P4+BACUS for developing high-performance software switches

Paper # 132

Abstract

High-level domain-specific languages (DSLs) for specifying modern packet processing functionality, like P4 [4] and Netbricks [19], are convenient and promising tools for networking protocol authors. An important target for these DSLs are software switches running on general-purpose hardware: software-based switching is ubiquitous in virtualized data centers and these DSLs allow specification of the desired switching logic through higher-level abstractions with strong type checking and safe runtimes. However, now the onus of translating succinct and safe DSL-based program specifications to high-performance software switch implementations shifts to the DSL compiler. We present a series of architecture-dependent optimization passes inside BACUS, a P4-to-C/DPDK compiler, and demonstrate that the resulting performance improvements are significant — up to 49% over a current state-of-the-art P4 compiler on common real workloads. Our results indicate that P4+BACUS is a compelling method for developing software switches: one can obtain the type-safety and succinctness of a high-level language, and the performance of low-level hand-optimized implementations.

1 Introduction

Software switches are ubiquitous in virtualized datacenters — for example, every hypervisor contains a software switch to switch packets between multiple VMs and the physical NICs. New and innovative protocols imply that newer packet processing functionality is implemented in these switches over time. Because these switches share resources with other VMs, optimization of these packet-processing programs becomes important for higher consolidation in cloud environments. Manual programming and optimization of these software switches is a daunting task: a software switch involves a large body of code, and performance engineering requires highly-skilled programmers; additionally, optimization of these programs is tedious, error-prone, and difficult to reason about, especially in the presence of multiple co-existing protocol functionalities.

Domain specific languages like Click [12], Netbricks [19], and P4 [4] are intended to bridge this gap, by allowing manual specification of protocol functionality using higher-level constructs. In this way, several low-level details are abstracted away from the programmer, and the programmer can focus on the protocol functionality without worrying about mapping and optimizing it for the low-level machine. The authors of a recent P4 compiler, PISCES [22], report that P4 programs are about 40 times shorter than equivalent C-based switch implementations. However, now the onus of efficiently mapping this high-level specification to the underlying machine shifts to the compiler. Several previous studies [11, 13, 2] show that the difference between an unoptimized and a carefully hand-optimized implementation for the same high-level specification can be significant. An ideal compiler should automatically bridge this performance gap between compiler-generated code and hand-optimized code.

We report our experiences with adding architecture-dependent optimizations to a P4 compiler that compiles a high-level P4 program to a lower-level C-based implementation that links with the DPDK infrastructure [9], and eventually gets executed on a multi-socket x86 machine. Our choice of the programming language — P4 — for compilation to a software target merits some discussion. Often, P4 is believed to be a language meant for programming hardware switches, and its support for software backends like C/DPDK is considered only incidental (primarily meant for testing and debugging purposes). Alternative languages like Netbricks [19] or library-based environments like VPP [26] are instead considered to be specifically designed for software switching functionality. We challenge this common belief. We show that P4 abstractions can be compiled to high-performance software implementations. We compare P4 abstractions with abstractions provided by other frameworks like Netbricks and VPP, for their amenability to compiler-based optimization in Sec. 6. In summary, the "higher-level" nature of P4 abstractions allows greater optimization leverage for an automatic compiler. The code generated by our P4-to-C/DPDK compiler, which we call BACUS, is always competitive, and often significantly more efficient than implementations produced by other comparable DSLs and library-based frameworks, including hand-optimized low-level implementations.

The salient features of our optimization strategy in-

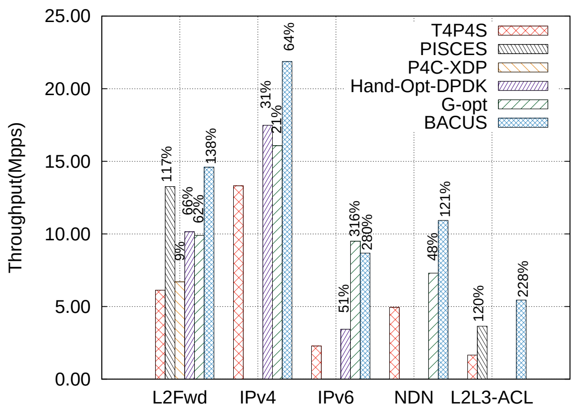

Figure 1: Comparison with other related work.

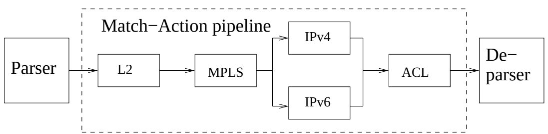

Figure 2: An example P4 program.

clude: (1) a calibration method to determine hardware characteristics automatically through microbenchmarks; (2) data-flow analyses to determine expected network traffic loads at different program points; (3) the table-join algorithm to identify an efficient data-structure layout for realizing the program logic involving match-action tables; and (4) a scheduling pass to exploit the memory-level parallelism available in a general-purpose computer.

PISCES [22] is a related effort aimed at optimizing P4 programs, that focuses on architecture-independent optimizations. In contrast, BACUS implements *architecture-dependent* optimizations that depend on the characteristics of the underlying general-purpose machine. In our evaluation, we assume that architecture-independent optimizations are already available and implemented, and demonstrate improvements achieved by our architecture-dependent optimizations over the current state-of-the-art. Fig. 1 compares the performance of the code generated by BACUS with other comparable systems for various benchmarks we use for evaluation: in summary, we either significantly outperform (by up to 228%) or are comparable to existing state-of-the-art systems (including those that require careful hand-optimization). Compared head-to-head, we obtain 49% performance improvement over PISCES on the identical benchmark used in the PISCES paper [22] for evaluation: L2L3-ACL. The evaluation section (Sec. 5) contains more details on this and other experiments.

2 A brief introduction to P4

While erstwhile data-plane functionality was hard-wired into high-speed ASICs, modern network programming

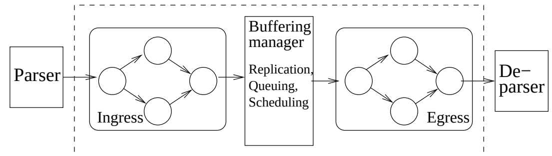

Figure 3: Buffering manager sits between ingress and egress pipelines in a P4 program.

constructs, such as software-defined networking (SDN) constructs, require greater flexibility, control and customization. P4 [16] is a language designed for efficient and easy programming of the data-plane of typical networking systems, and is *target-independent*, i.e., it can be compiled to a variety of hardware and software targets. P4's abstractions include a programmable *packet parser*, a programmable *match-action pipeline*, and a programmable *deparser* to serialize the contents of an output packet. Figure 2 shows an example P4 program that performs L2 switching on a network packet followed by MPLS, IP (v4 or v6 depending on the packet headers) and ACL (e.g., firewall) processing on it.

