REGISTOR: A Platform for Unstructured Data Processing Inside SSD Storage

SHUYI PEI, JING YANG, and QING YANG, University of Rhode Island and Shenzhen Dapu Microelectronics Co. Ltd.

This article presents REGISTOR, a platform for *r*egular *e*xpression *g*rabbing *i*nside *stor*age. The main idea of Registor is accelerating regular expression (regex) search inside storage where large data set is stored, eliminating the I/O bottleneck problem. A special hardware engine for regex search is designed and augmented inside a flash SSD that processes data on-the-fly during data transmission from NAND flash to host. To make the speed of regex search match the internal bus speed of a modern SSD, a deep pipeline structure is designed in Registor hardware consisting of a file semantics extractor, matching candidates finder, regex matching units (REMUs), and results organizer. Furthermore, each stage of the pipeline makes the use of maximal parallelism possible. To make Registor readily usable by high-level applications, we have developed a set of APIs and libraries in Linux allowing Registor to process files in the SSD by recombining separate data blocks into files efficiently. A working prototype of Registor has been built in our newly designed NVMe-SSD. Extensive experiments and analyses have been carried out to show that Registor achieves high throughput, reduces the I/O bandwidth requirement by up to 97%, and reduces CPU utilization by as much as 82% for regex search in large datasets.

CCS Concepts: • **Hardware** → **External storage**; • **Computer systems organization** → *Special purpose systems*;

Additional Key Words and Phrases: Regular expressions, processing in storage, near data processing, SSD storage, hardware accelerator

ACM Reference format:
Shuyi Pei, Jing Yang, and Qing Yang. 2019. REGISTOR: A Platform for Unstructured Data Processing Inside SSD Storage. *ACM Trans. Storage* 15, 1, Article 7 (March 2019), 24 pages.
https://doi.org/10.1145/3310149

1 INTRODUCTION

Staggering growth of big data has generated numerous challenges to both the research community and IT industry in terms of data processing. The most critical one is how to understand and extract meaningful information out of this huge amount of data, of which nearly 80% is unstructured data

This research was supported in part by NSF grants CCF-1439011 and CCF-1421823. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF. This work was also partly supported by a research contract between URI and Shenzhen Dapu Microelectronics Co., Ltd.

Authors' addresses: S. Pei, J. Yang, and Q. Yang, University of Rhode Island, 45 Upper College Road, Kingston, RI 02881, and Shenzhen Dapu Microelectronics Co. Ltd., Tian'an Cyber Park Huangge Road, Longgang Center, Longgang Dis, Shenzhen, Guangdong, 518100, China; emails: {spei, jyang, qyang}@ele.uri.edu.

1553-3077/2019/03-ART7 $15.00
https://doi.org/10.1145/3310149

[12, 13, 15, 25]. Obtaining useful information within unstructured data not only requires searching simple strings but also needs to apply complex patterns to obtain a deeper insight. Among many different methods, regular expression (regex) search provides a powerful and flexible approach for unstructured data analysis [42]. However, regex search in a file is compute intensive because it requires a full scan of the file and multiple state transitions to locate a complete match. Traditional software solutions such as *grep* and *awk* for regex search cannot keep pace with the rapid growth of data volume and the speed of hardware that offers a tens of gigabytes data rate.

Due to the importance of speeding up regex search, extensive research has been reported in the literature over the past decade in accelerating regex search. Some researchers exploit SIMD hardware available in many modern processors [8, 31, 44], multi-core architectures [38], and GPU widely used for parallel computing [30, 62]. Recent work [12] proposed a unified automata processor (UAP) that can be integrated with traditional CPU architectures and supports various automata models. Another line of research provides FPGA- or ASIC-based solutions [17, 23, 33, 56]. Micron's automata processor (AP) implements NFA and uses bit vectors and a routing matrix to perform state transitions, with one AP chip achieving a 1Gps line rate [10, 50]. The Helios regex processor from Titan IC, commercially used for network intrusion detections (NIDS), can deliver throughput up to 10GB/s based on FPGA acceleration [23]. IBM PowerEN integrates a regex engine (RegX) that splits regex into sub-patterns and processes in parallel, achieving scanning rates of 20 to 40 GB/s [33, 56]. HARE's ASIC RTL implementation, taking advantage of bit-split automata [51], can process data at a rate of 32GB/s, which matches the modern memory bandwidth [17].

Although existing research efforts successfully accelerate regex search to match the speed of DRAM, the main bottleneck of I/O bus has not been given enough attention in the research community. Terabytes of unstructured data, as exemplified by e-commerce [2], social computing [58], and bioinformatics [45], are stored in data storage, such as high-speed flash memory SSDs. All existing accelerator techniques require loading this huge amount of data from data storage, such as AWS S3 storage service [3], to the system DRAM before any analysis can be done. Moving such large data from storage to system DRAM places a great burden on the storage I/O bus. The typical high speed I/O bus in use today, such as PCIe 3.0, only provides 3.94GB/s with four lanes and 7.88GB/s with eight lanes [39]. Even the next-generation PCIe 4.0 is expected to offer only 7.88GB/s with four lanes and 15.75GB/s with eight lanes [40]. However, modern flash technologies exhibit great potential in matching the speed of high-performance computing. Flash SSD controllers are able to support 32 independent flash channels [35], each of which runs at 667 megatransfers per second (MT/s) [35]. The aggregated throughput of flash memories at the back end of modern SSDs reaches 32GB/s with the channel width of 16 bits [34]. Therefore, we have high-speed DRAM on one side and high-throughput flash memory SSD on the other, making the storage I/O bus the clear system bottleneck.

To truly speed up regex search and eliminate the system bottleneck, we propose a new approach to accelerating regex search, referred to as Registor (*r*egular *e*xpression *g*rabbing *i*nside SSD *stor*age). Registor brings computation to storage to avoid unnecessary data movement and thus eliminates the I/O bottleneck when processing sizable data stored in storage. A preliminary version of this work can be found Pei et al. [41]. We develop Registor hardware to perform on-the-fly regex search in storage, targeting the speed of internal bus. The idea is to find matching candidates and then examine them in parallel. In addition, Registor hardware is able to obtain file semantics from out-of-order data blocks and responds to a host's request by sending the data that match the regex exactly, associated with a line number, displacement, length, and so forth.

For the search engine of Registor to work for any applications running on the host, we develop a user library that includes APIs that can be called by user applications, a compiler that translates regex to the formats that are understandable by hardware, and an exception handler that improves

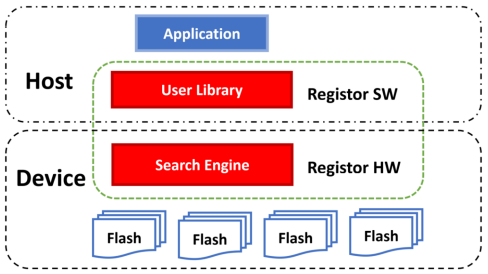

Fig. 1. Overview of the system showing where Registor sits.

robustness. The compiler is optimized for Registor hardware to make the search process more efficient. The data path for Registor hardware bypasses the long I/O stack of operating system (OS) to achieve low latency.

To assess the potential benefits of our proposed Registor and demonstrate its performance, we have implemented Registor augmented to our newly developed NVMe-SSD, which includes both the hardware accelerator in FPGA and the user library running on a Linux host computer. Extensive experiments show that Registor reduces I/O bandwidth requirement by up to 97% and CPU utilization by as much as 82%, eliminating I/O bottlenecks and providing high-performance regex search. In summary, our main contributions are as follows:

- A hardware search engine has been designed for on-the-fly regex search in the SSD. The search engine is fully pipelined, consisting of a file semantics extractor, matching candidates finder, regex matching units (REMUs), and results organizer. Each stage of the pipeline leverages parallel architecture to achieve high throughput.
- A user library has been developed that enables user applications to fully take advantage of Registor hardware. We also optimize the compiling process for the search engine and improve robustness by syntax checking and exception handling. The data transfer path from search engine to applications bypasses the long I/O stack in the host system, providing low latency.
- A working prototype of Registor has been built and integrated in our newly developed NVMe-SSD, including a search engine in FPGA and a user library running on a Linux host computer. The SSD with Registor can be treated as a regular block-level SSD storage with regex search functions. It is readily usable by applications with no need to modify the OS.

The rest of this article is organized as follows. In Section 2, we present the overall architecture of Registor followed by a detailed hardware design. Section 3 discusses the design of Registor's software including a user library and the data path. Section 4 describes the implementation of Registor, and Section 5 illustrates the experimental setup for evaluation purposes. The results are discussed in Section 6 to demonstrate the advantage of Registor over state-of-the-art solutions. Section 7 discusses related work, and Section 8 concludes the article.

2 REGISTOR HARDWARE

Consider a system shown in Figure 1. Registor sits between host applications and the SSD device. It consists of two major parts: the *hardware search engine* and *user library*. In this section, we focus on the architecture of the hardware search engine, including each functional module and interactions among different modules, and discuss how each module contributes to a high-performance regex engine in SSD storage. The user library will be discussed in the next section.

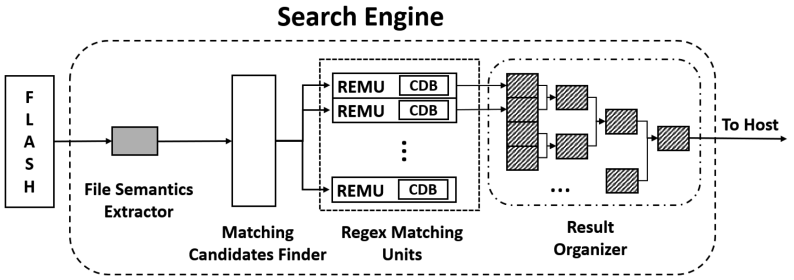

Fig. 2. Registor's hardware pipeline.

