Fast String Searching on PISA

Theo Jepsen
Università della Svizzera italiana
Barefoot Networks

Daniel Alvarez
Barefoot Networks

Nate Foster
Barefoot Networks
Cornell University

Changhoon Kim
Barefoot Networks

Jeongkeun Lee
Barefoot Networks

Masoud Moshref
Barefoot Networks

Robert Soulé
Università della Svizzera italiana
Barefoot Networks

ABSTRACT

This paper presents PPS, a system for locating occurrences of string keywords stored in the payload of packets using a programmable network ASIC. The PPS compiler first converts keywords into Deterministic Finite Automata (DFA) representations, and then maps the DFA into a sequence of forwarding tables in the switch pipeline. Our design leverages several hardware primitives (e.g., TCAM, hashing, parallel tables) to achieve high throughput. Our evaluation shows that PPS demonstrates significantly higher throughput and lower latency than string searches running on CPUs, GPUs, or FPGAs.

CCS CONCEPTS

• **Networks → In-network processing**;

KEYWORDS

Programmable switches; String searching

ACM Reference Format:
Theo Jepsen, Daniel Alvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Masoud Moshref, and Robert Soulé. 2019. Fast String Searching on PISA. In *SOSR '19: ACM Symposium on SDN Research, April 3–4, 2019, San Jose, CA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3314148.3314356

SOSR '19, April 3–4, 2019, San Jose, CA, USA

ACM ISBN 978-1-4503-6710-3/19/04…$15.00
https://doi.org/10.1145/3314148.3314356

1 INTRODUCTION

String searching is one of the most common and important functions run on computers. It is estimated that 80% of the world's data is unstructured [31], meaning that it cannot be easily queried using a fixed data model. Instead, users must search through large amounts of log data, JSON files, email, web pages and other documents to find the relevant patterns.

These searches can have a significant impact on application performance. For example, Palkar et al. [23] recently showed that using string searches to pre-filter data before feeding it into Spark can provide a 9× improvement in end-to-end application completion time.

Unfortunately, data is increasingly stored on devices that cannot provide good search performance. In order to improve the utilization of resources, many data centers have turned towards a disaggregated architecture [12], in which storage devices have weak CPUs and little memory. In the storage community, these devices are commonly referred to as JBODS—just a bunch of disks.

Besides the fact that these machines are "wimpy", performing search also requires paying the levitation cost to get data off the disk. When running search on an x86 CPU, this will be through PCI-Express, which is a known bottleneck [22, 36]. The fourth generation of this interconnect achieves 128Gbps over sixteen serial links [28].

In fact, we argue that the general approach of having an x86 CPU inspect every single byte of incoming data is fundamentally flawed. With bulk data for storage or big data applications, the main mode of operation from the x86's point of view should be DMA. In other words, the CPU should only handle metadata or parts of the data required for real computing (e.g., mapping or transforming).

Given that I/O is a bottleneck and data is distributed on a large number of machines, we believe there is an exciting opportunity to leverage the emerging trend of programmable network ASICs [4] to accelerate string search. After all, networking interconnects can easily reach high throughput performance of multi-Tbps. Moreover, in purely economic terms,

using a dedicated ASIC for search would be less expensive than increasing the specifications of storage servers.

However, implementing string search on an programmable networking ASIC—i.e., one that implements the Protocol Independent Switch Architecture (PISA) [5]—is non-trivial. The hardware imposes constraints on, among other things, the number of bytes that can be examined in the packet header and number of match-action stages in a pipeline. Moreover, the main language used to program the devices, P4, is intentionally computationally restricted, as it excludes looping constructs, which are undesirable in forwarding pipelines. Because string search requires both iteration and accessing deep in the payload, it is often used as the archetypal example of an application that is a poor fit for programmable network ASICs.

In contrast to this conventional wisdom, we argue that programmable ASICs provide a distinctive set of hardware features—a much larger number of interfaces compared to an FPGA or Smart NIC, a large amount of fast SRAM relative to a CPU, and a high-degree of parallelism—that make them well-suited to the task of high-throughput string search. The challenge is implementing string search in a way that best use these hardware features.

