REGISTOR: A Platform for Unstructured Data Processing Inside SSD Storage

Shuyi Pei, Jing Yang and Qing Yang
Dept. of Electrical, Computer, and Biomedical Engineering, University of Rhode Island
Kingston, RI, USA 02881
Shenzhen DAPU Microelectronics Co., Ltd
Shenzhen, China
{spei,jyang,qyang}@ele.uri.edu

ABSTRACT

This paper 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 flash SSD that processes data on-the-fly during data transmission from NAND flash to host. In order to make the speed of regex search match the internal bus speed of modern SSD, a deep pipeline structure is designed in Registor hardware consisting of file semantics extractor, matching candidates finder, regex matching units (REMUs) and results organizer. Furthermore, each stage of the pipeline makes 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 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 I/O bandwidth requirement by up to 97% and CPU utilization by as much as 82% for regex search in large data sets.

SYSTOR '18, June 4–7, 2018, HAIFA, Israel
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5849-1/18/06. . . $15.00
https://doi.org/10.1145/3211890.3211900

CCS CONCEPTS

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

KEYWORDS

Regular expressions, processing in storage, near data processing, SSD storage, hardware accelerator

1 INTRODUCTION

Staggering growth of big data has generated numerous challenges to both 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 [15, 16, 18, 24]. 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, regex search provides a powerful and flexible approach for unstructured data analysis [2]. However, regex search in a file is compute-intensive since 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 tens of gigabyte 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 [11, 29, 39], multi-core architectures [35], and GPU widely used for parallel computing[28, 56]. Recent work [15] proposed 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 [20, 22, 30, 50]. Micron's automata processor (AP) implements NFA and uses bit-vectors and routing matrix to perform state transitions, with one AP chip achieving 1Gps line rate [13, 44]. Helios regex processor from Titan IC, commercially used for network intrusion detections, can deliver throughput up to 10Gbps based on FPGA acceleration [22]. IBM PowerEN integrates a regex engine (RegX) that splits regex into sub-patterns and processes in parallel, achieving scanning rates of 20-40 Gbps [30, 50]. HARE's ASIC RTL implementation, taking advantage of bit-split automata [45], can process data at a rate of 32GB/s that matches the modern memory bandwidth [20].

While 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 are stored in data storage, such as high-speed flash memory SSDs, as exemplified by e-commerce [5], social computing[52] and bioinformatics [40]. All existing accelerator techniques require loading this huge amount of data from data storage, such as AWS S3 storage service [6], to the system DRAM before any analysis can be done. Moving such large data from storage to system DRAM places a great burden to the storage I/O bus. The typical high speed I/O bus in use today such as PCIe 3.0 only provides 3.94 GB/s with 4 channels and 7.88GB/s with 8 channels [36]. Even the next-generation PCIe 4.0 is expected to offer only 7.88GB/s with 4 channels and 15.75GB/s with 8 channels [37]. On the contrary, 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 [32], each of which runs at 667MT/s (megatransfers per second) [32]. The aggregated throughput of flash memories at the back end of modern SSDs reaches 32GB/s with the channel width of 16 bits [31]. 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.

In order to truly speed up regex search and eliminate the system bottleneck, we propose a new approach to accelerating regex search, referred to 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. 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. Also, Registor hardware is able to obtain file semantics from out-of-order data blocks and responds to host's request by sending the data that match the regex exactly, associated with line number, displacement, length and so forth.

In order for the search engine of Registor to work for any applications running on the host, we develop a user library which includes APIs that can be called by user applications, a compiler which translates regex to the formats that are understandable by hardware, and an exception handler that improves 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 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 SSD. The search engine is fully pipelined consisting of a file semantics extractor, matching candidates finder, regex matching units, 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 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 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 operating system.

The rest of this paper is organized as follows. In section 2, we present the overall architecture of Registor followed by 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 the experimental setup for evaluation purpose. The results are discussed in Section

5 to demonstrate the advantage of Registor over state-of-the-art solutions. Section 6 discusses related work and Section 7 concludes the paper.

2 REGISTOR HARDWARE

Consider a system shown in Figure 1. Registor sits between host applications and SSD device. It consists of two major parts: *hardware search engine* and *user library*. In this section, we focus on the architecture of 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.

