CaT: A Solver-Aided Compiler for Packet-Processing Pipelines

Xiangyu Gao
xg673@nyu.edu
New York University, USA

Divya Raghunathan
dr31@cs.princeton.edu
Princeton University, USA

Ruijie Fang
rjf@abstractpredicates.org
Princeton University, USA

Tao Wang
tw1921@nyu.edu
New York University, USA

Xiaotong Zhu
xz2532@nyu.edu
New York University, USA

Anirudh Sivaraman
anirudh@cs.nyu.edu
New York University, USA

Srinivas Narayana
sn624@cs.rutgers.edu
Rutgers University, USA

Aarti Gupta
aartig@cs.princeton.edu
Princeton University, USA

ABSTRACT

Compiling high-level programs to high-speed packet-processing pipelines is a challenging combinatorial optimization problem. The compiler must configure the pipeline's resources to match the semantics of the program's high-level specification, while packing all of the program's computation into the pipeline's limited resources. State of the art approaches tackle individual aspects of this problem. Yet, they miss opportunities to produce globally high-quality outcomes within reasonable compilation times.

We develop a framework to decompose the compilation problem for such pipelines into three phases—making extensive use of solver engines (e.g., ILP, SMT, and program synthesis) to simplify the development of these phases. *Transformation* rewrites programs to use more abundant pipeline resources, avoiding scarce ones. *Synthesis* breaks complex transactional code into configurations of pipelined compute units. *Allocation* maps the program's compute and memory to the pipeline's hardware resources.

We prototype these ideas in a compiler, CaT, which targets (1) the Tofino programmable switch pipeline and (2) Menshen, a cycle-accurate simulator of a Verilog description of the RMT pipeline. CaT can handle programs that existing compilers cannot currently run on pipelines and generates code faster than existing compilers, where the generated code uses fewer pipeline resources.

CCS CONCEPTS

• **Networks** → **Programmable networks**; **In-network processing**.

KEYWORDS

Programmable switches; program synthesis; code generation; packet processing pipelines; integer linear programming

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9918-0/23/03…$15.00
https://doi.org/10.1145/3582016.3582036

ACM Reference Format:
Xiangyu Gao, Divya Raghunathan, Ruijie Fang, Tao Wang, Xiaotong Zhu, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 2023. CaT: A Solver-Aided Compiler for Packet-Processing Pipelines. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3582016.3582036

1 INTRODUCTION

Reconfigurable packet-processing pipelines (e.g., RMT [27]) are emerging as important programmable platforms. They are embodied in many programmable high-speed switches and network interface cards (NICs) such as the Tofino [9], Trident [4], and Jericho switches [3]; the Pensando SmartNIC [1]; and Intel IPUs [8].

Programmable pipelines are organized into multiple stages, where each stage processes one packet in parallel, and hands it off to the next stage (§2.1). Each stage contains memory blocks to hold tables containing packet-matching rules and state (e.g., counters) maintained across packets. Header fields are extracted from packets to match the table rules. Once the packet's fields are matched against a rule, the packet or state can also be updated using an action.

P4 [12] is emerging as a popular language to program these pipelines. P4 offers the ability to parse packets according to custom header definitions, and specify the match types and actions on parsed packets. A P4 action may modify packet headers and state.

The compilation problem. The networking community has developed P4 programs targeting programmable pipelines for several research [24, 38, 47, 48] and production [6, 40, 45, 50] use cases. To enable these use cases, a compiler must translate P4 programs to pipeline configurations. This compiler must solve a combinatorial optimization problem with several challenging aspects to it:
(1) Multiple resource types: There are multiple pipeline resources, with some resources being scarce, e.g., pipeline stages, and others being abundant, e.g., arithmetic logic units (ALUs). Some resources must be allocated hand in hand (e.g., match memory and ALUs).
(2) Transactional guarantees: P4 actions can be annotated to have transactional guarantees [13, 18]: executing to completion on each packet before processing the next one. If such a transactional P4 action requires multiple pipeline stages, the compiler must be able to split the action into multiple ALUs and stages, ensuring the implementation respects the action's transactional semantics [53].

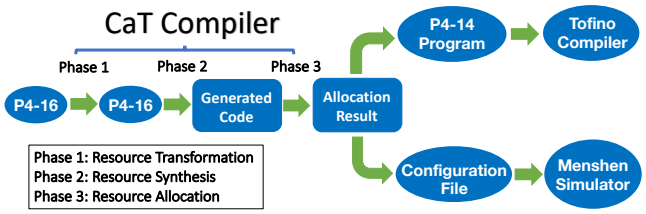

Figure 1: The workflow of the CaT compiler.

(3) All-or-nothing fit: A program for a high-speed pipeline can either run at the pipeline's highest throughput (typically line rate above a minimum packet size), or cannot run at all. Thus, it is important to "pack" all of the P4 program into the pipeline's limited resources.

Prior work has tackled several individual aspects of this compilation problem (§2.2). Such an approach loses opportunities to globally reduce resource usage (e.g., stages), which is necessary to fit complex programs on a pipeline. However, it is challenging to solve a single combinatorial optimization problem. Our goal is to find a good decomposition of the large problem into smaller pieces, enabling global optimization of resource usage, while keeping each piece small enough to solve efficiently.

Our approach. In this paper, we present an end-to-end compiler, CaT[1], that unifies prior approaches and translates high-level P4 programs into a low-level representation suitable for pipelined execution. We take inspiration from *high-level synthesis* (HLS)—a technology for improving productivity of hardware design for ASICs [16] and FPGAs [7].

Informally, HLS [29] takes as input a high-level algorithmic description of the hardware design with no reference to clocks or pipelining, and with limited parallelism in the description. An HLS compiler then progressively lowers this high-level description down to an optimized hardware implementation, pipelining the implementation if possible, executing multiple computations in parallel, scheduling computations in time, and converting these computations into a register-transfer level (RTL) design.

We believe such an approach to developing compilers targeting packet-processing pipelines will raise the user's level of programming abstraction, while retaining the performance of low-level pipeline programming. For a user developing algorithmic programs in P4 (such as those used for in-network computation, e.g., [38, 51, 62]), such an approach eliminates the labor of manually breaking the high-level algorithmic computation into actions spread over many pipeline stages (§3).

The workflow of our compiler, CaT, is shown in Figure 1. It consists of three phases. The input consists of P4 code containing tables that match on specific headers and action code blocks that modify packet headers and state. The action code blocks may be written without regard to their feasibility within a single pipeline stage. The first phase of CaT employs *resource transformations* that rewrite a high-level P4 program to another semantically-equivalent high-level P4 program; these rewrites are used to transform a computation's use of one scarce resource to its use of a relatively abundant resource, and potentially reduce the number of stages as well. The second phase performs *resource synthesis* to lower transactional blocks of statements in the high-level P4 program to a lower-level program suitable for hardware execution. In this step, individual ALUs in hardware are configured to realize the programmers' intent in the transactional action blocks, while respecting the ALUs' computational limits. The third phase performs *resource allocation* to allocate the computation units corresponding to the lowered program to physical resources such as ALUs and memory in the pipeline. Notably, our compiler workflow works within the confines of the widely used P4 ecosystem without requiring the development of a new domain-specific language (DSL) for packet processing.

Our contributions. The main technical contribution of CaT's three-phase approach is the modularization of the large combinatorial optimization problem of compilation into smaller problems, whose solutions still enable a high-quality global result (§7). These smaller problems can also be fed to solver engines, simplifying the process of solving them. Additionally, we improve upon the state of the art and introduce new techniques in each phase. In particular, our resource transformations (§4) are driven by a novel *guarded dependency analysis* that identifies false dependencies between computations, exposing more parallelism opportunities when rewriting programs to use more abundant resources. Our resource synthesis phase (§5) uses a novel synthesis procedure that quickly finds pipelined solutions with good-quality results for complex actions. It separates out stateful updates from stateless updates, to decompose a large program synthesis problem into smaller and more tractable subproblems; each subproblem uses a program synthesis engine (SKETCH) as a subroutine. Stateless code is synthesized into a minimum-depth computation tree, i.e., with the minimum number of stages. In comparison to prior work [34], this new synthesis algorithm allows CaT to handle many large actions, in a much shorter time, and with fewer computational resources needed for compilation. Finally, our resource allocation phase (§6) uses a constraint-based formulation that extends prior work [39] to handle complex multi-stage transactional actions; this formulation can be fed to either an ILP or SMT solver. Our techniques can support general P4 programs (including @atomic constructs [13]) efficiently, including programs translated into P4 from higher-level DSLs developed for pipeline programming [33, 34, 37, 53, 56].

Our prototype of CaT can target: (1) the Tofino pipeline, and (2) an open-source RMT pipeline called Menshen (that was previously implemented on an FPGA) [10, 61]. Existing commercial switches have proprietary instruction sets that preclude the kind of low-level resource allocation and control over ALU configurations needed by CaT. Therefore, our backend for Tofino [9] generates low-level P4 in lieu of machine code. To evaluate CaT in full generality, we extend Menshen's open-source register-transfer level (RTL) Verilog model with additional resources for our experiments. We generate code for the cycle-accurate simulator of Menshen, and also use it for testing the CaT prototype. Our results (§7) show that CaT can automatically compile programs that previously required manual changes to be accepted by the Tofino compiler. On other challenging benchmarks, CaT produces good quality code and does so about 3 times faster (on average) than prior work [34].

[1]CaT stands for **C**ode Generation **a**nd **T**able Allocation.

2 BACKGROUND AND RELATED WORK

2.1 Packet-Processing Pipelines

The compiler target in this paper is a programmable packet-processing pipeline following the Reconfigurable Match Tables (RMT) architecture [27]. Such pipelines are present in commercially available programmable switches such as the Barefoot Tofino, Broadcom Trident, and Mellanox Spectrum, and NICs such as the Pensando DPU. An RMT-style pipeline consists of (i) a programmable packet parser and (ii) a number of processing stages structured around match-action computation. We describe these components below.

A programmable parser takes in a programmer-specified header specification, and extracts packet header fields. This set of fields is termed the *packet header vector (PHV)*. PHV fields can be both read and written in each pipeline stage, termed a match-action stage. One match-action stage extracts relevant fields from the PHVs using a crossbar circuit. The fields are then matched against user-inserted rules in stage-local match memory. The memory may also contain *state*, i.e., values maintained on the switch and updated by every packet, such as a packet counter. Once a packet matches a rule, a corresponding set of actions is invoked. The actions are implemented using Very Long Instruction Word (VLIW) ALUs which may modify multiple PHV fields in one shot. Some match-action tables may be skipped entirely (e.g., due to control flow) through hardware components called *gateways*.

Three factors limit the available resources and expressiveness of packet-processing pipelines. First, to support high throughput (e.g., 6.5 Tbit/s in Tofino), pipelines are clocked at high frequencies (e.g., 1 GHz for Tofino). Thus, the pipeline must admit a new packet every clock cycle. Hence, stateful computations (read-modify-write) must finish in one clock cycle. Second, on-chip and stage-local memories are limited in size, to support fast lookup. Third, constraints on chip area and power limit the number of pipeline stages (e.g., 12 match-action stages in Tofino) and control circuitry (e.g., number of gateways and crossbars). Such exacting hardware constraints pose compiler challenges. Furthermore, program behavior is *all-or-nothing*: a program that fits into the pipeline resources would run at the pipeline's clock frequency; otherwise it cannot be run. There is no graceful degradation between these extremes.

2.2 Related Work

There has been significant interest in developing compilers and domain-specific languages (DSLs) for packet-processing pipelines. We categorize the existing compiler efforts based on their support for program rewriting, code generation, and resource allocation.

DSLs for programming packet pipelines. P4 and NPL are the most widely used languages to program packet pipelines. They share many syntactic and semantic aspects. Several academic projects have proposed new DSLs or extensions to P4 to remedy many of P4's shortcomings. For instance, microP4 [57] adds modularity to P4. Lyra [33] addresses the issue of portability of programs across multiple devices. Lyra and FlightPlan [58] address the problem of partitioning a program automatically across multiple devices. Lucid [56] introduces an event-driven programming model for control applications in the data plane. P4All [37] extends P4 to support 'elastic' data structures, whose size can grow and shrink dynamically based on the availability of switch resources. Domino [53] is a DSL that supports transactional packet processing: a programmer specifies a block of code that is executed on each packet in isolation from other packets. These languages can all be translated into P4, and in this paper, we directly take P4 programs as our starting point. Thus, our work is complementary to work on such new DSLs. One limitation of CaT is that it does not currently handle the problem of partitioning a *network-wide* program into per-device programs, like Lyra and Flightplan. Instead, our goal is to build a high quality compiler that inputs a P4 program for a *single device* and outputs a high-quality implementation for that device.

