MithriLog: Near-Storage Accelerator for High-Performance Log Analytics

Seongyoung Kang
seongyk3@uci.edu
University of California, Irvine
USA

Jiyoung An
jiyouna2@uci.edu
University of California, Irvine
USA

Jinpyo Kim
jkim@vmware.com
VMware
USA

Sang-Woo Jun
swjun@ics.uci.edu
University of California, Irvine
USA

ABSTRACT

This paper presents MithriLog, a log analytics platform with near-storage accelerators for high-performance, cost- and power-efficient unstructured log processing. MithriLog offloads log analytics queries to an efficient near-storage FPGA implementation of a token querying engine, which can take advantage of the high internal bandwidth of storage devices within the available chip resource limitations. This engine is flexible enough to handle complex queries including template search based on user-defined tree-based template libraries, as well as concurrent execution of multiple queries. MithriLog also uses a log-optimized version of a simple, high-throughput compression algorithm in order to further improve the effective bandwidth of backing storage.

Evaluated with complex search queries on large real-world log datasets, MithriLog achieves an order of magnitude higher performance over software systems, even against more expensive machines with enough DRAM to stage the entire dataset. Furthermore, MithriLog delivers constant performance regardless of query complexity, resulting in further improved performance benefits with more complex queries. By replacing costly DRAM with storage and power-hungry CPU threads with FPGAs, MithriLog dramatically improves the cost-effectiveness and accessibility of log analytics.

ACM Reference Format:
Seongyoung Kang, Jiyoung An, Jinpyo Kim, and Sang-Woo Jun. 2021. MithriLog: Near-Storage Accelerator for High-Performance Log Analytics. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3466752.3480108

1 INTRODUCTION

Collecting and analyzing log data in both cloud and edge deployments is a critical tool for large-scale system management including monitoring and insight [4, 47, 52, 82], as well as detecting abnormal behavior and security issues [12, 18, 34, 63, 79], and even program verification [1]. As such, logs are important components in many computer systems, especially in enterprise deployments [7, 70]. As hardware and software systems become exponentially more complex, the rate and volume of log collection is also increasing rapidly [19]. Due to their sheer size and rate of collection, typical use pattern of logs involves firstly storing everything to the storage and then running queries [66, 68].

One of the biggest hurdles of effective log analytics is the typically unstructured nature of logs [12, 19, 46, 63, 70]. Logs of interest are typically generated by wide range of independent software and hardware sources [46], via various tools spanning from `prinf()` and `System.out()` to specialized logging libraries [9, 14]. The unstructured nature of log text makes it a bad fit for regularly structured relational databases and its optimizations [12, 46, 63]. The algorithmic complexity of processing unstructured text becomes a primary bottleneck in scaling the performance of log analytics [53, 54, 70], making it a natural target for improvement.

Figure 1 shows a subset of such a real-world log file from a supercomputing environment [47, 48]. Unlike relational database tables with pre-determined schemas, each line in an unstructured log can have its own format and structure, as they are often generated by different programs using different *templates* for encoding relevant information into human-readable text. This renders structured table-based approaches ineffective, and the first step in automated log analytics is typically parsing unstructured log text to determine which template each log line belongs to. The example in Figure 1 shows three such templates, with underlines for key words — or key *terms* or *tokens* — used for template identification. For example, a line with terms `RAS`, `KERNEL`, and `INFO`, but not `FATAL` can be determined as belonging to template 2. In the rest of this paper, *term* and *token* are used interchangeably to refer to a textual word which is separated by delimiters. A library of templates for parsing purposes can be hand-constructed by a programmer or by an algorithm, and can contain hundreds of templates [41, 84]. The example templates have been automatically extracted by a template tree-based algorithm, which we will describe in more depth in Section 4.3.

In this paper, we present the design and evaluation of **MithriLog**, a log analytics system with near-storage hardware acceleration. Based on the observation that important workloads such as template identification works in *term* units instead of individual characters, MithriLog implements a hardware accelerator for complex *token-based* line-wise log filtering based on cuckoo hashing, instead of a

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10.
https://doi.org/10.1145/3466752.3480108

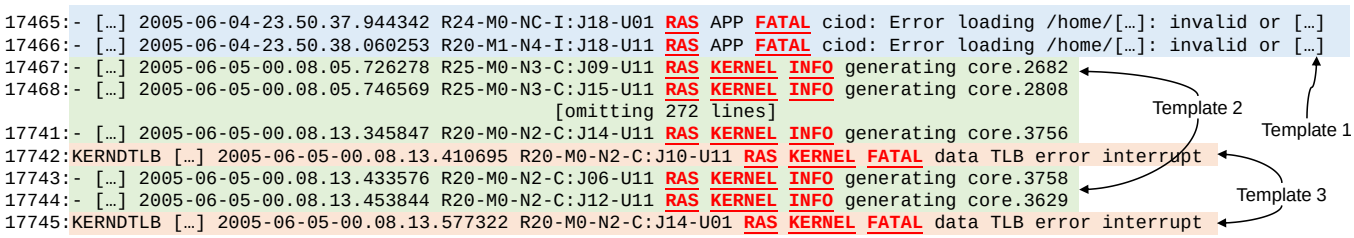


Figure 1: Example logs from the HPC4 dataset [47], with highlights for automatically identified keywords and templates [84].

more general-purpose regular expression matching. As a result, it can take advantage of the highly parallel hash access performance of hardware accelerators to reach almost 12 GB/s of throughput on an FPGA prototype.

Since such a high bandwidth far exceeds the performance of the PCIe-attached NVMe storage on our prototype system, MithriLog introduces two options to improve the effective bandwidth of backing storage, resulting in balanced performance between system components. First, MithriLog is configured as a near-storage accelerator taking advantage of the relatively high internal bandwidth of PCIe-attached SSDs compared to the PCIe or network bandwidth [5, 27]. It also uses a log-optimized compression algorithm fast enough to further improve the effective bandwidth of storage, saturating accelerator performance using the same back-end storage bandwidth. We present a novel, hardware-optimized compression algorithm for this purpose, that trades compression efficiency for high performance and low hardware overhead.

The design goal of the filtering engine and compression algorithm was to enable efficient hardware implementation. As a result, the resulting system is able to achieve significantly higher performance per chip resource compared accelerators implementing general-purpose approaches designed for software implementation, such as regular expression parsers [68] or LZ4 [38] compression.

MithriLog also includes software support including an in-storage inverted index implementation tuned for accelerator performance, allowing it to run real-world queries and evaluate its realistic end-to-end performance against popular commercial systems such as Splunk [62]. We evaluate the performance of MithriLog on various queries for log discovery and iterative exploration on real-world logs collected from supercomputers [47], using hundreds of queries of variable complexity, algorithmically generated from the datasets using the FT-Tree template extraction method [84, 85]. Queries generated by FT-Tree are based on a log template library using a frequency tree, and we demonstrate that the token filter engine is flexible enough to support multiple such queries with hundreds of terms, executing concurrently at no performance loss. Comparing the performance of the token filter as well as end-to-end performance, MithriLog demonstrates over an order of magnitude performance improvements over off-the-shelf systems such as MonetDB and Splunk, while reducing the overall power budget of the system thanks to power-efficient acceleration.