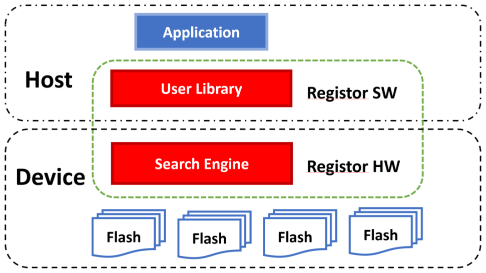

Figure 1: Overview of the system showing where Registor sits

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 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 engine to match the speed of data transfer in 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*, *regex matching units (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 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, results organizer reorders the intermediate results from REMUs before sending to host.

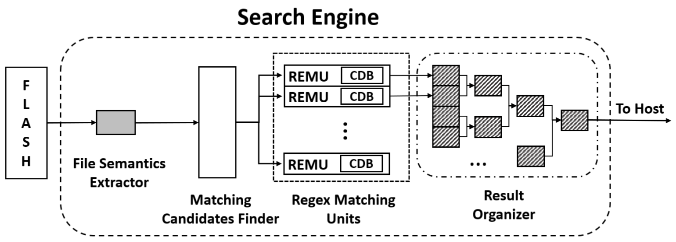

Figure 2: Registor's hardware pipeline

2.2 File Semantics Extractor

Files are stored in separate data blocks in SSD and retrieved from NAND flash regardless of ordering/sequence in order 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 as well as providing in-file locations of matched strings.

Retrieving File Layout: In 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 supper block from 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 file path, and obtain the entire file layout by parsing the inode.

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 in order to conforming file format, we design a reordering buffer (RoB) for block reordering. The RoB is a random access buffer with capacity of k blocks. The incoming data blocks are buffered in 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 then buffer size, we use a pointer indicating the logical start of 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 user library to prevent out-of-range LBNs that may cause RoB overflow.

2.3 Matching Candidates Finder

Matching Candidates Finder finds possible matches by checking whether the input character accepted by the start character of regex. During this process, 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 knows 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 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 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 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 (REMUs)

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 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 [47] and [12] to generate the code in 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 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, i.e. "PPAIR a,a" matches char "a" while "PPAIR a,b" matches character set "[ab]" and "PPAIR a,A" matches case-insensitive "a". This feature is useful and results in better code efficiency. An example executable code for searching regex "a(b|c)d" is shown in Figure 3. Note that "b|c" is interpreted using "SPLIT" and "JMP" and "[bc]", which has the same meaning as "b|c" 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.

```
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
```

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

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 program counter, PC), which holds the bit map of instruction buffer entries. That is, 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" which 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* which

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 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

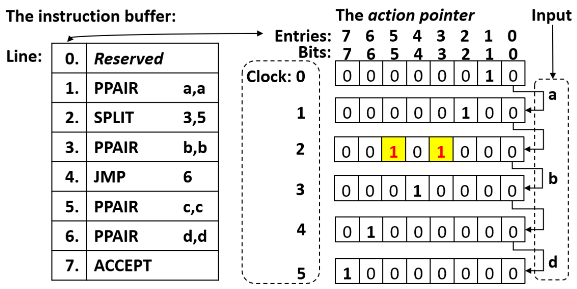

Figure 4: An example of execution in REMU

requires the next input character to be compared with both line 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 reject line 5, thus REMU continues to the line after line 3. After running for a few clock cycles, REMU reaches line 7 which indicates the string "abd" is accepted by regex "a(b|c)d".

Note that the above 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 [42, 55], DFA [17, 25], B-FSM [30, 50], bit-split automata [20, 45] 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 matching candidates finder independently and simultaneously. To provide data streams for REMUs, we deploy cyclic data buffer to replay the data stream for each REMU. The input data stream from NAND flash is saved in CDBs of matching candidates finder and flushed out at 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 Register, 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 users-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 takes three parameters: *slba* (starting logical block address), number of blocks and regex. The slba information can be obtained by calling registror_file_layout(). It will also check file access permission based on the access control information from file's inode and credentials of the current process.

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 [7]) or repurposed for data queries in NoSQL or SQL database.