In addition, P4 programs allow 1:N relationships in the program specification through a *buffering manager*. The P4 match-action pipeline can have three parts: an *ingress pipeline*, a *buffering manager*, and an *egress pipeline* (Fig. 3). Both ingress and egress pipelines map a processing function to each packet. For example, each packet that enters the ingress pipeline also leaves the ingress pipeline. Similarly, each packet that enters the egress pipeline also exits the egress pipeline. The buffering manager allows the P4 program to replicate, drop, or schedule the packets. The ingress pipeline can set a certain attribute called `egress_spec` in a packet's header/metadata fields[1], and based on the value of this attribute, the buffering manager can make decisions related to drop, replication, or scheduling. Each packet that is output by the buffering manager goes through the egress pipeline before reaching the deparser. More precisely, the buffering manager implements a function that maps the value of the `egress_spec` attribute to a collection of packet instances represented as triples: `(packet, egress_port, egress_instance)`. The first two fields in each triple represent packet contents and the egress network port respectively. The third field `egress_instance` represents target-specific semantics, and can be ignored for the discussion in this paper. In practice, an `egress_spec` value may represent a physical egress port, a logical port (such as a tunnel), or

[1] Packet header fields refer to fields directly derived from the incoming packet or fields that will be a part of the outgoing packet; metadata fields implement per-packet local storage required for passing information from one processing stage to another. The handling of both header and metadata fields is identical in P4 programs, and so we use the term 'headers' to represent both packet headers and packet metadata in the rest of the discussion.

Type declarations
parser : packet → packet headers
ingress : packet headers → packet headers
buffering_manager : packet headers stream → packet headers stream
egress : packet headers → packet headers
deparser : packet headers → packet

package(parser, ingress, buffering_manager, egress, deparser) =
(map (parser ≫ ingress))
≫ buffering_manager
≫ (map (egress ≫ deparser))

Figure 4: Functional representation of a P4 program.

a multicast group. For example, if it is a multicast group, the buffering manager would create multiple copies of the incoming packet, and output each with a different `egress_port` value.

The functional representation of a P4 program is shown in Figure 4. The parser takes as input a packet (a string of bytes) and outputs a set of packet header values. Similarly the deparser takes packet header values as input and outputs a packet. The ingress and egress components read/write to packet headers. The `map` functor lifts these functions to operate on *streams* of packets and packet-headers. The buffering manager sits between the ingress and the egress pipelines, and takes as input a stream of packet headers and outputs another stream of packet headers. A full P4 program is represented as a *package* that composes the individual components as shown in the figure. We focus our optimization efforts on the ingress and egress pipelines as they usually constitute the bulk of the processing time.

Most processing logic in the ingress and egress pipelines is typically represented through a set of match-action tables. The match-action table construct allows efficient implementation of match-action rules in the data-plane, and yet allows efficient and structured communication between the control-plane and the data-plane (we discuss match-action tables in more detail in the next paragraph). These match-action tables are connected through one or more "control" blocks that either compose the tables in sequence (through the '`;`' operator) or allow if-then-else style branching. A control block could additionally involve optional declarations of local variables for temporary storage and assignment statements which can assign header fields to values computed through expressions. The expressions are formed using standard non-excepting arithmetic operators, and a few bitvector-manipulation operators such as bit-slicing and bit-concatenation.

Figure 5 shows an example syntax of a match-action table, `ipv4_host`, that encodes a small fragment of common IPv4 forwarding functionality. In a match-action table, the *keys* represent packet header fields (e.g., `hdr.vrf` and `hdr.ipv4.dstAddr` in this example) whose values need to be *matched* with the values stored

```
//arguments obtained from action_data in table entry
action l3_switch(bit<9> port_id, bit<48> new_mac_da,
                 bit<48> new_mac_sa, bit<12> new_vlan) {
  hdr.egress_spec = port;
  hdr.ethernet.dstAddr = new_mac_da;
  hdr.ethernet.srcAddr = new_mac_sa;
  hdr.vlan_tag[0].vlanid = new_vlan;
  hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
}
action l3_l2_switch(...) { ... }
action l3_drop(...) { ... }
action noAction() {}

table ipv4_host {
  key = {
    hdr.vrf : exact;
    hdr.ipv4.dstAddr : exact;
  }
  actions = {
    l3_switch;  l3_l2_switch;
    l3_drop; noAction;
  }
  default_action = noAction();
}
```

Figure 5: Example syntax of a match-action table.

hdr.vrf	hdr.ipv4.dstAddr	action	action_data
1	192.168.1.10	l3_switch	"port_id = .., new_mac_da = .., new_mac_sa = .., new_vlan = .."
100	192.168.1.10	l3_l2_switch	"port_id = .."
1	192.168.1.3	l3_drop	<empty>
…	…	…	…

Figure 6: Example state of a match-action table in Fig. 5.

in the table. The specification of a "match" is done through *match-types*. Each key is associated with a match-type which represents the method of matching, e.g., an *exact* match-type requires that the packet's key value exactly matches the key value stored in the table; similarly, a *longest-prefix* (LPM) match-type requires that the value stored in the table represents the longest prefix that can be matched to the packet's key value. The P4 syntax supports a pre-defined, but extendable, set of match-types. Figure 6 shows an example state of the match-action table defined in Figure 5. Each entry in the table contains values for the two keys and the associated *action* along with its *action-data*. An action can be one of the procedures specified through the `actions` keyword in the table syntax (Fig. 5). If a match is found, the corresponding action is invoked with arguments obtained through the `action_data` field (the fourth column in Fig. 6). If a match is not found, and a *default action* exists in the table (e.g., the default action is `noAction` in the ipv4_host table), then the default action is executed. If a match is not found and no default action exists, the control plane is invoked on the current state of the table and the current state of the *environment*. The environment includes the values of the header fields of the packet. The control-plane would typically add or modify entries in the table before yielding control back to the data-plane, which would retry the table lookup on the updated table.

Type declarations

table(keys, actions).apply() : (packet headers, table state) $\rightarrow$ (packet headers, table state)

Operational semantics

$$\frac{H \vdash \texttt{keys} : \texttt{values} \qquad T \vdash \texttt{t.lookup(values)} = \texttt{hit(entry)} \qquad H \vdash \texttt{entry.action.apply(entry.action\_data)} : H'}{H, T \vdash \texttt{t.apply()} : H', T}$$

$$\frac{H \vdash \texttt{keys} : \texttt{values} \qquad T \vdash \texttt{t.lookup(values)} = \texttt{miss} \qquad \texttt{t}' = \text{ControlPlane}(\texttt{t}, H) \qquad T' = T[\texttt{t}'/\texttt{t}] \qquad H, T' \vdash \texttt{t}'\texttt{.apply()} : H', T''}{H, T \vdash \texttt{t.apply()} : H', T''}$$

ControlPlane($\texttt{t}$, H) = apply a sequence of add_entry/delete_entry/modify_entry operations on $\texttt{t}$

Figure 7: A fragment of the match-action table semantics. H represents the packet headers (including metadata) and T represents the state of the match-action tables. $T[\texttt{t}'/\texttt{t}]$ represents a state of the tables, where all tables are exactly as in T except that $\texttt{t}$ has been replaced by $\texttt{t}'$. Given H and T, the application of a table results in potentially new H' and T'. The first rule is for a table-hit and the second rule is for a table-miss. On a miss, the control-plane will update the table and the data-plane will re-apply the updated table on the packet headers. The updates to the table are made through the control-plane API functions.

Figure 7 shows some relevant rules from the operational semantics of a match-action table lookup. The second rule says that the control plane updates table t to a new table t' (represented by replacing the old table t with the new table t' in the table environment T) before retrying the lookup. Unlike the data-plane which is compiled from the high-level spec, the control-plane needs to be programmed manually by the programmer. To manipulate the table entries, the control-plane programmer needs a set of API functions: the add_entry(), delete_entry(), and modify_entry() functions may be used to add, delete, or modify a table entry respectively from within the control plane. We call these functions, *control-plane API functions*. Because the compiler has flexibility in choosing the data-structure for representing a match-action table, it is also responsible for generating the implementation for the control-plane API functions — clearly, the API implementation would depend on the data-structure choices made by the compiler. Control-plane API code is not performance-critical, as typically the calls to the control-plane are far and few. On the other hand, data-plane accesses to match-action tables constitute the bulk of the runtime.