Program rewriting. The open-source reference P4 compiler [11], which is the foundation for most P4 compilers including the widely-used Tofino compiler [9], employs rewrite rules to turn an input P4 program into successively simpler P4 programs. These rewrite rules consist of classical optimizations like common sub-expression elimination and constant folding. Rewrite rules are also employed by Cetus [44] and Lyra [33] to merge tables in different stages (under certain conditions) into a single "cartesian-product" table in a single stage, thereby saving on the number of stages. CaT uses rewrite rules to transform uses of scarce resources (gateways, stages) to more abundant ones (tables, memory, ALUs), in a style similar to Cetus.

Code generation for complex actions. Domino [53] and Chipmunk [34] tackle the problem of *code generation*: selecting the right instructions (i.e., ALU opcodes) for a program action expressed in a high-level language. These compilers have to respect the limited capabilities of each stage's VLIW ALUs while correctly implementing state updates according to @atomic semantics for transactions (§2.1). Domino largely uses rewrite rules and employs program synthesis to code-generate just the stateful fragments in the action, but minor semantic-preserving modifications to programs can cause compilation to fail. Chipmunk addresses this drawback of Domino by using program synthesis to exhaustively search for ALU configurations that could implement a high-level program, but at the expense of high compile time. Lyra [33] uses *predicate blocks*, chunks of code predicated by the same path condition, to break up algorithmic code into smaller blocks that have only inter-block (but no intra-block) dependencies. CaT's resource synthesis is faster than Chipmunk's and more reliable than Domino's (Table 5, §7.3). It generalizes Lyra's predicate block approach by considering ALUs expressed via a parameterizable grammar, such that the procedure is independent of the operations in the program's source code or intermediate representation.

Resource allocation. The problem of allocating specific resources required by a P4 program (e.g., match memory blocks, a specific number of ALUs, etc.) can be posed as an integer linear programming problem (ILP) [37, 39] or as a constraint problem [33] for Satisfiability Modulo Theory (SMT) solvers [23]. If the constraints of the hardware are modeled precisely, ILP-based techniques can improve resource allocation relative to greedy heuristics for resource allocation. To this end, CaT's resource allocation (§6) uses a fine-grained constraint-based formulation that models detailed pipeline resources and enables global optimization by considering dependencies across tables as well as within actions.

Table 1: CaT unifies prior work in P4 compilers (first column) to provide and improve various features (listed in other columns) in an end-to-end flow, and does so within the context of the P4 language without needing a new DSL.

Project	Program Rewriting	Code Generation	Resource Allocation	Retargetability		New Language Constructs
				Instruction Sets	Resource Constraints	
Domino [53]	Yes	Rewriting, some program synthesis		Atom templates		Packet transactions
Chipmunk [34]		Program synthesis		ALU DSL		Packet transactions
Lyra [33]	Yes		SMT		SMT constraints	Network-wide programs
Flightplan [58]		Resource rules	Resource rules		Resource rules	Network-wide programs
Cetus [44]	Yes		Table Merging, PHV Sharing		SMT constraints	
P4All [37]			ILP		ILP constraints	Elastic data structures
Jose et al. [39]			ILP		ILP constraints	
Lucid [56]		Memops		Memops		Event-driven programming
Tofino compiler [9]	Yes	Yes	Heuristics			
CaT (this work)	Yes	Min-depth tree synthesis	ILP/SMT	ALU grammars	ILP/SMT constraints	P4's atomic construct

Table 2: Detailed relationship of the 3 phases of CaT with prior work on compilers, HLS, and packet-processing pipelines.

CaT compiler phase	CaT technique	Builds on prior work	Differences in CaT	Other complementary work
1: Resource transformation	Rewrite rules	LLVM [43], HLS [7, 29], p4c [11]	Rewrite rules target RMT, based on novel guarded dependency analysis	p4c [11] uses platform-independent rewrites, Cetus [44] merges tables
2: Resource synthesis	Mapping operations to ALU pipeline	HLS operation binding [7, 29]	Stateful updates restricted to 1 stage	Lucid [56] uses syntactic rules to ensure operations map to Tofino
	Synthesis procedure uses SKETCH queries for program synthesis	Chipmunk [34]	Novel synthesis procedure: faster, more scalable, uses smaller queries	
	Target portability: via parameterizable grammars for ALUs	Sketch [55], Chipmunk [34]	Generate resource graph (used in Phase 3), not low-level ALU configs	
	Preprocessing: branch removal, SSA, SCC in computation graph Simplifications: const prop, expr simplification, deadcode elimination	Domino [53], SSA [30], VLIW [42] LLVM [43]	No backward control flow (similar to Domino) No backward control flow	
3: Resource allocation	Constraints for match memories	Jose et al. [39], Lyra [33]	Associates match memories with corresponding action resources (ALUs)	
	Constraints for multi-stage actions	HLS scheduling [29], Domino [53], Chipmunk [34]	Uses result of Phase 2 for intra-action dependencies and ALU output propagation	
	Constraints for multiple transactions	Domino [53], Chipmunk [34] handle a single transaction only	Enforces inter-table and intra-action dependencies for global optimization	
	Modeling real hardware constraints in backend FPGA target	Menshen provides a FPGA backend target [61]	Extended functionality of resources in comparison to Menshen	Tofino compiler uses heuristics

3 CAT: MOTIVATION AND OVERVIEW

Motivation. Today, P4 developers typically write down actions in P4 programs with the assumption that each action must finish in one stage. However, tracking the hardware-level feasibility of an action leads to thinking at an unnecessarily low level of abstraction, especially when developing high-speed algorithmic code. Consider the example pseudocode (motivating example ME-1) shown in Figure 2. This function implements the SipHash algorithm, used as a hash function to prevent collision-based flooding attacks [22]. The developer of a P4 version of this algorithm (distinct from the authors of this paper) started with a high-level transactional description of the algorithm (Table 3, [62]). The developer then *manually* changed it into a pipelined implementation (Table 4, [62]), because the algorithm as expressed cannot be compiled by the Tofino compiler since it cannot be finished in one stage. We believe that a good compiler should automate this process of *synthesizing* pipelined implementations from transactional specifications. Indeed, CaT can successfully handle this example (discussed in §7.3), without requiring an expert developer to manually pipeline their code. Furthermore, beyond automatically pipelining a single transaction, a compiler should be capable of pipelining *multiple* such transactions, generating pipeline configurations for their resulting implementations, and then *allocate* physical resources in the pipeline for these implementations. Finally, P4 programs can often be written in different ways, which consume different kinds of resources; if possible, a compiler must be able to transform uses of a scarce resource into uses of an abundant resource, e.g., using larger tables in lieu of more stages [44].

CaT's approach. CaT is an end-to-end compiler for P4-16 programs that takes inspiration from high-level synthesis (HLS) [7, 16]

```
#define ROTL(x, b) (uint32_t) ((x << b) | (x >> (32 - b)))
void siphash(uint32_t v0, uint32_t v1, uint32_t v2, uint32_t v3) {
    1) v0 += v1;          8) v0 += v3;
    2 ) v1 = ROTL(v1, 5);  9) v3 = ROTL(v3, 7);
    3) v1 ^= v0;          10) v3 ^= v0;
    4) v0 = ROTL(v0, 16); 11) v2 += v1;
    5) v2 += v3;          12) v1 = ROTL(v1, 13);
    6) v3 = ROTL(v3, 8);  13) v1 ^= v2;
    7) v3 ^= v2;          14) v2 = ROTL(v2, 16);
}
```

Figure 2: Motivating Example ME-1: SipHash was manually split into four stages and rewritten by P4 programmers [62].

to provide both: (1) a high level of abstraction for specifying packet-processing functionality, and (2) high quality of the compiler-produced implementation. While prior approaches to HLS for ASICs and FPGAs have sometimes resulted in poor quality of the generated implementation, we believe that the narrower domain of packet-processing pipelines is particularly well-suited for applying HLS gainfully for 2 reasons. First, HLS techniques are designed to systematically explore tradeoffs between functionality (e.g., which ALU can implement an operation?), capacity (e.g., how many ALUs, gateways, etc.?), and scheduling of resources (which stage should run an operation?)—a core challenge in compiling to packet-processing pipelines. Second, HLS techniques can be effective in pipelining transactional code with updates to state, while providing transactional semantics to the programmer: the illusion that each packet modifies headers and state in isolation from other packets.

CaT overview. To produce high quality implementations, CaT combines ideas from several prior P4 compiler projects (Table 1) into an end-to-end system for the first time. CaT divides up the process of compiling a P4-16 program into three phases, as shown in Figure 1; the detailed relationship of these phases to prior HLS and compiler research is shown in Table 2. First, *resource transformations* rewrite packet-processing programs in P4 from one form

(that makes use of a scarce resource) to another form (that makes use of a more abundant resource). Second, *resource synthesis* employs a novel algorithm based on program synthesis to synthesize low-level resource graphs with hardware ALUs, from a high-level transactional specification of a match-action table's action functionality. Third, *resource allocation* employs ILP or SMT solvers to allocate computations and data structures to memory blocks and action units, while respecting program dependencies and per-stage resource constraints. Throughout the 3 phases, CaT makes pervasive use of solver engines to simplify the development of and improve the quality of the compiler.

How CaT factorizes the compilation problem. The problem of optimal code generation in compilers is known to be NP-complete [19] in general. Within the context of P4, Vass et al. [60] show that the problem of compiling P4 programs to pipelines is NP-hard. These results suggest that a compiler may have to decompose the problem in some way to tradeoff optimality for reasonable performance. In CaT's approach, Phases 2 and 3 can be viewed as an *action-block based decomposition* of the P4 compilation problem. In Phase 2, we perform *local* resource synthesis for each individual action block (after transformations in Phase 1). Then, in Phase 3, we use these local synthesis results to perform a *global* resource allocation for *all* action blocks. This keeps the synthesis runtime manageable in practice while still attempting a good quality allocation of computation to resource units. Furthermore, our Phase 2 supports rich computations in action blocks that could require multiple stages as well as transactional (@atomic) semantics. In the rest of the paper, we refer to action block computations as *transactions*. The next three sections detail the three phases of our compiler.

4 PHASE 1: RESOURCE TRANSFORMATION

In the first phase of our compiler, we perform source-to-source rewrites in P4, with the goal of transforming a program that makes use of scarce resources, to one that makes use of more abundant resources. Rewrite rules provide a flexible and general mechanism for this purpose, and can be easily extended by adding more rules for new backend targets and resources. CaT includes rewrite rules for if-else statements in the control block of a P4 program. The standard p4c compiler transforms each action in an if-else branch into one default table, i.e., a table without a match key and with only one action. Our rewrites effectively merge together multiple (possibly nested) if-else statements into one bigger table with keys, thereby using fewer gateway resources (which are used to implement if-else branches) . Such rewrites are in turn driven by a novel *guarded dependency analysis* that identifies parallelism opportunities by eliminating false dependencies — this often leads to reduced usage of pipeline stages in a program.