The efficiency-aware compiler: The compiler translates regex to executable code that is 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

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 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

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 AST as the start of 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 preoder traversal and the second part is from the specific node to the start of preoder traversal. Recall that we find matching candidates based on the first character of regex. Our efficiency-aware compiler can reduce the total number of matching candidates, making the REMUs more efficient.
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 passes syntax check can benefit from our proposed Registor due to hardware resource limitation or 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 problem [8]. 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 host is restricted in size to less than the amount of data per I/O request. If the search result exceeds such limit, the excess is discarded and a bit in result to 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, Registor system features a well-designed data path that achieves low latency and avoids interfering with the normal I/Os of SSD. Registor hardware is placed aside the normal I/O data path. Normal I/Os do not go through the Registor path and hence 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 5. Recall that retrieving file layout needs file information, super blocks and inodes from SSD. The data path for file layout query involves virtual file system for file path, NVMe driver for data transfer, and SSD controller to load super blocks and inodes from NAND flash. Unlike file layout query that is executed only once per search, regex search that has significant impact on the overall performance. To reduce latency, we interface user library directly to the NVMe device drivers, bypassing file system. We avoid modifying the operating system by augmenting extended NVMe commands through 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 standard NVMe and no modifications are made in the operating system, making Registor readily available to user applications. The results from Registor hardware are regarded as normal data blocks requested by normal I/O read command and are sent directly to NVMe driver without the interference of SSD controller, which simplifies the internal control and reduces latency.

4 EXPERIMENTAL SETUP

For the purpose of evaluation of Registor, we have built a working prototype of Registor. This section presents details of its implementation, experimental setup, tools and workloads used in our evaluation.

4.1 Implementation

The entire Registor hardware has been implemented on Xilinx FPGA, the UltraScale+ chip, using Verilog

Table 3: Extended NVMe commands for Registor

Command	Description	Parameters
rgt_fsm_inst	Send executable code generated by the compiler to 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

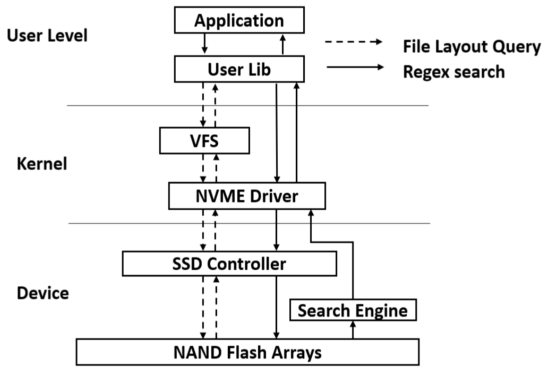

Figure 5: The data path of Registor

language. The RTL of the implementation is integrated in an in-house enterprise-level SSD prototype, as shown in Figure 6 and Table 4.

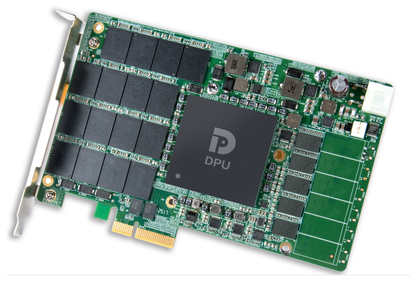

Figure 6: NVMe-SSD with Registor

The user library is implemented in C at 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 [27]. The extended NVMe command is implemented by using the reserved bits (15:08) of *Write-Command Dword 13* in NVMe standard revision 1.3a [14]. The file layout retrieval is implemented using *ioctl()* 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 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

Table 4: Specification of the SSD

FPGA	Xilinx Ultrascale+ 9P, xcvu9pflgb2104-2
DRAM	9X1GB in which 1GB for ECC
NAND flash	32x256GB, 8TB in total
Interface	PCIE Gen 3x4

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 (cyclic data buffers) 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 round robin algorithm to serve the read requests from two REMUs in turns.

4.2 Performance Measurement

We conduct our experiments using a host server with a quad-core Intel i7-7700 processor running at a clock rate of 3.6 GHz 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 the FPGA is just 100 MHz 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 Xilinx FPGA.

Benchmarks that we select to drive our measurements include network intrusion detection (NIDS), web data mining, and text processing. All the regex and files are either from third party or real world environment with the file size varied from 20MB to 60GB. Over 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 Snort community library [3] and generate files using the file generator proposed in [9]. 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 [51] 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 [1] where regex of various syntax are applied to portions of a famous book [53].