2.1 Overview

To achieve high-performance regex search inside SSD storage, we aim to make Registor capable of performing on-the-fly search while the data stream is being transferred from flash memory to the host. In this way, the in-storage processing time is completely hidden and transparent to users. However, it is challenging for such an engine to match the speed of data transfer in the SSD since the traversal of regex consumes multiple clock cycles. Moreover, the out-of-order data stream makes it difficult to handle file semantics in regex search. To tackle these challenges, we have designed four hardware modules, *file semantics extractor*, *matching candidates finder*, *REMUs*, and *results organizer*, fully exploiting parallelism and pipelining for maximal performance. Figure 2 shows the pipeline structure of Registor hardware. The file semantics extractor recovers file semantics from out-of-order data blocks retrieved from NAND flash and provides a data stream in file order to the matching candidates finder. The matching candidates finder locates possible matches through a fast scan of the data stream and associates contextual information with these matches to form tasks. Then, REMUs process these tasks to determine exact matches from these matching candidates by performing regex search. Cyclic data buffers (CDBs) are deployed to provide data streams for REMUs (see Section 2.4 for details). Since these REMUs work in parallel to gain speedup, the results organizer reorders the intermediate results from REMUs before sending to the host.

2.2 File Semantics Extractor

Files are stored in separate data blocks in the SSD and retrieved from NAND flash regardless of This work was also partly ordering/sequence to maximize backend bandwidth. However, the regex matching process requires not only *intra-block* semantics but also *inter-block* semantics. The inter-block semantics are necessary for matching regex across block boundaries and providing in-file locations of matched strings. To obtain inter-block semantics, file semantics extractor reorders data blocks based on the file layout provided by Registor software. In the following paragraphs, we briefly describe how to retrieve file layout in the host before carrying out the hardware design.

Retrieving file layout. In the SSD, the file-block mapping is stored in *inodes*. The data in inodes have different formats in different on-disk-file systems. To retrieve such information, we first read the super block from the SSD to get the type of file system and determine the right format. Then, we find the inode for the file by traversing the inodes in the file path and obtain the entire file layout by parsing the inode. Note that this function is realized at the user level in the host, which will be discussed further in Section 3.1.

Reordering blocks. Retrieving data blocks from NAND flash follows a first-ready-first-serve principle, which passes whatever is ready to the frontend interface regardless of sequence. To recombine data blocks to a conforming file format, we design a reordering buffer (RoB) for block

reordering. The RoB is a random access buffer with a capacity of k blocks. The incoming data blocks are buffered in the RoB based on their logic block numbers (LBNs). Blocks in the RoB are sent to the next stage of the pipeline in ascending order of their LBNs, although they may enter the RoB out of order. The input (write) and output (read) happens asynchronously for maximal performance. Since the file size can be larger than the buffer size, we use a pointer indicating the logical start of the RoB and turn it into a cyclic buffer. The RoB can be logically viewed as a sliding window of size k over the file being searched. The same size sliding window is used in the user library to prevent out-of-range LBNs that may cause RoB overflow.

2.3 Matching Candidates Finder

The matching candidates finder finds possible matches by checking whether the input character is accepted by the *start point* of the input regex. The start point is the character in the regex where we compare this character to the input character before we check the rest of characters in the regex. Therefore, the start point depends on how the regex is processed. In most cases, a regex is matched from the first character to the last, where the first character of the regex is the start point. We also introduce an approach of matching regex from an internal character in Section 3.1, where the internal character serves as the start point. While the matching candidates finder parses the input data to find a start point, we also record the line number by counting "\n" and displacement by counting characters. The displacement and line number within the file are recorded in separate registers. These matching candidates are encapsulated with their respective contextual information to form tasks to be processed by the next stage, the REMUs for exact matches. These tasks essentially contain the positions of matching candidates so that REMUs know from where to replay the data stream. This benefits the performance of REMUs in two aspects: on one hand, these tasks are independent from each other and thus can be executed in parallel in REMUs without changing the search results; on the other hand, each REMU only needs to check a small segment of the input stream and can quickly reject/accept a possible match.

Since the task generation is merely a one-character comparison and counter updates, it can scale up easily to match the bandwidth of the incoming stream. Note that when files (i.e., tables, log files) and results have special patterns, all related tasks can be assigned to one REMU in the worst case. To minimize the performance impact of input files, we incorporate randomness into the dispatching policy by shuffling the tasks generated within one clock cycle before dispatching to REMUs.

2.4 Regex Matching Units

Regex processing generally involves two steps: compiling and matching. The compiling process is interpreting the regex into a piece of code that can be executed on a computer and the matching process is executing such code against the input stream. We only ported the matching process to hardware in SSD, since the compiling process is required only once upon each query.

We use the similar method described in Thompson [53] and Cox [9] to generate the code in the compiler. In addition, we optimize the compiling process for the matching candidates finder, which will be discussed in detail in Section 3.1. Although these codes can be executed on the computer directly, a special hardware and an instruction set optimized for the hardware need to be developed to realize the matching process in storage. Since FPGA supports parallel computing naturally, we propose a new instruction set that is able to process more complex matching logic in one instruction as compared to traditional forms. As shown in Table 1, each instruction consists of an action and operands. For instance, *PPAIR* can be used to match a single character or two characters optionally—for instance, "PPAIR a,a" matches char "a," whereas "PPAIR a,b" matches character set "[ab]" and "PPAIR a,A" matches case-insensitive "a." This feature is useful and results

Table 1. The Instruction Set

Action	Operands	Description	Action	Operands	Description
PPAIR	a,b	Match char a or char b	JMP	p	Jump to line p.
NPAIR	a,b	Not match char a and char b	SPLIT	p,q	Track both p and q, where p and q are line numbers.
PRANGE	a,b	Match ASCII code of a to b	LSPLIT	n,l,u,s	If counter *n* is within lower bound l and upper bound u, increase counter *n* by 1. Otherwise, jump to line s.
NRANGE	a,b	Not match ASCII code of a to b	ACCEPT	void	The string is matched by the regex.

```
Line 0:  Reserved
Line 1:  PPAIR    a,a
Line 2:  SPLIT    3,5
Line 3:  PPAIR    b,b
Line 4:  JMP      6
Line 5:  PPAIR    c,c
Line 6:  PPAIR    d,d
Line 7:  ACCEPT
```

```
Line 0:  Reserved
Line 1:  PPAIR    a,a
Line 2:  PPAIR    b,c
Line 3:  PPAIR    d,d
Line 4:  ACCEPT
```

Fig. 3. An example code for search regex "a(b|c)d" (left) and "a[bc]d" (right). See Table 1 for the instruction set.

in better code efficiency. An example of executable code for searching regex "a(b|c)d" is shown in Figure 3. Note that "b|c" is interpreted using "SPLIT" and "JMP," whereas "[bc]," which has the same meaning as "b|c," is encoded into "PPAIR b,c." The current version of the compiler is not fully optimized for encoding efficiency, which is one of our future works.

We now design the REMU that can execute such code in FPGA, fully exploiting parallelism. The code generated by the compiler is stored in an instruction buffer in FPGA with each entry corresponding to a line in the code. We keep an *action pointer* (similar to a program counter, PC), which holds the bit map of instruction buffer entries. In other words, each bit in the action pointer corresponds to an entry in the instruction buffer. A value "1" in a bit position indicates the corresponding entry of the instruction buffer needs to be executed at the cycle. The action pointer is updated in each clock cycle to track where the code should be executed next. Initially, the action pointer points to the start of the code and executes line by line sequentially. When a *SPLIT* is encountered, the action pointer is able to track multiple entries at the same time, thus realizing parallel executions.

Figure 4 shows how REMU decides input string "abd" as a complete match of the regex in Figure 3. The action pointer is initialized to "1" in bit 1 and "0" in all other bits to denote that the code is executed from line 1. The input stream provides an "a" that is accepted by line 1 and then the action pointer is updated and REMU now executes line 2. Note that line 2 is a *SPLIT* that requires the next input character to be compared to both lines 3 and 5. Here, REMU tracks the two lines by marking bits 3 and 5 as "1" in the action pointer. Then, the input character "b" matches line 3 but rejects line 5, and thus REMU continues to the line after line 3. After running for a few clock cycles, REMU reaches line 7, which indicates that the string "abd" is accepted by regex "a(b|c)d."

Note that the previously described design of REMU is one of the many methods of implementing regex search using FPGA. It can be replaced by other designs, such as NFA [47, 61], DFA [14, 27],

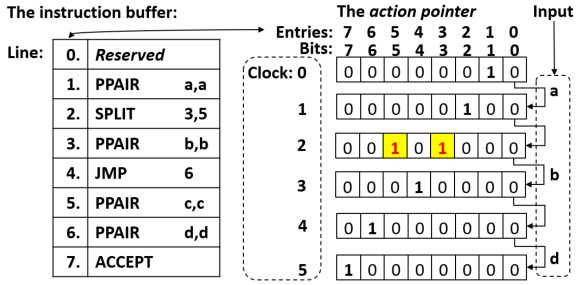

Fig. 4. An example of execution in REMU.

B-FSM [33, 56], bit-split automata [17, 51], and so forth. Each method has its advantage and best applicable field. Since our goal is to eliminate the I/O bottleneck in searching unstructured data, we focus on how to fit REMU into our proposed search engine, and adopting more advanced automata designs in place of REMU is part of our future research.