While queries evaluated for this work focus on the exploration and discovery aspect of log analytics, more complex analytical operations such as principal component analysis [79] or clustering [36] can also be implemented to benefit from the fast data extraction capability of MithriLog.

This paper claims the following contributions:

- Design of a flexible, high-performance token filtering engine for unstructured data based on cuckoo hashing, and its evaluation on complex queries such as template-based queries.
- Design and evaluation of architectural methods of improving the effective bandwidth of storage, including the near-storage acceleration configuration and log-optimized compression accelerators.
- Detailed analysis of our approaches in the context of large unstructured log queries, demonstrating good end-to-end performance and power efficiency against commercial off-the-shelf log analytics systems.

The rest of this paper is organized as follows: Section 2 presents relevant background and related works, and Section 3 describes the overall system architecture of a complete MithriLog system. The next three sections describe the prominent components of MithriLog in detail. Section 4 presents the design of the log token filtering engine, and Section 5 presents our log- and FPGA-optimized compression algorithm based on LZRW1. Section 6 describes the in-storage inverted index design. Section 7 presents the performance, cost, and power-efficiency evaluations of MithriLog. We conclude with future work in Section 8.

2 RELATED WORK

2.1 Log Analytics

2.1.1 Log Analytics Systems. Log analytics is typically considered not a good fit with conventional relational database management systems (RDBMS) for many reasons, including the unstructured nature of logs [19, 46, 70, 73], as well as the high overhead of ACID compliance, which is of less importance for log analytics [40, 72, 81]. As a result, log analytics software is designed specifically to handle unstructured logs.

Multiple off-the-shelf products support various aspects of log collection, archiving, and analytics. Examples include Splunk [62], Facebook LogDevice [7], Apache Kafka [10], Datadog [11], and Elasticsearch [83]. Most of these tools support collecting, parsing, indexing, querying, as well as some aggregation and visualization functionality. These systems typically implement log-optimized data structures such as inverted indices [26, 62, 83].

For use cases that require complex, application-specific analytics functions that are outside the scope of these tools, many

custom log analytics systems have been built and deployed. Popular platforms for system development include various NoSQL databases [39, 52, 81], as well as distributed computing frameworks such as Hadoop [67] and Spark [52, 72].

2.1.2 Hardware Accelerators. Application-specific hardware accelerators, especially those using easily deployable Field-Programmable Gate Arrays (FPGAs), are under active research due to their potential to reach superior performance and power efficiency compared to software running on general-purpose processors [3, 50, 64, 71].

HAWK and HARE are a family of accelerators for unstructured log analytics [13, 68], which use a parallel implementation of finite state machines to match string tokens at fast, deterministic rate. HAWK and HARE's projected ASIC implementation at 1 GHz can achieve a deterministic throughput of 32 GB/s for token-based search into unstructured logs. However, due to their complexity the FPGA implementation had to reduce parallelism to fit on chip, resulting in only 400 MB/s while consuming 12% of an Intel Arria V FPGA. It also did not take indexing structures into account, restricting itself to full scans into the dataset.

A great amount of research also focuses on FPGA implementation of regular expressions, a key component in many unstructured log analytics methods [23, 58, 80]. Such accelerators implemented on CPU-FPGA hybrid platforms have been integrated into DBMSs to achieve over an order of magnitude performance improvement in query performance [59], demonstrating string matching is a prominent performance bottleneck of database systems.

Other accelerator designs exploring similar computation patterns include pattern recognition from event streams [75] which also used an FPGA implementation of finite state machines to achieve network speeds. In the realm of semi-structured data, FPGAs have been used to parse and filter XML data using regular expressions [43], tree matching [45], and more.

2.1.3 Log Template Extraction. An important class of log analytics is template extraction, where patterns in unstructured logs are extracted in order to recognize log lines that are generated from the same template. Once a template library is extracted, log exploration and analytics can be reduced to focusing on their variable parameters. There are many proposed methods on template extraction. Some methods use a prefix tree based method, which extracts more common terms and organizes them into a tree where terms appearing earlier in a line are closer to the root [6, 15, 17]. Some methods use the frequency tree method, where positions of terms in each line are ignored, and terms which globally occur more commonly are placed closer to the root of the parse tree [16, 84, 85]. Other methods have used genetic algorithms to discover common patterns [42]. Figure 1 highlights the automatically extracted templates and their keywords, discovered by the frequency tree-based FT-tree algorithm [84, 85].

2.1.4 Log Compression. Compressing logs is also an important topic due to the sheer size of logs. While general-purpose algorithms such as DEFLATE [49] have also been used, many log-specific algorithms have also been proposed. Many such algorithms take advantage of the repeated patterns across log lines, and improve compression by aligning at line boundaries [55, 60], discovering consistent patterns across lines [37], and clustering lines of similar structure [8]. The existence of repeated patterns across log lines, even across different templates, can be seen in Figure 1. Parts of time stamps and template messages can be efficiently compressed with this insight. For regularly structured data, column-wise compression also has shown good performance [35].

2.2 Near-Storage Acceleration

For large-volume data typically stored in an array of secondary storage, placing computation on the storage itself has shown to be beneficial. Near-storage processing can reduce storage access latency and data movement overhead [30], as well as take advantage of the relatively high internal bandwidth compared to the economically provisioned communications link [5, 27]. As storage performance scaling outpaces interconnect and software performance [29], near-storage acceleration that removes these overheads is expected to continuously become more important.

The new system configuration is also a good opportunity to introduce power-efficient hardware accelerators [5, 24, 33] along with the new programming model. The near-storage paradigm has demonstrated performance, cost, and power efficiency benefits across many application domains, including machine learning [28], graph analytics [25, 32], and relational database queries [65, 78]. Many near-storage computation platforms exist, including the commercially released SmartSSD from Samsung [33], MIT's BlueDBM prototyping platform [24], and others [69].

3 SYSTEM ARCHITECTURE

Figure 2 shows the overall architecture of the MithriLog system. The system configuration consists of an off-the-shelf host server machine augmented with the MithriLog storage device plugged into PCIe. The storage device is a Solid-state Storage Device (SSD) equipped with decompressors and text filtering engines programmed onto a near-storage accelerator, taking advantage of the typically higher internal bandwidth of the storage device compared to the PCIe link. We assume the storage medium is NAND-flash, since it is the most prominent large-scale SSD technology. However, the design of MithriLog is still applicable to other high-performance solid-state storage technologies. Software on the host consists of the accelerator-aware log analytics software, which also includes an index structure for effective use of storage and computation resources. The inverted index is designed to minimize host memory usage during ingestion while still maintaining maximum storage performance utilization.

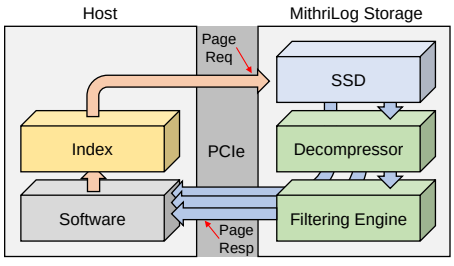


Figure 2: System architecture with MithriLog.

When querying, host software first sends configuration commands to the accelerator to program it with the tokens of interest, if any, and what to do with them. Once the accelerator is configured, the software can send page read requests to the storage device normally. For each query, the software can configure the decompressor and filter engine pipeline to either forward the raw page directly to host without doing any special processing, forward the page after decompressing, or after decompressing and also passing through the filtering engine. The resulting number of pages returned to host software may differ from the originally requested number, and the software is aware of this. The resulting stream of pages consisting of filtered text can be further processed by the host software to perform either complex analytics, or to simply display for the user to explore.