5 RESULTS AND DISCUSSIONS

Registor system is evaluated in terms of throughput, CPU utilization, I/O bus utilization, and power consumption. The throughput is computed by dividing file size in terms of the number of characters by the execution time of search the entire file. We use Linux Grep, a command-line utility in Linux, as the base line for performance comparison purpose. In addition, more advanced software packages for regex matching are also considered in our performance comparison such as RE2, PCRE and Onig-uruma. RE2 is developed and used by Google and PCRE, Perl Compatible Regular Expressions, is used by a number of programs including Apache HTTP Server, R scripting language and so forth. Onig-uruma is used by Ruby programming language as well as many other products, e.g. Atom, Tera Term, Sublime Text.

5.1 Throughput

Our first experiment is to measure the search throughput of Registor as compared with baseline. We pick up the first 2.1GB of *enwiki* file as microbenchmark. The search throughput of Registor is compared with Linux Grep. The measured throughput is depicted in Figure 7. It can be seen from Figure 7 that Registor shows much better performance compared to Linux Grep. Throughput of Linux Grep is 214MB/s while the throughout 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 the FPGA is running at 100MHz. Even with this compromised configuration, Registor still shows better performance than Linux Grep. By expanding bus width to 32B and raising 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 much higher clock rate and optimized I/O performance.

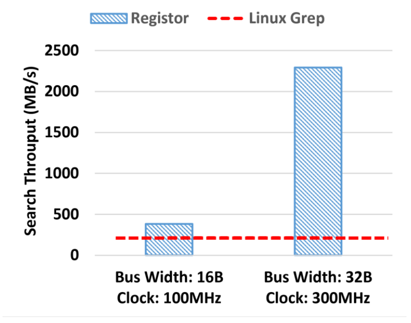

Figure 7: Throughput of Registor under different configurations compared to Linux Grep

Our next experiment is to measure throughput using UVM test (see Section 4.2). 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 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 8 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 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.

5.2 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 microbenchmark (also used in Section 5.1) and "ps" command in Linux system with Registor and Linux Grep, receptively. Figure 9 plots the CPU utilization over time. During the runtime, the average CPU utilization of Registor is 11.90% while that of Linux Grep is 70.09%, implying Registor consumes 82% less CPU resources then Linux Grep. This is because Registor offloads compute-intensive tasks to FPGA. The only functions that consumes CPU resources are the user library

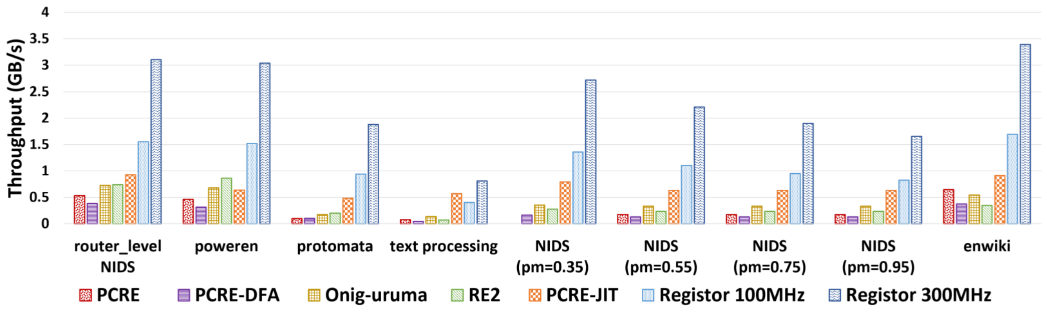

Figure 8: Throughput comparison among different search engines

and APIs that are simple and light weight. Therefore, Registor consumes almost no CPU clock cycles as compared to Linux Grep. The remaining CPU resources can be used by other applications.

