Whippersnapper: A P4 Language Benchmark Suite

Huynh Tu Dang[*] Han Wang[†‡] Theo Jepsen[*] Gordon Brebner[§]
Changhoon Kim[‡] Jennifer Rexford[¶] Robert Soulé[*‡] Hakim Weatherspoon[†]
[*]Università della Svizzera italiana [†]Cornell University
[§]Xilinx Inc. [‡]Barefoot Networks [¶]Princeton University

ABSTRACT

As P4 and its associated compilers move beyond relative immaturity, there is a need for common evaluation criteria. In this paper, we propose Whippersnapper, a set of benchmarks for P4. Rather than simply selecting a set of representative data-plane programs, the benchmark is designed from first principles, identifying and exploring key features and metrics. We believe the benchmark will not only provide a vehicle for comparing implementations and designs, but will also generate discussion within the larger community about the requirements for data-plane languages.

1. INTRODUCTION

The P4 language [7] has rapidly captured the attention of the networking community. The P4 Language Consortium boasts over 12 university members and 44 industry members, including companies such as Microsoft, Intel, Cisco, and VMWare. SIGCOMM 2016 included five papers related to P4 [1, 20, 31, 32, 30]. Using P4, developers have created a variety of powerful new applications including advanced network diagnostics and telemetry [24, 37] and responsive traffic engineering [18].

As interest in the language and potential applications grows, there has been a flurry of research and development in P4 compilers and targets. At the May 2016 P4 workshop, there were at least seven P4 compiler implementations demonstrated. Given the recency of these projects, the standard for evaluation of these tools has simply been "it exists". However, as P4 and its associated tools move beyond relative immaturity, there is a need for common evaluation criteria.

Typically, these criteria come in the form of a benchmark. Historically, benchmarks have been used to both encourage and evaluate innovation. As an example of the former, when the Datamation Sort benchmark [33] was first proposed in 1985, the winning entry took more than one hour to sort one million records. The same task can now be done in less than 1 second. This progress is partly due to the fact that researchers had a target for which to aim. As an example of the latter, consider how benchmarks such as TPC-C [39] are used in the database community. Research papers that do not include an evaluation against a known benchmark are viewed with skepticism, and are unlikely to be published. The benchmark provides a standard by which the community can evaluate a new technique or optimization.

Designing a benchmark suite for P4 is difficult for at least two reasons. First, P4 is used as a high-level interface to many diverse target platforms. Metrics on one target platform may not be interesting or relevant on another. Moreover, because P4 abstracts away from target internals, it is difficult to collect target-specific metrics using P4 primitives themselves. This makes it hard to build a generic test harness. Second, data-plane programming and applications are still relatively new, and it is not yet clear what constitutes a representative application and workload. Much of the excitement around P4 is a result of discovering new applications that have not been previously envisioned (e.g., Paxos [12]). As usage increases, the expected functionality will change.

In this paper, we propose Whippersnapper, a P4 benchmark suite that addresses these challenges. First, to cope with target heterogeneity, Whippersnapper is divided into platform-agnostic and platform-specific benchmarks. Moreover, it advocates a black-box benchmarking methodology, allowing for measurements in a target-agnostic manner. Second, in contrast to benchmarks organized around a set of representative programs and workloads [34, 35, 39, 2], Whippersnapper is a *synthetic benchmark*. It consists of artificial programs and workloads that evaluate essential P4 operations in isolation. Some of the first benchmarks widely-accepted by industry were synthetic benchmarks [11, 41] that evaluated key features of the Algol and Fortran languages. More recently, for file systems, researchers have argued for a systematic approach based on key metrics, rather than "realistic" workloads [36]. Taking a cue from these projects, Whippersnapper is designed from *first principles* to focus on the essential language features of P4, identifying a set of metrics to measure and parameters to change.