4.1 Guarded Dependencies

The sequence of program statements inside the `apply {...}` block of a P4 control block can be treated as a branching program (without loops) with (possibly nested) if-statements, reads and writes to PHV fields, and `apply` statements, which apply match-action tables. This program induces Read-after-Write (RAW), Write-after-Read (WAR), and Writer-after-Write (WAW) dependencies between pairs of program statements, which must be respected during synthesis and resource allocation. Conventionally, these dependencies are

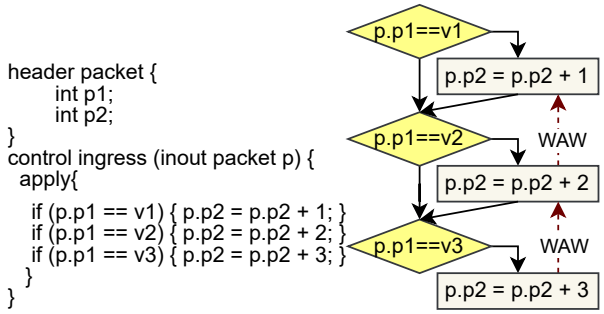

Figure 3: Motivating Example ME-2: The control flow graph of a P4 control block (left); snippet taken from a different portion of SipHash [15]. Dependencies shown as dotted red edges. $v1 \neq v2 \neq v3$ **are constants.**

defined between pairs of program statements without accounting for *path conditions* [41], i.e., conditions under which a control path in a program is executed. Specifically, a dependency due to variable v between two statements s_1 and s_2 is denoted as $(v@s_1 \rightarrow v@s_2, t)$, where $t \in \{RAW, WAR, WAW\}$.

Consider the motivating example ME-2 shown in Figure 3, inspired by a different portion of the SipHash program [15]. The WAW dependencies (shown on the right) cause the Tofino compiler to produce an implementation with 3 pipeline stages. However, *these WAW dependencies are not real*, since the if-conditions guarding these assignments are disjoint. Indeed, a developer of this program recognized the disjoint conditions and *manually changed* the program to use a single block of `if...else if...else if...` statements, thereby reducing the pipeline usage of the compiled program to 1 stage. *We aim to automate such rewrites.* In particular, p4c and the Tofino compiler miss these rewrites in ME-2, likely due to a conservative dependency analysis.

To solve this issue, we propose *guarded dependencies*, which take into account path conditions along control paths. Given a control-flow graph (CFG) C for a P4 control block, a *guarded dependency* between nodes $(n_1, n_2) \in C$ is defined as a tuple $(v@s_1 \rightarrow v@s_2, t, \phi)$, where v is the variable of concern at statement s_1 (in node n_1) and statement s_2 (in node n_2), $t \in \{RAW, WAR, WAW\}$, and ϕ (called a guard) is a formula that describes all the path conditions under which node n_2 may be visited after node n_1 is visited. A procedure based on symbolic execution [41] or model checking [25] that computes path conditions can be used to determine precise guarded dependencies for the program. In particular, we can use an SMT solver to identify *false dependencies*, i.e., dependencies where ϕ is unsatisfiable. Since running model checking or symbolic execution on the input program can be expensive, we next describe a faster lightweight analysis for analyzing guarded dependencies in CaT.

4.2 Lightweight Guarded Dependency Analysis for CaT Rewrites

We now describe a lightweight analysis that helps CaT determine *guarded dependencies* in a P4 program. First, we check that none of the assignment statements update any variables used in conditions of if-else statements. Such updates lead to WAR dependencies and complicates the analysis; we currently choose to not

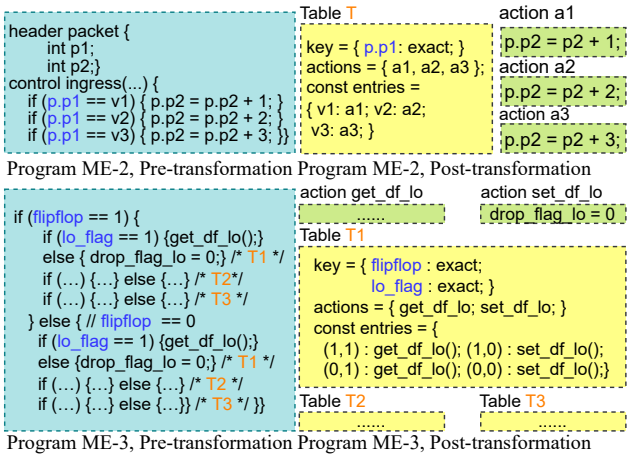

Figure 4: Illustration of Phase 1 Rewrites in CaT, on motivating examples ME-2 (from Figure 3) and ME-3 (from a UPF Rate_enforcer [14] example provided by P4 programmers).

perform any rewrites in such cases. When there are no such updates, the path condition for each CFG node is a simple logical AND of branch conditions, which we compute by a depth-first traversal of the CFG. During the depth-first traversal, branch conditions are pushed/popped on a stack at branch/merge points, respectively. For a pair of nodes $(n_1, n_2) \in C$ with a guarded dependency, the guard ϕ is a logical AND of the computed path conditions for n_1 and n_2. If ϕ is unsatisfiable, then this is a false dependency and removed; otherwise it is conservatively retained as a dependency. For our ME-2 example in Figure 3, this analysis finds that the shown WAW dependencies are false, and removes them.

4.3 Rewrites to Match-Action Tables

We now focus on (possibly nested) if-else statements where the branch conditions are tests on packet fields that can be implemented as keys in a match-action table. Based on the guarded dependency analysis, if there is no dependency between the branches, then we can rewrite them into a match-action table. The key of the generated table is comprised from packet fields used in the if-else conditions, and the actions are the computations within each branch. For example, Figure 4 illustrates our rewrites on two P4 programs – ME-2, and another motivating example ME-3 taken from a UPF Rate_enforcer example [46]. After rewriting, both ME-2 and ME-3 use only match-action tables and thus no gateway resources. ME-2 uses only 1 stage post-rewriting vs. 3 stages before rewriting (due to false WAW dependencies). ME-3 also uses only 1 stage post-rewriting vs. 2 stages before rewriting (due to needing too many gateway resources to fit into 1 stage). These motivating examples drawn from real-world P4 programs show the effectiveness of our approach, where *manual steps taken by a programmer to reduce resource usage are successfully automated by CaT.*

5 PHASE 2: RESOURCE SYNTHESIS

For the second phase, we propose a novel procedure to perform resource synthesis on each P4 action block. Like Chipmunk [34], we too use the SKETCH program synthesis tool [55] to generate a *semantically equivalent* pipelined hardware implementation using ALUs. However, there are several important differences from Chipmunk, which we summarize at the end of this section, after describing our procedure.

5.1 Preprocessing of a P4 Action

We preprocess each action block of the P4 program to prepare for synthesis. We first use some standard preprocessing steps, similar to Domino [53], including (a) branch removal (by replacing assignments under branches with conditional assignments), (b) creating two temporary packet fields for each stateful variable – *pre-state field* (denoting its value before update) and *post-state field* (denoting its value after update), and (c) conversion to static single-assignment (SSA) form [30].

In addition, and differently from Domino, we perform several static analyses during preprocessing: constant folding, expression simplification, and dead code elimination. These analyses are useful in simplifying the action block of a P4 program, thereby reducing the difficulty of the subsequent SKETCH queries. While preprocessing can create temporary packet fields, we neither add nor delete stateful variables during preprocessing simplifications.

5.2 Computation Graph for a P4 Action

After preprocessing, we construct a dependency graph (similar to Domino), with nodes for each program statement and an edge for each RAW dependency.[2] Edges in both directions are also added to/from the pre/post-state fields of each stateful variable. The strongly connected components (SCCs) of this graph correspond to stateful updates, which are condensed to form a *computation graph* G. Thus, G is a directed acyclic graph (DAG) with nodes for program computations (some with stateful updates) and edges for RAW dependencies. Nodes in G are partitioned into two sets: *stateful nodes* are formed from SCCs on the dependency graph, containing a set of program statements that describe an atomic stateful update; *stateless nodes* are the other nodes in the dependency graph. Each edge (u, v) is mapped to a packet field variable that appears in the LHS of the assignment at u and in the RHS of the assignment at v. We call source edges of G *primary inputs (PIs)*; each is associated with an input packet field variable. We call outgoing edges of G *primary outputs (POs)*, each is associated with a final value written to a packet field variable.

5.3 Synthesis Procedure for a P4 Action

Synthesis for a P4 action is now performed on the computation graph G. Rather than create a large synthesis query for the entire G, we decompose the problem into multiple smaller synthesis queries. Specifically, we generate individual synthesis queries for the following variables in G: (1) the output stateful variable of each stateful node (i.e., the LHS of the stateful update assignment), (2) each input variable to a stateful node (i.e., any variable in the RHS of a stateful update assignment), and (3) each primary output (PO) variable, which corresponds to a packet field. Each synthesis query finds an ALU-based implementation and is *parameterized* by an ALU grammar that specifies the functionality of the ALUs (stateful or stateless) available in a given hardware target. These implementations are then connected together according to G, to result in a *resource graph* R, where a node v represents an ALU, and an edge

[2] Conversion to SSA form removes WAW and WAR dependencies.

Input:
1. Computation graph $G = (V, E)$, with $V = \mathsf{Stateful} \cup \mathsf{Stateless}$, where Stateful is the set of stateful nodes and Stateless is the set of stateless nodes;
2. Primary outputs POs: Outgoing edges of G
3. Stateful ALU grammar A_1, stateless ALU grammar A_2;
4. Number of pipeline stages available in hardware, `numPipelineStages`.

Output: Synthesized code for each primary output (PO) and each stateful update.

```
1  // Step 1: Normalize the computation graph to ensure every stateful
     node in G has out-degree 1.
2  Normalize(G);
3  // Step 2: Perform predecessor packing and folding optimizations.
4  graphModified ← TRUE;
5  // Iterate until fixpoint
6  while graphModified do
7      // Folding: tryFold returns TRUE iff G changed
8      for (u, v) ∈ E do
9          if v ∈ Stateful ∧ u ∈ Stateless then
10             graphModified ← tryFold (G, u, v, A1);
11         end
12     end
13     // Predecessor packing: tryPack returns TRUE iff G changed
14     for (u, v) ∈ E do
15         if u ∈ Stateful ∨ v ∈ Stateful then
16             graphModified ← tryPack (G, u, v, A1);
17         end
18     end
19 end
20 // Step 3: Synthesis of stateful updates
21 for v ∈ Stateful do
22     s ← querySketchStateful (v, A1);
23     if s ≡ FAILURE then
24         abort(
25         "Error synthesizing stateful node " + v);
26     end
27 end
28 // Step 4: Min-depth solutions for stateless code
29 Os ← POs ∪ {inputs(v)|v ∈ Stateful};
30 Order elements of Os according to topological order in G;
31 for o ∈ Os do
32     // Compute the Backwards Cone of Influence (BCI) of o
33     spec ← computeBCI(o); // spec of o
34     i ← 1; // initial depth of solution tree
35     // Loop over i to find a minimum depth solution tree
36     while i ≤ numPipelineStages do
37         s ← querySketchStateless(spec, i, A2);
38         if s ≡ SUCCESS then
39             break;
40         else
41             i ← i + 1; // increment depth
42         end
43     end
44 end
```

Auxilliary procedures:

procedure tryFold(G, u, v, A_1): Query SKETCH to determine if edge (u, v) can be folded into stateful node v using stateful grammar A_1. If query succeeds, edge (u, v) is removed from G.

procedure tryPack(G, u, v, A_1): Query SKETCH to determine if nodes u, v can be packed into a single new stateful node using stateful grammar A_1.

Algorithm 1: CaT Synthesis Procedure. Calls that use a SKETCH query are highlighted in blue.

(u, v) indicates that the output of ALU u is connected to an input of ALU v. We prove that our synthesis procedure is correct: the resource graph R is functionally equivalent to G.

Our synthesis procedure is shown in **Algorithm 1**, which consists of four main steps: 1) normalization; 2) folding and predecessor packing optimizations; 3) synthesis of stateful updates; 4) synthesis of minimum-depth solutions for stateless code. The critical step is Step 3, which queries SKETCH to see if each stateful update assignment can be synthesized into configurations for a single stateful ALU. If any such query fails, then we terminate the procedure and provide feedback to the programmer. We create separate synthesis queries to perform optimizations in Step 2, to help Step 3 succeed. Finally, Step 4 creates synthesis queries to implement the POs and inputs to the stateful nodes.

Step 1: Normalization of computation graph G. In the typical hardware backends that we target (e.g., Menshen, Tofino), a stateful ALU can output a single value that is either the pre-state or the post-state value of one of its stateful registers. In this step, we normalize G to a graph such that each stateful node has only one output, and each packet field labelled as an out-edge from a stateful node is either the *pre-state field* or the *post-state field*. Normalization is performed by replicating stateful nodes that have multiple outputs.