3 Hardware calibration

We now discuss the typical nature of the performance-critical subsystems in a general-purpose computer for high-performance data-plane processing, and the algorithms to measure and summarize the relevant characteristics. In the process, we also roughly outline the general code generation strategy, which we detail in the next section. We expect these hardware measurements to be performed once for each machine, in an offline *calibration phase* of the compiler. Measuring the required characteristics, and not *reading* them from an architecture manual is preferable for at least two reasons: (1) The required characteristics may not be available in the architecture manuals; even if they are available, the information may be scattered across multiple manuals and automatically stitching them together seems difficult. Measuring through specifically-designed micro-benchmarks is usually easier and more accurate. (2) The runtime environment may be virtual or otherwise different from the original bare-metal characteristics described in the manual. In the following discussion, we outline the measurement strategy along with the results of our measurements on our test hardware described in Table 1.

CPU Subsystem	
Features	**Comment**
CPU Model	Haswell based Xeon E5-2640 v3
Speed	2.60 GHz, 16 Cores/socket, 2 Sockets
Cache Size	L1-d 32 KB, L2 256 KB, L3 20 MB
ROB Size	192 instructions
Memory Subsystem	
Feature	**Comment**
Size	64 GB, distributed on two sockets
#MSHRs	10

Table 1: Dell Poweredge R430 Rackserver details.

For a packet processing application, under normal operation, an incoming network packet is received by a NIC over the wire, and is DMAed to main-memory/last-level caches through PCIe transfers. Similarly, an outgoing network packet is DMAed from the last-level cache (or the main memory) to the outgoing NIC port. CPU processing takes place between these incoming and outgoing transfers. On modern machines, each of these three macro-level operations, i.e., receiving, processing, and transmitting, typically execute in parallel, due to multiple memory channels and banks, high bandwidth memory interfaces supporting multiple in-flight requests, independent DMA controllers, etc. Hence, the overall throughput of a packet processing application is usually limited by the throughput of the slowest operation (bottle-necked operation) among these three parallel operations. While the processing speed is application-dependent, the receive and transmit operations are common to all applications. There are two important aspects of the receive and transmit paths: (a) The PCIe link between the I/O device and the last-level cache/main-memory is a high-latency high-bandwidth interface. To ensure that multiple in-flight PCIe requests can proceed in parallel, the hardware-readable FIFO ring (device ring) needs to be large enough to facilitate this parallelism. (b) The transfer operation involves an expensive memory-mapped I/O (MMIO) operation for the CPU to communicate to the NIC that it has freed/consumed some space in the device ring. Batching of this MMIO operation cost over several packets (also called doorbell-batching) is an important optimization. The first step involves calibrating the required size of the device ring and the desired batch-size for doorbell batching (also called the I/O-batch size or IOBS). We use a simple *echo* application, i.e., an application that just forwards the packets from the receive NIC port to the transmit NIC port, to measure the optimal values of the ring-size (R) and IOBS. The application is run with different values of R and IOBS and the smallest values that yield the desired throughput (beyond which the throughput saturates) are used by the compiler. For our test hardware, R=256 and IOBS=32. In this optimal configuration, the echo application's per-core throughput with 64B packets is 22Mpps (million packets per second).

After the I/O paths have been configured, the focus shifts to the actual processing logic of the application which executes on the CPU-memory subsystem. We first measure the number of *free CPU cycles*, i.e., the time spent by CPU waiting for the I/O packets to arrive. For example, on our test machine, at the optimum IOBS=32, the CPU has around 32 free cycles available; i.e., the throughput of an application remains unaffected if the per-packet processing can complete within 32 cycles. As an aside, we note that for lower IOBS values, the CPU idle times are higher (e.g., 128 for IOBS=4).

Most P4 applications would consume more CPU than the available free cycles. Towards optimizing the CPU usage of a packet processing application, we make two observations: (a) CPU pipelines of modern out-of-order super-scalar (OOO) processors can automatically exploit the available instruction-level parallelism (ILP) in the executing code, provided that the parallel instructions fit in a single *reorder buffer window* (ROB). To allow the architecture to exploit ILP, the compiler needs to schedule the instructions to ensure that instructions that fit within a single ROB can be executed in parallel, i.e., they have minimal data and control dependencies. (b) P4 programs involve lookups into match-action tables: these tables are stored in memory and various levels of the cache hierarchy. This CPU-cache-memory path is a high-latency and high-bandwidth path, and thus hiding the memory latency through parallel outstanding memory requests becomes important. To make effective code-generation choices, the compiler needs to know the size of the reorder buffer window (ROBW), the memory access latencies, and the available parallelism on the CPU-memory interconnect, which we also call *memory-level parallelism* or MLP. We discuss micro-benchmarks designed to measure all three.

To measure ROBW, we execute an instruction sequence of the following form:

```
mem-load; nop; nop; ...  K nops; mem-load
```

We configure both memory-loads to miss all caches and measure the runtime of this sequence with varying K. When K equals the CPU's reorder buffer size, we notice a step-increase in the running time of this sequence, as the two memory accesses can no longer be issued in parallel. Similarly, to measure MLP, we use the following instruction sequence:

```
N mem-loads; nop; nop; ...  K nops; mem-load
```

This is quite similar to the previous instruction sequence, except that it involves N memory accesses before the sequence of K `nops`. For a given N, we expect a similar step-increase when $N+K$ equals the size of the reorder buffer. However, we also expect a step increase in the runtime after N equals the maximum available MLP (the $N+1th$ memory access will now need to be serialized after the Nth memory accesses). Figure 8 shows the result of this microbenchmark on our target machine: by looking at the curve for $N=1$, we conclude that the reorder buffer size of our test machine is 192 instructions[2]. Further, we see that the gap between the curves for $N=9$ and $N=10$ is significantly higher than the gap between the curves for lower values of N (e.g., $N=0$ and $N=1$). This indicates that the maximum achievable memory-level parallelism, or *maxMLP*, on our machine is 10. This measured MLP tallies with the published information about our machine which says that it has 10 miss-handling-status-registers (MSHRs); our measurement indicates that the number of MSHRs is the limiting factor for MLP on our test system.

Next, we are interested in the *effective memory access latency* for different levels of the cache hierarchy. The "effective" latency needs to include any effects due to latency-hiding through MLP exploitation — as we discuss later, a compiler needs to know the effective latency for different MLP settings to make effective code generation and scheduling decisions. The latency of any element in the cache hierarchy is measured by setting up the workload such that it hits that element with a high-probability. For example, if we are interested in measuring L3 latency, we randomly access an array which would just fit in the L3 cache[3]. The effective latency is computed by computing the average latency per memory request for each MLP setting. Fig. 9 plots the results of these measurements on our test machine, and we make two observations from these results: (1) latency hiding through MLP significantly reduces the effective memory latency, and (2) the reduction in the effective access latency is different for different levels of the cache hierarchy — more for memory elements that are farther from the CPU. For a compiler making data-layout and scheduling decisions, this has a significant fallout: e.g.,

[2]ROB size = #Nops + μ-ops for one memory access(186 + 6)

[3]We also measure the cache sizes through the same microbenchmark; we omit this discussion as these techniques are well-known [20].