Although P4 is still in its infancy, there is already a strong

SOSR '17, April 3–4, 2017, Santa Clara, CA, USA
© 2017 ACM. ISBN 978-1-4503-4947-5/17/04. . . $15.00
DOI: http://dx.doi.org/10.1145/3050220.3050231

need for a commonly accepted benchmark. There are numerous P4 compiler implementation efforts underway, targeting reconfigurable ASICs [38], NPUs [22], FPGAs [29, 40], GPUs [19], and general-purpose CPUs [27, 30, 17]. Without a benchmark, it is difficult for the community to compare and evaluate the various approaches. Moreover, the synthetic benchmark approach advocated by Whippersnapper is based on a systematic, multi-factor evaluation of compilers and targets via fundamental P4 abstractions, allowing it to be useful well after a large repository of real code examples exist.

We have used Whippersnapper to benchmark four different P4 compilers: PISCES [30], P4FPGA [40], BMV2 [6], and a Xilinx prototype [9] based on SDNet [29]. The results identify interesting opportunities for optimization in all four systems.

Just as the sorting benchmark spurred innovation in domain of data processing, a benchmark for P4 will help spur innovation in compiler design and implementation. We believe that proposing such a benchmark will generate discussion within the larger community about what are the requirements for P4 and data-plane languages in general, and what are the criteria by which they should be judged.

Clearly, a benchmark must be a community effort. Therefore, all source code for the P4 programs (or a script to generate a set of P4 programs) and packet capture (PCAP) files with workloads are available with an open-source license.

Overall, this paper makes the following contributions:

- It identifies key language features, metrics, and parameters for evaluating P4 programs.
- It describes a set of synthetic benchmarks that systematically explore and evaluate those key features.
- It demonstrates the utility of the benchmark on case studies involving four different P4 compilers.

The rest of this paper is organized as follows. In Section 2, we describe the target-independent benchmarks. Then we describe the target-dependent benchmarks in Section 3. Section 4 discusses long-term questions about data-plane benchmarks. In Section 5, we report results from case studies involving four different P4 compilers. Finally, we describe related work (Section 6) and conclude (Section 7).

2. TARGET-INDEPENDENT SUITE

P4 provides high-level abstractions for packet processing: packets are parsed and then processed by a sequence of tables; tables match packet header fields, and perform actions that forward, drop, or modify packets; actions may include stateful operations that read and write to memory.

Whippersnapper is designed to explore how various parameters impact the design and implementation of the key features provided by these P4 abstractions. Table 1 summarizes the benchmark suite. Entries in the table are grouped by a language *feature*, which correspond to the abstractions provided by P4. Each entry in the table roughly corresponds to an experiment in which a metric is measured as some *parameter* is changed. In all cases, except one, we expect that the metrics are *latency* and *throughput*; for the size of table benchmark, we expect that the metric is *memory consumption*. For all but one of the benchmarks, measurements are collected at runtime; for the size of table benchmark, the metric is collected at compiler time.

Table 1: Language features and parameters.

Feature	**Parameter**
Parsing	#Packet headers
	#Packet fields
	#Branches in parse graph
Processing	#Tables (no dependencies)
	Depth of pipeline
	Checksum on/off
	Size of tables
State Accesses	#Writes to different register
	#Writes to same register
	#Reads to different register
	#Reads to same register
Packet Modification	#Header adds
	#Header removes
Action Complexity	#Field writes
	#Arithmetic expressions
	#Boolean expressions

Parsing. When a packet enters the processing pipeline, the parser extracts header fields and metadata, which can be referenced in *match* rules in P4 tables. Parsers are typically implemented as finite state machines. Thus, the overhead due to parsing can result from either increasing the size or number of headers, or increasing the complexity of the parse graph. Whippersnapper includes three sets of programs to evaluate parsing performance: one that increases the number of nested packet headers, one that increases the number of fields in a packet header, and one that increases the number of branches in a parse graph (i.e., checking the value in a header, and transitioning to another state). For all three experiments, we are interested in latency and throughput.