The filtering engine uses multiple pipelines of cuckoo hashing-based filters to implement wire-speed filtering of complex boolean combinations of tokens, within the chip resource limitations. The abstraction is flexible enough to program not only straightforward filtering based on token presence, but also more complex queries based on log templates. Many design decisions within the accelerator implementation itself, such as datapath width, are also driven by performance per chip resource usage.

The following sections present details about the design and implementations of the major system components of MithriLog. More details about the filtering engine are given in Section 4. The low-overhead, high-performance log-optimized compression algorithm and its accelerator is described in Section 5, and the in-storage index structure in Section 6.

4 TOKEN FILTERING ENGINE

The heart of the MithriLog system is the near-storage accelerator for filtering tokens in unstructured text. A MithriLog storage instantiates multiple filter pipelines according to performance goals and resource availability. For example, our FPGA prototype instantiates four pipelines to achieve near wire-speed, almost 12 GB/s on our hardware platform. More details regarding this design decision can be found in Section 7.2.

The token filtering engine compares unstructured log text line-by-line against a complex query consisting of boolean combinations of up to hundreds of terms. The token filtering engine supports queries expressed in the form of unions (the *union set*, $\cup$) of one or more *intersection sets* ($\cap$) each consisting of one or more tokens, where each term can also be negated ($\neg$). An example of such a query is shown in Equation 1, which is a single union set composed of two intersection sets. We will show that this is a flexible enough representation for many important applications including template-based filtering. This format can also be used to either encode one complex query, or to evaluate multiple queries in parallel by joining them with unions ($\cup$).

$$(\neg A \cap B \cap C) \cup (\neg D \cap \neg E \cap F \cap G) \quad (1)$$

Figure 3 describes the internal architecture of a filter pipeline, which in turn includes multiple sub-modules working in parallel, connected over wide datapaths. The datapath width is 128 bits in our prototype implementation. This is a balance between higher

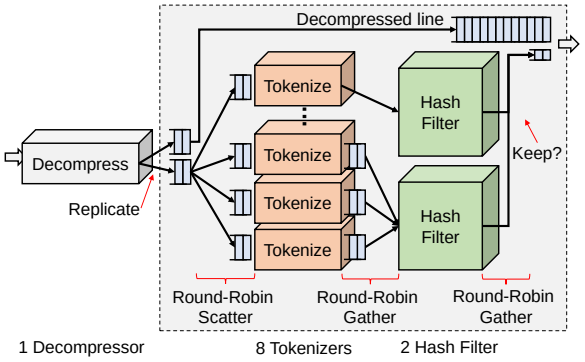


Figure 3: A token filter pipeline consists of multiple parallel sub-modules to achieve wire-speed.

chip resource requirements of a wide bus, and the token length distribution statistics presented in Section 7.4. The figure also depicts a decompressor module which will be presented in Section 5.

The token filter consists of two stages, the tokenizers, and hash filters. The input to the system is first scattered round-robin to an array of tokenizers, in textual line units. This process is made very simple by making the decompressor emit line-aligned words, as described in Section 5. The tokenized output is again collected by the hash filters in the same, round-robin order to ensure correct ordering. Hash filters evaluate the tokenized terms against queries encoded as cuckoo hashes, and emit a stream of boolean values on whether each line should be kept, or filtered out. The following subsections describe these components in more detail.

4.1 Tokenizer Design

The filter pipeline uses an array of simple tokenizers operating in parallel. After design-space exploration we designed our tokenizers to ingest two bytes per cycle, as it showed better performance per chip resource usage compared to designs processing one or four bytes per cycle, due to clock speed and timing issues. Our prototype instantiates eight such tokenizers per filter pipeline to achieve wire-speed on the 128-bit (16 byte) datapath. Log data is scattered across tokenizers line-by-line in a round-robin manner and gathered in the same order, assuring in-order processing at the hash filter.

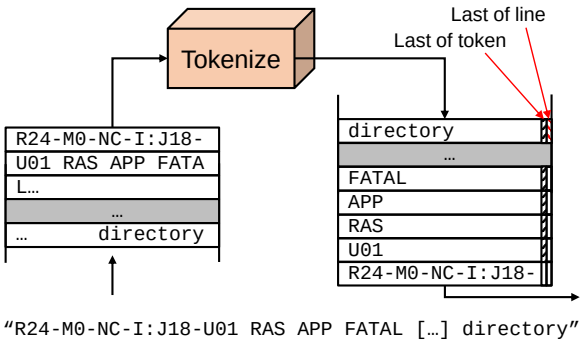


Figure 4: The tokenizer emits each token aligned to the datapath, along with two flags.

Figure 4 shows an example input and output for tokenizing a log line. The input is simply each log line streamed over multiple cycles, and the tokenizer emits a stream of tokens aligned to the datapath. Each output token is tagged with two single-bit flags. One tells the downstream hash filter if this token is the last of the current token, and is used if a token is larger than the datapath of 16 bytes and must be sent over multiple cycles. The other flag is set if this token is the last of the current line.

Emitted tokens are aligned to the datapath, and tokens that have less data than the datapath are padded with zero bytes. This means the tokenizer output may suffer data amplification, where the data rate of tokenized output may be larger than the original data due to padding bytes. We will show in Section 7.4 that there is typically a factor of two data amplification, driving our design decision of having two downstream hash filter modules per pipeline even though each is capable of sustaining wire-speed. As a result, each filter pipeline is capable of typically achieving wire-speed despite the data amplification, processing 16 bytes of unstructured text per cycle. This is 3.2 GB/s of throughput per pipeline on our prototype clocked at 200 MHz.

4.2 Hash Filter Design

4.2.1 Cuckoo Hashing. MithriLog uses cuckoo hashes [51] to evaluate tokens against multiple queries at wire-speed. Queries are given to the accelerator in the form of hash tables, into which the software has encoded one or more queries. The query also includes a number of bitmaps, which is described in Section 4.2.3.

Cuckoo hashing, a variant of hash tables, resolves collisions by using two hash functions instead of one. A token can be inserted into any of the two hash function results, if one or more slots are empty. If both locations are occupied, the value stored in one slot is evicted to store the new value. The evicted value is moved to its alternate location, in turn evicting the already stored value, if any. Insertion can fail if this chain falls into an infinite loop, meaning such a query cannot be offloaded to our accelerator and must fall back to conventional software processing. However, hash table construction should succeed for typical queries since cuckoo hashes are known to typically succeed with load factor of 0.5 or below [31, 56]. We over-provision our hash table resources for this purpose.

Compared to more general approaches such as regular expression matching, a token-based filter requires much less chip resources per unit bandwidth, which is critical for our goal of extracting the best performance from storage devices. We present a detailed performance evaluation in Section 7.4. While token-based filtering cannot handle some queries such as those including substring matches, it still supports enough important query classes to be useful.

The benefit of cuckoo hashes for encoding tokens is twofold: First, hash lookups can be done in a single cycle using on-chip Block RAM, unlike an array or tree. Second, because cuckoo hashing statistically can achieve placement if the number of values is less than half the hash table capacity [31], it is a more compact method than a standard hash table which can fail placement at the first hash collision.