Step 2: Folding and predecessor packing optimizations. We iterate the following two optimizations until convergence.

Folding to reduce input edges. A stateful node with too many in-edges could cause Step 3 to fail, due to a limited number of inputs available in ALUs. The **folding optimization** finds opportunities to reduce the number of in-edges to a stateful node. We consider *dependent inputs*, i.e., inputs that are themselves functions of other inputs to the same stateful node. For each such candidate i, we query SKETCH to check if the function that computes i can be *folded into* the stateful node itself, such that the enlarged node fits into a single stateful ALU. If the synthesis query is successful, i is removed. Figure 5 shows an example benchmark—BLUE (decrease) [32]—where this works successfully. Here, *folding* reduces an edge between the top two nodes in the computation graph G (extreme left of Figure 5), thereby reducing the pipeline usage by 1.

Predecessor packing to merge nodes. Even after folding, the stateful update in a single node in G might not *fully utilize* an available stateful ALU in hardware. Consider again the BLUE (decrease) example in Figure 5, where the middle box shows G after folding. Here, a single Tofino stateful ALU can actually implement *both* stateful updates (in blue boxes) in a single stage, as shown by a merged node on the right. To achieve this compaction, we use a simple heuristic called **predecessor packing**, inspired by technology mapping for hardware designs [28]. The key idea is to pack more into a stateful ALU by attempting a merge of nodes u and v, where at least one node is stateful and where predecessor u has only one out-edge (to v). Like folding, we implement the packing attempt via

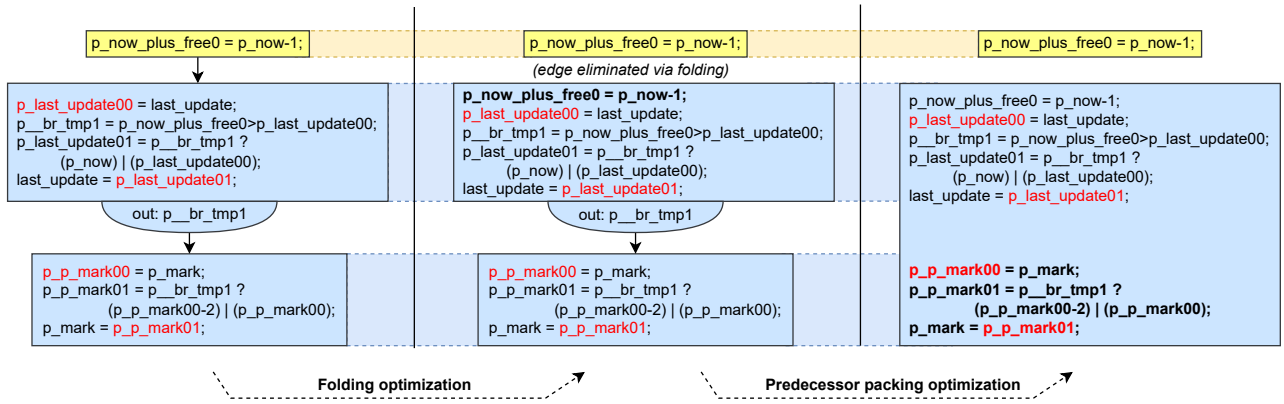

Figure 5: Computation graph for the BLUE(decrease) [32] (leftmost) and optimizations performed by CaT when targeting the Tofino ALU. Stateful nodes are in blue, stateless nodes are in yellow, pre/post-state fields are in red, modified parts are in bold.

a SKETCH query, and merge the nodes if the query is successful. In our evaluations (§7.3), we show that these optimizations are effective in compiling to fewer pipeline stages.

Step 3: Synthesizing stateful updates. We are now ready to synthesize the outputs of the stateful nodes in G. To preserve the transactional semantics of the program, each stateful update must be completed within a single pipeline stage, i.e., *the update operation must fit in a single stateful ALU*. Accordingly, for each stateful node in G, we generate a SKETCH query to check if the stateful update operation can be implemented by a single stateful ALU. The functionality of the stateful ALU available in hardware is specified using an ALU grammar A_1, which is expressed as a large block of multiple if-else statements with one case for each opcode. We assert that each such query succeeds; if any query fails, our procedure exits with an error, giving feedback to the programmer.

Step 4: Minimum-depth solutions for stateless code. In the last step, we synthesize code for the POs and inputs to the stateful nodes in G (line 29). For each such variable o to be synthesized, we first compute its backwards cone of influence (BCI), which is often used in verification/synthesis tasks to determine the dependency region up to some (boundary of) inputs [36]. In graph-theoretic terms, $\text{BCI}_G(o)$ is a subgraph in G derived by going recursively backward from o, stopping at a PI or an output of a stateful node. Essentially, the BCI provides the *functional specification* for o in terms of a set of inputs, where each input is a PI or the output of a stateful node in G. Note that these specifications are stateless, i.e., they do not include any stateful nodes.

We model a switch's stateless ALU functionality using an ALU grammar A_2 (expressed as a large block of if-else statements). We use SKETCH to find a *minimum-depth tree* solution for o, where each tree node represents a stateless ALU, and the leaf nodes represent the inputs in $\text{BCI}_G(o)$. A minimum-depth solution helps reduce the number of pipeline stages – this is explained in more detail in the next section (§5.4). Since SKETCH does not support optimal synthesis, we invoke it in a loop to minimize depth, where each iteration tries to find a solution tree of a given depth i (line 35), starting from 1 and continuing until i exceeds the maximum number of pipeline stages. An example computation graph with a single stateful update (blue box) and the associated synthesis query results are shown in Figure 6.

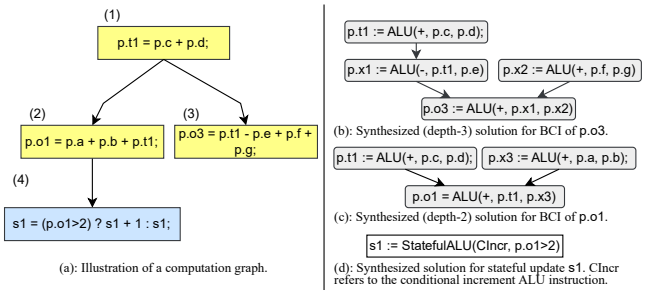

Figure 6: Example of a computation graph (left) and the synthesis query results (right) targeting Banzai ALUs [53]. Stateful nodes in blue and stateless nodes in yellow. The POs are: {`p.o1`,`p.o3`}. `p.o1`'s BCI contains nodes 1 and 2; `p.o3`'s BCI contains nodes 1 and 3.

5.4 Staged-Input Tree Grammar for Synthesis

We now describe details of the grammar used for the synthesis queries in Step 4, where each query (in line 37) tries to find a solution tree of a given depth i for implementing a given variable o. Initially, we used a simple recursive tree grammar for the SKETCH query, where each tree node is an ALU (specified by a stateless ALU grammar A_2) and its children are the ALU operands; and a leaf node is an input in $\text{BCI}_G(o)$, i.e., either a primary input (PI) or an output of a stateful node in G. By iteratively incrementing i, we were able to find a minimum-depth tree solution for o.

However, even with a minimum-depth tree solution for each variable o, when we compose together these solution trees according to G, the number of pipeline stages for the entire action may not be the minimum possible. This is because with this simple grammar, the depth is optimized to be minimum *within an individual synthesis query for* o, without considering the larger scope of the entire action. As a concrete example, consider the computation graph G for the Flowlet switching benchmark [52] shown in Figure 7. As before, blue nodes are stateful nodes and yellow nodes are stateless. In addition, we show two synthesized solutions for the variable `p_br_tmp0`, with the specification `p_br_tmp0 = (p_arrival0 - p_last_time_0 > 2)`. Its BCI has two inputs: `p_arrival0` is a PI, and `p_last_time_0` is the output of the stateful node 1.

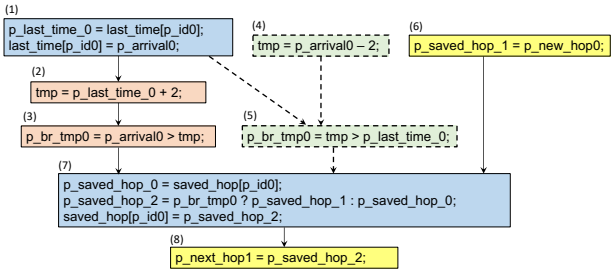

Figure 7: Computation graph for the Flowlet switching [52], showing two minimum-depth solutions for variable `p_br_tmp0` using Banzai ALUs: nodes {2, 3} and nodes {4, 5}.

Note first that the BCI input `p_last_time_0` can only be available after stage 1 within the overall action (stages are numbered starting from 1), since the implementation of node 1 occupies one stage. Now, consider a minimum-depth solution with nodes 2 and 3 (both shown in orange), where node 1 provides an input to the ALU operation in node 2, which in turn provides an input to the ALU operation in node 3, which computes `p_br_tmp0`. Hence, `p_br_tmp0` is computed in stage 3 and is available at the end of stage 3.

Consider a second minimum-depth solution shown by dashed nodes and edges, with nodes 4 and 5 (both shown in green). Like the first solution, it also has two ALU operations and the same minimum depth 2. However, the ALU operation in node 4 is *independent* of `p_last_time_0`, and can be computed in parallel with node 1. This allows `p_br_tmp0` to be computed in stage 2, making it available at the end of stage 2. This example shows that although both solutions have the minimum depth 2, the second is better because `p_br_tmp0` can be computed in an earlier stage for the overall action.

Since the number of pipeline stages is often a critical resource in compiling P4 programs, we consider the larger scope of the action in each individual synthesis query. We achieve this by augmenting our tree grammar for a query, where an input in the BCI is now associated with a *stage*, which denotes the stage *within the action* at which the input is available to be used. We call this grammar a *staged-input tree grammar*. We regard a primary input (PI) in G as being available for use at stage 1, and the output of a stateful node being available for use at some stage $s > 1$, where s depends on its own implementation. In each individual synthesis query, we now look for a minimum-depth tree solution that produces the output at the *earliest possible stage*, based on stage information of the inputs in its BCI. To compute the latter, in Step 4, we use a topological ordering over the set of outputs o in G (line 30), such that any input in $\text{BCI}_G(o)$ is already implemented before the synthesis query for o. For the example in Figure 7, our synthesis query with a staged-input tree grammar returns the solution with nodes 4 and 5 (in green) for the output `p_br_tmp0`. The complete SKETCH input for this query is shown in Appendix A, which includes the grammars for a staged-input tree and for a stateless ALU.

5.5 Final Result of the Synthesis Procedure

The final result of the synthesis procedure is represented in the form of a resource graph R for a given P4 action block, where each node v in R represents a stateful or a stateless ALU, and an edge (u, v) in R indicates that the output of ALU u is connected to an input of ALU v. These resource graphs play an important role in resource allocation, the next phase of our compiler. We now state and prove correctness of our synthesis procedure.

Theorem 1 (Correctness). *The result of the CaT synthesis procedure (Algorithm 1) on a computation graph G is correct.*

Proof sketch. The synthesis procedure works by decomposing G (after correctness-preserving normalization and optimizations in Steps 1 and 2, respectively) into subgraph components comprising of: (1) outputs and inputs of stateful nodes, (2) inputs of stateful nodes and their stateless BCIs, and (3) POs and their stateless BCIs. Each such subgraph of G represents a *specification* for a synthesis query (in Steps 3 or 4), which generates a corresponding *implementation* using ALUs, i.e., a subgraph in the resource graph R. Based on correctness of program synthesis in SKETCH [55], each stateful node output, stateful node input, and PO in R is functionally equivalent to that in G. Hence the synthesis procedure is correct. □

5.6 Comparison with Synthesis in Chipmunk

Our motivation for a new synthesis procedure was improving the performance of synthesis in Chipmunk [34], which also uses SKETCH. CaT and Chipmunk have several differences.