Processing. Once a packet has been parsed, it is processed by a sequence of tables, which match header fields and perform actions. Tables are composed using P4 control-flow abstractions. The overhead due to processing can result from: (i) the number of tables, (ii) the depth of the pipeline (i.e., number of tables that have dependencies), (iii) if the table requires a checksum computation, and (iv) the size of the tables measured in number of entries. The processing benchmarks isolate each of these parameters.

State Accesses. P4 allows for stateful operations that can read and write to memory cells called registers. Thus, Whippersnapper includes experiments that vary the number of reads and writes to different registers. We note that P4 does not guarantee that reads and writes to registers for a given packet will be atomic. Therefore, the performance (and correctness) can vary quite a bit, depending on how state accesses are implemented. The state access benchmarks include experiments that vary the number of reads and writes to the same register, in order to evaluate the potential overhead of concurrency control. While the main metrics for the state access benchmarks are latency and throughput, the experiments based on increasing reads and writes to different registers also test the capacity that registers can support on a given target. Our benchmark does not currently test for correctness, since correctness is not defined in the current P4 specification.

Packet Modification. A P4 program can modify packets using action components. Whippersnapper separates actions that modify packets in different ways. The packet modification benchmarks evaluate the overhead for adding and removing packet headers. In these benchmarks, we expect to measure latency and throughput as the number of header adds or removes increases.

Action Complexity. P4 can also modify packets by writing to header fields. Moreover, field writes can include arithmetic or boolean expressions. An action can be made increasingly complex by varying the number of expressions. Whippersnapper includes separate benchmarks that vary the number of field writes, and vary action complexity by increasing the number of expressions.

Intentionally Omitted. Of course, as with any evaluation or model, it is important to not only identify those metrics that you are concerned with, but also those metrics that you ignore. Whippersnapper intentionally does not address:

- *Compilation time.* Compilation time is quite long (e.g., synthesis on an FPGA can take hours). At the same time, we expect that this is a one-time cost, as users do not frequently re-compile and deploy new data-plane programs.
- *Power consumption and area used.* These metrics are interesting, but they depend on "white box" profiling and reporting from the compilers.
- *Rule installation time.* The time to install new flow rules can be impacted by a variety of factors, including the target, controller, and network latency. While this is an interesting metric, we have omitted it because it is not necessarily a compiler issue. (Although, a compiler may impact installation time, for example, if it generates code that uses a slow TCAM to load new rules, when an SRAM would have been a fine alternative.)
- *Reliability and correctness.* We do not include experiments that evaluate reliability concerns (e.g., dropped/lost packets) or correctness. We assume that these issues are covered by unit tests. Instead, we focus on performance.

Overall, Whippersnapper is a benchmark suite that systematically evaluates the core P4 language features in isolation.

3. TARGET-DEPENDENT SUITE

In addition to the above benchmarks, Whippersnapper includes a set of target-dependent benchmarks. These benchmarks are summarized in Table 2. Below, we briefly discuss these benchmarks in more detail.

General Purpose CPU. Higher performance is often achieved by running multiple processing threads in parallel on separate cores. Therefore, performance is often unpredictable due to scheduling or locking on shared state. To evaluate this, Whippersnapper includes benchmarks that vary the workload by changing the working set of flows quickly, changing size of flows, and reading/writing to a single register. We note that there is often significant overhead due to processing packets in the OS kernel. Consequently, several projects have explored kernel-bypass approaches, such as DPDK [13] and NetMap[28]. With DPDK, a single thread runs on a separate CPU core that is directly attached to one Ethernet Network Interface and one of the Non-Uniform Memory Access (NUMA) nodes of the machine. Usually, each thread can process packets independently and does not need to share any state with the other threads.

NPU. Like CPUs, NPUs typically have multiple processing cores executing in parallel, each with separate memory for storing instructions Multiple-threads may be assigned to a processor, but only a single thread is active at a time. Processors are divided into clusters. Within a cluster, processors are connected by a bus, and share access to cluster memory. Clusters are connected to chip memory by a chip bus. As with CPUs, the performance of NPUs can be unpredictable, due to issues such as thread scheduling and locking to protect memory accesses. To explore this unpredictability, Whippersnapper includes benchmarks that vary the workload by changing the working set of flows quickly, changing size of flows, and reading/writing to a single register.