4.2.2 Hash Table Structure. Figure 5 describes the structure of a cuckoo hash table used by MithriLog. The hash table in our prototype implementation has 256 rows, and we demonstrate it is large enough to support realistic queries. However, it is trivial to make both it much larger using on-chip Block RAM resources.

Each table entry stores a token, an optional offset into the overflow table, and an array of flags. The overflow offset is used if the token length is longer than the statically sized slot in the hash table. In our prototype, each hash entry has 16 bytes provisioned for tokens, which is the same as the datapath width. If the length of a token is longer than these 16 bytes, the remainder of the token is stored in contiguous entries in the overflow table, which the overflow offset points to. The overflow table entries are also flagged whether each entry is the last entry for this token.

The flag array consists of multiple pairs of *valid* and *negative* flags. The valid flag specifies whether it is a valid entry in the cuckoo hash, and the negative flag specifies whether this query term is negative (¬), meaning this token should not exist in the log line. Each pair of flags is used to encode whether a token exists in each of the N intersection sets (∩). As a result, the number of intersection sets MithriLog can support in a query is limited by the number of flag pairs in the hash table. Our prototype provisions eight flag pairs, supporting a union set (∪) of up to eight intersection sets.

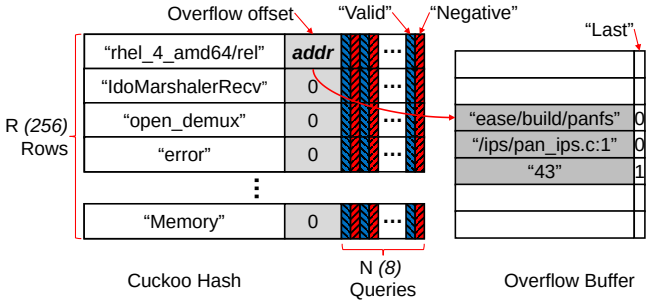


Figure 5: MithriLog achieves wire-speed token matching using a cuckoo hash, coupled with an overflow buffer for long tokens.

4.2.3 Query Filtering Process. During querying, each token from the tokenizer is hashed using the two hash functions in order to access the hash tables, as well as the overflow table if the query is longer than the datapath. The two hash entries are compared against the input token, where at most one token will match. If no entry matches, this input token can be ignored. If a match exists, the flag array of the matched entry is used to update an array of bitmaps used to keep track of whether all query terms in each intersection set exist in the log line. This process is shown in more detail in Figure 6, using a simple configuration with only three flag pairs specifying three intersection sets. The query also includes three bitmaps of size 256, which will be described in more detail below.

For each log line, the engine keeps N bitmaps of width R. For example, when using a hash table configuration with 256 entries and three flag pairs, the engine will keep track of three bitmaps,

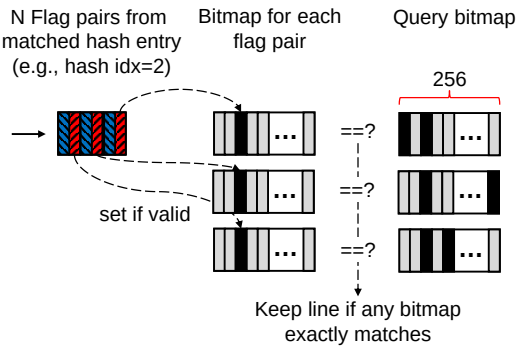


Figure 6: One bitmap is used to keep track of whether each intersection set in the query is satisfied. Example shows three intersection sets in a hash table with 256 rows.

each with 256 bits. Each bitmap corresponds to an intersection set, and each bit in the bitmap corresponds to an entry in the hash table.

Each input flag pair is first checked whether the *valid* flag is set. If not, this token is of no interest to the query and is ignored. If the *valid* flag is set, and the *negative* flag is also set, this line violates the negative term condition, and the corresponding intersection set is marked as not satisfied. If a *valid* flag is set and the *negative* bit is *not* set, then this token is part of the query. The index of this token in the hash table is used to update the bitmap of the corresponding intersection set. For example, if the token is from entry number 2 in the hash table, and the first flag pair is valid but not negative, bit number 2 of the first bitmap is set to 1.

After all tokens of a line has been processed, each line can be filtered out without forwarding to software if one of two conditions are met: First, all intersection sets violated the negative term condition. Or, none of the bitmaps exactly match their corresponding query bitmap, meaning one or more terms of the intersection set does not exist in the input line. The query bitmap has bits set at all cuckoo hash indices with both *valid* bit set and *negative* bit unset, representing the positive terms in the intersection set that must be satisfied. If any of the bitmaps is an exact match while not violating the negative term condition, the log satisfies the query and can be forwarded to software.

4.3 Use on Tree-Based Template Filtering

Our cuckoo hash-based filtering is flexible enough to support queries beyond simple token presence, such as log filtering based on a frequency tree-based template library. Specifically, we target log templates based on the Frequent Pattern Tree (FP-Tree) [16], where tokens that occur more frequently in the dataset are located closer to the root node. Figure 7 shows a small example parse tree, where the global frequency of token occurrence in descending order is `A,B,C,D,E`.

Normally, determining whether a line of tokens belongs to a template requires sorting the tokens according to global frequency, and then traversing the parse tree, each of which is an expensive operation. Instead, since we are targeting log search and exploration, we can map multiple templates of interest into our *unions of intersections* query format. For example, traversing the tree for template 1 involves visiting `A`, and then `B`, but not visiting its sibling, `C`. However, because `C` has lower frequency than the leaf node `B`, we do not need to explicitly check for ¬`C`. This can be expressed with the boolean formula $(A \cap B)$. Template 3 can similarly be mapped to $((A \cap C \cap \neg B) \cap D \cap E)$. Since neither of these two formulas involves using a union ($\cup$) operator, we can join both queries into the following single, offloadable query:$(A \cap B) \cup ((A \cap C \cap \neg B) \cap D \cap E)$. This way, MithriLog can support querying up to N templates at once, as long as the total tokens involved fit into the R slots of the hash table.

We note that the engine can also trivially support not only frequency tree-based templates, but **also prefix tree-based templates** where tokens appearing earlier in a line appear closer to the root. To support prefix trees, a small field is added to the hash table entry specifying the column each token should appear at, and tokenizer modified to also emit an increasing column counter per token. This does not change the performance datapath at all, still supporting near wire-speed throughput.

5 LOG-OPTIMIZED COMPRESSION

MithriLog uses a high-performance hardware implementation of a log compression algorithm to improve the effective bandwidth of storage. In principle, any compression algorithm can be used, as long as it is performant and effective enough to improve the effective bandwidth of the storage beyond what is supported by the token filter pipelines. However, most existing algorithms, even hardware implementations of performance-optimized ones like LZ4, do not support the multi-GB/s bandwidth MithriLog requires. To remedy this, we have designed a log- and hardware-optimized compression algorithm which trades a small amount of compression efficiency for superior performance per chip resource utilization. We justify our design with a detailed comparison of performance and resource efficiency between hardware implementations of viable compression algorithms in Section 7.3.

