1]We include both multicore SoC SmartNICs & end hosts.

stated objective function) 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, 43].

We believe that superoptimizing compilers are a great fit to optimize performance of small amounts of packet-processing code over SmartNICs and end hosts, for two reasons. First, superoptimizing compilers can handle multiple instruction set architectures (ISAs) with relatively small engineering effort compared to traditional compilers, which must be heavily re-engineered for each ISA. Second, the stringent performance requirements for high-speed packet processing at 100 Gbit/s and beyond makes it desirable to squeeze every last bit of performance, rather than just 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 take advantage of execution on multiple parallel cores? Can we effectively incorporate memory access costs, patterns, and data layouts? Can compilation scale to reasonable-size network programs and run within a reasonable time? Could one formulate a flexible intermediate representation like LLVM to support multiple ISAs for superoptimization?

## 2.3 Synthesizing Approximate Programs

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 [38]. 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 [46, 37, 48, 27] already exist to target hardware with instructions that reduce energy. Recently, a more general *approximate program synthesis* framework [24, 23] has emerged. This framework has been used, among other things, to improve the performance of some programs by an order of magnitude [23] while producing approximate results.

***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, incrementing, etc., is it possible to synthesize measurement sketches (e.g., count-min sketch) that capture a statistic with guaranteed memory-accuracy tradeoffs?

## 2.4 Synthesizing Program Repairs

Developers of network data plane code 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 an all-or-nothing pipeline, rewrites for offending eBPF program code to move past eBPF verification errors [14], 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 [20]? 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?

## 3. CASE STUDY: SYNTHESIS FOR SWITCH PIPELINES

We now expand on the first application from the previous section. We use SKETCH to build Chipmunk, a compiler for Banzai [51], a hardware simulator for programmable switch pipelines. We briefly explain Banzai; [51] has a more detailed explanation. We then describe how Chipmunk synthesizes the configuration of a

| 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: Banzai hardware configurations

Banzai machine to implement the program supplied to Chipmunk. While we developed Chipmunk in the context of a switch pipeline, we believe Chipmunk's techniques are also applicable to similar NIC pipelines [33].

## 3.1 The Banzai machine model

Banzai extends RMT [25] with stateful computation. It provides a simple model of programmable switch hardware 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. 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 stage. Similarly, a program's state variables are stored within stateful ALUs.

ALUs are Banzai'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. Banzai allows us to experiment with different kinds 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 Banzai's configurations.

## 3.2 The Chipmunk compiler

Chipmunk compiles programs expressed as packet transactions in the Domino language [51]: all operations in the program execute atomically on each packet from start to finish, similar to P4's @atomic construct [9]. In addition to the 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. The specification for the sketch is the packet transaction; the holes are various 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 need to be allocated to PHV containers. 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: 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 [21] 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 (Fig. 3); 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 $I[s, x, y]$ tracks if state variable $s$ is assigned to stateful ALU $y$ in stage $x$. 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 $s_i$ can be 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. 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 settings to ALUs.*** We use SKETCH holes to represent the opcode used by each ALU and the mux controls. The size of opcode holes depends on an ALU's expressiveness; the size of mux holes depends on the number of PHV containers and the pipeline width. In experiments, we find that constraining opcode holes to take on fewer values than the hardware allows