FPGA. One of the major challenges when compiling to an FPGA is handling resource constraints. FPGA hardware has a number of fixed and finite components, including Input/Output, block RAMs, logic cells, digital signal processing blocks, transceivers, and other blocks. Of course, FPGAs come in many different sizes, and with different feature mixes. So, the resources will vary across devices. To explore the constraints of a particular target, Whippersnapper includes an FPGA-specific benchmark that checks if increasingly large P4 programs, as measured by number of tables and table sizes, fit on a given device. The target-independent benchmarks also check this aspect, for example, different sizes of parsers and numbers of packet modification consume different amounts of FPGA logic resources.

We note that ASICs also typically impose resource constraints. However, with FPGAs, there are two subtle differences. First, while FPGAs come in a variety of different sizes, ASICs typically come in only one or a few sizes. So, when fitting a given program on hardware, users may have to accept that they could be wasting unused resources. Second, with an ASIC, a program must be mapped to a generic pipeline in a hardware substrate. With an FPGA, compilers attempt to map the hardware substrate to fit the program.

ASIC. The common hardware architecture for programmable high-performance packet-processing ASICs is

Table 2: Platform-dependent metrics.

Target	Metric(s)	Parameter	Measurement Time
General Purpose CPU	Latency,Throughput	Changing working set of flows quickly	Runtime
	Latency, Throughput	Changing size of flows	Runtime
	Latency, Throughput	Read, modify, update same register	Runtime
NPU	Latency, Throughput	Changing working set of flows quickly	Runtime
	Latency, Throughput	Changing size of flows	Runtime
	Latency,Throughput	Read, modify, update same register	Runtime
FPGA	Does it fit?	#Tables	Compile-time
	Does it fit?	Table size	Compile-time
ASIC	Does it fit?	#Tables	Compile-time
	Does it fit?	Table size	Compile-time
	Does it fit?	#Depth of dependency in expression	Compile-time

often dubbed PISA (Protocol Independent Switch Architecture) [21, 8]. In essence, PISA is a special kind of VLIW machine with a huge amount of I/O capacity and on-chip memory necessary to realize *match-action units* (i.e., exact or ternary lookup logic coupled with simple ALUs) and packet buffers, combined with generic packet parsing logic [15]. Unlike the conventional VLIW architecture used for CPUs, however, PISA does not have logic for heap, instruction memory, or stack pointers. Hence looping is not possible, and data dependencies between packets are resolved via a sequential pipeline of match-action units. The amount of all physical resources on a PISA chip is determined and fixed at the chip design time. Important resource constraints include number of physical stages, number of match-action units in each stage, amount of on-chip memory per stage, amount of containers that can carry header fields and metadata, capabilities of match-action units and state-processing ALUs, etc.

CPUs and NPUs can spend virtually as many resources (cycles and memory) as needed to process each packet. In contrast, PISA machines strictly limit the resources usage. This is necessary because a PISA machine must ensure deterministically high performance; the performance target of a PISA chip today is multi Tbps or billions of packets per second of throughput, and a sub-microsecond processing latency (excluding queuing delay).

Note a PISA machine's throughput and latency are characterized by a P4 program at compile time; at run time (packet processing time) the machine's throughput and latency do not change. Hence, with Whippersnapper we propose to benchmark PISA machines' target-specific capabilities in terms of number and size of tables, the depth of the table dependency graph, and the complexity of action definition the target can support.

4. DISCUSSION

The proposal of a P4 benchmark suite raises several interesting questions related to both benchmarking methodology, and about potential impact on the P4 language itself, including future areas of research. We expand the discussion of some of these topics below.