We call our algorithm **LZAH**, or LZ Aligned Header. We modify the simple, high-performance LZRW1 compression algorithm [74] for high performance and low chip resource utilization, while maintaining effective compression of logs. Much like LZRW1, LZAH uses a hash table to discover recent occurrence of each word. If a recent occurrence is discovered, a single-bit header and the table index are emitted. If not, a single-bit header and the literal word are emitted. On this basis, LZAH introduces two new characteristics to facilitate efficient, high-performance compression accelerators: (1) most of the algorithm is word-aligned, and (2) it groups multiple

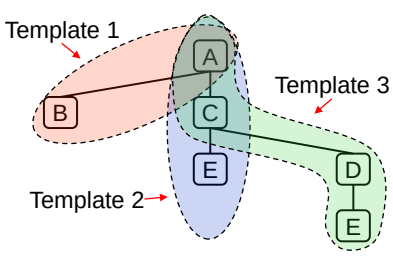


Figure 7: Three templates in an example frequent template tree.

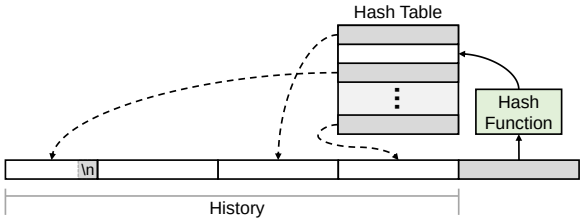


Figure 8: LZAH moves the window forward in fixed intervals unless a newline is found.

header-payload pairs into chunks. The performance and efficiency evaluations of LZAH and the hardware decompressor implementation are presented in Section 7.3.

First, LZAH removes the requirement of costly variable-amount shifters in hardware by moving a fixed, word-size window across the input stream in fixed-size, word-aligned steps. As this word size dictates the amount of data processed per hardware clock cycle, our current implementation uses a wide, 16-byte word to match the width of the filter datapath. Figure 8 illustrates an overview of this process. However, moving the window in word-aligned steps instead of sub-words results in a significant drop in compression efficiency. LZAH reclaims some of this performance by specially treating the newline (\n) character, exploiting the fact that patterns in logs appear at similar positions in each line. When a newline character is encountered, the window moves to the character immediately after the newline character, instead of advancing in the same fixed-size unit. The current word is padded with zero bits after the newline for storing in the hash table, such that characters from the next line are not included.

LZAH achieves further efficiency by grouping multiple headers and corresponding payloads into larger chunks, and aligning the chunks at word boundaries. In our implementation, we group 128 header-payload pairs in each chunk to match the size of the collected header to the datapath width. Figure 9 shows an example LZAH-compressed file with 128 header-payload pairs grouped into chunks, as well as padding between chunks and at the end of the page. Since headers are aligned at word boundaries, the decoder can parse chunk headers without shifting. As a result, parsing payloads can be done with efficient, multy-cycle shifters without losing performance. Furthermore, each compressed data in each storage page can be decompressed independently by aligning chunks at page boundaries.

As a result, LZAH supports extremely simple and efficient decoder implementations, as seen in Figure 10. All input word are either header chunks, which is stored in shift registers, or payload

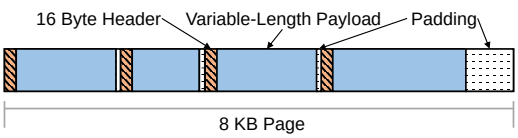


Figure 9: Compressed files are aligned both at word and page boundaries. Word size is 16 bytes in our prototype.

chunks, which is iteratively parsed by the multi-cycle shifter according to the header bits. The shifter is also aware of chunk sizes, and flushes remaining padding bits every time all payloads per chunk are parsed and emitted. According to the header bits, the shifted value can be interpreted as hash table indices, or as literal words by the downstream modules.

If the resulting word has a newline character, depending on the decompressor configuration it can either emit a zero-padded word to make the tokenizer's work easier, or another multi-cycle shifter can be used to shift down the subsequent word and remove the padding bits added during encoding.

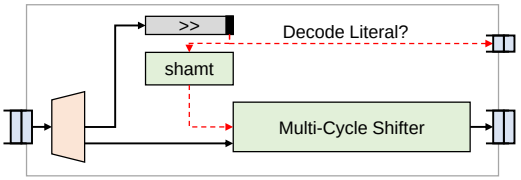


Figure 10: LZAH allows extremely efficient decoder implementations.

6 IN-STORAGE INVERTED INDEX

MithriLog also includes an efficient, storage-optimized inverted index for fast querying, which allows us to realistically evaluate complex queries on MithriLog against off-the-shelf log analytics platforms such as Splunk. Inverted index is one of the most popular methods to quickly locate information in unstructured text data, and is used by many prominent systems including Splunk and Elasticsearch [26, 62, 83]. Our in-storage inverted index emphasizes a simple design with a small memory footprint to maintain low host resource requirements, especially during ingest. It also focuses on extensibility and efficient use of storage bandwidth. Our in-memory footprint is around 256 MB during steady operation, and is fast enough to saturate even the near-storage accelerator bandwidth.

We also emphasize that while we demonstrate the efficiency and performance of MithriLog using our inverted index implementation, the core, near-storage accelerator platform can be coupled with any indexing strategy that accesses storage, including those used by other state-of-the-art systems, as long as the index can generate a stream of page addresses for the accelerator.

Figure 11 shows the overall structure of our inverted index implementation. The storage layout consists of index pages and data pages, where data pages store the log text in compressed format. The index implementation includes an in-memory hash table, as well as a linked list in storage. Each node in the linked list is an in-storage tree. Each in-memory hash table entry includes a small, fixed-size buffer of in-storage data page offsets that each token appears in. The in-storage index is used only for hash indices where the number of pages exceed the array capacity. In our prototype implementation, the per-index buffer size is 16 page addresses.

6.1 Using Linked List of Trees

The in-storage index consists of a linked list of shallow trees, each with height of two. This design aims to take advantage of the simplicity of a linked list-based index, while still saturating flash/SSD

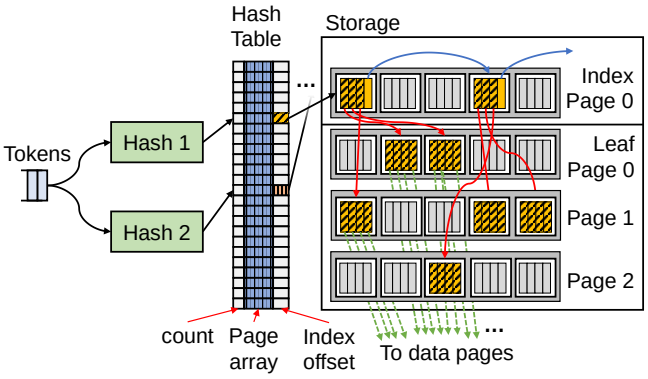


Figure 11: Inverted index of the MithriLog prototype uses two hash functions, each pointing to a linked list of tree nodes.

performance. While the latency of flash storage is superior to mechanical disks, traversing a linked list can still result in low performance due to its latency-bound nature. For example, a storage device with a reasonable 100μs latency can only visit 10,000 index nodes per second. To saturate a 4 GB/s PCIe SSD within these boundaries, each index node must store page indices of over 100 4 KB data pages. This simple approach can result in very large memory requirements. This is because during ingest, partially full index nodes must be buffered in memory until they are filled, at which point they can be flushed to storage. Maintaining write buffers for hundreds of elements for all hash table entries can quickly exceed multiple GBs of memory footprint.