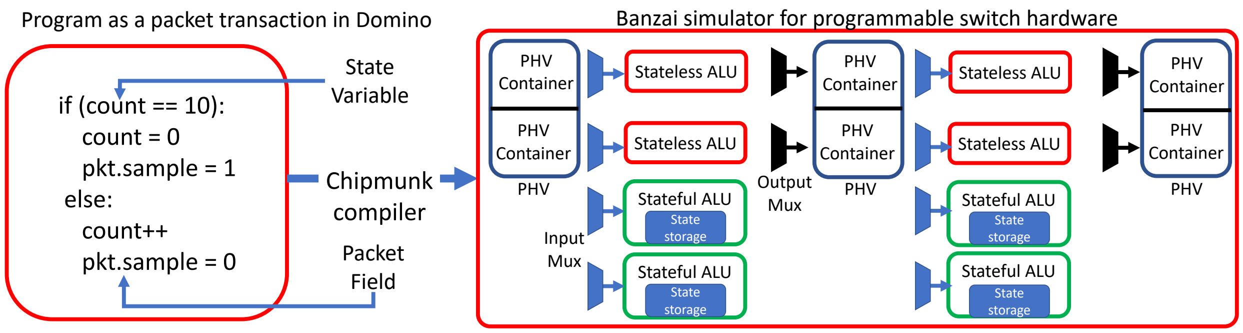


Figure 2: Chipmunk compilation to a 2-by-2 Banzai grid. Wires into input, output muxes are elided for clarity.

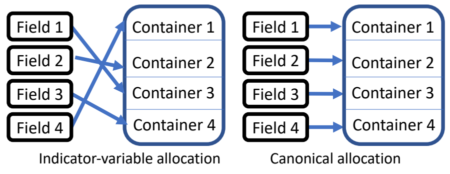
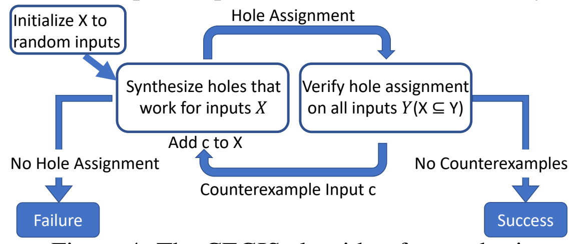


Figure 3: An indicator-variable allocation can be transformed into a canonical allocation.



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
No Counterexamples
Counterexample Input c
Failure
Success

Figure 4: The CEGIS algorithm for synthesis

can sometimes speed up synthesis (for example, 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 indeed 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, SKETCH's problem can be stated as: find an assignment of values to all holes so that the sketch and the specification program have the same output for all possible input packets and initial values of state.

To solve this problem, SKETCH uses an algorithm called *counterexample-guided inductive synthesis* (CEGIS, Fig. 4) [54, 53]. The CEGIS algorithm is initialized with a set of random packet and state inputs. Then, CEGIS iteratively alternates between (1) finding holes that make the sketch produce the same outputs as the specification for its current set of inputs, and then (2) verifying that those hole assignments work for all inputs. If verification fails, SKETCH's built-in verifier generates a counterexample, which is then added to its input set, and the cycle of synthesis and verification repeats. The algorithm terminates when either the verification succeeds or the synthesis fails.

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 sizes, we decouple the input ranges for synthesis and verification (an idea proposed in prior work [17, 32]) and use a theorem prover to scale verification to much larger input ranges. Specifically, we implement our own 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 [13] to verify that these holes 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 its own small input set.

***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 implementing theory-based constant synthesis proposed in recent work [15]. 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 [11] requires translating Chipmunk's holes to low-level switch configurations. We have not yet designed a translator.

## 4. CASE STUDY EVALUATION

We now compare our synthesis-based compiler,

| Programs | Chipmunk | Domino | Compile time (sec) |
|---|---|---|---|
| RCP [57] | 100 % | 100% | 17.7 |
| Stateful Firewall [19] | 100 % | 90 % | 2295 |
| Sampling [51] | 100% | 0% | 7.3 |
| BLUE (increase) [28] | 100% | 0% | 10.7 |
| BLUE (decrease) [28] | 100% | 0% | 52.5 |
| Flowlet switching [50] | 70% | 100% | 3648 |
| Detecting new flows [39] | 100% | 0% | 7.7 |
| Detecting flow reordering [39] | 100% | 0% | 8.3 |

Table 2: Compilation rate and time for Chipmunk

Chipmunk, against the current compiler for Banzai, Domino [51]. Domino is based largely on classical compiler techniques using rewrite rules on the abstract syntax tree of the program, e.g., branch elimination and data flow analysis.[2] For comparisons, we pick a set of test programs drawn from several sources [39, 51, 34] and compile them with both Domino and Chipmunk. We measured compilation quality using two metrics: (1) whether the same program is compiled or rejected by either compiler and (2) if compiled, the number of stages used and the maximum number of ALUs used per stage.