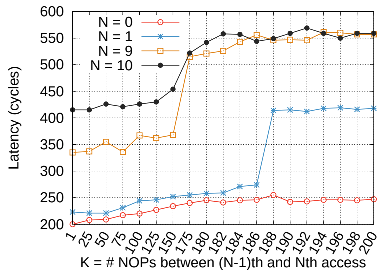

Figure 8: Measuring the size of ROB and degree of MLP.

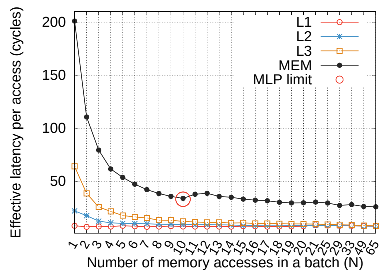

Figure 9: Measuring the effective latency of L3 and main memory.

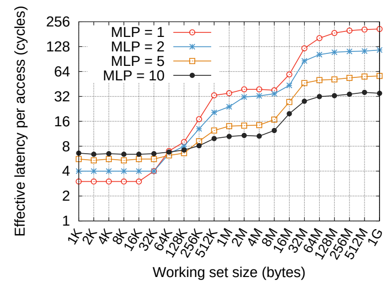

Figure 10: Effective latency vs. working-set size (`WSS`).

if the compiler is generating code such that the expected MLP is ≈ 10, then the *effective* latency difference between L3 and main-memory is much smaller (≈ 20 cycles) than the *actual* latency difference (≈ 165 cycles at MLP=1). Hence a compiler should be less worried about spilling the working-set from L3 to main memory at high MLP.

We summarize these measurements through a function that indicates the effective memory latency ($L^{\texttt{eff}}$) for a given working-set size (`WSS`) and MLP value: $L^{\texttt{eff}}(\texttt{WSS}, \texttt{MLP})$. Fig. 10 plots $L^{\texttt{eff}}$ for some values of MLP for our test machine. For example, if the total working set size of the application is 256 Kb, and we are able to exploit MLP=10 for a particular table access, then the effective latency of that table access is 8 cycles. We use $L^{\texttt{eff}}$ for making code transformation choices (next section).

A well-known idea to exploit available ILP and MLP for packet-processing applications is *packet batching* [11], wherein B packets are processed in one batch. Batching usually creates independent processing threads that can be executed in parallel within the same reorder buffer window. However, it is also well-known that batching can be counter-productive beyond a certain point, due to an associated increase in the working-set size by virtue of handling multiple packets simultaneously, and its precipitous cache-pollution effects. For example, on our test machine, a batch-size greater than 64 is counter-productive. To capture this, we estimate the maximum batch-size B_{max}, before batching becomes counter-productive for a given hardware — B_{max} is largely independent of the application characteristics, but depends on the hardware characteristics and the packet-size, and is thus measured during the calibration phase. B_{max} has important implications on the scheduling transformations discussed in the next section.

4 Code generation and optimization

The P4 program logic is first converted to a High-Level Intermediate Representation (HLIR) which represents the program as a packet-flow or a control-flow graph. Broadly speaking, a node in HLIR could either be a *table node* (indicating the application of a match-action table on the packets in the packet stream) or a *conditional node* (indicating conditional control flow such as if/else based on some condition on the packet's header fields). The nodes are linked through directed edges, e.g., a conditional node would have two outgoing edges representing the true and false branches of the condition respectively. The syntactic conversion of a P4 program to HLIR syntax is straightforward and our code generation and optimization algorithms work on the HLIR syntax.

As with most compilers, we structure our code optimization strategy as a series of analysis and transformation passes. Our optimizer involves three transformation passes, namely *traffic estimation and batch-size allocation*, *table-join*, and *memory-access scheduling and prefetching*.

We first group the HLIR nodes into *basic blocks*. As in traditional compilation literature, a basic block is a maximal path in the HLIR graph that contains no incoming edges (except at the first node of the path) and contains no outgoing edges (except at the last node of the path). We discuss each transformation pass separately in the following sections.

4.1 Traffic estimation and batch-size allocation

The traffic-estimation pass involves estimating the relative traffic profile at each basic-block in the HLIR graph. The traffic profile indicates the expected number of packets processed at the node, relative to the number of packets received at the ingress port. For example, if a conditional node receives traffic T at its input, then it may transmit traffic $T * p$ and $T * (1 - p)$ (for some p, s.t. $0 \leq p \leq 1$) respectively on its two outgoing branches. Similarly, if the buffering manager may drop (or replicate) a packet, then its outgoing traffic would be less (or more) than its incoming traffic. We estimate the average conditional probabilities, drop-probabilities, and replication probabilities at compile time – these estimates could be based on static analysis of the program or could be aided through available execution profiles or programmer-provided hints.

The estimation of the steady-state traffic at each HLIR node involves a forward data-flow analysis pass on the HLIR graph. The incoming traffic at an HLIR node is the sum of the traffic on the incoming edges at that node. For nodes with a single outgoing edge, the outgoing traffic at that node equals the incoming traffic at that node. For nodes with multiple outgoing edges, the traffic is split based on some estimated split ratio p. These functions that relate the output traffic of a node to its input traffic, are also called the dataflow analysis's *transfer functions* [1]. The transfer function of the buffering manager is special: it needs to capture the effect of operations like drop, replicate, etc. After the transfer functions have been estimated, the computation of the traffic profile through data-flow analysis is straightforward. It is easy to see that after this analysis, the estimated incoming traffic at all nodes within a single basic block will always be identical.

The traffic profile is needed to make packet-scheduling decisions. For example, if the start node S of the HLIR graph has incoming traffic T, and another node in the graph, say X, has incoming traffic $T/16$, then, in theory, we should schedule S sixteen times more often, on average, than we schedule X. To realize this, we implement packet queues at each HLIR node: the execution of a node X, through its function `processX`, involves consuming a packet from one of X's incoming queues, processing the packet, and producing a new packet at one of X's outgoing queues. Thus, in our example, `processS` should be invoked sixteen times more often than `processX`. The queues are sized to be large enough to absorb expected traffic jitter. Any estimation errors, or large fluctuations in traffic patterns, could potentially cause some of the queues to become empty or full: to handle this, we need to ensure (either through static analysis or dynamic runtime checks) that a node's processing logic is executed only if its incoming queue is not-empty and its outgoing queue is not-full.

Batch-size allocation: Apart from packet scheduling, traffic estimation also indicates the potential parallelism that may be exploitable at each node through packet batching. Because packet batching has limits (it becomes counter-productive after B_{max} batch-size), traffic analysis is used to identify the optimal batch-sizes that should be used at each HLIR node. We first identify the HLIR nodes with the maximum traffic, and allow them the maximum batch-size B_{max}. The batch sizes of all other nodes are based on their respective traffic profiles relative to these maximum-traffic nodes, e.g., if the traffic at one HLIR node X is T_X, and the traffic at another node Y is $\frac{T_X}{16}$, then the batch-size at Y should ideally be a sixteenth of the batch-size at X. Thus, batch-sizes for all nodes are *allocated* in proportion to their relative traffic profiles and are upper-bounded by B_{max}. This strategy ensures that the contribution of packet data to the program's working set remains upper-bounded by B_{max} packets.