Our design reduces the memory requirement by making the index nodes smaller, only 16 entries in the prototype, without sacrificing performance. The new in-storage index structure divides each linked list node into a small N-ary tree with a height of two. The root nodes of each tree form a linked list, and are stored in *index pages* in storage. The leaf nodes are stored in a pool of *leaf pages*. This layout is presented in Figure 11. Since each latency-bound linked list node visit results in many parallel leaf node accesses, this approach can achieve high performance by retrieving enough data page addresses per linked list node access, even when root and leaf node sizes are much smaller than the index node sizes required by the naive index list approach. In our prototype, the size of the root and leaf nodes are both 16 elements, resulting in 256 data page accesses per root node visit. Furthermore, this uniformly growing structure has very low overhead for append-only data patterns such as logs.

6.2 Using Two Hash Functions

In order to allow correct operation with small table sizes, we use the hash table as a probabilistic structure which does not keep track of the actual tokens hashed to each index. This avoids the issue of hash table entries running out with many tokens, but also means more than one token can be mapped to each index. While this still results in correct operations since unnecessary data will be filtered out by the filtering engine, it may incur performance overhead if a query token shares an index with another very common token.

To remedy this issue, the hash table is indexed by two hash functions. During index construction, the page addresses for each token is pushed into the index with the lesser number of total pages so far. Each hash table entry also includes a counter to keep track of this. During querying, entries for both hash tables are accessed. By spreading tokens with large occurrences across two indices, our experiences showed this method has a statistically lower number of pages accessed compared to using a single hash function.

6.3 Querying Tokens

Constructing the linked list is done by inserting new nodes into the head of the list, because updating older nodes already in storage will incur performance overhead. As a result, traversing the list starting from the hash table returns data page addresses in revere chronological order. During querying data page addresses are first read into memory and their order reversed. Thankfully this overhead is not very high, first because each element represents a page, and also because when two or more tokens are given as query predicates, the intersection of the resulting lists can first be calculated in read order, before reversing the likely much smaller list.

In order to support time-based queries, our design also supports snapshots. Whenever the number of leaf pages created since the last timestamp created exceeds a certain threshold, the whole in-memory hash table is flushed to storage, and a separate tree-based data structure keeps track of the index pages created during the flush, along with the time of the flush event. This allows coarse-grained time-based indexing into the log structure.

7 EVALUATION

We evaluate MithriLog in the context of complex search queries, and show that MithriLog is a desirable system for log analytics. The near-storage configuration coupled with efficient compression significantly improves the effective performance of the storage device, and the hardware implementation of the token-based query filter can make use of the improved performance by delivering an order of magnitude higher performance compared to software implementations. Coupled with a compact in-storage inverted index, MithriLog can not only improve performance, but also dramatically reduce the system resource requirements, as well as power consumption of high-performance log analytics.

7.1 Datasets and Benchmarks

We evaluate our system using the real-world HPC4 log dataset [47], which is the largest open system log dataset we could find. The log dataset consists of log files from four supercomputer deployments in Sandia National Labs and Lawrence Livermore National Labs. These logs have been used widely in the research community to evaluate analytics methods including system fault and alert detection [17, 48, 61, 86]. Table 1 describes the datasets of interest.

In order to evaluate a realistic workload untainted by human biases, we have used a machine-extracted set of queries generated by the FT-tree method [84, 85], which constructs a parse tree with the more commonly occurring terms near the root. FT-tree is a modern log parsing method, which has shown success in many log analytics use cases such as anomaly detecting using neural

	BGL2	Liberty2	Spirit2	Thunderbird
Lines (M.)	4.7	265.5	272.2	211.2
Size (GB)	0.7	30	38	30
Templates	93	197	241	125

Table 1: Logs span hundreds of millions of lines and dozens of GBs, queried using machine-extracted template queries.

networks [41]. We have used configuration parameters presented in the original paper to construct a parse tree with hundreds of templates for each dataset. The number of templates extracted from each dataset is also presented in Table 1.

We evaluate the performance of each system using all generated queries, as well as a library of batched queries. Batched queries include 100 random combinations of two query pairs connected using `OR`, as well as 16 random combinations of eight queries. The same set of randomly generated combinations were used for all systems tested.

7.2 Evaluation Platforms

We implemented a prototype MithriLog system using MIT's BlueDBM near-storage accelerator prototyping platform [24], configured for performance similar to a common NVMe storage device with a PCIe Gen3 ×4 link, specifically the Samsung SmartSSD platform [33]. We do not use the SmartSSD for this work since it separates the FPGA and SSD by a PCIe link, not allowing the FPGA to take advantage of the high internal bandwidth.

Figure 12 shows the platform configuration. It consists of four BlueDBM storage cards, each supporting up to 1.2 GB/s bandwidth, plugged into two Xilinx VC707 FPGA development boards connected over a 20 Gbps Aurora link, collectively emulating a single storage device. Only one FPGA board is connected to the host via a PCIe Gen2 ×8 link, delivering up to 3.1 GB/s of useful bandwidth via DMA.

Four MithriLog pipelines were instantiated across the two FPGAs. Each running at 200 MHz, the four pipelines were able to effectively make use of the backing storage bandwidth, as described in Section 7.4. Table 2 shows the chip resource utilization of a single MithriLog pipeline, as well as the total resource utilization of our prototype on a VC707 FPGA, including PCIe, flash controllers, and aurora links. The decompressor and tokenizer modules are small enough to allow wide replication for high performance.

Module	LUTs	RAMB36	RAMB18
1x Decompr.	4,245 (1.4%)	4 (0.4%)	0 (0%)
1x Tokenizer	1,134 (0.3%)	0 (0%)	0 (0%)
1x Filter	30,334 (10%)	10 (1%)	2 (0.1%)
1x Pipeline	61,698 (20%)	66 (6.4%)	18 (0.9%)
Total	225,793 (74%)	430 (41%)	43 (2%)

Table 2: Chip resource utilization of MithriLog on VC707.

This is a realistic configuration for an SSD device with near-storage acceleration. Each VC707 board can support up to two BlueDBM cards, and the two board configuration emulates a device with internal bandwidth higher than the PCIe link. The four

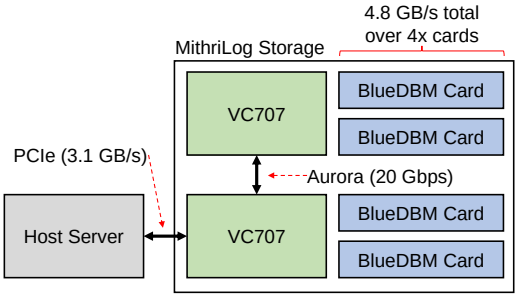


Figure 12: Four BlueDBM cards are used to emulate reasonable performance.

BlueDBM cards add up to 4.8 GB/s of bandwidth, 1.5× the effective PCIe bandwidth at 3.1 GB/s. This is a realistic performance differential considering the 1.8× internal to external difference published by Samsung [27]. The total capacity of the two last-generation Virtex 7 FPGAs are also similar to the single KU15P FPGA used in the Samsung SmartSSD device [33].