Towards this goal, we present PISA-based Parallel Search (PPS), a system for locating occurrences of string keywords stored in the payload of packets. The intuition behind PPS is to store DFA state transitions in the match tables of the pipeline. However, to achieve high-performance, PPS relies on two key techniques. First, PPS partitions the DFA into a set of smaller DFAs that run in each stage, allowing PPS to benefit from the pipeline parallelism inherent in the ASIC design. This also allows PPS to use the relatively small (compared to DRAM) on-chip SRAM more efficiently. Second, in order to increase throughput, PPS uses switch SRAM and TCAM to support larger DFA *stride* sizes (i.e., the number of characters matched per transition). Beyond these key techniques, PPS provides an optimization that trades-off accuracy for memory usage by matching on the CRC16 hash of the input characters, rather than the characters themselves.

We have implemented a prototype of PPS, which includes a compiler, a custom network controller, and a small client application to send data to the network. We have also incorporated our prototype into Apache Spark, allowing PPS to accelerate basic data analytics.

We evaluated PPS on two Spark jobs inspired by real-world applications, and a set of micro-benchmarks. Our experiments show that offloading Spark filtering to PPS provides a speedup up to 6.5x. And, the micro-benchmarks demonstrate that running the full PPS workflow—including pattern compilation, rule installation, and streaming the data through a remote ASIC—has a faster search completion time than running Unix `grep` locally.

In summary, this paper makes the following contributions:

- Provide the first implementation of per-packet string matching on a programmable ASIC.
- Describe techniques to leverage the inherent ASIC pipeline parallelism by partitioning a DFA into smaller DFAs running in each stage.
- Present techniques that increase throughput by efficiently using SRAM and TCAM to increase the DFA stride size.

The rest of this paper is organized as follows. We provide a short summary of various approaches to string searches (§2). We then discuss the design of PPS (§3), our pattern compilation technique (§4) and prototype (§5). After a discussion (§6), we present the results from our experimental evaluation (§7), discuss related work (§8), and conclude (§9).

2 BACKGROUND

There is a long history of research on string searching algorithms, and we are unlikely to improve the performance via a purely algorithmic solution. Instead, we seek to adapt an existing algorithm to best utilize the characteristics of the new domain-specific machine: the programmable networking ASIC. Below, we discuss the most well-known solutions, focusing on the suitability for PISA.

More formally, the string search problem is defined as follows. Let Σ be an arbitrary alphabet. Given a text string $t = t_1 \dots t_n$ and the pattern string $p = p_1 \dots p_m$, where each t_i and p_i are characters in Σ, then output the set of all positions in t where an occurrence of p starts as a substring.

The naïve algorithm iterates over every index of t, and checks if the string starting from that index matches p, running in $\Theta(nm)$ time. Searching for multiple patterns requires iterating over the string multiple times, once for each pattern. This algorithm could be implemented on PISA, but it would be unnecessarily slow.

The Boyer-Moore algorithm [6], used by Unix `grep` matches on the tail of the pattern, rather than the head, and uses information gathered in a pre-processing step to jump ahead multiple characters, rather than advancing one index at a time. This reduces the best case running time to $\Omega(n/m)$, but the worst case time is still $O(mn)$. It also requires $\Theta(k)$ space, where $k = |\Sigma|$ is the size of the alphabet. The implementation would be similar to the naïve algorithm.

The Rabin-Karp algorithm [18] speeds up the comparison of the pattern with the substring of t by using a hash function. This reduces the running time to $\Theta(m)$ time. However, Rabin-Karp is not a good match for a PISA for two reasons. First, it requires computing the sliding hash of the text being searched. This would require a large number of hash units, which does not scale on hardware. Second, there may be hash collisions, in which case the algorithm reverts to comparing the two strings index by index anyways.