Since REMU usually takes multiple clock cycles to determine a complete match, several REMUs are marshaled to multiply the processing rate, where each REMU processes tasks dispatched by the matching candidates finder independently and simultaneously. To provide data streams for REMUs, we deploy a CDB to replay the data stream for each REMU. The input data stream from the NAND flash is saved in CDBs of the matching candidates finder and flushed out at the results organizer by manipulating a read and write pointer.

2.5 Results Organizer

In real-world applications, a search engine should present the matched results in their order of positions in a file. However, the intermediate results from REMUs are in separate streams that need to be sorted in order. The results organizer merges the ordered streams into one by popping out the results of minimal displacement at a time until all the results are sorted. Note that the comparison among all REMUs is required each clock cycle. We pipeline the process, as shown in Figure 2, using a tree structure where the results are merged hierarchically to hide the time for comparisons.

3 REGISTOR SOFTWARE

We now describe the software design of Registor, including a user library and the data path. We focus on how these software components are designed to coordinate with Registor's hardware.

3.1 User Library

The user library consists of *APIs* for user-level applications, *a compiler* generating executable code for Registor's hardware, and *an exception handler* to improve the robustness of the system.

The APIs. Table 2 lists two levels of APIs to users. The higher-level APIs, registor_sync_read() and registor_async_read(), function in a similar way to Linux Grep. These functions take two basic parameters: file name and regex. The lower-level APIs, registor_blks_sync_read() and registor_blks_async_read(), provide functions similar to direct I/O and let users process data based on a single page or a page group. These functions take three parameters: *slba* (starting logical block address), the number of blocks, and the input regex. The slba information can be obtained by calling registor_file_layout(). This function uses APIs (e.g., ioctl() [32] function with FS_IOC_FIEMAP as its parameter) provided by the file system to obtain the file layout. The OS will check access

Table 2. APIs for User Applications

API Functions	Description	Parameters
registor_sync_read()	Read file synchronously with Registor processing	file_name, regex
registor_async_read()	Read file asynchronously with Registor processing	file_name, regex
registor_file_layout()	Retrieve file layout from the SSD	file_name
registor_blks_sync_read()	Synchronously read certain blocks with Registor processing	slba, length, regex
registor_blks_async_read()	Asynchronously read certain blocks with Registor processing	slba, length, regex

```
Line  0: Reserved
Line  1: PPAIR     a,a
Line  2: PPAIR     p,p
Line  3: PPAIR     p,p
Line  4: PPAIR     l,l
Line  5: PPAIR     e,e
Line  6: PPAIR     s,s
Line  7: NOP
Line  8: PRANGE    0,9
Line  9: PRAMGE    1,9
Line 10: ACCEPT
```

Fig. 5. An example code for search regex "[1-9]\dapples."

permissions based on the access control information obtained from files' inodes and credentials of the current processes.

It is worth pointing out that these APIs can be used for a wide span of real-world applications, and Registor can be easily tailored to large-scale text annotation for search engines (e.g., Lucene [4]) or repurposed for data queries in NoSQL or SQL databases.

The efficiency-aware compiler. The compiler translates regexes to executable codes that are understandable by Registor hardware. It consists of a lexer and a parser, where the lexer breaks the regex into tokens and the parser generates the abstract syntax tree (AST) using these tokens. The executable code can be obtained through the preorder traversal of the AST.

We also provide another function that generates the executable code to improve the efficiency of REMU. The idea is to find a more *specific* node in the AST as the start of the executable code. For instance, a deterministic character "a" is more specific than a character class "\d." In this case, the executable code starts with the specific node and consists of two parts. The first part is from the specific node to the end of the preorder traversal, and the second part is from the specific node to the start of the preorder traversal. Recall that we find matching candidates based on the start point in the input regex. The matching candidates finder actually checks whether the input character is accepted by the start of the executable code. To enable this optimization, we only need to generate the code by traversing the AST from the internal node and keep the operations of the matching candidates finder unchanged. Since the code starts from a specific node, the efficiency-aware compiler can reduce the total number of matching candidates, making the REMUs more efficient. Figure 5 provides an example code for regex "[1-9]\dapples," where the character class "[1-9]" and the escape character "\d" are not specific nodes. Thus, the matching candidates finder checks the input stream for "a" instead of the start character "[1-9]," and REMU searches "apples" first. When REMU executes line 7, it will reset the current position to the start offset (the position of "a" in the input stream) and then scans the data stream backward to match "\d[1-9]." Note that this optimization comes at the cost of losing parallel execution of multiple lines in a code (e.g., some

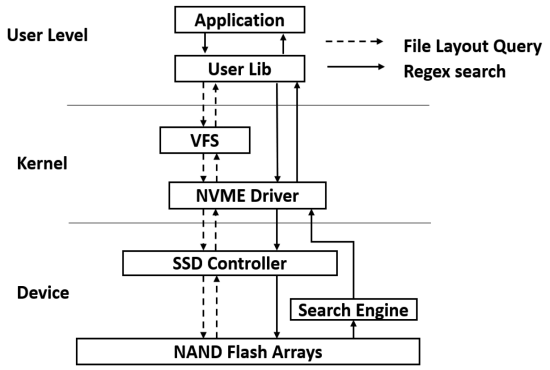

Fig. 6. The data path of Registor.

lines require forward scanning, whereas other lines need backward scanning). Therefore, this optimization that is currently used for those regexes consists only of characters and character classes.

The exception handler. Supporting an enterprise-level system requires the software platform to achieve robustness, reliability, and availability beyond a simple and accessible interface. Any command sent to Registor hardware is validated through the syntax check function and resource check function. When a syntax error occurs, the error handler returns with a code to notify the type of exceptions. However, not all regex that pass the syntax check can benefit from our proposed Registor due to hardware resource limitation or Registor hardware being occupied by other applications. For instance, the maximum times of backtracking/loop and the allowable length of executable code supported in REMU are subject to hardware resource constraints. The preemption in Registor hardware may cause data consistency and integrity problems [5]. To address this issue, we add a lock to Registor to prevent preemption. When an invalid input for hardware is detected (e.g., over-depth backtracking, Registor hardware unavailable), the error handler calls the integrated software regex engine instead of using Registor hardware.

In our design, the amount of data returned to the host is restricted in size to less than the amount of data per I/O request. If the search result exceeds such a limit, the excess is discarded and a bit in result to the host is set to indicate overflow. The exception handler then checks this bit and reports an overflow to upper-level applications.

3.2 Data Path

To ensure system-level performance, the Registor system features a well-designed data path that achieves low latency and avoids interfering with the normal I/Os of the SSD. Registor hardware is placed aside the normal I/O data path. Normal I/Os do not go through the Registor path and hence are not interfered by it. The Registor data path is activated only upon a search request issued by an application. In this case, there are two types of data paths for Registor corresponding to the two phases of processing: *file layout query* and *regex search*, as depicted in Figure 6. Recall that retrieving the file layout needs file information, super blocks, and inodes from the SSD. The data path for the file layout query involves a virtual file system for the file path, an NVMe driver for data transfer, and an SSD controller to load super blocks and inodes from NAND flash. Unlike the file layout query that is executed only once per search, regex search has significant impact on the overall performance. To reduce latency, we interface the user library directly to the NVMe device drivers, bypassing the file system. We avoid modifying the OS by augmenting extended NVMe commands through an optional command field defined in NVMe standards. The added NVMe command set is listed in Table 3. These newly added NVMe commands are compatible with

Table 3. Extended NVMe Commands for Registor

Command	Description	Parameters
rgt_fsm_inst	Send executable code generated by the compiler to the device	devID, code
rgt_sync_pis_read	Read Registor results synchronously	devID, slba, len
rgt_sync_data_read	Read raw data synchronously for exception handling	devID, slba, len
rgt_async_data_read	Read raw data asynchronously for exception handling	devID, slba, len

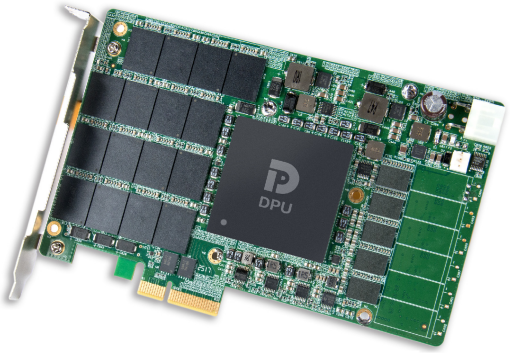

Fig. 7. NVMe-SSD with Registor.

Table 4. Specification of the SSD

FPGA	Xilinx Ultrascale+ 9P, xcvu9pflgb2104-2
DRAM	9 × 1GB, in which 1GB is for the ECC
NAND flash	32 × 256 GB, 8TB in total
Interface	PCIE Gen 3 × 4

a standard NVMe and no modifications are made in the OS, making Registor readily available to user applications. The results from Registor hardware are regarded as normal data blocks requested by a normal I/O read command and are sent directly to the NVMe driver without the interference of an SSD controller, which simplifies the internal control and reduces latency.

4 IMPLEMENTATION

For the purpose of evaluation of Registor, we have built a working prototype of Registor under the resource constraints of the SSD platform. This section presents the details of its implementation, followed by discussions on several practical issues and solutions through which Registor shows advantages over traditional techniques.

4.1 Overview

The entire Registor hardware has been implemented on the Xilinx FPGA, the UltraScale+ chip, using Verilog language. The RTL of the implementation is integrated in an in-house enterprise-level SSD prototype, as shown in Figure 7 and Table 4.