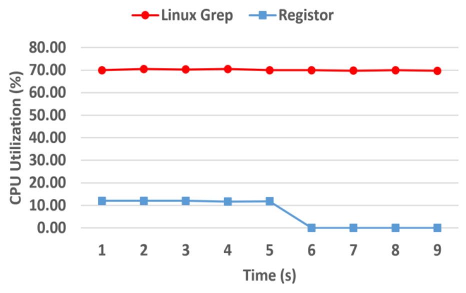

Figure 9: CPU utilization of Registor and Linux Grep

5.3 I/O Bus Utilization

To better illustrate reduction in I/O bus utilization of Registor over other software solutions, we measured data transfer ratio (a value between 0% to 100%), calculated by dividing the size of data transferred from SSD to host by file size. Since NVMe's read command can request up to 128kB data per I/O and the minimum size of data transfered to host is 4kB, the data transfer ratio in the best scenario is 4kB/128kB = 3.125%. For software solution, the data transfer ratio is a constant value of 100% because the whole file is loaded to host for further scanning.

Figure 10 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 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 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%, the minimum value of data transfer ratio in our design. It is obvious that Registor exhibits much less negative impacts on other applications in I/O bandwidth, a tiny fraction of 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, data transfer ratio of 3.125% means that a 100K IOPS SSD with Registor enabled provides regex search applications with equivalent 3.175 million IOPS of SSD without Registor.

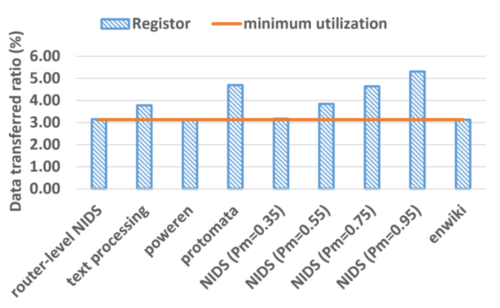

Figure 10: Data transfer ratio (the size of data transferred from SSD to host divided by file size)

To further illustrate how Registor alleviates I/O bottleneck problem, we analyze *effective throughput* and *required throughput* by applications. The effective throughput is what Registor can offer to regex applications while the required throughput is a measure for the necessary I/O throughput in order for an application to achieve a desired effective throughput. Figure 11 plots the effective throughput and required throughput of Registor when it is implemented in 1GHz ASIC with internal bus width of 64B. It can be seen from this figure that, for all

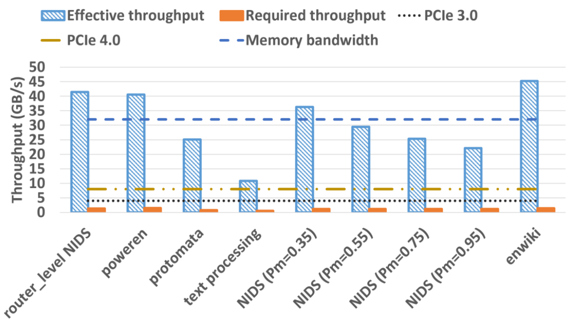

Figure 11: The effective throughput and required throughput of Registor

Table 5: Summary of power consumption

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

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

5.4 Area and Power Consumption

In our current implementation, the SSD prototype uses 78% of LUTs and logic cells in Xilinx 9p FPGA board including 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 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 5, Registor hardware consumes 0.119W which is only a tiny fraction (less then 1%) of the total power consumption of the SSD prototype.

6 RELATED WORK

Regex search acceleration: Extensive research has been reported in accelerating regex search over the past decade. Some researchers take advantage of GPU [28, 56], SIMD [11, 29, 39], and multi-core architectures [35] while others focus on FPGA/ASIC based solutions [10, 17, 20, 22, 25, 38, 42, 43, 46, 55].

Early work discusses regex search in FPGA/ASIC by mapping non-deterministic finite automata (NFA) [42, 55] and deterministic finite automata (DFA) [10, 17, 25] to programmable logic. Recent study [20] by Vaibhav et al. proposed HARE, extended from [46], compiles regex into subexpressions and runs bit-split automata [45] 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 [30, 50]. Another generalized ASIC-based accelerator is Micron's Automata Processor (AP) [13]. It is capable of processing large NFA whose state transitions are stored in bit vectors and being executed via customizable routing matrix. The most recent work by Subramaniyan and Das [44] 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. propose Unified Automata Processer (UAP) architecture that features a programmable engine for finite automata (FA) and supports a wide range of FA models [15].