Language Evolution. The design of the P4 language is an active area of research and development. The language itself has quickly evolved from the $P4_{14}$ specification [26] to the $P4_{16}$ specification [25]. The internals of the reference compiler implementations have also evolved quickly, transitioning between several standardized intermediate representations [16, 3]. The synthetic benchmark approach advocated by Whippersnapper should be able to evolve with the language as well. The methodology, which is based on a systematic, multi-factor evaluation via fundamental language abstractions, is useful beyond the syntactic details of a particular language version.

Methodology challenge. One challenge in using a blackbox approach to benchmarking is collecting measurements with timestamps of sufficient granularity to measure differences in performance. Collecting such measurements requires access to hardware timestamps. In our sample runs, we have used three different tools: a hardware simulator with sub-nanosecond accuracy, the MoonGen Packet Generator [14] and a custom packet generator running in an FPGA. P4 does not provide a generic way to access hardware timestamps, and in general, such target-specific functionality must be implemented with "extern" functions. It would be useful to consider extensions to the P4 API that would allow for accurate benchmarking in the P4 program itself.

Another methodological challenge is generating workloads. Our implementation of the benchmark suite includes PCAP files with sample packets for each benchmark. To vary workload parameters, such as number of packets, number of flows, duration, and interpacket gaps, we rely on features of the packet generator (e.g., [14]).

Underspecified semantics. The benchmarks related to state accesses present a particular challenge. Neither the current draft of the $P4_{14}$ spec nor the pre-release of the $P4_{16}$ spec identify a concurrency model for state accesses. In other words, P4 does not guarantee that reads and writes to registers for a given packet will be atomic. Therefore, both the performance and correctness can vary quite a bit, depending on how state accesses are implemented. For these experiments, it may be worth adding a metric that reports the semantics that one can expect on a given platform.

Target diversity. P4 provides a target-agnostic API. However, in reality, there are compilers for many different targets, and performance metrics vary significantly on different hardware. For example, consider that INT allows a developer to measure the depth of a queue on an ASIC when a packet is buffered. But, on a software switch [30], it is unclear how queue depth relates to queuing delay (especially if system overheads prevent line-rate processing), or on which buffer a packet is stored (e.g., in the kernel or on the NIC).

Extensibility. As the language evolves, and new use cases for P4 emerge, the benchmark will need to be extended. To add a new benchmark to the suite, a developer must conceptually identify a metric to measure, and a set of parameters that would cause the value of the metric to change. Concretely, they would need to provide a set of P4 programs (or script to generate them) that change those parameters, and a PCAP file with a representative workload.

5. EXAMPLE USE CASES

To demonstrate potential use cases for Whippersnapper, we have performed a set of experiments using four different P4 compilers targeting different hardware:

- *PISCES [30]* is a software hypervisor switch that extends Open vSwitch [23] with a protocol independent design.
- *P4FPGA [40]* is an open-source framework that compiles and runs high-level P4 programs to various FPGA targets such as Xilinx and Altera FPGAs. P4FPGA first transforms P4 programs into a data-plane pipeline expressed as a sequence of hardware modules written in Bluespec System Verilog [5]. P4FPGA then compiles these Bluespec modules to FPGA firmware.
- *Xilinx SDNet [29]* is a development environment that allows for scalability across the range of Xilinx FPGAs. We used an early research prototype compiler implementation [9] that builds on SDNet. It first maps the source P4 program to the high-level PX [10] language. Then, SDNet compiles programs from PX to a data-plane implementation on a Xilinx FPGA target, at selectable line rates from 1G to 100G without program changes.
- *BMV2* is the reference compiler implementation on P4.org which targets the P4 Behavioral Model software switch [6].

We ran the bmv2 and PISCES experiments on a server with 12 cores (dual-socket Intel Xeon E5-2603 CPUs @ 1.6GHz), 16GB of 1600MHz DDR4 memory. The OS was Ubuntu 14.04. On a separate server, we ran the MoonGen packet generator. To measure latency, the packet generator applied a timestamp to each packet at departure and arrival time. The two servers were connected with 2 10Gbps links on 2 distinct SFP+ ports. The P4FPGA results were collected using hardware simulation.