The user library is implemented in C language at the application layer of the host system running Linux OS Ubuntu 16.04. The standard parser and lexer in our compiler are developed based on Flex and Bison [29]. The extended NVMe command is implemented by using the reserved bits

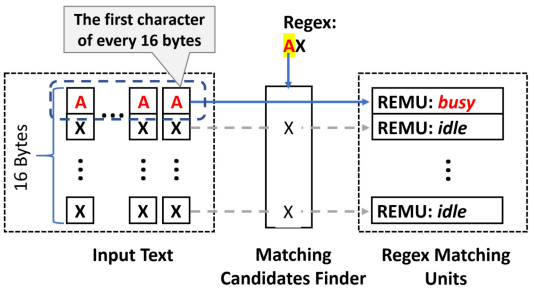

Fig. 8. An example of the worst case for Registor using the fixed dispatching mechanism.

(15:08) of *Write-Command Dword 13* in NVMe standard revision 1.3a [11]. The file layout retrieval is implemented using the *ioctl()* [32] system call provided by Linux kernels for userspace to get file extent mappings.

To make the system latency insensitive, we implemented the hardware pipeline with a back-pressure mechanism that the latter stage in the pipeline can request the former stage to temporarily stop its production of data. We use 1MB RoB, and the data block size is 4KB. Since the internal data bus width of our SSD prototype is 16B, we implemented 16 REMUs that can be expended to 32 or more if the bus width is expanded. The length of code supported in each REMU is limited to 32 for the demonstration purpose. To reduce the use of RAM resources, we deploy 8 CDBs for 16 REMUs where 2 REMUs share 1 CDB and the size of each CDB is 4KB, same as the size of one data block. When contention occurs, we use the round-robin algorithm to serve the read requests from two REMUs in turns.

4.2 Load Balance in REMUs

Based on the design described in Section 2.4, REMUs have the highest computational complexity and consume the most hardware resources among all stages in the pipeline of Registor HW. Efficiently utilizing these hardware resources is critical to the overall performance of Registor. In particular, we would like to make all REMUs work in parallel with a balanced workload to avoid bottleneck problems. Without proper load balancing, REMUs could suffer from a severe performance penalty. The worst case happens when all tasks generated by the matching candidates finder are dispatched to the same REMU, resulting in up to 16 times slowdown as compared to the best case. This slowdown is caused by the fixed dispatching mechanism in the Registor design. When using fixed dispatching, the 16 output ports of the matching candidates finder and 16 REMUs are one-to-one matched and the connections remain unchanged across different clock cycles. This drags down performance severely in some scenarios, such as when an input text has a pattern where the first character of every 16 bytes matched the search string (Figure 8). Only the first character can pass the matching candidate finder every clock cycle. If we use the fixed dispatching mechanism, all of these tasks are dispatched to the same REMU. To make efficient use of all REMUs and avoid the worst-case performance, we implemented a random task dispatcher as follows.

Recall that tasks are dispatched from the matching candidate finder to REMUs, with a one-to-one mapping from 16 sources to 16 destinations, which is the root cause for potential unbalance of workloads among REMUs. A straightforward way is to randomly generate one of the possible permutations of 16 objects. If a lookup table were used to do this, it would have 16! entries, which is virtually impossible to implement in practice. Instead of directly dealing with the permutations of 16 objects, we proposed a practical method, named *pseudo random shuffle*, to implement a workload

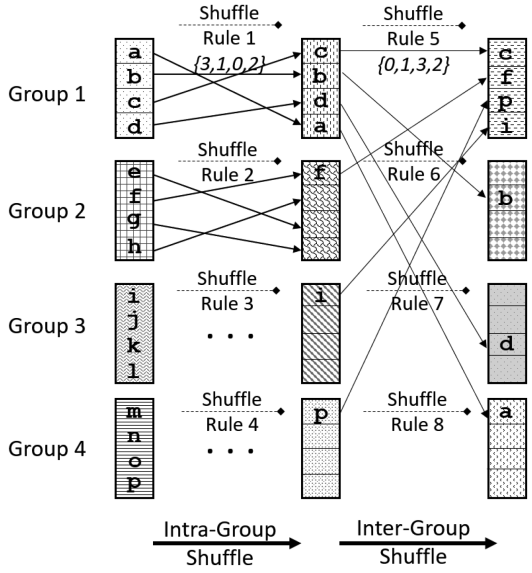

Fig. 9. Random shuffle.

balance mechanism. This is illustrated in Figure 9. We divide 16 objects into 4 groups (i.e., each group has 4 objects). Then, the random shuffle consists of two phases: the intra-group phase and the inter-group phase. Each shuffling is performed based on a permutation of the set $\{0, 1, 2, 3\}$, which we call a *shuffling rule*. For example, $\{3, 1, 0, 2\}$ is a possible shuffling rule to map input positions of 0, 1, 2, and 3 to output locations of 3, 1, 0, and 2. In intra-group phase, the 4 objects within a group are shuffled based on the rule, where each number in the rule denotes the new position of an object accordingly. In Figure 9, the 4 objects named $\{a, b, c, d\}$ in group 1 are rearranged into $\{c, b, d, a\}$ if rule 1 is $\{3, 1, 0, 2\}$. Specifically, the "3" in the rule means that object "a" should be the third object of the group after shuffling. Then, we perform inter-group shuffling by selecting 1 object from each group to form new groups. In this step, we only mix objects of the same position in different groups. For example, by selecting the first object in each group, we have $\{c, f, i, p\}$. Similarly, if rule 5 $\{0, 1, 3, 2\}$ is applied to $\{c, f, i, p\}$, we obtain final group 1 as $\{c, f, p, i\}$ after inter-group shuffling. This shuffling process is feasible in hardware implementation because it requires permutations of only 4 objects. The entire table can be easily stored in a lookup table with $4! = 24$ entries. If we wish to do shuffling every clock cycle, we only need to choose 8 entries from the table as shuffling rules. In our implementation, we use eight linear feedback shift registers (LFSRs) to make the choices. Each LFSR is configured to generate an 8-bit random number (to make the outcome appear to be more random) in which we use 5 bits to randomly select from the range of 1 to 31. Since the the lookup table has 24 entries that are smaller than the output range of LFSR, the values greater than 23 are considered as 0 by default. The preceding process is shown in Figure 10.

Although pseudo random shuffle provides less randomness than the straightforward method because it only picks one permutation from $(4!)^8$ possibilities, it is practical in implementation. This mechanism is better than other simple mechanisms (e.g., one-step linear shift of 16 objects) by preventing Registor from falling into the traps resulting from linear patterns that may exist in input text.

To quantitatively demonstrate the necessity of random dispatching in Registor design, we carry out experiments based on the UVM test, which will be described later in Section 5. We construct

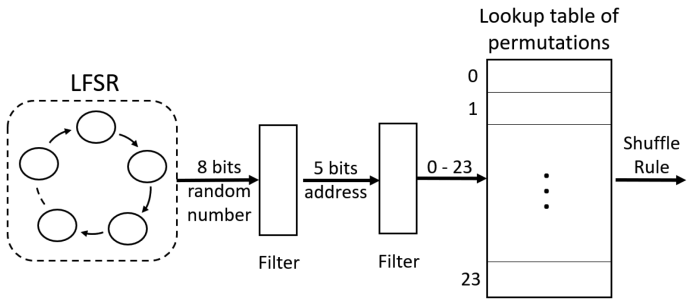

Fig. 10. Pseudo random number generator.

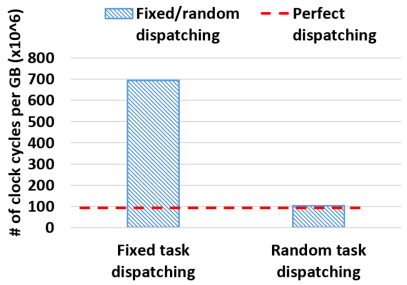

Fig. 11. Performance comparison between the fixed dispatching and random dispatching.

a 10MB file where each line is 32B and matched strings only appear in a fixed position in each line. We use IP addresses as matched strings and use regex to locate all the IP addresses in the file. Thus, the matched strings are at a fixed position in every 32B. The perfect dispatching in this case is the one-step linear shift. Figure 11 shows the comparison between the fixed dispatching (i.e., without shuffling) and random dispatching in terms of the number of clock cycles. We observe that the performance of random dispatching is close to perfect dispatching and is 6.7x faster than fixed dispatching in this case.

4.3 Advanced Syntax

One of the advantages of the current design of Registor is that it keeps the integrity of regex instead of splitting regex into parallel executables, protecting the syntax relationship within regex. Thus, it is easy for Registor to support advanced syntax where an internal relationship is usually necessary. In this section, we focus on how the following features are implemented in our design.

The caret anchor. The caret anchor matches a position before characters. For example, regex " ^a" matches "a" in text "abc," whereas regex "^b" does not match any character. This is a useful syntax that helps in filtering out meaningless matches in real-world applications. Recall that the *matching candidates finder* in Registor HW uses the start character of a regex to filter the input text when the regex contains operators other than characters. The caret anchor can be easily supported in Registor by examining both the character itself and the character before it at the same time in the matching candidates finder. Since the input text is chopped into 16B at each clock cycle, the last character of 16B needs to be saved for further access in the next clock cycle. In our implementation, we also let users define the specific meaning of "^" to be one of the eight characters: "\n," " " (space), "\t," "," ":", "(," "[," "{." This function enhances the efficiency of the matching candidates finder and potentially reduces the workload sent to REMUs. Since the caret anchor combined with the start character of a regex is more *specific* than one character, the compiler will always process

the regex from the start to the end when the regex contains a caret anchor. The cost of supporting the caret anchor in hardware is 1B extra storage and additional logic from extending one character comparison to two.