The above mentioned work achieves encouraging progress in accelerating regex search and most of them are designed for network intrusion detection or in-memory pattern matching. With different optimization objectives and different architecture level from above mentioned work, our proposed Registor works in different manners: (1) It is located inside SSD for the purpose of eliminating I/O bottlenecks on processing large amount of data stored in storage; (2) It is capable of extracting file semantics from data blocks and providing host with contextual information of search results.

Near data processing (NDP): The benefits of NDP have been demonstrated by many researchers at different levels of system hierarchy such as in-memory computing [4, 19, 26, 54] and processing in storage [8, 21, 23, 41, 48, 49].

Some existing work exploits the computational power of SSD controller by offloading the data-intensive tasks to the embedded cores [48, 49]. Tiwari and et al. [48] present a detailed energy and performance models for data analysis using embedded cores in SSD. Tseng et al [49] 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 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 an FPGA and have it sit in the internal data path between NAND flash to host interface to achieve on-the-fly regex search.

Some other groups integrate FPGA/ASIC in SSD for computing purpose [21, 23, 41]. Willow SSD by Seshadri et al. [41] allows users to implement customized features to support particular applications by deploying several RISC processors in SSD, using a BEE3 FPGA-based prototype [33]. Gu et al [21] propose Biscuit, an NDP framework, that includes a hardware pattern matcher for string search in each channels of NAND chips. BlueDBM [23] applies in-storage processing to big data analytics which integrates Morris-Pratt (MP) string search engine in SSD [34]. Different from the above mentioned work, Registor eliminates I/O bottlenecks in unstructured data processing that requires regex search, which has higher computational complexity than string search.

7 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 transfered from NAND flash to host. The search engine achieves high processing rate that matches the speed of internal bus in SSD by fully exploiting the parallelism in FPGA. The deep pipeline structure of Registor consists of file semantics extractor, matching candidates finder, regex matching units, and results organizer. Furthermore, we developed a user library to facilitate the upper-layer applications to take advantage of the search engine. In order 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 I/O bottleneck. Our future work includes adopting more advanced automata designs in our Registor and further optimization of I/O path inside our SSD prototype.

8 ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments that helped greatly in improving the quality of the paper. The authors are thankful to Shuqun Xie, Qingchun Zhu, Ying Yang, Archie Wu, Pan Qin for providing guidance to this work. This research is supported in part by the 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. It is also partly supported by a research contract between URI and Shenzhen Dapu Microelectronics Co., Ltd, Shenzhen Peacock Plan (KQTD2015091716453118).

REFERENCES

[1] [n. d.]. Performance comparison of regular expression engines. http://sljit.sourceforge.net/regex_perf.html. ([n. d.]). Accessed April 4, 2017.
[2] [n. d.]. Regular expression library. http://regexlib.com/. ([n. d.]). Accessed April 4, 2017.
[3] [n. d.]. Snort - Network Intrusion Detection and Prevention System. https://www.snort.org/. ([n. d.]). Accessed April 4, 2017.
[4] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 105–117.
[5] 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.
[6] Amazon. 2018. Amazon S3. (2018). https://aws.amazon.com/s3/
[7] apache. 2018. Lucene. (2018). https://lucene.apache.org/
[8] 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, 56–61.
[9] Michela Becchi, Mark Franklin, and Patrick Crowley. 2008. A workload for evaluating deep packet inspection architectures. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE, 79–89.
[10] 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.
[11] 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, 139–150.
[12] Russ Cox. 2009. Regular expression matching: the virtual machine approach. *URL: http://swtch. com/rsc/regexp/regexp2. html* (2009).
[13] 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.
[14] NVM Express. 2018. NVM Express Revision 1.3a October 24, 2017. (2018). http://nvmexpress.org/wp-content/uploads/NVM-Express-1_3a-20171024_ratified.pdf
[15] Yuanwei Fang, Tung T Hoang, Michela Becchi, and Andrew A Chien. 2015. Fast support for unstructured data processing: The unified automata processor. In *Microarchitecture (MICRO), 2015 48th Annual IEEE/ACM International Symposium on*. IEEE, 533–545.

[16] 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, 55–68.

[17] 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.

[18] 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.

[19] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. 2015. Practical near-data processing for in-memory analytics frameworks. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*. IEEE, 113–124.