Packet modification. Our first experiment is from a subset of the packet modification benchmarks, in which we measure the latency as we increase the number of header additions and removals. As a compiler, we used the Xilinx research prototype mentioned above. As a target hardware, we used a Xilinx Virtex UltraScale+ XCVU13P, which can achieve a 100Gb/s line rate, and throughput of 150 million minimum-size packets per second. The measurements came from running the benchmark test packet through a hardware simulation at sub-nanosecond accuracy, using the Verilog hardware design generated from the P4 description.

Figure 1a shows the results. While the general trend conforms to our intuitions (i.e., more adds/removes results in higher latency), the relative performance is surprising: it is slower to remove a header than add a header.

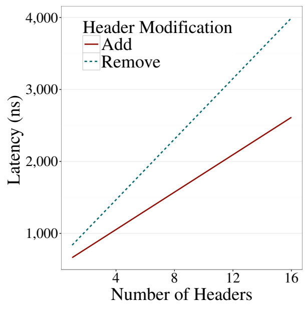
This behavior is due to a non-optimized feature in this particular $P4_{14}$-to-PX mapper. It adds a separate removal stage for each header (e.g., 16 removal stages for the 16-removal case) before deparsing. However, all insertions are done in a single stage at the same time as deparsing. In practice, the removals could also be done at the same time as deparsing, but the prototype does not take this approach.

We note that as the P4 language transitions to a new version, the $P4_{14}$-to-PX mapper we used in our experiments has been replaced with a $P4_{16}$-to-PX mapper, based on a new code base that does not exhibit the same behavior. SDNet itself has the same latency for adding and removing.

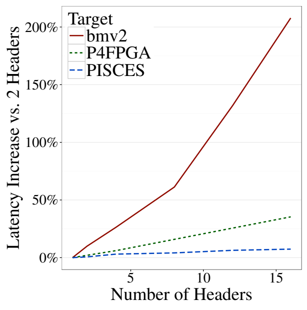
Parsing. This subset of the benchmark measures the latency for parsing an increasing number of headers using PISCES, P4FPGA, and the bmv2 switch. The results appear in Figure 1b. All results are normalized to the latency for each particular system for parsing 1 header, which are shown in Table 3. We see that as we increase the number of headers, the latency for bmv2 increases rapidly. In contrast, the latency for PISCES is only about 7% worse for 16 headers. We were surprised to find that P4FPGA scales much worse that PISCES, about 30% worse for 16 headers.

The benchmark exposes an opportunity for optimization in P4FPGA. The current parser is designed without consideration for the size of the header. In this case, the headers are 2 bytes, and the data width is 16 bytes. The parser only reads one header at a time, which takes 1 clock cycle. Multiple headers could have been read at the same time, which would have reduced latency.

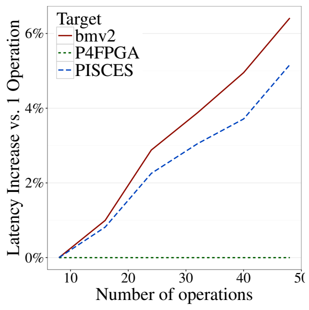
Action complexity. Figure 1c shows the latency for performing an increasing number of field writes using PISCES, P4FPGA, and the bmv2 compiler. P4FPGA scales better than either of the two software switches. P4FPGA leverages hardware parallelism to schedule packet modify operations with no read-after-write dependencies into the same clock cycle. If all field write operations are writing to different packet fields, then all operations will be scheduled to the



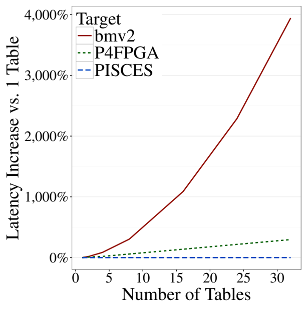
(a) Header add/removes vs. latency (Xilinx prototype).