Instead, PPS is based on the Aho-Corasick algorithm [2]. Aho-Corasick, which was the basis of the original `fgrep` command, runs in $O(n + m + z)$ time, where z is the number of matches. The algorithm constructs a finite-state machine that resembles a trie, with additional edges between nodes that share a common prefix.

3 DESIGN OVERVIEW

At a high-level, PPS implements a finite-state machine in the pipeline of a PISA switch to search for patterns in the payload of packets. PPS extends this basic design with optimizations to effectively utilize the switch hardware.

The PPS state machine works at byte level granularity. Thus, PPS works with binary data or ASCII strings. It does not make any assumptions about the character encoding (e.g., UTF-8, UTF-16, etc.) or length of the pattern.

Our prototype implementation assumes that input packets have Ethernet, IP, and UDP headers. It begins searching at the UDP payload. However, this is not inherent to the design, and PPS could just as easily start the search from the beginning of the packet. PPS searches for patterns throughout the entire packet, using recirculation to examine deep into the payload. Furthermore, it assumes the switch is not oversubscribed and thus no packet loss.

If PPS detects a matching packet, it emits a new packet that contains a custom header, indicating which packet matched, as well as the offset of where the match occurred.

3.1 Expected Deployment

PPS can be deployed in two ways: as a *dedicated appliance*, or as a network switch that also does *bump-in-the-wire* processing. However, in order to search deep into the packet payload, PPS relies on re-circulation and it must discard the initial portion of the packet at each iteration. Therefore, a bump-in-the-wire deployment would require access to some external memory architecture to buffer packets [19]. When deployed as an appliance, the ASIC program is relieved of the responsibility of forwarding packets, and more resources can be dedicated to searching.

In either case, the switch is configured to run the PPS data plane pipeline. Conceptually, the pipeline consists of a sequence of tables that contain state machine transitions. Note that the pipeline is completely static. It is compiled once when PPS is deployed, and does not change when search patterns are updated.

The PPS controller process compiles patterns and installs table entries on the switch. These entries are dynamic, and are re-generated whenever there is a new search pattern.

The controller process is divided into two parts: the server sends rules to the switch and the agent receives the rules and installs them via the control plane API. PPS uses a custom controller agent to reduce serialization overhead.

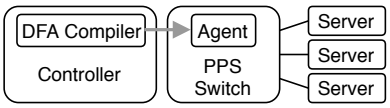


Figure 1: Expected deployment.

Applications that use PPS run on servers connected to the switch. In order to use PPS, they must send the data to be searched to the switch. Our prototype implementation includes a small library that reads a stream of data and sends it to the switch, one chunk per packet.

Figure 1 shows an example deployment, which is the same deployment used in the Spark experiments described in Section 7. There are three servers connected to the PPS switch. A fourth server acts as the controller process. The servers have multiple NIC interfaces—as is common in data center deployments—and the second interface is used to provide external network connectivity.

4 PATTERN COMPILATION

The PPS compiler has two main tasks. First, it converts a set of search patterns into a k-stride DFA. Second, it maps the DFA into a pipeline of match action tables.

Patterns to k-stride DFA. PPS takes a set of search patterns as input. Patterns can be arbitrary regular expressions. However, all of our experiments use inputs that are finite strings (i.e., only concatenation operator), which are converted to acyclic DFAs. As we will discuss in Section 6, handling arbitrary DFAs is expensive.

PPS differs from the standard Aho-Corasick algorithm in that it uses a k-stride DFA. The *stride* size of a DFA refers to how many characters are read for each transition. For example, a stride size of 2 means that the implementation reads 2 character per transition.

Increasing the stride size increases the throughput. However, the throughput improvement comes at the cost of memory, as the size of the state transition table increases with the stride. With a stride size of s and an alphabet Σ, there are $|\Sigma|^s$ transitions per state.

The algorithm to convert a set of patterns to a k-stride DFA is as follows. The compiler first converts the patterns to a nondeterministic finite automata (NFA) using the Aho-Corasick algorithm [2]. It then transforms the NFA to a DFA using subset construction [3]. Finally, to convert the DFA to a k-stride DFA, the compiler computes the k-stride closure on each node in the 1-stride DFA—for each state s, a new transition is added for each state s' that is reachable in k characters from s. In Figure 2, the DFA on the right is the 2-stride equivalent of the DFA on the left. Both machines match the string "dog".