First, CaT creates multiple *smaller* synthesis queries for SKETCH. Although Chipmunk uses a slicing technique to create per-output queries, the scope for each such query is the entire transaction. Our procedure separates queries for stateful update operations from those on stateless operations in Steps 3 and 4, respectively. The scope for a stateful query is a single stateful ALU: these queries are small and also critical; if any fails, synthesis cannot succeed. The scope for a stateless query is typically smaller than an entire transaction, since its BCI stops at outputs of other stateful nodes. Overall, smaller synthesis queries lead to significant performance improvement over Chipmunk, as demonstrated in evaluations (§7). We note that because SKETCH queries are independent of each other in both CaT and Chipmunk, both lose opportunities to share common computations across multiple queries.

Second, for stateless operations, we use multiple SKETCH queries to synthesize solutions of *minimum-depth*, i.e., the minimum number of pipeline stages, while searching the space over all possible equivalent programs. Although Chipmunk also considers the space of all possible programs, its queries do not guarantee minimum-depth solutions within a given bound.

Third, Chipmunk creates synthesis queries in the form of low-level holes in an ALU grid architecture that are filled by SKETCH. In contrast, our synthesis queries ask for ALU-based implementations that we represent as resource graphs. These resource graphs are used during resource allocation (in Phase 3) for handling *multiple* transactions, which are not supported by Chipmunk.

Finally, similar to Chipmunk's ALU DSL, our synthesis queries are *parameterized* by an ALU grammar that specifies the functionality of ALUs available in a given hardware target. This enables the same synthesis procedure to be used for different hardware backends, providing compiler retargetability. CaT currently supports three different ALU grammars: Tofino ALUs [9], Banzai ALUs [53], and Menshen ALUs [61]; more can be supported as needed. As long as compiler developers have access to the documentation of a hardware ALU in the target backend, it is straight-forward to write a complete and correct ALU grammar describing its capabilities.

Table 3: Constraint formulation for resource allocation.

	Name	Definition
Constants	N_S	maximum number of pipeline stages
	N_{alu}	maximum number of ALUs in each stage
	N_P	number of packet fields that must remain available through the pipeline
	N_{table}	maximum number of logical table IDs per stage
	$N_{entries}$	maximum number of match entries per table per stage
	e_t	maximum number of entries in table t in program
Indices	t	index for Table
	i	index for partition of a Table, partition denoted $t[i]$
	a	index for Action
	u	index for ALU
	s	index for pipeline Stage
Variables	M_{tis}	binary, set to 1 iff match of $t[i]$ is assigned to stage s, 0 otherwise
	$stage_u$	integer, stage assigned to ALU u
	$stage_{us}$	binary, set to 1 iff ALU u is assigned to stage s
	beg_u	integer, stage where ALU u output is computed
	end_u	integer, ≥ last stage where ALU u is used as an input
	$prop_{us}$	binary, set to 1 iff output of ALU u is propagated in stage s
Sets	R_{tia}	Resource graph for action a of table partition $t[i]$
	V_{tia}	Vertices in R_{tia}, each represents an ALU
	E_{tia}	Edges in R_{tia}, each represents a connection between ALUs
	AP_{tia}	Set of ALUs in R_{tia}, whose outputs may need to be propagated across stages
	UV_{tiau}	Set of ALUs v in R_{tia}, s.t. $(u, v) \in E_{tia}$, i.e., ALU u is an input to ALU v
Constraints similar to prior work [39, 44]		
Match table capacity		$\forall s: \ \sum_{t,i} \ M_{tis} \leq N_{table}$
Match action pairing		$\forall s \ \forall t, i, a \ \forall u \in V_{tia}: \ stage_{us} \rightarrow M_{tis}$
Table dependency		$\forall i_1, i_2, a_1, a_2, \forall u_1 \in V_{t_1\ i_1\ a_1}, \ \forall u_2 \in V_{t_2\ i_2\ a_2}: stage_{u_1} < stage_{u_2}$
New constraints in our work		
ALU allocation 1		$\forall t, i, a \ \forall u \in V_{tia}: \ 1 \leq stage_u \leq N_S$
ALU allocation 2		$\forall s \ \forall t, i, a, \ \forall u \in V_{tia}: \ stage_u = s \leftrightarrow stage_{us}$
Action dependency		$\forall t, i, a \ \forall (u, v) \in E_{tia}: \ stage_u < stage_v$
ALU propagation 1		$\forall t, i, a \ \forall u \in AP_{tia}: \ beg_u = stage_u \wedge beg_u < end_u \wedge end_u \leq N_S$
ALU propagation 2		$\forall t, i, a \ \forall u \in AP_{tia}, \forall v \in UV_{tiau}: \ end_u \geq stage_v$
ALU propagation 3		$\forall t, i, a \ \forall u \in AP_{tia}, \forall s \in \{1, \ldots, N_S\}: \ (beg_u < s \wedge s < end_u) \leftrightarrow prop_{us}$
ALU propagation 4		$\forall s \ \sum_{\forall t,i,a \ \forall u \in AP_{tia}} (stage_{us} + prop_{us}) \leq N_{alu} - N_P$

6 PHASE 3: RESOURCE ALLOCATION

After performing synthesis for each P4 action block, the third phase of our compiler performs *global* resource allocation for the full P4 program by using a constraint-based formulation, shown in Table 3. The top part lists the definitions of constants, indices, variables, and sets that are used to automatically generate the constraints. The bottom part shows the full set of constraints, divided into a first set that is similar to prior work [39, 44], and a second set that is new. Our new constraints address: (1) ALU resources in action computations, (2) multi-stage actions, (3) fitting multiple action blocks in the same pipeline stage, and (4) propagation of ALU outputs. Prior efforts either do not consider allocation of ALU resources and multi-stage actions [39, 44], or do not address multiple action blocks [34, 53]. Another novel feature of our approach is that we use the resource graph R synthesized for each action block (in Phase 2), to perform global optimization in this phase.

6.1 Constraints Similar to Prior Work

If a match table in the program has too many entries to fit within a single stage, it is partitioned into b_t separate tables, where $b_t = \lceil (e_t / N_{entries}) \rceil$. Currently, we only support exact matches; hence, a packet will match at most one of the partitions $t[i]$ that have the same actions as table t. The first constraint ensures that the number of match tables allocated in a stage is less than or equal to the number of table IDs available. The second ensures that ALUs in action blocks are accompanied by the associated match table. The third enforces four types of table dependencies: match, action, successor, and reverse-match [39]. If table t_2 depends on table t_1, all ALUs of t_2 are allocated after ALUs of $t1$. For successor and reverse-match, $<$ is replaced by $\leq$.

6.2 New Constraints in Our Work

The constraints for ALU allocation (1,2) ensure that each ALU in each action is assigned to one and only one pipeline stage. The Action dependency constraint uses the edges in R_{tia} (synthesized in Phase 2) to enforce dependencies between ALUs. Together with the Table dependency constraint, this allows ALUs from multiple action blocks to be assigned in the same pipeline stage, *while respecting both inter-table and intra-action dependencies.*

We support a multi-stage action under the condition that it does not modify the table's match key, by duplicating the match entries at each stage to ensure that the entire action is executed. As an example, suppose a match entry m in table t is associated with action A that takes 2 stages. We can allocate table t in two consecutive stages, such that if a packet matches entry m in table t in stage s, it will match entry m in table t in stage $s+1$ as well, resulting in action A being executed completely over the two stages.

We allow allocation of multiple actions in the same stage and also allow assigning an Action A to *non-consecutive* stages. In the latter case, we need additional ALUs in the intermediate stages to *propagate the intermediate results.* The ALU propagation constraints (1-4) handle allocation of these additional ALUs. Here, AP_{tia} is the set of ALUs in R_{tia} whose outputs may need to be propagated across stages, and UV_{utia} is a set of ALUs v in R_{tia} that use ALU u as an input. The ALU propagation constraints 1-3 ensure that an ALU $u \in AP_{tia}$ is propagated until the largest stage where it is used as an input. The ALU propagation constraint 4 enforces the ALU capacity constraint in each stage, where N_P ALUs are pre-occupied to carry packet fields that remain live through the whole pipeline (e.g., IP TTL) or are updated (by ALUs not in any AP_{tia}); the remaining $(N_{alu} - N_P)$ ALUs must be enough for the sum over all ALUs u in any AP_{tia} that are either assigned to or propagated in that stage. (Appendix B shows the formulation of ALU propagation 3 using the well-known Big-M method).

6.3 Solving the Constraint Problem

We can use either an ILP solver (Gurobi [5]) or an SMT solver (Z3 [31]) to find an optimal or a feasible solution. We specify an objective function to find an optimal solution, e.g., we add the constraint min *cost* to minimize the number of stages, where *cost* is ≥ the stage assigned to any ALU. i.e., $\forall t, i, a, \ \forall u \in V_{tia}: \ cost \geq stage_u$. To find a feasible solution, we use a trivial objective function (min 1) with Gurobi (none is needed with Z3).

7 IMPLEMENTATION AND EVALUATION

We implement the CaT compiler with the workflow shown in Figure 1. The resource transformation phase is implemented on top of p4c [11]. We also use p4c to identify the action blocks and table dependencies needed in CaT's resource synthesis and resource allocation phases. For the backend, ideally the CaT compiler should directly output machine code for the targets. However, due to the undocumented and proprietary machine code format of the Tofino chipset, we generate a low-level P4 program by using a best-effort encoding for the resource constraints, based on known information about the Tofino chipset. For the Menshen backend, we extend the open-source RMT pipeline [10, 61] by writing additional Verilog to support richer ALUs, e.g., the IfElseRAW ALU [53]. The CaT

compiler directly outputs machine code to configure various programmable knobs (e.g., opcodes) within Menshen's Verilog code.

Sanity checking of CaT prototype. We check CaT's output for Menshen using its cycle-accurate simulator, which can be fed input packets to test the generated machine code. We create P4-16 benchmarks starting with a subset of the switch.p4 program [54], consisting of 2–6 tables randomly sampled from switch.p4. Then, we add new actions to the tables using @atomic blocks for transactional behavior. The logic within these atomic blocks consists of one of 8 Domino benchmark programs [53]; the IfElseRaw ALU [53] in our simulator is not expressive enough for the remaining 6. We also test the 8 benchmark programs in isolation, generating 24 benchmarks in total, many of which have multiple transactions and thus stress both resource synthesis and resource allocation. We randomly generate test input packets and inspect the output packets from the simulation. So far, all our sanity checks have passed.

7.1 Evaluation Setup and Experiments

We address the following evaluation questions:

Q1: Resource Transformation. How much does CaT's resource transformation help in terms of the resource usage? We select 3 benchmarks [2] extracted from real P4 programs and compare resource usage for pre- and post-transformed programs (§7.2).

Q2: Resource Synthesis. How does CaT's resource synthesizer compare to existing ones? We compare CaT with Chipmunk on several dimensions using ALUs drawn from Tofino [9] and Banzai [53], along with controlled experiments on the predecessor packing and preprocessing optimizations (§7.3).

Q3: Resource Allocation. How good is the CaT compiler in terms of resource usage? We use Gurobi as the default solver for resource allocation and compare the runtime of the Gurobi and Z3 solvers. In addition, we compare 2 modes: finding either an optimal or a feasible solution (§7.4).

Q4: Retargetable Backend. Can CaT easily perform compilation for different hardware targets? Our synthesis experiments with the Banzai and Tofino ALUs already demonstrate this feature. Additionally, we run the CaT compiler on different *simulated* hardware configurations, compile switch.p4 under varying constraints and report the results (§7.4).

Benchmark selection. We use different benchmarks to demonstrate the benefits of each phase of the compiler.

- Resource transformation: 3 benchmarks (ME-1, ME-2, ME-3) extracted from SipHash and UPF (real P4 programs developed by other P4 programmers).
- Resource synthesis: 14 benchmarks together with their semantically equivalent mutations (10 for each benchmark, hence 140 in total) from the Chipmunk paper [34].[3]
- Resource allocation: Same as the benchmarks we use for sanity checking our prototype. We use the full switch.p4 program for experiments that vary hardware resource parameters in the Menshen backend.

Machine configuration. We use a 4-socket AMD Opteron 6272 (2.1 GHz) machine with 64 hyperthreads and 256 GB RAM to run all our experiments for both CaT and Chipmunk. Additionally, we note that Chipmunk requires performing a grid search on pipeline geometries (within an upper bound) using multiple such machines in parallel to find an implementation that consumes a small number of pipeline resources. By contrast, CaT does not require multiple machines since CaT's resource synthesis (Algorithm 1) directly tries to minimize pipeline depth without a parallel grid search.