(b) Number of headers vs. normalized latency.



(c) Number of field writes vs. Normalized latency.



(d) Pipeline depth vs. normalized latency

Figure 1: Whippersnapper identifies opportunities for optimization for four different P4 compilers.

Table 3: Latencies for parsing 1 header, writing 1 field, processing through 1 table.

Target	**Parse 1 header**	**Write 1 field**	**Depth of 1**
bmv2	11.2 ms	11.1 ms	14.4 ms
PISCES	5.2 μs	5.5 μs	4.8 μs
P4FPGA	0.3 μs	364 ns	364 ns

same clock cycle, yet still maintain the intended semantics of source P4 program. We note that in PISCES (and Open vSwitch), if the same field is set multiple times, it will result in a single set-field action. However, if multiple distinct fields are set, there will be a separate set-field action for each field. These actions are executed in sequence.

Processing Pipeline. Figure 1d shows the latency for processing packets as we increased the table depth. In this experiment, we see the benefits of optimizations in PISCES, in contrast to the bmv2 software switch. There is no latency increase when the table count increases, as the match-action pipeline is converted into a single match-action rule. P4FPGA implements a similar optimization.

These experiments demonstrate the types of phenomenon that a P4 benchmark can uncover, and how compiler implementors can use the benchmark to better understand the implications of their designs and potential optimizations.

6. RELATED WORK

This work is motivated both by the many P4 compiler implementation efforts underway, and inspired by benchmarking efforts in other communities.

P4 Compilers. P4.org provides a reference compiler [27] with backends that target the behavioral model software switch, and Berkeley Packet Filters. Barefoot Networks is developing a compiler that targets their Tofino chips [38]. There are a few projects that target FPGAs, including SDNet from Xilinx [29] and P4FPGA [40]. P4c [17] translates P4 to DPDK. PISCES [30] targets a modified version of Open vSwitch [23]. P4gpu [19] generates code for GPUs.

Benchmarks. There is a long history of benchmarks in different domains. Unix Systems have the Spec benchmarks [34]. For web applications, there is SpecWeb [35]. Java VM developers use the DaCapo benchmark [4]. In databases, TPC-C [39] is one prominent example. Whippersnapper is a *synthetic benchmark*, based on artificial programs and workloads that evaluate key features of a language or system. Whetstone [11] is a synthetic benchmark used to evaluate Algol 60. Dhrystone [41] was developed as an alternative to Whetstone, and was based on characterizations of programs in terms of common language constructs.

7. CONCLUSION

Whippersnapper is a synthetic benchmark suite for P4 that systematically evaluates the key language components. Whippersnapper not only provides a way to evaluate compilers or compiler/target combinations, but it will also help spur innovation. Indeed, using Whippersnapper, we have already identified interesting opportunities for optimizations in four P4 compilers. As the ecosystem of P4 tools and compilers grows, Whippersnapper addresses the pressing need for a common evaluation criteria.

Availability

All code for the benchmark suite and sample packet capture files to generate workloads are publicly available under an open-source license at: http://p4benchmark.org.

Acknowledgments

This research is partially supported by the SNF (166132 and 159537), DARPA CSSG (D11AP00266), NSF (1053757, 1440744, and 1422544), Cisco, a gift from Intel, and a Google Faculty Research Award. It is also partially supported by the European Union's Horizon 2020 research and innovation program under the SSICLOPS project (agreement No. 644866).

8. REFERENCES

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

[2] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: A Stream Data Management Benchmark. In *International Conference on Very Large Data Bases*, volume 30, pages 480–491, 2004.

[3] The B Intermediate Representation. https://github.com/OpenNetworkingFoundation/PIF-Open-Intermediate-Representation/blob/master/docs/BIR.TR.1.0.pdf, 2015.

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. *ACM SIGPLAN Notices*, 41(10):169–190, Oct. 2006.

[5] Bluespec. www.bluespec.com, 2013.