Backreference and lookahead assertion. Backreference matches the string as previously matched by a capture group where the capture group is usually wrapped by a pair of parentheses. For example, regex "(\w)a\1" could match "dad" or "bab" but could not match "bad" because the symbol "\1" forces the last part of the string matching the content within the parentheses. The number represents a specific capture group according to the order of positions in the regex when multiple capture groups exist. Lookahead in the context of regex syntax is a zero-length assertion, which means that the string matched by lookahead is not included in the results. For example, finding regex "\d(?=cm)" in text "3cm and 4mm" only yields the result "3." The engine will check whether the number is followed by "cm" but will not include this part in the result.

Fully supporting these two functions is known to be challenging even in software regex engines because of the irregularity in the syntax where usual finite automata (FA) construction does not work. Moreover, when regex has backreference, the matching process is NP-hard. To implement these two functions, Registor SW recognizes the states in the FA where backreference or lookahead assertion is needed. These special states are marked as *breakpoints* and identified by REMU. Recall that REMU executes FA based on an *action pointer*, which looks at multiple states simultaneously. However, breakpoints have to be handled separately due to the requirement of special actions like loading previously saved strings and backtracking the input text. To make these features work with the action pointer, we add a stack in each REMU. When any of the breakpoints is encountered during execution, we save the current value of the action pointer in the stack and then set the breakpoint as the only activated state. Once all the states evoked by the breakpoint are finished and the stack is not empty, indicating that the matching process is not finished, REMU pops the stack to restore the remaining activated states. Since multiple breakpoints are possible to be activated at the same time, priority has to be assigned to different types of breakpoints to avoid conflict. We assign a higher priority to the breakpoints that are more likely to put an end to the matching process to achieve better performance. Thus, lookahead assertion has higher priority than backreference. For demonstration purposes, we limit the depth of the stack to two, which means that each regex can have at most one backreference and one lookahead assertion in our implementation.

***Greedy* and *lazy* quantifiers.** A greedy quantifier always finds the longest possible match, whereas a lazy quantifier yields the shortest match. Most regex engines are greedy by default, and a "?" is used to notify the compiler whether the quantifier is greedy or lazy. For example, finding regex "ab[2,3]" in text "abbbc" gives the result "abbb" by default. To enable lazy match, we use "ab[2,3]?" instead and then the result is "abb." During the compiling process, Registor SW knows which mode is required by users, and this information is included in the message sent to Registor HW along with the executable code. Registor HW supports both greedy and lazy quantifiers by using different rules to determine when to stop searching. Specifically, when the lazy quantifier is enabled, REMU stops searching once the acceptance bit is set, regardless of other activated bits in the action pointer. With the greedy quantifier, REMU may need more cycles to find a match. An additional register is used in each REMU to keep the results (length and position of the matched strings) when the acceptance bit is set. It is then updated when a longer match is found. REMU stops searching until all bits in the action pointer are deactivated, meaning that there is no possibly longer match beyond this point.

5 EXPERIMENTAL SETUP

We conduct our experiments using a host server with a quad-core Intel i7-7700 processor running at a clock rate of 3.6GHz and 8GB memory. The NVMe-SSD card is directly plugged into a PCIe

slot of the server. Our SSD prototype card is functioning at the time of this submission but not very stable. The clock speed of FPGA is just 100MHz during our measurement experiments. It is currently being optimized and tuned for higher clock speed. For reporting Registor performance and comparative analysis, it serves the purpose. Besides actual measurements on the prototype, we carry out simulation experiments using System Verilog Universal Verification Methodology (UVM). As for power consumption, we apply both actual measurement and Vectorless Power Analysis, a standard tool for power estimation and analysis in the Xilinx FPGA.

Benchmarks that we select to drive our measurements include NIDS, web data mining, and text processing. All the regex and files are either from a third party or real-world environment with the file size varying from 20MB to 60GB. More than 100 regex are tested that have a variety of patterns and fit the hardware restriction of Registor. The first group of benchmarks is for NIDS. We extract regex from the Snort community library [49] and generate files using the file generator proposed in Becchi et al. [6]. The file generator features an adjustable parameter P_m, denoting the probability of experiencing malicious traffic. The *NIDS* ($P_m = value$) benchmarks are pathological, where the higher value of *Pm* means more cycles in regex processing. We also collect router data (named *router-level NIDS*) from our high-performance computer lab by using PSAD (Intrusion Detection and Log Analysis with iptables) and use suspicious IP addresses as regex. The second group of workloads is *protomata* and *poweren* from ANMLZoo [57] for the evaluation of automata-processing engines. For web data mining, we use *enwiki* from Wikipedia and perform string search on this sizable file (60GB). The benchmark for *text processing* is from a third-party test [22] where regex of various syntax are applied to portions of a famous book [59].

6 RESULTS AND DISCUSSIONS

The Registor system is evaluated in terms of throughput, CPU utilization, I/O bus utilization, and power consumption. The throughput is computed by dividing the file size in terms of the number of characters by the execution time of searching the entire file. We use Linux Grep, a command-line utility in Linux, as the baseline for performance comparison purpose. All experiments are done with cold cache to ensure fairness in comparisons between the baseline and Registor. In addition, more advanced software packages for regex matching are also considered in our performance comparison, such as RE2 [18], PCRE [20], and Onig-uruma [26]. RE2 is developed and used by Google, and PCRE is used by a number of programs including Apache HTTP Server and R scripting language. Onig-uruma is used by the Ruby programming language and many other products, such as Atom, Tera Term, and Sublime Text. In our experiments, we have PCRE run under three different configurations: PCRE (standard), PCRE-DFA, and PCRE-JIT by using pcre2_match(), pcre2_dfa_match(), and pcre2_jit_compile() functions, respectively. The algorithm provided by pcre2_match() function is based on NFA [21], whereas the algorithm provided by pcre2_match() function is based on DFA [21]. PCRE-JIT is optimized for performance through extra processing of regexes before pattern matching is performed.

6.1 Throughput

Our first experiment is to measure the search throughput of Registor as compared to baseline. We pick up the first 2.1GB of *enwiki* file as the microbenchmark. The search throughput of Registor is compared to Linux Grep. The measured throughput is depicted in Figure 12. It can be seen from Figure 12 that Registor shows much better performance compared to Linux Grep. Throughput of Linux Grep is 214MB/s, whereas the throughput of Registor is 382MB/s. Note that these throughputs were measured on the SSD prototype that is still under development and being tuned for better I/O performance. The internal bus width is 16B, and FPGA is running at 100MHz. Even with this compromised configuration, Registor still shows better performance than Linux Grep.

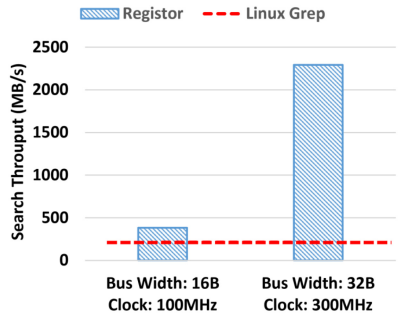

Fig. 12. Throughput of Registor under different configurations compared to Linux Grep.

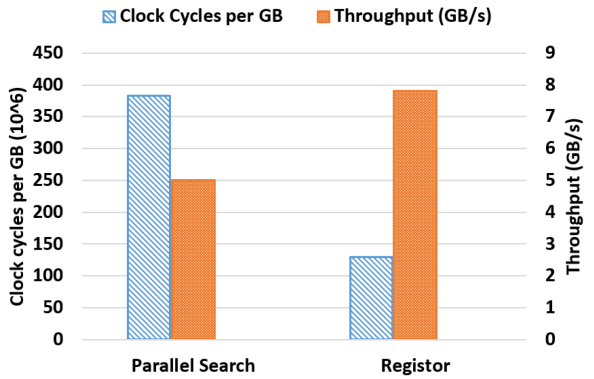

Fig. 13. Comparison between parallel software search and Registor.

By expanding the bus width to 32B and raising the clock speed to 300MHz in FPGA, commonly seen in modern SSDs, Registor is able to achieve the throughput of 2.3GB/s and outperforms Linux Grep by more than 10x. We expect much better throughput if Registor is implemented in an ASIC with a much higher clock rate and optimized I/O performance.

To further demonstrate Registor's advantages in throughput, we also compared Registor to parallel software search implemented using Linux Grep. We have the software search run on a powerful server that has an eight-core 16-thread Xeon CPU and Memblaze PBlaze5 SSD whose 128KB sequential read speed is up to 6GB/s. Although Grep itself is a single-threaded program, we open 16 subprocesses to bring the power of the server into full play. The file we used in this experiment is *enwiki*, which is partitioned into 16 segments of approximately the same size to feed the 16 subprocesses. For Registor, we use the results from the preceding tests with the bus width set to 32B and the clock speed set to 300MHz. We first measured the number of clock cycles that are used to complete the search. Figure 13 plots the number of clock cycles per gigabyte used by parallel software search and Registor, respectively. The results show that software search uses 383×10^6 clock cycles, whereas Registor only requires 130×10^6 clock cycles, which is only one third of the software search. We then estimate the performance of the two cases in terms of throughput using the realistic clock speeds. The server we used is running at 1.8GHz, and we expect Registor to run at 1GHz if implemented in ASIC. As shown in Figure 13, Registor runs 50% faster than parallel software search.