Figure 2: DFAs with different strides for "dog".

Match		Action
state	chars	
0	do	set_state(1)
3	og	accept(4)
1	g*	accept(2)
0	*d	set_state(3)

Figure 3: Table for 2-stride (k=2) DFA

Patterns can occur at any offset within the input string, and patterns may not be multiples of the stride size, k. This means that some transitions need to ignore some characters to match the start and end of the string. For example, the 2-stride DFA in Figure 2 with pattern "dog" has start transitions *d and do ("*" matches any character). Likewise, some of the terminal transitions also include "*" and can be implemented with ternary matching.

***k*-stride DFA to MAU Pipeline.** The DFA is then translated to the match-action abstraction. The switch has a pipeline of tables that is compiled once and can be used to execute any DFA. The DFA is installed in the tables at runtime.

The first step is to represent the DFA as a state transition table. Figure 3 shows the table corresponding to the 2-stride DFA in Figure 2. A naïve approach would be to store one big transition table in the pipeline that performs a single transition. However, this would limit the number of characters consumed per pipeline pass. Instead, we replicate the transition table on all stages. This way, multiple transitions are performed per pass, increasing throughput by the number of stages. Given the current state and the current k input characters, each stage transitions to the next state.

To store the DFA on the switch, we leverage different types of memory. DFA transitions that consume exactly k characters require an exact match, which is stored in SRAM. The start and end transitions that match less than k characters require ternary matching; these transitions are stored in TCAM. In each stage, first the exact match table is applied; if no entry matches, then the TCAM table is applied. If none of the tables match, then the default action is executed, setting the state to 0. This is an implicit transition to the start state.

4.1 Optimizations

Multiple DFAs. Memory is a scarce resource on switches. To reduce memory usage, we split the DFA into multiple smaller DFAs, which run in parallel on the switch. We do this by partitioning the patterns into multiple subsets, and constructing a DFA from each subset of patterns. The aggregate size of these DFAs (# transitions) is smaller than that of one large DFA containing all the patterns. This is because patterns with similarities can cause an explosion in the number of transitions. An optimal partitioning of the patterns yields a set of DFAs with the smallest aggregate size. We leave finding the optimal partitioning for future work. Currently, we partition the patterns randomly multiple times and pick the best one. We chose to use 3 parallel DFAs in our pipeline because: it is the most number of splits before the returns diminish; and, coincidentally, it is the most that we can fit in the our hardware switch due to resource limitations.

Tunable pipeline. In the pipeline design outlined above, every stage in the ingress and egress pipeline performs a DFA state transition. To support more patterns, we can make two changes: (i) reduce the stride size of the DFA; and (ii) use the resources of multiple stages to perform a single transition. By reducing the stride size, the number of transitions in the DFA is reduced, producing a more compact DFA. By performing fewer transitions, we can combine the resources of multiple stages. For example, if the DFA representation does not fit within the resources of a single stage, we can split it across two stages. These optimizations come at the expense of throughput, which we explore in the evaluation.

4.2 Approximation

We observe that for some applications accuracy is not a strict requirement, such as with approximate streaming analytics [26]. We can trade-off accuracy for reduced memory usage by storing a hash of characters. Matching each character in the stride size, k, requires storing k bytes per transition. Instead, the switch can compute a CRC16 hash of the k characters. Some hash collisions are detected at compile time (i.e. two different transitions out of the same state have the same hash). In this case, we use perfect hashing: the compiler finds a different hash function (CRC with a different polynomial) that doesn't produce collisions, and updates the pipeline to use that hash function. Hash collisions that occur at runtime will produce false positives and false negatives.