[3] Chipmunk can compile all 14 benchmarks by using Banzai ALUs [53], and 10 of the 14 benchmarks by using Tofino ALUs [34]. For Banzai ALUs, we also show the Domino pipeline usage as reported in the Chipmunk paper [34].

Table 4: Resource usage with/without CaT's transformation.

Program	Without CaT transformations			With CaT transformations		
	#gateways	#tables	#stages	#gateways	#tables	#stages
ME-1	15	15	5	15	15	4
ME-2	3	3	3	0	1	1
ME-3	19	12	2	0	3	1

7.2 Results for Resource Transformation

The resource transformation phase of the CaT compiler performs a best-effort rewrite of if-else statements in the P4 program into match-action tables. Table 4 shows the resource usage of compiling benchmarks ME-{1,2,3} to the Tofino architecture with and without the CaT rewrites. As expected, the rewrites help in reducing the number of gateways. Furthermore, they may merge together multiple tables without match entries (i.e., *default* tables), thereby reducing the total number of tables. More importantly, for all benchmarks shown, the rewritten program consumes *fewer pipeline stages* due to either reduced gateway usage (ME-2, ME-3) or the removal of false control flow dependencies (ME-1). CaT does this automatically without the developer engaging in trial-and-error compilation [44].

7.3 Results for Resource Synthesis

In all our experiments, the resource synthesis phase consumes the most time, and the SKETCH synthesis queries dominate the overall runtime of CaT. In this section, we focus on evaluating this phase. We compare the CaT and Chipmunk compilers on the SipHash benchmark (cf. Figure 2) and on all benchmarks used in the Chipmunk work [34]. For the latter, we target both Tofino ALUs and Banzai ALUs, to evaluate the performance of CaT on different instruction sets and various input programs. This also demonstrates CaT's retargetability via different ALU grammars.

Results for SipHash. For the SipHash P4 program, the CaT compiler was successful with the Tofino ALU, and took about 40 hours to complete. In comparison, Chipmunk failed to generate the output even after 150 hours. After investigation, we found two main reasons for the long runtime of CaT: (1) 1 multistage action required 4 pipeline stages – the synthesis query for this action has a large search space and took more than 30 hours in SKETCH. (2) SipHash includes bitvector operations in addition to integer arithmetic. This results in a harder synthesis problem for SKETCH: SipHash uses 32-bit bitvectors, while SKETCH's default for integers is 5 bits.

We plan to explore new ideas for handling deep multistage actions in future work. For handling 32-bit bitvectors more efficiently, we enhanced our basic procedure as follows. We first run the SKETCH synthesis query on a program with a constrained input space, and verify separately whether the generated solution works for the full input space. If it does, then we have found a correct solution; otherwise, we add the generated solution as a counterexample in SKETCH and repeat the procedure. The hope is to quickly generate a solution from the constrained input space that can be proven correct for the whole input space. For the SipHash example,

Table 5: CaT vs. Chipmunk and Domino; Tofino or Banzai ALUs. (pred: Predecessor packing, ppa: Preprocessing, ✗: failed, Std Dev: sample standard deviation).

Program	ALU	CaT					Chipmunk [34]			CaT Speedup	Domino [53]
		Mean Time (s)	Std Dev (s)	Avg #stages			Mean Time (s)	Std Dev (s)	Avg #stages	wrt Chipmunk	Avg #stages
				default	w/o pred	w/o ppa					(from [34])
BLUE (increase) [32]	**Tofino ALU**	19.04	0.43	1	2	1	159.78	59.03	2	8.39 ×	
BLUE (decrease) [32]	**Tofino ALU**	18.72	0.84	1	2	1	142.5	42.5	2	7.61 ×	
Flowlet switching [52]	**Tofino ALU**	19.76	0.69	2	✗	2	962.83	1170.16	2	48.73 ×	
Marple new flow [49]	**Tofino ALU**	6.65	0.52	1	✗	1	5.2	1.71	1	0.78 ×	
Marple TCP NMO [49]	**Tofino ALU**	13.24	0.53	2	✗	✗	6.56	0.36	2	0.50 ×	**N/A**
Sampling [53]	**Tofino ALU**	14.03	0.57	1	✗	1	22.87	10.68	1	1.63 ×	
RCP [59]	**Tofino ALU**	20.19	0.59	1	✗	1	65.13	20.93	1	3.23 ×	
SNAP heavy hitter [21]	**Tofino ALU**	3.58	0.25	1	1	1	26.83	13.63	1	7.49 ×	
DNS TTL change [26]	**Tofino ALU**	20.84	1.97	2	3	✗	36.34	50.55	2	1.74 ×	
CONGA [20]	**Tofino ALU**	10.24	0.43	1	✗	1	3.02	0.17	1	0.29 ×	
BLUE (increase) [32]	**Banzai ALU:** pred raw	40.69	1.41	4	4	4	166.88	36.59	4	4.10 ×	✗
BLUE (decrease) [32]	**Banzai ALU:** sub	38.83	1.48	4	4	4	1934.82	1611.66	4	49.83 ×	✗
Flowlet switching [52]	**Banzai ALU:** pred raw	25.37	0.94	3	3	3	185.84	81.41	3	7.33 ×	8.3
Marple new flow [49]	**Banzai ALU:** pred raw	13.79	0.44	2	2	2	12.31	0.18	2	0.89 ×	✗
Marple TCP NMO [49]	**Banzai ALU:** pred raw	28.12	2.60	3	4	✗	15.3	0.49	3	0.54 ×	✗
Sampling [53]	**Banzai ALU:** if else	11.52	0.65	2	2	2	33.39	11.09	2	2.90 ×	✗
RCP [59]	**Banzai ALU:** pred raw	25.08	0.85	2	2	2	31.21	7.55	2	1.24 ×	5.6
SNAP heavy hitter [21]	**Banzai ALU:** pair	3.45	0.23	1	1	1	69.07	19.36	1	20.02 ×	3.3
DNS TTL change [26]	**Banzai ALU:** nested if	32.63	34.91	3	5	✗	211.67	22.65	3	6.49 ×	✗
CONGA [20]	**Banzai ALU:** pair	10.27	0.55	1	1	1	19.47	8.05	1	1.90 ×	✗
Stateful firewall [21]	**Banzai ALU:** pred raw	2499.43	4638.58	4	4	✗	6749.89	6349.94	4	2.70 ×	15.5
Learn filter [53]	**Banzai ALU:** raw	31.01	0.73	3	3	3	212.32	4.47	3	6.85 ×	17.5
Spam Detection [21]	**Banzai ALU:** pair	3.51	0.21	1	1	1	59.95	17.75	1	17.08 ×	3.1
STFQ [35]	**Banzai ALU:** nested if	20.99	2.04	3	3	3	22.73	6.94	2	1.08 ×	✗

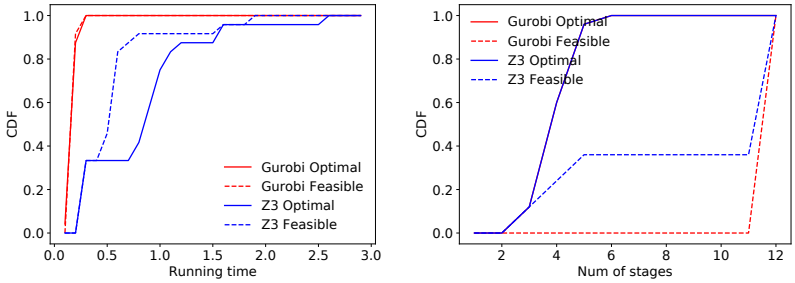

Figure 8: Gurobi vs. Z3: Running time, Num of stages.

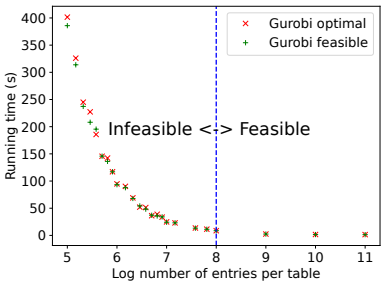

Figure 9: Varying # of entries/table.

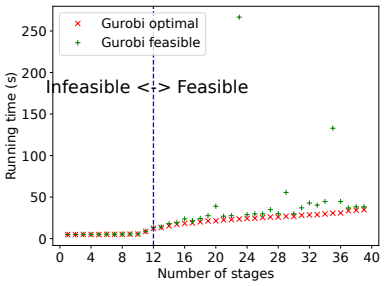

Figure 10: Varying # of stages.

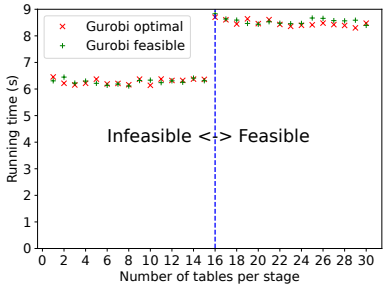

Figure 11: Varying # of tables per stage.

we constrained the input space to use only 16 bits (by setting the higher 16 bits of the input 32-bit bitvectors to 0). CaT separately verified that the generated solution is also correct for unconstrained 32-bit bitvectors. This approach reduced CaT's runtime to under 2 minutes, showing the promise of such an approach.

Results for Chipmunk benchmarks. The results are shown in Table 5, for Tofino and Banzai ALUs, respectively. We report the runtime of full compilation; for CaT, this includes the resource allocation time, whereas Chipmunk does not perform any resource allocation. We consider 10 semantically equivalent mutations of each of the benchmarks, which are identical to those in Chipmunk [34]. We report the mean and sample standard deviation of compilation time across all mutations.[4] These experiments evaluate if CaT can effectively handle semantically equivalent programs.

We report the resource consumption of the generated code produced by CaT, in terms of the total number of pipeline stages required on average across the mutations ("#stages default" column). Stages are the most scarce resource in programmable switches (e.g., 12 for Tofino). For evaluating the effectiveness of our predecessor packing optimization (pred) and the preprocessing analyses (ppa) (§5.3), we also report the number of stages without these optimizations in columns "#stages w/o pred" and "#stages w/o ppa." Gray-ed entries indicate a difference from the default setting.

[4]The runtimes in Table 5 are similar to, but slightly different from that in Table 2 of the Chipmunk paper [34]. The differences arise due to Chipmunk's use of SKETCH's parallel mode, which introduces non-determinism due to thread interleaving.

Our results show that CaT is able to compile *all* programs successfully compiled by Chipmunk, with almost all compiled results having a matching number of pipeline stages. Furthermore, *CaT is often much faster and more stable (in running time) than Chipmunk*. Specifically, for the Tofino ALUs (top section of Table 5), CaT finishes compilation within a few seconds, 2.75x faster on average (geometric mean) than Chipmunk. The max speedup is 48x for *flowlet switching*, a minutes-to-seconds improvement (≈16 minutes in Chipmunk vs. 20 seconds in CaT). In *BLUE (increase)* and *BLUE (decrease)*, CaT generates a solution with fewer stages than Chipmunk. In all other benchmarks the number of stages is the same. For the BLUE benchmarks, since the Tofino stateful ALU contains two registers, CaT's optimizations enabled it to pack a successive pair of stateful updates into a single stateful ALU (§5.3). In comparison, Chipmunk mapped the two stateful updates to two ALUs in two stages. This shows that CaT's approach can find additional opportunities for fully utilizing the functionality of available hardware resources. Predecessor packing is also effective in 9 of 10 benchmarks, enabling compilation to succeed or reducing the number of stages; preprocessing is also useful in 2 benchmarks.