4.2 Table-join transformation pass

The high-level representation of packet processing logic in P4 as a sequence of match-action tables opens opportunities for the compiler to potentially reduce the number of table lookups by *joining* multiple tables into a single larger table in the optimized implementation. Often this reduction in the number of lookups may also translate into a reduction in processing time. Our table-join transformation pass is aimed at performing such transformations wherever profitable.

The table-join transformation is a *local optimization* pass [1], i.e., it transforms each basic-block in isolation. The input to the table-join algorithm is a basic block (a sequence of nodes N) and the allocated batch-size (B) at that basic block as computed in the traffic-estimation pass. The output of the algorithm is a potentially transformed sequence of nodes N' that is logically equivalent to N and is expectedly more efficient to execute for the given B. There are three important considerations during the table-join pass: (1) We need to correctly handle all read-after-write (RAW), write-after-write (WAW) and write-after-read (WAR) data-dependencies between tables. (2) The table-join should be performed only if we expect the lookup into the joint (larger) table to be faster than the two individual lookups into the smaller tables; in other words, the join should be *profitable*. (3) The join operation should be completely transparent to the control-plane, i.e., the control-plane API functions used to manipulate the table entries need to be appropriately re-implemented such that their semantic effect on the joint table is identical to their intended effect on the individual tables. For dependency analysis, we utilize the high-level information about table keys and actions available in program syntax, to label each HLIR graph edge with one of the four data-dependence properties, namely no-dependence, RAW, WAR or WAW, in a graph pre-processing step.

Algo. 1 presents our algorithm for the table-join pass. The `JoinTables` function takes two arguments, the input sequence of nodes N in the basic block, and the allocated batch-size B, and returns N', the new (potentially transformed) sequence of nodes. Each node in N (except potentially the last node) represents a match-action table. `cantJoin` represents a set of table-pairs that *cannot be joined*, and is populated as the function executes. It may not be possible to join two tables either because they have data-dependencies that cannot be handled, *or* their joins are not expected to be profitable.

Function *JoinTables(N, B)*
cantJoin = {}
$N' = N$
while *true* **do**
(found,T1,T2) = pickTablePair(N', cantJoin)
if $\neg$*found* **then**
return N'
end
else
if *mkTblsAdjacent(N', T1, T2)* $\wedge$ *joinIsProfitable(T1, T2, B)* **then**
N' = joinTablePair(N', T1, T2)
end
else
cantJoin.insert((T1,T2))
end
end
end

Function *mkTblsAdjacent(N', T1, T2)*
while *!tblsAreAdjacent(N', T1, T2)* $\wedge$ *swapRight(N', T1)* **do**
end
while *!tblsAreAdjacent(N', T1, T2)* $\wedge$ *swapLeft(N', T2)* **do**
end
return tblsAreAdjacent(N', T1, T2)

Algorithm 1: Table join algorithm.

N' is initialized to N, and the while loop greedily selects two tables to be joined. This selected pair of tables, `T1` and `T2`, should not already be present in `cantJoin`. The greediness of the algorithm stems from greedily picking the pair of tables that would yield the most profitable join (we discuss the notion of profitability later). The chosen pair of tables need not be adjacent, and so the `mkTblsAdjacent` function attempts to make them adjacent by iteratively swapping the first table `T1` with its right neighbor (as far as possible), and then iteratively swapping the second table `T2` with its left neighbor (as far as possible). These swapping (or reordering) operations can only be performed if no data dependencies exist between the two tables being swapped. If `mkTblsAdjacent` succeeds in bringing `T1` and `T2` adjacent, and the resulting join is profitable, then we join the two tables into one table, and update N' accordingly. Otherwise, we update `cantJoin` to avoid considering the same pair of tables again.

Fig. 11 shows an example HLIR graph demonstrating the `swapLeft` and `swapRight` operations. As discussed previously, the `swapLeft` and `swapRight` functions succeed only if the two nodes (or tables) being swapped do not have a data-dependence.

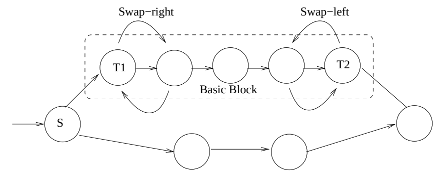

Figure 11: An example HLIR graph demonstrating `swapLeft` and `swapRight` operations.

For determining if a join is profitable (`joinIsProfitable()` function), we use the following logic: we first estimate the working-set size of the current set of tables (of the whole program). Let this be $\mathtt{WSS}^{\mathtt{cur}}$. We then estimate the new working-set size ($\mathtt{WSS}^{\mathtt{new}}$) of the program, if the join is effected: $\mathtt{WSS}^{\mathtt{new}} = \mathtt{WSS}^{\mathtt{cur}} - \mathtt{T1.size} - \mathtt{T2.size} + \mathtt{T1T2.size}$, where `T1.size`, `T2.size`, and `T1T2.size` represent the sizes of `T1`, `T2`, and the joined table `T1T2`. The join is profitable only if: $L^{\mathtt{eff}}(\mathtt{WSS}^{\mathtt{new}}, B) \leq 2 * L^{\mathtt{eff}}(\mathtt{WSS}^{\mathtt{cur}}, B) + \Delta$. Recall that $L^{\mathtt{eff}}(\mathtt{WSS}, B)$ represents the effective average memory-access latency for a given `WSS` and batching-factor B (B acts as a proxy for the available MLP), and was measured through micro-benchmarks (Fig. 10). Δ represents the computational cost of indexing a table (sans the actual memory access). For example, $\Delta = 42$ cycles for our test machine.

The actual joining of the match-action tables is based on the *natural-join* operator $\bowtie$ in relational algebra (also commonly used in relational databases). However, there are some subtleties related to joining P4 match-action tables that merit discussion. First, we cannot join across tables with arbitrary match-types: we join only across tables that use only `exact` match-types. Joining across exact matches is easier than joining across more sophisticated match-types, such as longest-prefix match. Also, exact matches are the most common match-types found in P4 programs, and are thus a good candidate for optimization. We leave the generalization of the table-join algorithm to other match-types for future work.

Assuming no RAW dependencies between `T1` and `T2`, the *joint-table* ($\mathtt{T1} \bowtie \mathtt{T2}$) can be formed by *joining* the individual entries (with matching common keys). The *joint-entries* are indexed using the union of the keys of the two individual tables (which we call the *joint-key*), and the corresponding actions are formed by sequencing the individual actions in T1 and T2 in the original execution order, i.e., T1's actions are executed before T2's actions. This preservation of the sequence of actions of the two tables correctly handles any WAW or WAR dependencies between T1 and T2. If for a particular key, one of the tables' actions is the default action, then the default action of the respective table is used as its action in the corresponding joint-entry. Similarly, if both tables have a default action, then the joint-table's default action involves executing the two individual default actions in their original execution order.

While joining, we also need to correctly handle table *misses*, i.e., cases where the required entry is not present in the table, and there is no default action. Recall that the control-plane is invoked in these situations. To handle this correctly, for any joint-key that would miss in `T2` (but hit in `T1`), we create an entry in the joint-table to appropriately indicate that the miss happened in `T2` and not in `T1`; a "hidden action" (hidden from the user) is used for these entries to transfer control to the control-plane after executing `T1`'s respective action for that entry. Thus, for the control-plane, it appears as though `T1` hit and `T2` missed.