Our next experiment is to measure throughput using the UVM test (see Section 5). We measured Registor's throughput and compared it to several widely used software regex engines. This method is based on real bitfile and is accurate to clock cycle. For software regex search engines, we load the

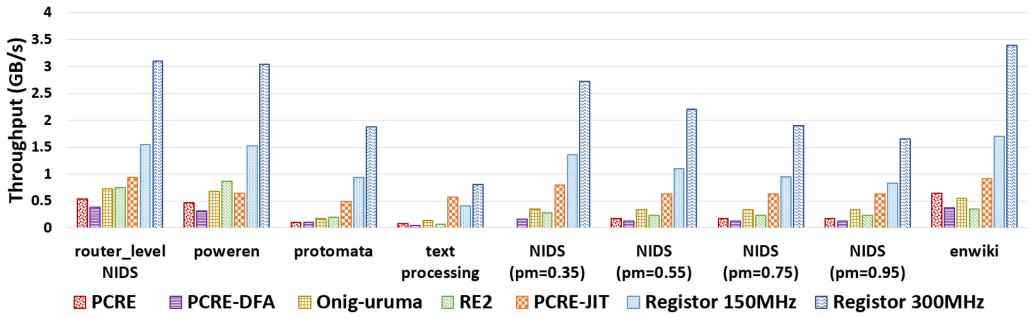

Fig. 14. Throughput comparison among different search engines.

Table 5. Throughput (GB/s) for the Text Precessing Benchmark

No.	Regexes	PCRE	PCRE-DFA	Onig-uruma	RE2	PCRE-JIT	Registor 150MHz	Registor 300MHz
1	Twain	2.67	1.62	1.01	5.03	0.97	1.45	2.91
2	(?i)Twain	0.36	0.25	0.20	0.26	0.92	1.45	2.91
3	[a-z]shing	0.06	0.03	1.05	0.17	1.01	1.53	3.07
4	Huck[a-zA-Z]+|Saw[a-zA-Z]+	0.87	0.82	0.48	0.32	5.41	1.53	3.07
5	[a-q][^u-z]{13}x	0.05	0.01	0.34	0.01	7.07	0.07	0.14
6	Tom|Sawyer|Huckleberry|Finn	0.67	0.62	0.42	0.30	0.72	1.38	2.76
7	(?i)Tom|Sawyer|Huckleberry|Finn	0.08	0.07	0.05	0.21	0.27	0.70	1.39
8	Tom.{10,25}river|river.{10,25}Tom	0.35	0.30	0.27	0.29	1.12	1.45	2.91
9	[a-zA-Z]+ing	0.03	0.02	0.03	0.15	0.25	1.45	2.89
10	\s[a-zA-Z]{0,12}ing\s	0.06	0.04	0.32	0.21	0.20	0.29	0.57
11	([A-Za-z]awyer|[A-Za-z]inn)\s	0.03	0.02	0.12	0.18	0.52	0.21	0.42
12	["'][^"']{0,30}[?!\.]["']	0.40	0.29	0.28	0.28	1.59	1.38	2.77

file into memory and then run regex search engines with the basic counting function excluding the time of loading files, results formatting, and displaying. Figure 14 shows that the throughputs vary among different benchmarks since the regex and files are of different patterns and types. As expected, the NIDS benchmarks of higher *Pm* value results in lower throughput because of more cycles in processing. For most of the applications, Registor achieves higher throughput than software even when running at 150MHz. When running at 300MHz clock speed (usually in ASIC implementation), the throughput is as high as 3.2GB/s, which outperforms traditional regex search engines by 16x.

We noticed that Registor shows limited advantage over existing approaches for the text processing benchmark in Figure 14. To understand why text processing of Registor did not show as large an improvement as other benchmarks, we carried out additional experiments for regexes that we used in the text processing benchmark. The corresponding throughputs are listed in Table 5. The top three ranking throughputs for each regex are marked with light red to highlight regex engines that have better performance. We also plot Registor's throughput in Figure 15 to provide an intuitive view of how the throughput is varied when searching different regexes. By comparing Table 5 and Figure 15, we observe that Registor loses its advantages in regex No. 1. Although there is no decrease in throughput when searching for raw strings (see Figure 15), Registor falls behind other software engines (see Table 5), where better algorithms are usually used for string search.

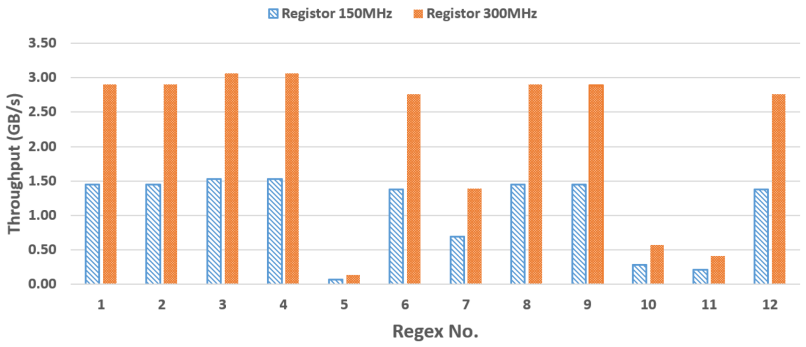

Fig. 15. Throughput for the text processing benchmark.

For example, Linux Grep uses the Boyer-Moore algorithm when searching for strings to achieve sublinear matching speed. In addition, we also noticed that throughputs drop severely for regex Nos. 5, 10, and 11. What those regexes have in common is character classes (e.g., "[a-zA-Z]") and whitespace characters ("\s"). Since the input text is an English book, "[a-zA-Z]" and "\s" cover almost every character in the text. Therefore, it is difficult for the matching candidates finder to reduce the workload for REMUs efficiently. Moreover, these regexes are more complex than others because they contain either quantifiers or alternations, which requires more clock cycles to process in REMUs. Another notable performance drop in Figure 15 is regex No. 7. The difference in throughput between regex Nos. 6 and 7 is caused by case-insensitive indicator "(?i)." This is mainly because only four characters (T,S,H,F) can pass the matching candidates finder for regex No. 6, whereas eight characters (T,t,S,s,H,h,F,f) can pass the matching candidates finder for regex No. 7. From the preceding analysis, one way to improve Registor's performance is to add a dedicated module for string search that uses algorithms like Boyer-Moore and Aho-Corasick. Another thing that can help is to implement different REMUs according to the different patterns of regexes. For example, bit-split automata [51] can be applied to regexes that are not selective because the performance is independent from the matching candidates finder. However, it cannot easily handle some syntaxes (e.g., capture groups) where our current implementation of REMUs are still useful. The use of different search modules can be determined by analyzing the characteristic of regexes during compiling in Registor SW.

6.2 CPU Utilization

To evaluate Registor's effect on CPU workload, we measured the CPU utilization of Registor and compared it to Linux Grep. The experiment is conducted by using a microbenchmark (also used in Section 6.1) and the "ps" command in the Linux system with Registor and Linux Grep, respectively. Figure 16 plots the CPU utilization over time. During the runtime, the average CPU utilization of Registor is 11.90%, whereas that of Linux Grep is 70.09%, implying that Registor consumes 82% fewer CPU resources than Linux Grep. This is because Registor offloads compute-intensive tasks to FPGA. The only functions that consumes CPU resources are the user library and APIs that are simple and lightweight. Therefore, Registor consumes almost no CPU clock cycles as compared to Linux Grep. The remaining CPU resources can be used by other applications.

6.3 I/O Bus Utilization

To better illustrate reduction in I/O bus utilization of Registor over other software solutions, we measured the data transfer ratio (a value between 0% and 100%), calculated by dividing the size of data transferred from the SSD to host by file size. Since NVMe's read command can request up to

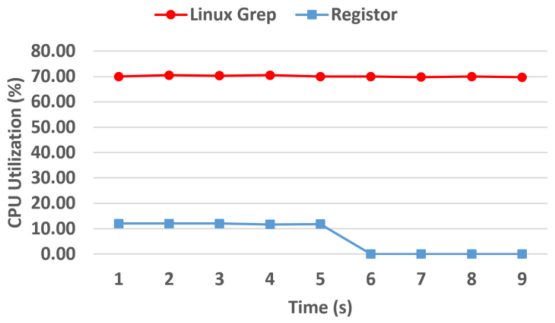

Fig. 16. CPU utilization of Registor and Linux Grep.

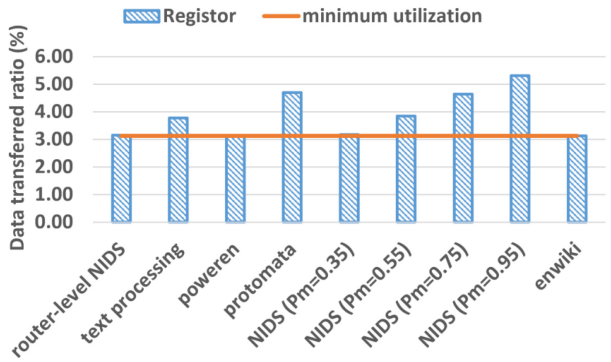

Fig. 17. Data transfer ratio (the size of data transferred from the SSD to the host divided by the file size).

128kB data per I/O and the minimum size of data transferred to the host is 4kB, the data transfer ratio in the best scenario is 4kB/128kB = 3.125%. For the software solution, the data transfer ratio is a constant value of 100% because the whole file is loaded to the host for further scanning.