5 IMPLEMENTATION

We have implemented a PPS prototype running on a Barefoot Tofino switch. At its core, the prototype includes a DFA compiler written in Python (365 LOC), as well as the DFA data plane program written in P4 (4754 LOC).

Controller. The Barefoot Networks switch agent offers a Thrift [1]-based API, which requires serializing and installing each table entry one-by-one. To reduce overhead, we implemented a custom agent that uses a binary serialization format and installs entries in bulk. This optimization reduces entry installation time from tens of seconds to milliseconds.

Data Levitation. Our prototype includes a client library that sends data to the switch. Because PPS searches one packet (chunk) at a time, it is most suitable for data that can

be partitioned into chunks (e.g. lines, records). If the data is a continuous stream of bytes that cannot be partitioned, the client can format the data as *overlapping* chunks.

The client library is implemented in C and runs in userspace. It would be possible to use DPDK [9], which could support higher throughput and lower CPU load. Recent work by Kim et al. demonstrates that a Tofino switch can serve as an RDMA end-point [19]. One could imagine connecting the switch to JBODS via RDMA. Accessing the data from the storage servers would completely eliminate the use of the server CPU for searching, allowing it to focus on other storage tasks, e.g., error correction, de-duplication, etc.

6 DISCUSSION

Our prototype provides significant performance benefits to applications as described in our evaluation. However, there are some limitations on the formatting of input data and the types of search patterns supported.

Input Alignment. Data chunks sent to PPS must be aligned to the packet. If a chunk spans multiple packets, a potential match split across the packets will not be detected. To address this, as described above, our client can generate overlapping chunks, which uses more bandwidth. To ensure that matches are detected, the overlap must be the size of the longest search pattern. Furthermore, the chunk must be less than the network Maximum Transmission Unit (MTU).

Recirculation. In one pipeline pass, PPS can search a fixed number of bytes in the packet: k times the number of pipeline stages. To search deeper, the packet is recirculated through the pipeline. At the end of each pipeline pass, the bytes that were just searched are truncated from the packet. For example, with 4 recirculations, the first pass searches the entire packet; the second $\frac{3}{4}$ of the packet; the third $\frac{2}{4}$; and the fourth $\frac{1}{4}$. Thus, one packet actually uses more bandwidth: $1 + \frac{3}{4} + \frac{2}{4} + \frac{1}{4} = \frac{10}{4}$. This is $\frac{6}{4}$ times the bandwidth for a single pass. The bandwidth overhead factor can be generalized to $\frac{n-1}{n+1}$, where n is the number of recirculations. Although this reduces the overall throughput, it is still orders of magnitude higher than that of other solutions (discussed in Section 8).

Generalizing to Regular Expressions. The technique we described for searching fixed patterns also generalizes to Regular Expression (RegEx) matching. Our compiler supports the Kleene star operator, alternation, and concatenation. It also supports character classes using alternation. However, we found that complex expressions result in a DFA state space explosion, using more switch memory. This is especially the case for character classes, which require exact transitions for all the characters in the class, which cannot leverage the ternary matching of TCAMs. To enable more efficient RegEx matching, it could be possible to translate each input byte to a symbolic value, reducing the number of table entries.

7 EVALUATION

Our evaluation focuses on three questions: (i) How does PPS help with end-to-end application performance? (ii) How does PPS performance vary with the number of patterns? and (iii) How does PPS performance compare to state-of-the-art software and hardware solutions.

Experimental setup. We ran PPS on a 32x100G port Barefoot Tofino switch connected to a 4-node cluster with QSFP+ breakout cables. Each server has 12 cores (dual-socket 1.6GHz Intel Xeon E5-2603 CPUs), 16GB of 1600MHz DDR4 memory, and an Intel 82599ES 10Gb Ethernet controller. For our microbenchmarks, we randomly selected non-disjoint "content" patterns from the Snort [30] community ruleset.