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

[21] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang K-won, Chanho Yoon, Sangyeun Cho, et al. 2016. Biscuit: A framework for near-data processing of big data workloads. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 153–165.

[22] Titan IC. 2018. Hyperion F1 10G Regex File Scan. (2018). http://titan-ic.com/products/hyperion-f1-10g-regex-file-scan

[23] 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 *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 1–13.

[24] Avita Katal, Mohammad Wazid, and RH Goudar. 2013. Big data: issues, challenges, tools and good practices. In *Contemporary Computing (IC3), 2013 Sixth International Conference on*. IEEE, 404–409.

[25] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. 2006. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *ACM SIGCOMM Computer Communication Review*, Vol. 36. ACM, 339–350.

[26] 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, 475–476.

[27] John Levine. 2009. *Flex & Bison: Text Processing Tools*. " O'Reilly Media, Inc.".

[28] 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 Trans. Comput.* 62, 10 (2013), 1906–1916.

[29] Dan Lin, Nigel Medforth, Kenneth S Herdy, Arrvindh Shriraman, and Rob Cameron. 2012. Parabix: Boosting the efficiency of text processing on commodity processors. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*. IEEE, 1–12.

[30] 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 Computer Society, 461–472.

[31] Micron. 2018. MT29F16G08ABCCBH1-10ITZ. (2018). https://www.micron.com/parts/nand-flash/mass-storage/mt29f16g08abccbh1-10itz?pc=

[32] Micron. 2018. MT29F2T08CUHBBM4-3R. (2018). https://www.micron.com/parts/nand-flash/3d-nand/mt29f2t08cuhbbm4-3r?pc=

[33] Microsoft. 2018. BEE3 Established: February 26, 2008. (2018). https://www.microsoft.com/en-us/research/project/bee3/

[34] James Morris Jr and Vaughan Pratt. 1970. *A linear pattern-matching algorithm.*

[35] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. 2014. Data-parallel finite-state machines. In *ACM SIGARCH Computer Architecture News*, Vol. 42. ACM, 529–542.

[36] PCI-SIG. 2018. Frequently Asked Questions PCI Express - 3.0. (2018). https://pcisig.com/faq?field_category_value%5B%5D=pci_express_3.0&keys=

[37] PCI-SIG. 2018. Frequently Asked Questions PCI Express - 4.0. (2018). https://pcisig.com/faq?field_category_value%5B%5D=pci_express_4.0&keys=

[38] Indranil Roy, Ankit Srivastava, Marziyeh Nourian, Michela Becchi, and Srinivas Aluru. 2016. High performance pattern matching using the automata processor. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 1123–1132.

[39] Valentina Salapura, Tejas Karkhanis, Priya Nagpurkar, and Jose Moreira. 2012. Accelerating business analytics applications. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*. IEEE, 1–10.

[40] 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.

[41] Sudharsan Seshadri, Mark Gahagan, Meenakshi Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A User-Programmable SSD.. In *OSDI*. 67–80.

[42] Reetinder Sidhu and Viktor K Prasanna. 2001. Fast regular expression matching using FPGAs. In *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*. IEEE, 227–238.

[43] 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, 403–415.

[44] Arun Subramaniyan and Reetuparna Das. 2017. Parallel Automata Processor. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 600–612.

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

[46] Prateek Tandon, Faissal M Sleiman, Michael J Cafarella, and Thomas F Wenisch. 2016. Hawk: Hardware support for unstructured log processing. In *Data Engineering (ICDE),*

2016 IEEE 32nd International Conference on. IEEE, 469–480.

[47] Ken Thompson. 1968. Programming techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (1968), 419–422.

[48] 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 *FAST*. 119–132.

[49] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. 2016. Morpheus: creating application objects efficiently for heterogeneous computing. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 53–65.

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

[51] Jack Wadden, Vinh Dang, Nathan Brunelle, Tommy Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, Chunkun Bo, Gabriel Robins, Mircea Stan, et al. 2016. ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on*. IEEE, 1–12.

[52] 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).

[53] www.gutenberg.org. 2018. The Entire Project Gutenberg Works of Mark Twain by Mark Twain. (2018). http://www.gutenberg.org/ebooks/3200?msg=welcome_stranger

[54] 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, 2.

[55] 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, 30–39.

[56] 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, 18.