[6] P4 Behavioral Model. https://github.com/p4lang, 2015.

[7] 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. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, July 2014.

[8] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, Aug. 2013.

[9] G. Brebner. P4 for an FPGA target. In *P4 Workshop*, 2015.

[10] G. Brebner and W. Jiang. High-Speed Packet Processing using Reconfigurable Computing. *IEEE Micro*, 34(1):8–18, Jan. 2014.

[11] H. J. Curnow and B. A. Wichmann. A synthetic benchmark. *The Computer Journal*, 19(1):43–49, Jan. 1976.

[12] H. T. Dang, M. Canini, F. Pedone, and R. Soulé. Paxos Made Switch-y. *ACM SIGCOMM Computer Communication Review*, 46(1):18–24, May 2016.

[13] Dpdk. http://dpdk.org, 2011.

[14] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *ACM Internet Measurement Conference*, 2015.

[15] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design Principles for Packet Parsers. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2013.

[16] The P4 HLIR Specification. https://github.com/p4lang/p4-hlir/blob/master/HLIRSpec.pdf, 2015.

[17] S. Laki. High-Speed Forwarding: A P4 Compiler with a Hardware Abstraction Library for Intel DPDK. In *P4 Workshop*, 2016.

[18] J. Lee and J. Zeng. LBSwitch: Your Switch is Your Server Load-Balancer. In *P4 Workshop*, 2016.

[19] P. Li and Y. Luo. P4GPU: Accelerate Packet Processing of a P4 Program with a CPU-GPU Heterogeneous Architecture. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2016.

[20] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *ACM SIGCOMM Conference*, 2016.

[21] N. McKeown. PISA: Protocol Independent Switch Architecture. In *P4 Workshop*, 2015.

[22] Intelligent Server Adapters. http://open-nfp.org/media/Netronome_NFP-6480_2x40GigE_Product_Brief_11-15_laQID9Y.pdf, 2016.

[23] Open vSwitch. http://www.openvswitch.org, 2009.

[24] J. T. P. Lapukhov. Using INT to Build a Real-time Network Monitoring System Scale. In *P4 Workshop*, 2016.

[25] The P4 Language Specification Version 1.0.2. http://p4.org/wp-content/uploads/2015/04/p4-latest.pdf, 2015.

[26] The P4 Language Specification Version 1.1.0. http://p4.org/wp-content/uploads/2016/03/p4_v1.1.pdf, 2016.

[27] The P4 Reference Switch. https://github.com/p4lang/behavioral-model, 2015.

[28] L. Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *USENIX Annual Technical Conference*, 2012.

[29] SDNet. http://www.xilinx.com/products/design-tools/software-zone/sdnet.html, 2014.

[30] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford. PISCES: A Programmable, Protocol-Independent Software Switch. In *ACM SIGCOMM Conference*, 2016.

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

[32] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable Packet Scheduling at Line Rate. In *ACM SIGCOMM Conference*, 2016.

[33] Sort Benchmark. http://sortbenchmark.org, 1985.

[34] Standard Performance Evaluation Corporation (SPEC). https://www.spec.org, 1988.

[35] SPECweb2009. https://www.spec.org/web2009, 2009.

[36] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer. Benchmarking File System Benchmarking: It *IS* Rocket Science. In *USENIX Workshop on Hot Topics in Operating Systems*, 2011.

[37] T. Tofigh. Dynamic Analytics for Programmable NICs Utilizing P4. In *P4 Workshop*, 2016.

[38] Barefoot Tofino. https://www.barefootnetworks.com/technology, 2015.

[39] TPC-C benchmark, revision 5.11. http://www.tpc.org/tpcc, 2010.

[40] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon. P4FPGA: A Rapid Prototyping Framework for P4. In *ACM SIGCOMM Symposium on SDN Research*, 2017.

[41] A. Weiss. Dhrystone Benchmark: History, Analysis, 'Scores,' and Recommendations White Paper. Technical report, EEMBC Certification Laboratories, LLC, Oct. 2002.