Figure 17 shows the data transfer ratios of Registor for different benchmarks with different file sizes. We observed that all values are below 5%, indicating that Registor reduces I/O bus utilization dramatically for all of our experiments. Although the ratio depends on how selective the regex is and can possibly reach 100% in some extreme cases, Registor is able to reduce the data transfer ratio to exactly the regex matches, which is a small fraction of total data in most applications. In most of the cases in our experiments, which are from third-party and real-world applications, the ratios are close to 3.125%, which is the minimum value of data transfer ratio in our design. It is obvious that Registor exhibits much less negative impact on other applications in I/O bandwidth, a tiny fraction of the file to the host. Putting it in a different perspective, such reduction on I/O bus utilization can also be interpreted as increased IOPS that Registor can offer. For example, a data transfer ratio of 3.125% means that a 100K IOPS SSD with Registor enabled provides regex search applications with an equivalent 3.175 million IOPS of SSD without Registor.

To further illustrate how Registor alleviates the I/O bottleneck problem, we analyze *effective throughput* and *required throughput* by applications. The effective throughput is what Registor can offer to regex applications, whereas the required throughput is a measure for the necessary I/O throughput for an application to achieve a desired effective throughput. Figure 18 plots the effective throughput and required throughput of Registor when it is implemented in a 1GHz ASIC with an internal bus width of 64B. It can be seen from this figure that for all benchmarks, the

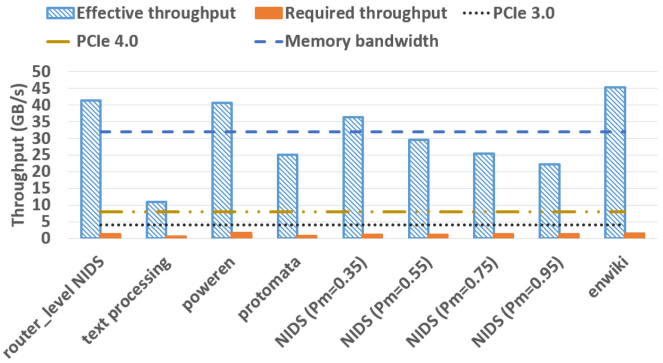

Fig. 18. The effective throughput and required throughput of Registor.

Table 6. Summary of Power Consumption

DRAM	0.4198W
FPGA	8.7918W Registor: 0.119W
NAND	2.7888W
Others	3.2396W
Total Power	15.24W

required throughput of Registor is far below what the current PCIe can provide. For about half of the benchmarks, the effective throughputs of Registor match the bandwidth of modern DRAM memory.

6.4 Area and Power Consumption

In our current implementation, the SSD prototype uses 78% of LUTs and logic cells in a Xilinx 9p FPGA board including the SSD controller, AXI bus, LDPC, and some other modules that are necessary for a working SSD. Registor hardware consumes 9% of the board's logic cell, contributing about 11.5% of the total logic cells of the SSD prototype.

As for power consumption, we apply both actual measurement and the STA method (introduced in Section 4.2) to measure energy efficiency. Measured power consumption of the SSD prototype and the power usage of Registor hardware are shown in Table 5. As summarized in Table 6, Registor hardware consumes 0.119W, which is only a tiny fraction (less than 1%) of the total power consumption of the SSD prototype.

7 RELATED WORK

Regex search acceleration. Extensive research has been reported in accelerating regex search over the past decade. Some researchers take advantage of GPU [30, 62], SIMD [8, 31, 44], and multi-core architectures [38], whereas others focus on FPGA/ASIC-based solutions [7, 14, 17, 23, 27, 43, 47, 48, 52, 61].

Early work discusses regex search in FPGA/ASIC by mapping non-deterministic finite automata (NFA) [47, 61] and deterministic finite automata (DFA) [7, 14, 27] to programmable logic. A recent study by Gogte et al. [17] proposed HARE, extended from Tandon et al. [52], compiles regex into subexpressions and runs bit-split automata [51] on each subexpression in parallel to achieve high throughput. IBM PowerEN integrates an ASIC-based regex engine (RegX) that splits regex into sub-patterns to reduce the size of states (in DFA) and then uses a local results processer to check if the partial results are in the right order [33, 56]. Another generalized ASIC-based accelerator

is Micron's AP [10]. It is capable of processing large NFA whose state transitions are stored in bit vectors and being executed via a customizable routing matrix. The most recent work by Subramaniyan and Das [50] breaks the the bottleneck on Micron's AP by parallelizing NFA execution by means of leveraging AP's flow and special properties of NFA. Fang et al. [12] propose a UAP architecture that features a programmable engine for FA and supports a wide range of FA models.

The works mentioned previously achieves encouraging progress in accelerating regex search, and most of them are designed for NIDS or in-memory pattern matching. With different optimization objectives and a different architecture level from the preceding work, our proposed Registor works in a different manner. First, it is located inside the SSD for the purpose of eliminating I/O bottlenecks on processing large amounts of data stored in storage. Second, it is capable of extracting file semantics from data blocks and providing the host with contextual information on search results.

Another technique that relates to our work is Amazon's S3 Select [3] that enables applications to retrieve only the data needed by using SQL expressions. Since the amount of data transferred is reduced, applications with S3 Select can achieve significant performance gains. Registor's approach of avoiding data movement bears some similarity to S3 Select's approach. S3 Select reduces data transferred from S3 to applications, whereas Registor reduces data from flash storage to the host by offloading regex matching to storage. SSDs with Registor can be equipped in data centers with S3 Select to further reduce data movement from the storage device to the host.

Near data processing. The benefits of near data processing (NDP) have been demonstrated by many researchers at different levels of system hierarchy such as in-memory computing [1, 16, 28, 60] and processing in storage [5, 19, 24, 46, 54, 55].

Some existing work exploits the computational power of the SSD controller by offloading the data-intensive tasks to the embedded cores [54, 55]. Tiwari et al. [54] present detailed energy and performance models for data analysis using embedded cores in the SSD. Tseng et al. [55] implement Morpheus-SSD, targeted at object deserialization on a hybrid architecture of GPU, CPU, and embedded cores. They reduce the overhead of data transmission between embedded cores and the GPU by using the NVMe-P2P mechanism. Unlike their approaches, Registor's design considers scenarios when embedded cores are heavily loaded by SSD control functions. We offload computations to FPGA and have it sit in the internal data path between the NAND flash to host interface to achieve on-the-fly regex search.

Some other groups integrate FPGA/ASIC in the SSD for computing purposes [19, 24, 46]. Willow SSD by Seshadri et al. [46] allows users to implement customized features to support particular applications by deploying several RISC processors in the SSD, using a BEE3 FPGA-based prototype [36]. Gu et al. [19] propose Biscuit, an NDP framework that includes a hardware pattern matcher for string search in each channel of NAND chips. BlueDBM [24] applies in-storage processing to big data analytics, which integrates the Morris-Pratt (MP) string search engine in the SSD [37]. Different from the preceding work, Registor eliminates I/O bottlenecks in unstructured data processing that requires regex search, which has higher computational complexity than string search.

8 CONCLUSION

We presented Registor, a platform for regex processing in storage. It features a hardware search engine that applies regex search on-the-fly while data is transferred from NAND flash to the host. The search engine achieves a high processing rate that matches the speed of the internal bus in the SSD by fully exploiting the parallelism in FPGA. The deep pipeline structure of Registor consists of the file semantics extractor, matching candidates finder, REMUs, and results organizer. Furthermore, we developed a user library to facilitate the upper-layer applications to take advantage of the

search engine. To quantitatively evaluate Registor's performance, we built a working prototype of Registor that was integrated into an NMVe-SSD card. The implementation of Registor needs no OS changes, making Registor readily available to user applications. Using the Registor prototype, we carried out extensive experiments to show its superb advantages over existing solutions in terms of eliminating the I/O bottleneck. Our future work includes adopting more advanced automata designs in our Registor and further optimization of the I/O path inside our SSD prototype.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments that helped greatly in improving the quality of the article. The authors are thankful to Shuqun Xie, Qingchun Zhu, Ying Yang, Archie Wu, and Pan Qin for providing guidance to this work.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. IEEE, Los Alamitos, CA, 105–117.

[2] Shahriar Akter and Samuel Fosso Wamba. 2016. Big data analytics in E-commerce: A systematic review and agenda for future research. *Electronic Markets* 26, 2 (2016), 173–194.

[3] Amazon. 2018. Amazon S3. Retrieved February 26, 2019 from https://aws.amazon.com/s3/.

[4] Apache. 2018. Lucene. Retrieved February 26, 2019 from https://lucene.apache.org/.

[5] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. 2017. It's time to think about an operating system for near data processing architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. ACM, New York, NY, 56–61.

[6] Michela Becchi, Mark Franklin, and Patrick Crowley. 2008. A workload for evaluating deep packet inspection architectures. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'08)*. IEEE, Los Alamitos, CA, 79–89.

[7] Benjamin C. Brodie, David E. Taylor, and Ron K. Cytron. 2006. A scalable architecture for high-throughput regular-expression pattern matching. *ACM SIGARCH Computer Architecture News* 34, 2 (2006), 191–202.

[8] Robert D. Cameron, Thomas C. Shermer, Arrvindh Shriraman, Kenneth S. Herdy, Dan Lin, Benjamin R. Hull, and Meng Lin. 2014. Bitwise data parallelism in regular expression matching. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. ACM, New York, NY, 139–150.

[9] Russ Cox. 2009. Regular Expression Matching: The Virtual Machine Approach. Retrieved February 26, 2019 from https://swtch.com/~rsc/regexp/regexp2.html.

[10] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. 2014. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 12 (2014), 3088–3098.

[11] NVM Express. 2018. NVM Express Revision 1.3a October 24, 2017. Retrieved February 26, 2019 from http://nvmexpress.org/wp-content/uploads/NVM-Express-1_3a-20171024_ratified.pdf.

[12] Yuanwei Fang, Tung T. Hoang, Michela Becchi, and Andrew A. Chien. 2015. Fast support for unstructured data processing: The unified automata processor. In *Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. IEEE, Los Alamitos, CA, 533–545.