7.1 End-to-end Application Performance

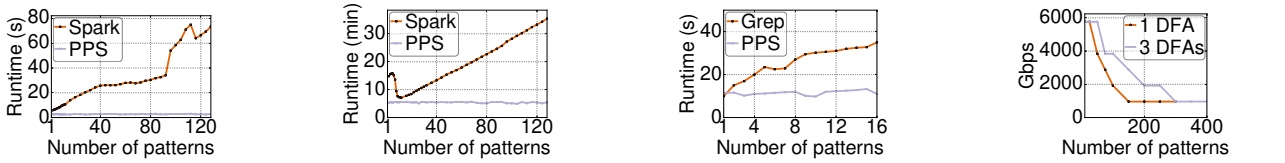
We used PPS to accelerate two Spark filtering jobs: scanning e-mails and filtering Twitter tweets. For the baseline, we use Spark SQL, which partitions the filtering among the worker cores. We compare this to a Spark program that pipes the data to PPS for filtering. In both cases, the Spark job executes a reduce stage that aggregates the sum of matching lines or JSON records. As we increase the number of patterns, the end-to-end runtime with PPS stays constant.

Scanning E-mails with Spark. We did a line-by-line search of the "Podesta E-mails": a collection of 50K e-mails (4.7GB) published by WikiLeaks in 2016 [25]. Figure 4a shows the results for searching an increasing number of patterns. Spark's execution time increases steadily and jumps above a minute after 96 (a multiple of 12 cores) because of query planning. PPS consistently searches the entire file in under 3 seconds.

Filtering Tweets with Spark. Inspired by the evaluation in Sparser [23], we filtered 212GB of JSON tweets collected using the Twitter Streaming API [15]. In addition to using unstructured patterns, we also searched for structured JSON key/values, e.g. `"lang":"ro"` and `"filter_level":"medium"`. Figure 4b shows the results. Spark SQL chooses a poor query plan with fewer than 12 patterns (which explains the dip); at 12 patterns, it has similar performance to PPS, but slowly reduces as the number of patterns increases. For 128 search strings, Spark takes 35.4 minutes, compared to 5.43 min for PPS (6.5x speedup).

7.2 Microbenchmarks

PPS vs Grep. For a direct comparison to `grep`, we measure the end-to-end time to search a 12GB log file. The file is stored in a RAM disk, because otherwise disk I/O (not compute) is the dominant factor in the search time. For PPS, the end-to-end time includes: (i) sending the patterns to the controller, (ii) compiling rules from the DFA, (iii) installing the rules and sending an acknowledgement to the client, and (iv) streaming



(a) Filtering e-mails. **(b) Filtering tweets.** **(c) Searching a 12GB log file.** **(d) Calculated tput feasibility**

Figure 4: PPS end-to-end experiments and micro-benchmarks.

the data to the switch. Figure 4c shows the search time for an increasing number of patterns. For a single pattern, both have similar performance. For multiple patterns, `grep`'s runtime increases, while PPS remains constant. This demonstrates that there are clear performance benefits, even including the overhead of sending data to the network.

Pattern Complexity vs. Throughput. Ideally, the switch would have an unlimited amount of memory that could hold as many patterns as necessary. However, in reality, to be able to process data at line rate, the switch has a fixed amount of memory. As the number (or complexity) of patterns increases, the stride size of the DFA has to be reduced to fit the DFA in the switch memory. To further reduce memory usage, we can also split the DFA into multiple DFAs (§ 4.1).

We calculated the throughput achievable for workloads with an increasing number of patterns. We randomly selected patterns from the Snort ruleset that are up to 32 characters long, which is the case for over 80% of the rules. We then compiled the patterns into a DFA with the largest stride size that would fit on the switch. Figure 4d shows the throughput using a single DFA (red) and multiple DFAs in parallel (green). Note that these are theoretical values that we calculated. It is reasonable to calculate the throughput (instead of testing it experimentally), because a compiled P4 program runs at the speed of the architecture with a fixed number of stages and bounded memory access time.