Performance comparisons were done using state-of-the-art database and log analytics platforms MonetDB [2, 21] and Splunk [62], running on a machine with 12-thread Intel i7-8700K CPU as well as four DDR-4 2133 DRAM cards adding up to 32 GB. This machine is also equipped with a RAID-0 array of two Samsung SSD 970 EVO Plus NVMe storage, adding up to 2 TB of capacity and 7 GB/s of measured peak storage bandwidth. Table 3 summarizes the relevant performance numbers. We emphasize that the *storage performance of the comparison system is much higher than MithriLog*, to err on the side of caution in comparing performance.

	MithriLog	Comparison
Computation	2x Virtex-7	i7-8700K
Storage Bandwidth	3.1 GB/s (PCIe) 4.8 GB/s (Internal)	7 GB/s

Table 3: Computation and storage of compared platforms.

7.3 Compression Evaluation

7.3.1 Accelerator Resource Efficiency. The LZAH compression algorithm enables significantly higher effective bandwidth of storage, via efficient compression and fast decompression performance. We used a modestly sized 16 KB hash table for compression, and achieved an average of 5.96× compression over the four datasets. As for throughput, our decompression accelerator has deterministic performance, always emitting a word of decompressed data per cycle. Running at 200 MHz, it consistently delivers 3.2 GB/s of uncompressed data regardless of compression efficiency.

Table 4 compares the resource efficiency of FPGA implementations of prominent performance-oriented compression algorithms, implemented on similar Xilinx FPGAs. One LZAH pipeline achieves higher throughput compared to all other algorithms, at a lower resource utilization (1,000 LUTs) compared to all but LZRW. LZAH achieves superior resource efficiency in terms of bandwidth per LUT. As MithriLog with LZAH already uses 70% of the available

Algorithm	GB/s	KLUT	GB/s/KLUT	Source
LZ4	1.68	35	0.048	[76]
LZRW	0.175	0.64	0.27	[20]
Snappy	1.72	35	0.049	[77]
LZAH	**3.2**	4	**0.8**	This

Table 4: The hardware-optimized LZAH algorithm achieves superior performance per chip resources (GB/s/KLUT).

chip resources, *no other algorithm presented can maintain our target wire-speed performance* within chip resource limitations.

7.3.2 Compression Efficiency. Table 5 compares the compression ratios of LZAH compared to LZRW1, LZ4, and Gzip. Compared to other algorithms, LZAH trades more compression effectiveness for better performance and less chip resource utilization, because its foremost goal is to improve the effective bandwidth of the storage device. It achieves its purpose, and provides high enough compression ratio and decompression performance to saturate the accelerator performance (shown in Section 7.4) at a small enough resource utilization level to leave space for performant filtering engine deployment, as seen in Table 2. It even achieves better compression efficiency compared to LZRW1 on some datasets.

7.4 Filtering Engine Performance

7.4.1 Accelerator Throughput Analysis. As discussed in Section 4, all sub-components of the filtering engine are replicated to support 16 bytes of useful data processed per cycle. The decompressor invariably emits 16 bytes per cycle, and the eight tokenizers ingest 16 bytes per cycle, each tokenizer ingesting two bytes per cycle. The only variable in performance is the percentage of tokens that are shorter than 16 bytes. If too many tokens are short, the number of padding bits in the tokenized datapath would negatively affect performance.

Figure 13 shows the percentage of useful bits in the tokenized datapath, excluding padding bits. Generally, about half of the 16 bytes tokenized datapath is useful data. This observation has driven the design of the filtering engine. For example, the 16-byte datapath is a balance between bandwidth and the ratio of padding bits. An 8-byte datapath was too slow, requiring too many pipelines, and the performance benefits of a 32-byte datapath were limited due to too many padding bits. Furthermore, the hash filter module is replicated twice to account for this decrease in effective bandwidth. The eight tokenizers in each pipeline is divided into exclusive groups of two, each group connected to one hash filter. Each replicated pipeline ingests 8 bytes of un-tokenized data per cycle, and processes maximum 16 bytes of tokenized data per cycle including padding bits. Considering the data amplification, two such pipelines adds up to statistically 16 bytes of useful data processed per cycle, which is the maximum bandwidth supported by the datapath.

	BGL2	Liberty2	Spirit2	Thunderbird
LZAH	2.63x	3.85x	6.60x	7.35x
LZRW1	4.39x	5.79x	6.00x	3.89x
LZ4	5.95x	27.27x	27.14x	9.68x
Gzip	11.82x	47.93x	45.04x	15.79x

Table 5: Compression effectiveness against other algorithms.

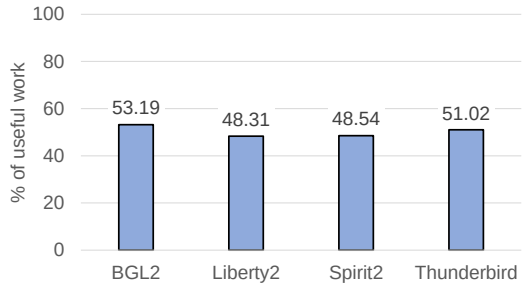


Figure 13: Percentage of useful bits in the tokenized datapath.

Figure 14 shows the total effective throughput across the four filtering engine pipelines, based on how much actual, decompressed text is processed per second. The total throughput of the filtering engines span between 11 GB/s and 12 GB/s for all datasets, almost four times the PCIe bandwidth.

The upper bound of achievable performance is set by the four decompressor pipelines, which can deliver a total of 12.8 GB/s of decompressed bandwidth given sufficiently high storage bandwidth. While the high compression ratio of LZAH coupled with the 4.8 GB/s internal bandwidth of the backing storage device was enough to keep the four pipelines completely busy, there is a slight performance difference between the filtering engines and the maximum decompressor performance. This difference comes from various sources, including the ratio of useful bits in the tokenized datapath, as presented in Figure 13. It also comes from the imbalance between lengths of consecutive log lines, causing certain tokenizers to become the bottleneck for short periods of time.

We also notice that for the datasets `Liberty2`, `Spirit2`, and `Thunderbird`, LZAH provided enough compression to keep the four decompressors always completely busy. However, the compression ratio of `BGL2` is relatively low, as seen in Table 5, resulting in a slightly lower performance of 12.62 GB/s. However, even then the effective bandwidth of the decompressed stream was fast enough to keep the filter engine busy. For `Liberty2`, `Spirit2`, and `Thunderbird`, adding more pipelines to the same storage device will improve performance, but for `BGL2`, we have reached to limit of performance attainable with the backing storage.

7.4.2 Comparison Against Optimized Software. We first evaluate the performance of our token filtering engine independently from other database effects such as indexing, against a well-optimized software implementation. We have chosen MonetDB as it has often demonstrated to be one of the fastest SQL databases [44]. We have also experimented with other platforms including `grep`, but we present MonetDB as it consistently delivered highest performance, especially due to its column-oriented compression helping overcome the PCIe bottleneck.

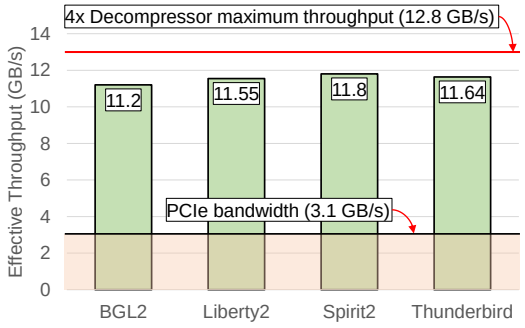


Figure 14: Near-storage configuration and compression together support much higher filtering bandwidth than PCIe.