[13] Yuanwei Fang, Chen Zou, Aaron J. Elmore, and Andrew A. Chien. 2017. UDP: A programmable accelerator for extract-transform-load workloads and more. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, New York, NY, 55–68.

[14] Domenico Ficara, Stefano Giordano, Gregorio Procissi, Fabio Vitucci, Gianni Antichi, and Andrea Di Pietro. 2008. An improved DFA for fast regular expression matching. *ACM SIGCOMM Computer Communication Review* 38, 5 (2008), 29–40.

[15] Amir Gandomi and Murtaza Haider. 2015. Beyond the hype: Big data concepts, methods, and analytics. *International Journal of Information Management* 35, 2 (2015), 137–144.

[16] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. 2015. Practical near-data processing for in-memory analytics frameworks. In *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT'15)*. IEEE, Los Alamitos, CA, 113–124.

[17] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. 2016. HARE: Hardware accelerator for regular expressions. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, Los Alamitos, CA, 1–12.

[18] Google. 2019. RE2. Retrieved February 26, 2019 from https://github.com/google/re2.
[19] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, et al. 2016. Biscuit: A framework for near-data processing of big data workloads. In *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. IEEE, Los Alamitos, CA, 153–165.
[20] Philip Hazel. 2019. PCRE—Perl Compatible Regular Expressions. Retrieved February 26, 2019 from https://www.pcre.org/.
[21] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2001. Introduction to automata theory, languages, and computation. *ACM SIGACT News* 32, 1 (2001), 60–65.
[22] Github. 2017. Performance Comparison of Regular Expression Engines. Retrieved February 26, 2019 from https://zherczeg.github.io/sljit/regex_perf.html.
[23] Titan IC. 2018. Hyperion F1 10G Regex File Scan. http://titan-ic.com/products/hyperion-f1-10g-regex-file-scan
[24] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, et al. 2015. Bluedbm: An appliance for big data analytics. In *Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. IEEE, Los Alamitos, CA, 1–13.
[25] A. Katal, M. Wazid, and R. H. Goudar. 2013. Big data: Issues, challenges, tools and good practices. In *Proceedings of the 6th International Conference on Contemporary Computing (IC3'13)*. IEEE, Los Alamitos, CA, 404–409.
[26] K. Kosako. 2019. PCRE—Perl Compatible Regular Expressions. Retrieved February 26, 2019 from https://github.com/kkos/oniguruma.
[27] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. 2006. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. *ACM SIGCOMM Computer Communication Review* 36, 4 (Oct. 2006), 339–350.
[28] Snehasish Kumar, Arrvindh Shriraman, Vijayalakshmi Srinivasan, Dan Lin, and Jordon Phillips. 2014. SQRL: Hardware accelerator for collecting software data structures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. ACM, New York, NY, 475–476.
[29] John Levine. 2009. *Flex and Bison: Text Processing Tools*. O'Reilly Media Inc.
[30] Cheng-Hung Lin, Chen-Hsiung Liu, Lung-Sheng Chien, and Shih-Chieh Chang. 2013. Accelerating pattern matching using a novel parallel algorithm on GPUs. *IEEE Transactions on Computers* 62, 10 (2013), 1906–1916.
[31] Dan Lin, Nigel Medforth, Kenneth S. Herdy, Arrvindh Shriraman, and Rob Cameron. 2012. Parabix: Boosting the efficiency of text processing on commodity processors. In *Proceedings of the IEEE 18th International Symposium on High Performance Computer Architecture (HPCA'12)*. IEEE, Los Alamitos, CA, 1–12.
[32] Linux. 2019. Source to sys/ioctl.h. Retrieved February 26, 2019 from https://unix.superglobalmegacorp.com/Net2/newsrc/sys/ioctl.h.html.
[33] Jan Van Lunteren, Christoph Hagleitner, Timothy Heil, Giora Biran, Uzi Shvadron, and Kubilay Atasu. 2012. Designing a programmable wire-speed regular-expression matching accelerator. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 461–472.
[34] Micron. 2018. MT29F8G16ADADAH4-IT. Retrieved February 26, 2019 from https://www.micron.com/products/nand-flash/slc-nand/part-catalog/mt29f8g16adadah4-it.
[35] Micron. 2018. MT29F2T08CUHBBM4-3R. Retrieved February 26, 2019 from https://www.datasheets.com/datasheet/mt29f2t08cuhbbm4-3r:b-micron-technology-75292692.
[36] Microsoft. 2018. BEE3 Established: February 26, 2008. https://www.microsoft.com/en-us/research/project/bee3/
[37] D. E. Knuth, J. H. Morris Jr., and V. R. Pratt. 1977. Fast pattern matching in strings. *SIAM Journal on Computing* 6, 2 (1977), 323–350.
[38] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. 2014. Data-parallel finite-state machines. *ACM SIGARCH Computer Architecture News* 42, 1 (March 2014), 529–542.
[39] PCI-SIG. 2018. Frequently Asked Questions: PCI Express - 3.0. Retrieved February 26, 2019 from https://pcisig.com/faq?field_category_value%5B%5D=pci_express_3.0&keys=.
[40] PCI-SIG. 2018. Frequently Asked Questions: PCI Express - 4.0. Retrieved February 26, 2019 from https://pcisig.com/faq?field_category_value%5B%5D=pci_express_4.0&keys=.
[41] Shuyi Pei, Jing Yang, and Qing Yang. 2018. REGISTOR: A platform for unstructured data processing inside SSD storage. In *Proceedings of the 11th ACM International Systems and Storage Conference*. ACM, New York, NY, 13–25.
[42] RegexLib. 2017. Regular Expression Library. Retrieved February 26, 2019 from http://regexlib.com/.
[43] Indranil Roy, Ankit Srivastava, Marziyeh Nourian, Michela Becchi, and Srinivas Aluru. 2016. High performance pattern matching using the automata processor. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium*. IEEE, Los Alamitos, CA, 1123–1132.
[44] Valentina Salapura, Tejas Karkhanis, Priya Nagpurkar, and Jose Moreira. 2012. Accelerating business analytics applications. In *Proceedings of the IEEE 18th International Symposium on High Performance Computer Architecture (HPCA'12)*. IEEE, Los Alamitos, CA, 1–10.

[45] Eric E. Schadt, Michael D. Linderman, Jon Sorenson, Lawrence Lee, and Garry P. Nolan. 2010. Computational solutions to large-scale data management and analysis. *Nature Reviews Genetics* 11, 9 (2010), 647.

[46] Sudharsan Seshadri, Mark Gahagan, Meenakshi Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, et al. 2014. Willow: A user-programmable SSD. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. 67–80.

[47] Reetinder Sidhu and Viktor K. Prasanna. 2001. Fast regular expression matching using FPGAs. In *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*. IEEE, Los Alamitos, CA, 227–238.

[48] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, New York, NY, 403–415.

[49] Snort. 2017. Snort—Network Intrusion Detection and Prevention System. Retrieved February 26, 2019 from https://www.snort.org/.

[50] Arun Subramaniyan and Reetuparna Das. 2017. Parallel automata processor. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, New York, NY, 600–612.

[51] Lin Tan and Timothy Sherwood. 2005. A high throughput string matching architecture for intrusion detection and prevention. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, Los Alamitos, CA, 112–122.

[52] Prateek Tandon, Faissal M. Sleiman, Michael J. Cafarella, and Thomas F. Wenisch. 2016. Hawk: Hardware support for unstructured log processing. In *Proceedings of the IEEE 32nd International Conference on Data Engineering (ICDE'16)*. IEEE, Los Alamitos, CA, 469–480.

[53] Ken Thompson. 1968. Programming techniques: Regular expression search algorithm. *Communications of the ACM* 11, 6 (1968), 419–422.

[54] Devesh Tiwari, Simona Boboila, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. 2013. Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. 119–132.

[55] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. 2016. Morpheus: Creating application objects efficiently for heterogeneous computing. In *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. IEEE, Los Alamitos, CA, 53–65.

[56] Jan van Lunteren and Alexis Guanella. 2012. Hardware-accelerated regular expression matching at multiple tens of Gb/s. In *Proceedings of the 2012 IEEE INFOCOM Conference*. IEEE, Los Alamitos, CA, 1737–1745.

[57] Jack Wadden, Vinh Dang, Nathan Brunelle, Tommy Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, et al. 2016. ANMLzoo: A benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'16)*. IEEE, Los Alamitos, CA, 1–12.

[58] Fei-Yue Wang, Kathleen M. Carley, Daniel Zeng, and Wenji Mao. 2007. Social computing: From social informatics to social intelligence. *IEEE Intelligent Systems* 22, 2 (2007), 79–83.

[59] Project Gutenberg. 2018. The Entire Project Gutenberg Works of Mark Twain by Mark Twain. Retrieved February 26, 2019 from http://www.gutenberg.org/ebooks/3200?msg=welcome_stranger.

[60] Sam Likun Xi, Oreoluwa Babarinsa, Manos Athanassoulis, and Stratos Idreos. 2015. Beyond the wall: Near-data processing for databases. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*. ACM, New York, NY, 2.

[61] Yi-Hua E. Yang, Weirong Jiang, and Viktor K. Prasanna. 2008. Compact architecture for high-throughput regular expression matching on FPGA. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, New York, NY, 30–39.

[62] Xiaodong Yu and Michela Becchi. 2013. GPU acceleration of regular expression matching for large datasets: Exploring the implementation space. In *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, New York, NY, 18.

Received October 2018; revised January 2019; accepted January 2019