For joint-keys that would have missed in `T1`, we simply do not emit an entry in the joint-table — misses in the joint-table are treated as `T1` misses, as far as the control-plane is concerned.

The handling of default actions is similar: if `T1` has a default action but `T2` does not have a default action, then the joint table's default action transfers control to the control-plane after executing `T1`'s default action (similar to the hidden-action construct).

After the control-plane executes its logic to update the table and resubmit the request, the data-plane needs to resume packet handling at the table that missed. For example, if `T2` missed, then the packet processing needs to be resumed at `T2` (and not at the joint-table). To achieve this, we maintain a copy of the original nodes N in the basic-block (in addition to the transformed basic-block nodes N') in the program. On a table miss, the control-plane is invoked, which in turn, transfers control to the respective *original* table in N after miss-handling (e.g., control is transferred to the original copy of `T2` if it was a `T2` miss). Thus, our transformed program has redundant paths: a fast-path with joined tables N' which is traversed on the data-path, and an equivalent original slow-path N which is executed when the control-plane yields after miss-handling. Notice that iterative joining of tables in N' has no effect on N.

Finally, a table-join transformation also needs to ensure that the control-plane APIs, such as `add_entry`, `delete_entry` and `modify_entry` are appropriately re-implemented to update both the original, as well as the joint-tables accordingly.

In general, the problem of identifying the optimal tables to join, given data-dependence and profitability constraints, is NP-complete. Our linear-time greedy algorithm may not yield the optimal solution, but we expect it to work well in practice.

Table-join across RAW dependencies: While handling of WAR and WAW dependencies during table-joins is straightforward, table-joins across RAW dependencies are more subtle. In general, joins across RAW dependencies are not possible because the key of the second table `T2` may depend on the result of some computation performed by `T1`'s action. However, most P4 programs involve actions that set the header fields to constants. In these cases, it may be possible to determine the output values of header fields uniquely and deterministically based on the input values of the table keys (the choice of which action gets executed still depends on the input key value). We refer to such dependencies, where the first table `T1`'s output header value is uniquely determined by `T1`'s key-values, as $\text{RAW}^{\texttt{fd}}$ dependencies: the written values are functionally-dependent (`fd`) on the key values of the first table. For $\text{RAW}^{\texttt{fd}}$ dependencies, the joint-table can be computed statically by first computing `T2`'s key for each `T1` key (at compile time), and then emitting the corresponding joint-entry with the computed joint-key.

In practice, because many RAW dependencies in P4 programs are $\text{RAW}^{\texttt{fd}}$ dependencies, joins across such dependencies become possible. We have implemented joins across $\text{RAW}^{\texttt{fd}}$ dependencies, and find that they are consequential for the optimization of many P4 programs (including the ones discussed in our evaluation section).

4.3 Memory-access scheduling/prefetching

After table-join, the compiler chooses data-structures to implement each (potentially joined) match-action table. Our choice of data-structures is identical to that of previous work: we use the cuckoo-hash table for exact-match tables [28] and DIR24-8 trees [7] for longest-prefix-match tables. Based on these data-structure choices, each table-lookup operation gets transformed into potentially multiple data-structure lookup operations. At this stage, we implement code transformations for memory-access scheduling and prefetching to maximize MLP.

Memory-access scheduling and prefetching involves reordering memory accesses for different packets within a single node of the basic block. For this, we first identify the *likely-expensive* memory accesses (that are likely to miss the L1 cache) in the table-lookup logic at a single node. All accesses involving indexing into a match-action table are identified as likely-expensive. Subsequently, the code is transformed such that M likely-expensive memory accesses from M different packets are scheduled together, so they fit within a single ROB window. Note that M (the degree of grouping of expensive memory accesses within a single table lookup) is usually different from B (the packet batch-size at that HLIR node/table) — while B is aimed at optimizing MLP for packet access, M is aimed at optimizing MLP during table access. However, it is true that $M \leq B$, as B dictates the maximum available parallelism during table access. In previous work [2], this distinction between *B* and *M* has been called batching and *sub-batching* respectively. Further, the memory accesses, whose result is not needed immediately (by virtue of sub-batching), are transformed to use the more efficient software-prefetch instruction. Software-prefetching improves performance because it provides performance hints to the hardware for its internal scheduling of memory accesses [14].

5 Evaluation

We have implemented our algorithms inside a P4 compiler, which we call BACUS. BACUS is derived from an existing P4 compiler, called `T4P4S` [13, 23]. We first compare the implementations produced by BACUS with and without the optimizations described in the paper, before comparing BACUS with other state-of-the-art compilers for P4 programs. For our experiments, we measure the per-core throughput of the compiled code — because packet processing is data-parallel, the throughput typically scales well with increasing number of cores till the PCIe bandwidth capacities of the machine. We run the compiled switch on a standalone machine (characteristics described in Table 1) and use another machine to generate traffic for this switch. The machines are connected through two 40GBe network ports and two 10GBe network ports: this network bandwidth is enough to saturate the processing resources in the system under test. Because we are interested in measuring the limits of the system, we perform our experiments with minimum-sized 64-byte packets, unless otherwise specified; our conclusions translate similarly to larger packet sizes.

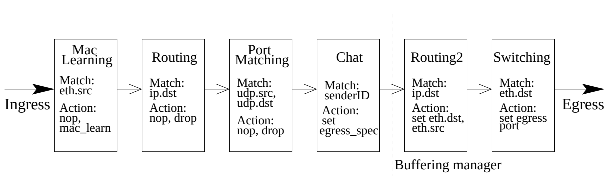

Figure 12: ChatServer application control flow.

We use five different applications to test our compiler. For some applications, we use two configurations: one where the size of the tables is small, and another where the size of the tables is large. Wherever possible, we use the exact same table sizes in our applications as used in previous work. (1) L2Fwd: a simple L2 switch with MAC learning: in this application, for each packet, two table lookups are performed in sequence, one each involving the source and destination MAC addresses of the incoming packet. For small table sizes, the two tables are populated each with 256 entries. For large table sizes, the two tables are populated each with 16M entries (as also done in previous work [10]). (2) IPv6 Forwarding: a longest-prefix match lookup to a table containing 200,000 random entries is performed on the destination address to find the egress port.The packet size used for this application is 78 bytes, as in previous work [10]. (3) L2L3-ACL: an L2-L3 switch with access control functionality having complex control-flow and if-then-else branching, with 7 table lookups (5 exact type, 1 ternary type and 1 LPM type). This application was used in the PISCES paper [22]. In our experiments for small table sizes, all tables for this benchmark have 4 entries. For large table sizes, the largest tables have 16 million entries. (4) ChatServer: a stateless L5 chat application on UDP that forwards messages from senders to receivers in a given chat room (multicast). The multi-cast forwarding decision is based on two layer-5 custom header fields — sender ID and chatRoom number. The P4 implementation of the ChatServer contains six table lookups — MAC learning, routing, UDP port matching, sender ID matching, egress routing, and egress switching. The two routing lookups are LPM-type and the rest are exact-match type. In the fourth table of this application (Fig. 12), the lookup on the sender ID returns the chat-room number which is written to `egress_spec`. The buffering manager replicates the packets based on the number of members in the chatroom. For small table sizes, we populate all tables with 256 entries; for large table sizes, the largest tables have 16 million entries. The ChatServer is intended to demonstrate a higher-level application programmed in P4 that involves replication with a complex pipeline of match-action transformations. (5) MPLS: Refer figure 2. For small table sizes, we populate all tables except ACL with 256 entries. For large table sizes, the L2 table has 16 million, MPLS has 256, and IPv4 / 6 have 200,000 entries respectively. ACL has 4 entries in both the cases. The packet size for this benchmark is 82 bytes.