For the Banzai ALUs (lower section of Table 5), we additionally report results on stages output by the Domino compiler [53] (which only handled Banzai ALUs), with the average number of stages across different program mutations shown in the last column (as reported in [34]). Note first that CaT takes no more than 1 minute on most successful benchmarks. Although it takes 40 minutes for *stateful firewall*, Chipmunk is much slower, requiring more than 1.5 hours. CaT provides 3.94x speedup on average (geometric mean) and 49x maximum, with respect to Chipmunk. CaT is slower only on *Marple new flow* and *Marple TCP NMO*, but finishes both within 30 seconds. Note that Chipmunk must use multiple machines in parallel for synthesis, while CaT only uses one machine for synthesis. In terms of number of stages, CaT generates code with the same number of stages as Chipmunk for all benchmarks except the *STFQ* example (3 in CaT vs. 2 in Chipmunk). Upon investigation, we find that this is due to separation between queries for stateful and stateless nodes in our synthesis procedure. Although our predecessor packing optimization can often mitigate this negative effect, we plan to improve it further in future work. Still, both predecessor packing and preprocessing optimizations are effective in some benchmarks here as well. Finally, Domino either fails to compile (8 of 14 examples), or uses many more stages (other 6 examples). *Overall, CaT generates high-quality code comparable to Chipmunk, but in much less time and with fewer compute resources.*

Results for controlled experiments. We selectively turned on 2 optimizations: (1) Predecessor packing, (2) Preprocessing analyses (constant folding, algebraic simplification, dead code elimination). According to the results in Table 5, for Banzai ALUs, without predecessor packing, our compiler uses additional stages in two examples (*Marple TCP NMO*, and *DNS TTL change*), showing that predecessor packing can reduce the number of pipeline stages; for the Tofino ALUs, predecessor packing was even more beneficial: disabling predecessor packing resulted in compilation errors for 6 examples (*flowlets, Marple new flow, Marple TCP NMO, Sampling, RCP*, and *CONGA*). The reason is that the Tofino ALU supports very limited stateless computations and cannot handle relational or conditional expressions. Packing such expressions into adjacent stateful ALUs was essential for compilation to succeed. For Banzai ALUs, without preprocessing analyses, 3 of the examples could not be compiled. The runtime of preprocessing is less than 0.1 sec in all examples. *Overall, CaT's optimizations allow compilation to succeed where it would fail otherwise and reduce the number of pipeline stages.*

7.4 Results for Resource Allocation

We experiment with two solvers (Gurobi and Z3) and two modes (optimal and feasible) on our benchmark examples. The results are in Figure 8 with more detailed data in Appendix C, Table 6. The results show that for checking feasibility, Gurobi returns suboptimal solutions that use all the pipeline stages, while Z3 finds feasible solutions that are better than Gurobi's but takes marginally more time. However, Gurobi finds an optimal solution almost as quickly as a feasible solution. For these benchmarks, Gurobi is faster than Z3. Thus, Gurobi with optimization is a good default.

In additional experiments, we study the resource allocation time of switch.p4 as a function of the parameters of the Menshen backend target. We vary the maximum number of entries per table, number of stages, and number of tables per stage, and plot the runtime of Gurobi in both optimal and feasible mode in Figures 9, 10, 11. A vertical line indicates the transition from infeasibility to feasibility for the constraint solver. Across a variety of hardware configurations, we find that the runtime of both modes are quite similar. Figure 9 shows that runtime increases as the maximum number of entries decreases because of an increase in the number of partitions of a table as the maximum number of entries decreases. Figure 10 shows that runtime increases as the number of stages increases because of the increase in the number of indicator variables tracking which stage a table belongs to. In Figure 11, the number of Gurobi variables is constant as we vary the number of tables per stage; The runtime is similar for optimal and feasible modes, but varies significantly depending on whether there is a solution.

8 CONCLUSION

We introduce a new decomposition of the compilation problem for packet pipelines into 3 phases: resource transformation, resource synthesis, and resource allocation, where solver engines (e.g., ILP, SMT, program synthesis) are employed extensively within these phases. We prototype CaT, a compiler for P4 programs based on this decomposition. CaT can handle more programs, reduce pipeline resource usage, compile faster, and requires fewer compute resources than existing compilers. We hope our results encourage compiler engineers for such pipelines to adopt similar ideas.

ACKNOWLEDGEMENTS

We are grateful to the anonymous ASPLOS reviewers, Jiaqi Gao, and Aurojit Panda, for their valuable comments on previous drafts of this paper. We thank Tiancheng Hou, Danny (Xiaoqi) Chen, Sata Sengupta, Divyam Madaan, and Kexin Jin for their help with compiler improvement and providing motivating benchmarks. This work was supported in part by grants from the Network Programming Initiative and the National Science Foundation: NSF-2008048, NSF-1837030, NSF-2107138, NSF-2019302, NSF-1910796.

DATA AVAILABILITY STATEMENT

The source code and data are publicly available in the repository https://github.com/CaT-mindepth.

A EXAMPLE OF SKETCH QUERY WITH GRAMMARS

Figure 12 presents one SKETCH synthesis query generated by CaT to implement a stateless variable in the flowlets.

B ILP ENCODING FOR ALU PROPAGATION CONSTRAINTS

We use the big-M method to obtain an ILP formulation of the constraint

$$\forall u \in I, \forall s \ (beg_u < s \wedge s < end_u) \leftrightarrow prop_{us} = 1$$

For each $u \in I$ and $s \in \{1, \ldots N_S\}$, we use a binary variable lo_{us} as an indicator for $beg_u < s$ and a binary variable hi_{us} as an indicator for $s < end_u$. M is a large constant (e.g., $N_S + 5$).

The following constraints ensure that lo_{us} is 1 if $beg_u < s$ and 0 otherwise.

$$s - beg_u \leq M lo_{us} \quad (1)$$

$$s - beg_u > -M(1 - lo_{us}) \quad (2)$$

If $s - beg_u > 0$ then $lo_{us} = 1$ (1) and if $s - beg_u \leq 0$ then $lo_{us} = 0$ (2).

The following constraints ensure that hi_{us} is 1 if $s < end_u$ and 0 otherwise.

$$s - end_u < M(1 - hi_{us}) \quad (3)$$

$$s - end_u \geq -M hi_{us} \quad (4)$$

If $s - end_u < 0$ then $hi_{us} = 1$ (4) and if $s - end_u \geq 0$ then $hi_{us} = 0$ (3).

The following constraints use lo_{us} and hi_{us} to make $prop_{us}$ an indicator for $beg_u < s < end_u$.

$$lo_{us} + hi_{us} - 2 < M prop_{us} \quad (5)$$

$$lo_{us} + hi_{us} - 2 \geq -M(1 - prop_{us}) \quad (6)$$

If $lo_{us} + hi_{us} - 2 \geq 0$ then $prop_{us} = 1$ (5) and if $lo_{us} + hi_{us} - 2 < 0$ then $prop_{us} = 0$ (6). This means that $prop_{us} = 1$ only if both $lo_{us} = 1$ and $hi_{us} = 1$. Hence, $prop_{us} = 1$ if $s > beg_u$ and $s < end_u$, otherwise $prop_{us} = 0$.

C ADDITIONAL RESULTS FOR RESOURCE ALLOCATION

We experiment with two solvers (Gurobi vs. Z3) and two modes (optimal and feasible solutions) on all our 24 benchmarks. We report both time spent running the solvers and the final number of stage usage to compare between different solvers and different modes. Table 6 shows the detailed results for the running time.

D ARTIFACT APPENDIX

D.1 Abstract

This artifact appendix section describes how to reproduce results demonstrated in this paper by running CaT on Amazon EC2 instances.

D.2 Artifact Check-list (Meta-information)

- **Data set:** Our data for reproducing Table 5 comes from https://github.com/CaT-mindepth/benchmarks repo.
- **Run-time environment:** AWS image with number: ami-0bf331cf0e574fa8b.
- **Hardware:** Amazon EC2 Instances (c5ad.16xlarge).
- **Metrics:** Compilation time and resource (e.g., number of pipeline stages) usage.
- **Output:** Compilation time and resource (e.g., number of pipeline stages) usage.
- **How much disk space required (approximately)?:** 128GB (on EC2 instance).
- **How much time is needed to complete experiments (approximately)?:** 15 hours.
- **Publicly available?:** Yes.
- **Archived (DOI)?:** https://doi.org/10.5281/zenodo.7592970 [17].
- **Latest update:** We put the latest FAQs and updates in this file (https://github.com/CaT-mindepth/CaT-AE/blob/main/UPDATES.md) and please check it before reproducing experiment results.

Table 6: Comparing optimal and feasible for Gurobi and Z3

Benchmark	Gurobi opt		Gurobi sat		Z3 opt		Z3 sat	
	Time (s)	Stages	Time (s)	Stages	Time (s)	Stages	Time (s)	Stages
stateful fw	0.133	4	0.14	12	0.251	4	0.273	4
Blue increase	0.1	4	0.12	12	0.218	4	0.244	4
marple new flow	0.102	2	0.11	12	0.201	2	0.225	2
sampling	0.102	2	0.103	12	0.21	2	0.226	2
flowlets	0.127	3	0.124	12	0.232	3	0.254	3
rcp	0.131	2	0.135	12	0.234	2	0.253	2
learn_filter	0.151	3	0.146	12	0.241	3	0.272	3
marple_tcp	0.117	3	0.119	12	0.223	3	0.241	3
benchmark9	0.169	4	0.161	12	0.81	4	1.8	12
benchmark10	0.165	4	0.163	12	0.781	4	0.501	4
benchmark11	0.19	4	0.18	12	1.13	4	0.639	12
benchmark12	0.178	4	0.168	12	0.967	4	0.502	11
benchmark13	0.18	3	0.173	12	0.88	3	0.506	11
benchmark14	0.202	3	0.182	12	0.889	4	0.523	11
benchmark15	0.241	4	0.22	12	1.033	4	0.585	11
benchmark16	0.198	4	0.183	12	0.963	4	0.553	12
benchmark17	0.159	3	0.142	12	0.845	4	0.459	12
benchmark18	0.156	3	0.146	12	0.922	4	0.447	12
benchmark19	0.169	4	0.154	12	0.946	4	0.531	12
benchmark20	0.181	5	0.17	12	1.086	4	0.6	11
benchmark21	0.145	3	0.148	12	0.733	4	0.462	12
benchmark22	0.162	3	0.168	12	1.578	4	1.578	12
benchmark23	0.215	3	0.201	12	2.544	4	0.798	12
benchmark24	0.174	3	0.149	12	1.53	4	0.595	12

D.3 Description

D.3.1 ***How Delivered.*** The latest version of CaT compiler is open-source in github repo: https://github.com/CaT-mindepth, and the detailed process of reproducing the experiment results is shown in the artifact appendix.

D.3.2 ***Hardware Dependencies.*** Amazon EC2 Instances with ≈ 64 cores such as c5ad.16xlarge.

D.3.3 ***Software Dependencies.*** We have already installed all the software dependencies in the shared Amazon EC2 image.

D.4 Experiment Workflow

D.4.1 ***Fetch the Latest Updates from Github Repos:***

```
$ cd /home/ubuntu/workspace/cat_eval
```

```
$ ./init.sh
```

```
// SKETCH input file for one synthesis query
// ALU grammar specification
int alu(int opcode, int pkt_0, int pkt_1, int pkt_2, int
     immediate_operand) {
    if (opcode == 0) {
        return immediate_operand;
    } else if (opcode == 1) {
        return pkt_0 + pkt_1;
    } else if (opcode == 2) {
        return pkt_0 + immediate_operand;
    } else if (opcode == 3) {
        return pkt_0 - pkt_1;
    } else if (opcode == 4) {
        return pkt_0 - immediate_operand;
    } else if (opcode == 5) {
        return immediate_operand - pkt_0;
    } else if (opcode == 6) {
        return pkt_0!=pkt_1;
    } else if (opcode == 7) {
        return (pkt_0 != immediate_operand);
    } else if (opcode == 8) {
        return (pkt_0 == pkt_1);
    } else if (opcode == 9) {
        return (pkt_0 == immediate_operand);
    } else if (opcode == 10) {
        return (pkt_0 >= pkt_1);
    } else if (opcode == 11) {
        return (pkt_0 >= immediate_operand);
    } else if (opcode == 12) {
        return (pkt_0 < pkt_1);
    } else if (opcode == 13) {
        return (pkt_0 < immediate_operand);
    } else if (opcode == 14) {
        return pkt_0 != 0 ? pkt_1 : pkt_2;
    } else if (opcode == 15) {
        return pkt_0 != 0 ? pkt_1 : immediate_operand;
    } else if (opcode == 16) {
        return ((pkt_0 != 0) || (pkt_1 != 0));
    } else if (opcode == 17) {
        return ((pkt_0 != 0) || (immediate_operand != 0));
    } else if (opcode == 18) {
        return ((pkt_0 != 0) && (pkt_1 != 0));
    } else if (opcode == 19) {
        return ((pkt_0 != 0) && (immediate_operand != 0));
    } else {
        return (pkt_0 == 0);
    }
}
// staged-input tree grammar for implementation (vars0, vars1, and
      vars are specific to each query and are defined in the
     harness below)
generator int expr(fun vars0, fun vars1, fun vars, int bnd){
    if (bnd == 0){
        return vars0();
    }
    int t = ??(1);
    if (t == 0) {
        return vars();
    }
    else {
        return alu(??, expr(vars0, vars1, vars, bnd-1), expr(vars0
    , vars1, vars, bnd-1), expr(vars0, vars1, vars, bnd-1), ??);
    }
}
// specification function, with BCI inputs as arguments
int comp_5(int pkt_arrival, int pkt_last_time00) {
    bit pkt__br_tmp1;
    pkt__br_tmp1 = pkt_arrival-pkt_last_time00>5;
    return pkt__br_tmp1;
}
// harness for synthesis
harness void sketch(int pkt_arrival, int pkt_last_time00) {
    generator int vars0(){
        return {| pkt_arrival |};
    }
    generator int vars1(){
        return {|pkt_last_time00|};
    }
    generator int vars(){
        return {| pkt_arrival | pkt_last_time00|};
    }
    // synthesized expression must be equivalent to specification
    assert expr(vars0, vars1, vars, 2) == comp_5(pkt_arrival,
     pkt_last_time00);
}
```

Figure 12: One example of the generated synthesis query for SKETCH.