7.3 Comparison to State-of-the-Art

To provide context for PPS performance, we briefly report results from comparable state-of-the-art solutions. To be clear, these results are not direct benchmark comparisons and are not collected using the same workloads. Using a GPU, Hsieh et al. [13] reached 150Gbps for 20 Snort patterns. Titan IC's Helios ASIC [34] reports 100Gbps for 1 million rules. DFC [8] achieves 45Gbps using x86 servers. With a single recirculation and a stride size of 4, PPS can search 100 Snort patterns at 3.8Tbps on a 64-port Tofino.

8 RELATED WORK

DFAs. Prior work has explored compacting DFAs [24]. Sherwood et al. [32] proposed splitting a DFA into DFAs that match a subset of the input bits. The NetKat [29] compiler matches packets using a Binary Decision Diagram, which has a structure similar to a DFA.

Hardware solutions. There are several techniques for using TCAMs, including compacting transition symbols and states [14], using LPM to share state [7] and variable striding [20]. DFC [8] uses cache-friendly data structures for pattern matching on general purpose CPUs. Others have implemented pattern matching on GPUs [13, 16, 35]. HAWK [33] implements an FPGA pipeline using a bitsplit technique proposed by Sherwood et al.. HARE [10] adds RegExp support to HAWK by adding a character class translation stage to the pipeline, as well as counters for RegExp quantifiers.

In-network computation. Programmable ASICs have been used for telemetry [21] and stream processing [17]. Sonata [11] accelerates basic filters and aggregations from Spark queries in the data plane. Sapio et al. [27] also perform aggregation tasks in programmable switches. None of this prior work performs string searches.

9 CONCLUSION

PPS is inspired by the observation that PISA is well-suited for particular computing tasks. Some of the common characteristics of those tasks are: (i) the I/O to compute ratio is high, (ii) the space complexity of the computing algorithm (i.e., amount of memory required during computing) is low and independent of the size of the input workloads fed to PISA via I/O, and (iii) the computing algorithm is branch heavy. String search has these characteristics.

Searching in strings is a fundamental problem in computer science, and improving the performance of the algorithm can have significant impact on a wide variety of applications. We have described a set of implementation techniques that build on the classic Aho-Corasick algorithm, while efficiently utilizing hardware primitives (e.g., TCAM, hashing, parallel tables) to achieve high throughput string searching on a programmable network ASIC. Compared to state-of-the-art alternatives on CPUs, GPUs, and ASICs, PPS offers orders of magnitude improvements in throughput.

ACKNOWLEDGEMENTS

This work is partially supported by SNF grant 200021_166132. We thank Pietro Bressana and Greg Watson for their help.

REFERENCES

[1] A. Agarwal, M. Slee, and M. Kwiatkowski. 2007. Thrift: Scalable Cross-Language Services Implementation. https://thrift.apache.org/static/files/thrift-20070401.pdf.

[2] Alfred V. Aho and Margaret J. Corasick. 1975. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM (CACM)* 18, 6 (June 1975), 333–340.

[3] Alfred V. Aho and Jeffrey D. Ullman. 1972. *The Theory of Parsing, Translation, and Compiling*. Vol. 1. Prentice Hall.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Computer Communication Review (CCR)* 44, 3 (July 2014), 87–95.

[5] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. 99–110.

[6] Robert S. Boyer and J. Strother Moore. 1977. A Fast String Searching Algorithm. *Communications of the ACM (CACM)* 20, 10 (Oct. 1977), 762–772.

[7] Anat Bremler-Barr, David Hay, and Yaron Koral. 2010. CompactDFA: Generic State Machine Compression for Scalable Pattern Matching. In *29th IEEE Conference on Computer Communications (INFOCOM)*. 659–667.

[8] Byungkwon Choi, Jongwook Chae, Muhammad Jamshed, KyoungSoo Park, and Dongsu Han. 2016. DFC: Accelerating String Pattern Matching for Network Applications. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 551–565.

[9] DPDK 2018. DPDK. http://dpdk.org/.

[10] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. 2016. HARE: Hardware Accelerator for Regular Expressions. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. Article 44, 12 pages.

[11] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven Streaming Network Telemetry. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. 357–371.