We first study the effects of our transformations on L2Fwd, a simple application involving two lookups per packet. We isolate the five transformations, namely batching (`B`), table-join (`TJ`), memory-access scheduling or sub-batching (`S`), and software prefetching (`P`). We characterize the effect of our transformations on tables of different sizes, from 16 to 4M entries in the L2Fwd tables. For this experiment, we deliberately modified our `joinIsProfitable` function in the table-join algorithm to always return true (i.e., we always join the two tables). Figure 13 plots our results. Notice that `TJ`, when used without other transformations, usually degrades performance for tables with more than 64 entries: this is because the latency of accessing one larger table is higher than the sum of latencies for accesses to two smaller tables. However, in the presence of other transformations that exploit MLP, `TJ` yields significant speedups till the tables are of size 1K entries each. Beyond a certain point (4K entries in this experiment), TJ

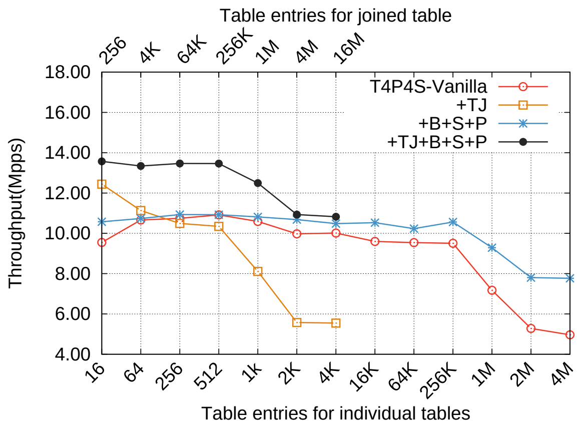

Figure 13: Effect of TableJoin on L2Fwd application.

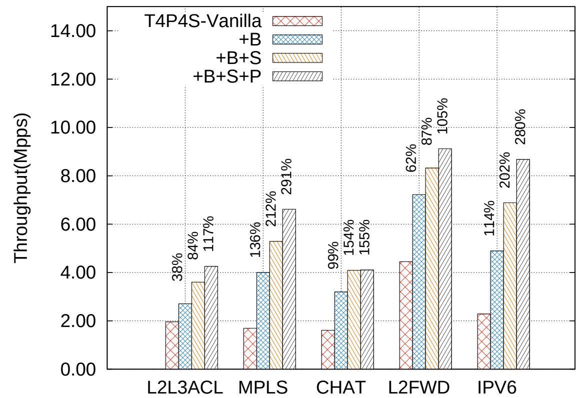

Figure 15: Effect of scheduling and prefetching.

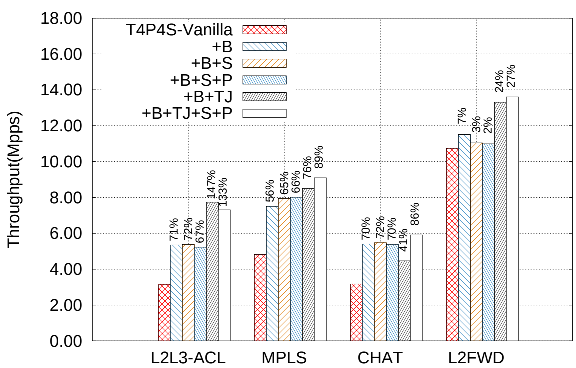

Figure 14: Effect of optimizations for small tables.

becomes infeasible as the joint-table exceeds the size of the allocated memory. It is also evident that the advantages of `B+S+P` increase with increase in table size, due to greater benefits from MLP.

Effect of optimizations on small tables: We next study the effect of our transformations on all our applications configured with small tables (Fig. 14). We make some interesting observations: (1) `B` provides significant gains by enabling higher ILP and MLP. (2) `B+S+P` does not improve significantly over `B` at small table sizes. (3) `B+TJ` is much more effective at improving performance by reducing the table-lookup overheads. (3) `B+TJ+S+P` usually provides extra performance by increasing MLP while accessing the larger joint tables. It is interesting to see that while `B+TJ` usually performs better than `B+S+P` for these experiments involving small table sizes, it performs worse for the ChatServer. This is because while the joint table fits into the L3 cache for all other applications, it spills to main memory for the ChatServer. Using `TJ` becomes unprofitable if the larger table is accessed through main memory, unless `S+P` are used to hide the extra latency of the main memory. These results clearly highlight the relevance of using the effective latency (Fig. 10), and not the actual latency while making table-join decisions.

Effect of optimizations on large tables: We next study the effects of our optimization passes for cases where the sizes of the tables are relatively large. In these cases, `TJ` becomes irrelevant, as the joint-table sizes can no longer fit in the allocated memory. Our algorithm is able to infer this automatically and defaults to the using `B+S+P` transformations for these workloads. The results (Fig. 15) clearly demonstrate the incremental value of each transformation over other transformations.

Comparison with existing tools and frameworks: Figure 1 compares BACUS with existing compilers for the P4 language, namely (1) T4P4S[13] and (2) PISCES [22]. We also compare with other tools that involve manual annotation/optimization of these programs: (3) G-opt[10] requires the programmer to annotate the expensive memory accesses in C code and employs a source-to-source compiler to transform the code to hide the memory latency, (4) XDP [27] is a high performance, programmable network data path available in the Linux kernel and P4C-XDP [18] is a backend for the P4 compiler [17] targeting XDP, and (5) hand-optimized DPDK [9] code. We only show comparisons where they were possible, e.g., where hand-optimized implementations are available. We could not compare with PISCES for all workloads because we could not get PISCES to compile some of the applications. In summary, the performance of BACUS-compiled code is always competitive and often significantly better than the implementations produced by other competing frameworks, including state-of-the-art compilers like PISCES, kernel-based frameworks like XDP and hand-optimized implementations on top of library-based frameworks like DPDK.

We omit direct performance comparisons with Netbricks as we find that Netbricks performance is usually inferior to hand-optimized DPDK code, as also reported in their paper [19]. The current Netbricks compiler supports only basic optimizations like zero-copy [24] and batching [19] that are already implemented in most hand-optimized DPDK implementations, and are also implemented in BACUS. Further, we find that unlike BACUS, the support for batching in Netbricks is relatively ad-hoc, e.g., they do not analyze the traffic pattern and upper-bound the maximum batch-size by B_{max}. In our experiments with the ChatServer, we have noticed that ad-hoc batching implementations that do not bound the maximum batch-size at any node to B_{max} may result in significant drop in throughput. For example, if the expected replication factor in our multi-cast ChatServer is 4, then the maximum batch-size used at program entry should be $B_{entry} = \frac{B_{max}}{4}$. If larger batch-sizes are used at program entry, e.g., if $B_{entry} = B_{max}$, we observe a drop in throughput by 36%. We expect similar performance anomalies with Netbricks, where the batching factor at program entry is assumed to be fixed and independent of the application. In contrast to Netbricks, we have tested BACUS on complex packet-processing programs involving deep pipelines and potential packet replication.