*D.4.2 **Part I: Reproduce the Result in Table 5.***

In general, the 10 mutations for program name <X> may be found at the following folder: /home/ubuntu/workspace/-cat_eval/benchmarks/Domino_mutations/<X>/

The running script for Banzai ALU:

Default mode:

```
$ ./quickrun-domino.sh <absolute path to input Domino program> <user-specified absolute path to output JSON file> <Banzai ALU name>
```

Without Predecessor Packing mode:

```
$ ./quickrun-domino-noPredPack.sh <absolute path to input Domino program> <user-specified absolute path to output JSON file> <Banzai ALU name>
```

Without Preprocessing mode:

```
$ ./quickrun-domino-noPreprocessing.sh <absolute path to input Domino program> <user-specified absolute path to output JSON file> <Banzai ALU name>
```

See the **"num_pipeline_stages"** field in the output JSON file for the number of pipeline stages usage.

As for Tofino ALU:

Default mode:

```
$ ./quickrun-tofino.sh <absolute path to input Domino program> <user-specified absolute path to output P4 file>
```

Without Predecessor Packing mode:

```
$ ./quickrun-tofino-noPredPack.sh <absolute path to input Domino program> <user-specified absolute path to output P4 file>
```

Without Preprocessing mode:

```
$ ./quickrun-tofino-noPreprocessing.sh <absolute path to input Domino program> <user-specified absolute path to output P4 file>
```

See the **"num pipeline stages"** in the output P4 file for the number of pipeline stages usage.

As for Chipmunk and Domino compiler:

Should you wish to reproduce the results generated by Chipmunk and Domino compilers, please refer to their artifact evaluation instructions, with links listed below:

Chipmunk: https://github.com/chipmunk-project/chipmunk-project.github.io

Domino: http://web.mit.edu/domino/

*D.4.3 **Part II: Reproduce the Result in Figure 8 and Table 6.***

```
$ cd /home/ubuntu/workspace/cat_eval/CaT-AE/figure_gen
```

```
$ bash figure8.sh
```

*D.4.4 **Part III: Reproduce the Result in Figure 9, Figure 10, and Figure 11.***

Generating the running time and #stages used in ILP.

```
$ cd /home/ubuntu/workspace/cat_eval/CaT-AE/figure_gen
```

```
$ bash figure9.sh
```

```
$ bash figure10.sh
```

```
$ bash figure11.sh
```

Generating the infeasible and feasible boundary.

Infeasible boundary in Figure 9:

```
$ time python3 Gurobi_opt_vs_fea.py 128 16 12 Optimal
```

Feasible boundary in Figure 9:

```
$ time python3 Gurobi_opt_vs_fea.py 256 16 12 Optimal
```

Infeasible boundary in Figure 10:

```
$ time python3 Gurobi_opt_vs_fea.py 256 16 11 Optimal
```

Feasible boundary in Figure 10:

```
$ time python3 Gurobi_opt_vs_fea.py 256 16 12 Optimal
```

Infeasible boundary in Figure 11:

```
$ time python3 Gurobi_opt_vs_fea.py 256 15 12 Optimal
```

Feasible boundary in Figure 11:

```
$ time python3 Gurobi_opt_vs_fea.py 256 16 12 Optimal
```

D.5 Evaluation and Expected Results

The results of resource usage should be exactly the same as those reported in all tables and figures. However, in terms of the running time, they should be within the same magnitude if you use the similar machines to rerun the experiments.

D.6 Notes

We put the latest FAQs and updates in https://github.com/CaT-mindepth/CaT-AE/blob/main/UPDATES.md. If you have any questions, feel free to open an issue there or let us know through email.

D.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] AMD Pensando Infrastructure Accelerators. https://www.amd.com/en/accelerators/pensando.
[2] Benchmarks used to show resource transformation difference. https://anonymous.4open.science/r/program_transformation_ex-6EE7.
[3] Broadcom Jericho2 Ethernet Switch Series. https://www.broadcom.com/products/ethernet-connectivity/switching/stratadnx/bcm88690.
[4] Broadcom Trident Ethernet Switch Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series.
[5] Gurobi Optimizer. https://www.gurobi.com/.
[6] HPCC++: Enhanced High Precision Congestion Control. https://www.ietf.org/id/draft-miao-tsv-hpcc-01.html.
[7] Intel FPGA's High Level Synthesis Compiler. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.
[8] Intel Infrastructure Processing Unit (Intel IPU). https://www.intel.com/content/www/us/en/products/network-io/smartnic.html.
[9] Intel Tofino Programmable Ethernet Switch ASIC. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.
[10] Menshen: An Isolation Mechanism for High-Speed Packet-Processing Pipelines. https://isolation.quest/.
[11] Open-source p4c compiler. https://github.com/p4lang/p4c.
[12] P4-16 language specification. https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html.
[13] P4_16 language specification. https://p4lang.github.io/p4-spec/docs/P4-16-v1.2.2.html#sec-concurrency.
[14] Rate enforcer P4 program github repo. https://github.com/Princeton-Cabernet/AHAB/blob/7c3b1bb/p4src/include/rate_enforcer.p4#L366.
[15] SipHash P4 program github repo. https://github.com/Princeton-Cabernet/p4-projects/blob/master/SipHash-tofino/p4src/siphash24_ingressonly.p4#L587.
[16] Synopsys Introduces Synphony High Level Synthesis. https://news.synopsys.com/index.php?s=20295&item=123096.
[17] Zenodo DOI for CaT compiler. https://doi.org/10.5281/zenodo.7592970.
[18] Actions, P4-16 specification v1.2.3. https://p4.org/wp-content/uploads/2022/07/P4-16-spec.html#sec-table-action-list, 2022.
[19] Alfred V. Aho, Stephen C. Johnson, and Jeffrey D. Ullman. Code Generation for Expressions with Common Subexpressions. *Journal of the ACM*, 1977.
[20] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *ACM SIGCOMM*, 2014.
[21] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful network-wide abstractions for packet processing. In *ACM SIGCOMM*, 2016.
[22] Jean-Philippe Aumasson and Daniel J. Bernstein. Siphash: a fast short-input prf. Cryptology ePrint Archive, Paper 2012/351, 2012. https://eprint.iacr.org/2012/351.
[23] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*.
[24] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. PINT: Probabilistic In-Band Network Telemetry. In *ACM SIGCOMM*, 2020.
[25] Dirk Beyer, Sumit Gulwani, and David A. Schmidt. Combining model checking and data-flow analysis. In *Handbook of Model Checking*. Springer, 2018.
[26] Leyla Bilge, Engin Kirda, Christopher Kruegel, and Marco Balduzzi. EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis. In *NDSS*, 2011.
[27] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *ACM SIGCOMM*, 2013.
[28] J. Cong and Yuzheng Ding. FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1994.
[29] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andre Takach. An Introduction to High-Level Synthesis. *IEEE Design Test of Computers*, 2009.
[30] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 1991.
[31] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.
[32] Wu-chang Feng, Kang G. Shin, Dilip D. Kandlur, and Debanjan Saha. The BLUE Active Queue Management Algorithms. *IEEE/ACM ToN*, 2002.
[33] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. In *ACM SIGCOMM*, 2020.
[34] Xiangyu Gao, Taegyun Kim, Michael D Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. In *ACM SIGCOMM*, 2020.
[35] Pawan Goyal, Harrick M. Vin, and Haichen Chen. Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *ACM SIGCOMM*, 1996.
[36] Gary D. Hachtel and Fabio Somenzi. *Logic synthesis and verification algorithms*. 2006.
[37] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. Modular Switch Programming Under Resource Constraints. In *USENIX NSDI*, 2022.
[38] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *ACM SOSP*, 2017.
[39] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling Packet Programs to Reconfigurable Switches. In *USENIX NSDI*, 2015.
[40] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *SIGCOMM Industrial Demo Session*, 2015.
[41] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 1976.
[42] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *ACM PLDI*, 1988.
[43] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *ACM/IEEE CGO*, 2004.
[44] Yifan Li, Jiaqi Gao, Ennan Zhai, Mengqi Liu, Kun Liu, and Hongqiang Harry Liu. Cetus: Releasing P4 Programmers from the Chore of Trial and Error Compiling. In *USENIX NSDI*, 2022.
[45] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: High precision congestion control. In *ACM SIGCOMM*, 2019.
[46] Robert MacDavid, Xiaoqi Chen, and Jennifer Rexford. Scalable Real-Time Bandwidth Fairness in Switches. In *IEEE INFOCOM*, 2023.
[47] Roland Meier, Petar Tsankov, Vincent Lenders, Laurent Vanbever, and Martin Vechev. NetHide: Secure and Practical Network Topology Obfuscation. In *USENIC Security*, 2018.
[48] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *ACM SIGCOMM*, 2017.

[49] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-Directed Hardware Design for Network Performance Monitoring. In *ACM SIGCOMM*, 2017.

[50] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *ACM SIGCOMM*, 2021.

[51] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-Network Computation is a Dumb Idea Whose Time Has Come. In *ACM HotNets*, 2017.

[52] Shan Sinha, Srikanth Kandula, and Dina Katabi. Harnessing TCPs Burstiness using Flowlet Switching. In *ACM HotNets*, 2004.

[53] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *ACM SIGCOMM*, 2016.

[54] Anirudh Sivaraman, Changhoon Kim, Ramkumar Krishnamoorthy, Advait Dixit, and Mihai Budiu. DC.P4: Programming the forwarding plane of a data-center switch. In *ACM SOSR*, 2015.

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

[56] John Sonchak, Devon Loehr, Jennifer Rexford, and David Walker. Lucid: A Language for Control in the Data Plane. In *ACM SIGCOMM*, 2021.

[57] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. Composing dataplane programs with μP4. In *ACM SIGCOMM*, 2020.

[58] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. Flightplan: Dataplane disaggregation and placement for p4 programs. In *USENIX NSDI*, 2021.

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

[60] Balázs Vass, Erika Bérczi-Kovács, Costin Raiciu, and Gábor Rétvári. Compiling Packet Programs to Reconfigurable Switches: Theory and Algorithms. In *EuroP4*, 2020.

[61] Tao Wang, Xiangrui Yang, Gianni Antichi, Anirudh Sivaraman, and Aurojit Panda. Isolation Mechanisms for High-Speed Packet-Processing Pipelines. In *USENIX NSDI*, 2022.

[62] Sophia Yoo and Xiaoqi Chen. Secure Keyed Hashing on Programmable Switches. In *ACM SIGCOMM Workshop on Secure Programmable Network INfrastructure*, 2021.

Received 2022-10-20; accepted 2023-01-19