[12] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. 2013. Network Support for Resource Disaggregation in Next-generation Datacenters. In *Workshop on Hot Topics in Networks (HotNets)*. Article 10, 7 pages.

[13] Cheng-Liang Hsieh, Lucas Vespa, and Ning Weng. 2016. A high-throughput DPI engine on GPU via algorithm/implementation co-optimization. *J. Parallel and Distrib. Comput.* 88 (2016), 46–56.

[14] Kun Huang, Linxuan Ding, Gaogang Xie, Dafang Zhang, Alex X. Liu, and Kavé Salamatian. 2013. Scalable TCAM-based regular expression matching with compressed finite automata. *Architectures for Networking and Communications Systems* (2013), 83–93.

[15] Introduction to Tweet JSON 2016. Introduction to Tweet JSON. https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/intro-to-tweet-json.html.

[16] Muhammad Asim Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. 2012. Kargus: a highly-scalable software-based intrusion detection system. 317–328.

[17] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. 2018. Life in the fast lane: A line-rate linear road. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*.

[18] Richard M. Karp and Michael O. Rabin. 1987. Efficient Randomized Pattern-matching Algorithms. *IBM J. Res. Dev.* 31, 2 (March 1987), 249–260.

[19] Daehyeok Kim, Yibo Zhu Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan Seshan. 2018. Generic External Memory for Switch Data Planes. In *Workshop on Hot Topics in Networks (HotNets)*. 6.

[20] Chad R. Meiners, Jignesh Patel, Eric Norige, Eric Torng, and Alex X. Liu. 2010. Fast Regular Expression Matching Using Small TCAMs for Network Intrusion Detection and Prevention Systems. In *USENIX Security Symposium*.

[21] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-directed hardware design for network performance monitoring. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. 85–98.

[22] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. 327–341.

[23] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2018. Filter Before You Parse: Faster Analytics on Raw Data with Sparser. *The VLDB Journal* 11, 11 (July 2018), 1576–1589.

[24] J. Patel, A. X. Liu, and E. Torng. 2014. Bypassing Space Explosion in High-Speed Regular Expression Matching. *IEEE/ACM Transactions on Networking* 22, 6 (Dec. 2014), 1701–1714.

[25] Podesta Emails 2016. The Podesta Emails. https://wikileaks.org/podesta-emails/.

[26] Do Le Quoc, Ruichuan Chen, Pramod Bhatotia, Christof Fetzer, Volker Hilt, and Thorsten Strufe. 2017. StreamApprox: Approximate Computing for Stream Analytics. In *17th ACM/IFIP/USENIX International Conference on Middleware*.

[27] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Workshop on Hot Topics in Networks (HotNets)*. 150–156.

[28] PCI Sig. 2003. PCI Express base specifications revision 1.0 a. *PCI SIG* (2003).

[29] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. 2015. A fast compiler for NetKAT. In *International Conference on Functional Programming (ICFP)*. 328–341.

[30] Snort 2018. Snort. http://snort.org/.

[31] Darin Stewart. 2013. Big Content: The Unstructured Side of Big Data. https://blogs.gartner.com/darin-stewart/2013/05/01/big-content-the-unstructured-side-of-big-data/.

[32] Lin Tan and T. Sherwood. 2005. A high throughput string matching architecture for intrusion detection and prevention. In *32nd International Symposium on Computer Architecture (ISCA)*. 112–122.

[33] P. Tandon, F. M. Sleiman, M. J. Cafarella, and T. F. Wenisch. 2016. HAWK: Hardware support for unstructured log processing. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 469–480.

[34] Titan IC 2018. Helios Product Brief. http://titan-ic.com/assets/img/news/Final-RXPA-APR-18.pdf.

[35] Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P. Markatos, and Sotiris Ioannidis. 2009. Regular Expression Matching on Graphics Hardware for Intrusion Detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*. 265–283.

[36] Noa Zilberman, Andrew W. Moore, and Jon A. Crowcroft. 2016. From photons to big-data applications: terminating terabits. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 374, 2062 (2016).