6 Related Work

In recent years, the increasing relevance of virtualization and software defined networking has brought programmable network functions into focus. Some previous studies have shown that packet processing workloads on general-purpose PCs are usually bottle-necked by the performance of the shared interconnect (the "front-side bus") connecting the CPUs to the memory subsystem [3, 25, 6, 5]. Routebricks authors implemented these optimizations by hand for a fixed set of workloads (namely L2 forwarding, IP routing, and IPSec encryption), running on a fixed machine architecture. Subsequent work in this direction involved studying *batching* [11], efficient hashing strategies [28], and hiding memory latency through manually-orchestrated "context-switching" among threads [10], and the use of commonly available accelerators, e.g., GPUs, for packet processing [8]. Our work builds upon ideas presented in these previous works, albeit in the setting of an automatic compiler.

The PISCES paper [22] discusses architecture-independent functional optimizations and the authors claim performance improvements of up to 50% over their baseline compiler through these optimizations. We are able to achieve significant performance improvements over PISCES (49% in head-to-head comparisons) through additional architecture-dependent optimizations.

Netbricks [19] is a related effort aimed at designing higher-level abstractions for eliminating virtualization overheads in network processing pipelines. Netbricks abstractions are richer and more general, e.g., they support stateful abstractions for higher-layer networking protocols like TCP, such as bytestream processing. However, their packet-processing abstractions are lower-level than P4. For example, they allow arbitrary user-defined functions (UDFs) with the map functor, in contrast to P4 which mandates that the logic be specified through structured match-action tables. For this reason, while some of our optimization passes, such as traffic-estimation and memory-access scheduling and prefetching are applicable to Netbricks programs, our table-join transformation pass is not: while each individual UDF can be highly optimized, it seems difficult to automatically combine multiple UDFs into a single more efficient UDF. Further, Netbricks does not provide a language-based separation between the data-plane and the control-plane: to achieve such functionality, programmers need to manually manipulate the data-structures used by the UDFs in the control-plane. This tight integration between the control-plane and data-plane in Netbricks thwarts optimization opportunities; in contrast, P4's control-plane API functions abstract this interface allowing for greater optimization flexibility. Overall, Netbricks design is aimed at alleviating virtualization overheads, and not at exposing optimization opportunities for an optimizing compiler. Empirically, the performance of the code generated by the current Netbricks compiler is either similar or significantly inferior to the performance of the code generated by P4+BACUS.

Library-based frameworks like VPP [26, 15] provide API support for specifying network protocol functionality; unlike DSLs, API-based frameworks largely depend on hand-optimization by the programmer for good performance. While some APIs provide interfaces for easy manual specification of common optimization patterns such as batching and vectorization, more involved optimizations relying on dataflow-analyses, such as traffic-estimation and table-join, are difficult to realize in these frameworks.

Our table-join algorithm achieves similar goals as OVS's mega-flow cache [21], albeit using compile-time optimizations (in contrast to run-time caching). Unlike table-join, mega-flow caching is best-effort and could incur significant runtime overheads. In general, the two approaches are orthogonal and mutually inexclusive: a mega-flow cache could be enabled with a BACUS-optimized program to obtain cumulative benefits.

7 Conclusions and future directions

To conclude, we propose algorithms for architecture-dependent optimizations in BACUS, a P4 compiler for generating software switches. The resulting performance improvements are significant and make a compelling case for using P4+BACUS for developing high-performance software packet-processing switches. In future, understanding if P4 abstractions can be elegantly extended to also support stateful abstractions seems like an intriguing direction of exploration.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, & Tools with Gradiance*. Addison-Wesley Publishing Company, USA, 2nd edition, 2007.

[2] A. Bhardwaj, A. Shree, V. B. Reddy, and S. Bansal. A preliminary performance model for optimizing software packet processing pipelines. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, pages 26:1–26:7, New York, NY, USA, 2017. ACM.

[3] A. Bianco, R. Birke, D. Bolognesi, J. M. Finochietto, G. Galante, M. Mellia, M. L. N. P. P. Prashant, and F. Neri. Click vs. linux: two efficient open-source ip network stacks for software routers. In *HPSR. 2005 Workshop on High Performance Switching and Routing, 2005.*, pages 18–23, May 2005.

[4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[5] N. Egi, M. Dobrescu, J. Du, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, L. Mathy, and S. Ratnasamy. *Understanding the Packet Processing Capability of Multi-Core Servers*.

[6] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy. Towards high performance virtual routers on commodity hardware. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, pages 20:1–20:12, New York, NY, USA, 2008. ACM.

[7] P. Gupta, S. Lin, and N. Mckeown. Routing lookups in hardware at memory access speeds. pages 1240–1247, 1998.

[8] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: A gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[9] *Intel Data Plane Development Kit*. http://dpdk.org/.

[10] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen. Raising the bar for using gpus in software packet processing. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 409–423, Oakland, CA, 2015. USENIX Association.

[11] J. Kim, S. Huh, K. Jang, K. Park, and S. Moon. The power of batching in the click modular router. In *Proceedings of the Asia-Pacific Workshop on Systems*, APSYS '12, pages 14:1–14:6, New York, NY, USA, 2012. ACM.

[12] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[13] S. Laki, D. Horpácsi, P. Vörös, R. Kitlei, D. Leskó, and M. Tejfel. High speed packet forwarding compiled from protocol independent data plane specifications. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 629–630, New York, NY, USA, 2016. ACM.

[14] J. Lee, H. Kim, and R. Vuduc. When prefetching works, when it doesn't, and why. *ACM Trans. Archit. Code Optim.*, 9(1):2:1–2:29, Mar. 2012.

[15] L. Linguaglossa, D. Rossi, S. Pontarelli, D. Barach, D. Marjon, and P. Pfister. *High-speed Software Data Plane via Vectorized Packet Processing*. https://perso.telecom-paristech.fr/drossi/paper/vpp-bench-techrep.pdf.

[16] *P4 Language Specification*, May 2017. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf.

[17] *P4C Compiler for P4*. https://github.com/p4lang/p4c/.

[18] *P4C-XDP: Backend for the P4 compiler targeting XDP*. https://github.com/vmware/p4c-xdp.

[19] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. Netbricks: Taking the v out of nfv. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 203–216, Berkeley, CA, USA, 2016. USENIX Association.

[20] D. A. Patterson and J. L. Hennessy. *Exercise 5.2 of "Computer Architecture: A Quantitative Approach (Third Edition)"*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[21] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, Oakland, CA, 2015. USENIX Association.

[22] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford. Pisces: A programmable, protocol-independent software switch. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 525–538, New York, NY, USA, 2016. ACM.

[23] *T4P4S - Translator for P4 Switch programs*. https://github.com/P4ELTE/t4p4s.

[24] M. N. Thadani and Y. A. Khalidi. *An efficient zero-copy I/O framework for UNIX*. Sun Microsystems Laboratories, 1995.

[25] B. Veal and A. Foong. Performance scalability of a multi-core web server. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS '07, pages 57–66, New York, NY, USA, 2007. ACM.

[26] *Vector Packet Processing*. https://wiki.fd.io/view/VPP.

[27] *XDP - eXpress Data Path*. https://www.iovisor.org/technology/xdp.

[28] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 97–108, New York, NY, USA, 2013. ACM.