***Test programs and ALUs.*** We started with 8 programs drawn from multiple sources [39, 51, 34]. Because these programs were previously compiled with Domino, they were written to facilitate successful compilation 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 that should all in theory be successfully compiled because the original 8 programs could be successfully compiled. For each of the mutations, we used the stateful ALU that was reported to successfully execute the original program; we determined this from the sources of the original program [39, 51, 34]. For the stateless ALU, we developed a stateless ALU implicitly used by Domino [51], which supports arithmetic, boolean, relational, and conditional operators.

***Results.*** We supply the 8*10=80 programs 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 compile (Table 2). On this metric, we find that Chipmunk is significantly better. Domino fails to compile most mutations of the original programs due to its limited support for program optimization. Chipmunk's compilation rate is close to 100%, which is expected because the original programs do compile to Banzai. In the one case (e.g., flowlet switching) where Chipmunk does not successfully compile, this was because Chipmunk's compilation times are variable and sometimes exceeded our timeout. Increasing the timeout causes Chipmunk to successfully compile several failed programs.

[2]Domino does employ synthesis in a limited form, but for engineering expediency rather than for code optimization.

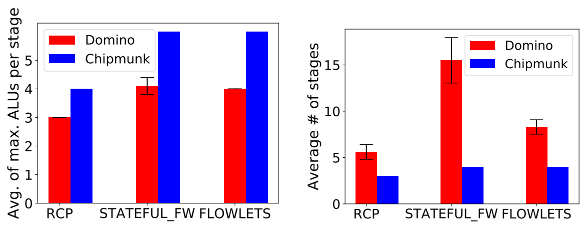


Figure 5: Resources consumed by Chipmunk, Domino.

When both Domino and Chipmunk successfully compile a program, we find that Chipmunk's output significantly reduces the number of pipeline stages (Figure 5), which is a scarce resource on programmable switches, e.g., Tofino has 12 stages [1]. Chipmunk's output is comparable and sometimes worse than Domino on maximum number of ALUs per stage, which is considerably more abundant (e.g., RMT has around 200 ALUs per stage [25]). Chipmunk's main drawback relative to Domino, whose compilation time is a few seconds, is increased and more variable compile time (Table 2).

## 5. RELATED WORK

Program synthesis has been applied to several areas of networking: synthesis of network updates [40, 45], synthesis of routing table configurations from policies [26, 55], the inverse problem of synthesis of policies from configuration [22], and synthesis of control planes [56]. 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. Several programming frameworks exist for NICs [42, 35]. Our work is complementary and provides efficient code generation for these frameworks.

## 6. CONCLUSION

Writing fast packet-processing code for programmable network substrates is challenging, and today is best left to experts with a deep understanding of 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 compilers for programmable network substrates.

## 7. REFERENCES

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

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

[3] https://github.com/packet-transactions: A machine model for line-rate programmable switches. https://github.com/packet-transactions/banzai.

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

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

[6] Mellanox BlueField SmartNIC 25Gb/s Dual Port Ethernet Network Adapter. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf.

[7] Microsoft bets big on sdn. https://azure.microsoft.com/en-us/blog/microsoft-showcases-software-defined-networking-innovation-at-sigcomm-v2/.

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

[9] P4-16 language specification. https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html.

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

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

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

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

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

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

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

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

[18] 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.

[19] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker. SNAP: Stateful network-wide abstractions for packet processing. In *SIGCOMM*, 2016.

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

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

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

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

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

[25] 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.

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

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

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

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

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

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

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

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

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

[35] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. iPipe: A Framework for Building Distributed Applications on Multicore SoC SmartNICs. In *SIGCOMM*, 2019.

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

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

[42] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *OSDI*, 2018.

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

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

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

[46] 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.

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

[48] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing Performance vs. Accuracy Trade-offs with Loop Perforation. In *ESEC/FSE*, 2011.

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

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

[51] 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.

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

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

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

[55] K. Subramanian, L. D&#039;Antoni, and A. Akella. Genesis: Synthesizing Forwarding Tables in Multi-tenant Networks. In *POPL*, 2017.

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

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

[58] 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.