In order to separate the text performance of MonetDB from other database effects including indexes and column-oriented operations, we store all lines for each dataset in a table with a single `VARCHAR` column, forcing MonetDB to scan the whole table for each query. We emphasize that performance including indexing effects are compared and presented in the following subsection. MithriLog was also configured to not use the inverted index, and scan the whole dataset for each query.

Performance was compared using *effective throughput*, calculated by dividing the original size of each dataset by the time elapsed per query. This means the effective throughput can exceed storage performance if compression or indexing is used effectively. Each system executed the single and combination benchmarks described in Section 7.1.

Figure 15 shows the performance histogram of MonetDB and MithriLog for the four datasets. We note that the x-axis of Figure 15 is not linear, in order to better present the performance distribution.

Given the higher bandwidth of the storage device equipped for MonetDB compared to MithriLog (Table 3), the overall lower performance of MonetDB shows that processing is largely bottlenecked by computation performance of the CPU. In contrast, MithriLog performance is constantly high thanks to the high throughput of hardware accelerators, reporting over 11 GB/s of effective throughput regardless of query contents.

For MonetDB, single queries typically show higher performance compared to larger combinations, showing the histogram distribution moving left with larger combination queries. This is likely due to the computation bottleneck of text processing as more terms are involved. Storage performance profiling also reinforces this assessment, as typically less than 1 GB/s of bandwidth usage was observed, while all cores on the CPU were running at maximum capacity. For `BGL2`, MonetDB incurred no storage access after the very first query loaded the small dataset entirely into memory, but still resulted in similar levels of effective throughput.

Table 6 also presents the average effective bandwidth of the 1-, 2- and 8-query combinations on MonetDB and MithriLog, as well as the average improvement over total number of queries, for all datasets. MithriLog demonstrates significantly better performance, often above an order of magnitude, thanks to not only the hardware accelerator performance, but also the effective storage performance improvement due to the near-storage configuration and compression.

7.4.3 Comparison Against Existing Accelerators. Compared to a hypothetical log analytics accelerator with state-of-the-art implementations of general-purpose compression and regular expression matching, MithriLog achieves superior performance within resource restrictions.

For example, HARE [13] implements log querying based on regular expression matching, and achieves 400 MB/s with 12% (~55K Logic Elements) of an Intel Arria V SoC FPGA. Assuming similar capabilities between Arria V Logic Elements and Virtex 7 Logic Cells for a back-of-the-envelope estimation, a HARE accelerator coupled with a resource-efficient LZRW accelerator [20] requires about 145K LUTs per 1 GB/s of bandwidth. On the contrary, MithriLog with LZAH only requires about 19K LUTs per 1 GB/s, almost an order of magnitude better efficiency.

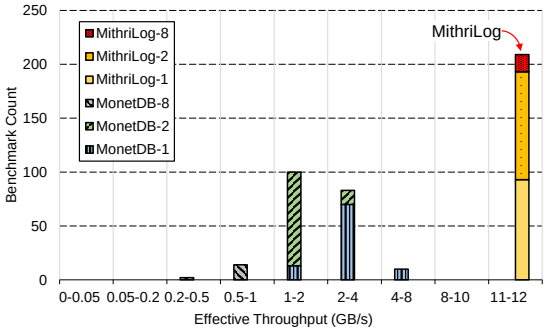
7.5 End-To-End Query Evaluation

We also compare the end-to-end query performance of MithriLog against a prominent log analytics platform, Splunk. MonetDB was not used for this experiment because it is difficult to effectively map free-form text, such as the logs under observation, to regularly structured relational databases such as MonetDB. On the other hand, log analytics platforms such as Splunk are optimized for free text processing using structures such as inverted indices. Splunk was not used for the previous token filtering experiment because we could not find a way to disable or restrict the indices in Splunk without changing the quality of the output.

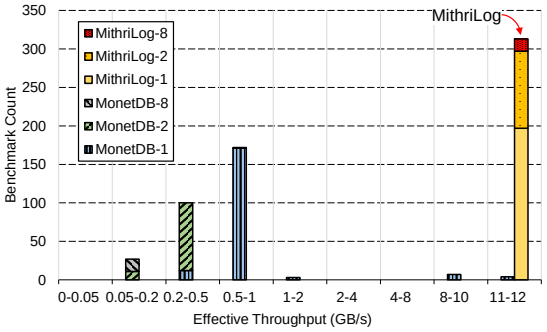
Another characteristic of Splunk is that each search query is handled by a single thread, meaning the advantage of parallelism is only available if many search queries are issued concurrently. In order to perform a fair comparison in favor of Splunk, we first measured the elapsed time for each query in Splunk, and divided it by the number of hyper-threads — 12 in our test platform across 6 physical cores — in order to get the amortized, upper-bound performance. Considering Intel's advertised benefit of hyperthreading is 30% at most [22] we are choosing to err on the side of caution, in favor of Splunk by deciding to divide by 12 instead of 7.8, as ($6 \times 1.3 = 7.8$). We also notice that other management functionalities of Splunk often added almost 100% processing overhead on top of the search query thread, resulting in two cores being completely occupied during a single query. However, we decided to again err

System	BGL2	Liberty2	Spirit2	Thunderbird
MonetDB1	2.57	0.64	2.84	0.65
MithriLog1	11.2	11.55	11.8	11.64
MonetDB2	1.53	0.24	0.16	0.24
MithriLog2	11.2	11.55	11.8	11.64
MonetDB8	0.58	0.07	0.05	0.06
MithriLog8	11.2	11.55	11.8	11.64
Average Improve.	5.82x	23.91x	6.01x	84.79x

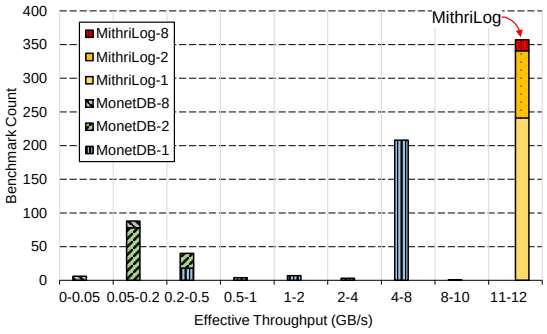
Table 6: Average effective throughput of batched queries in GB/s.



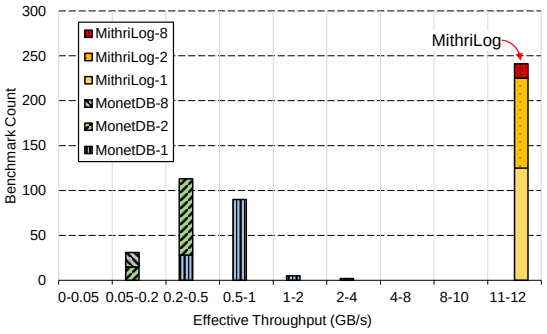
(a) Query performance with the BGL2 dataset.



(b) Query performance with the Liberty2 dataset.



(c) Query performance with the Spirit2 dataset.



(d) Query performance with the Thunderbird dataset.

Figure 15: Performance histogram shows hardware filtering demonstrating consistently superior performance against MonetDB.
(Right is better)

on the side of caution for fair comparison, and assume each Splunk search query takes at most a single thread to service.

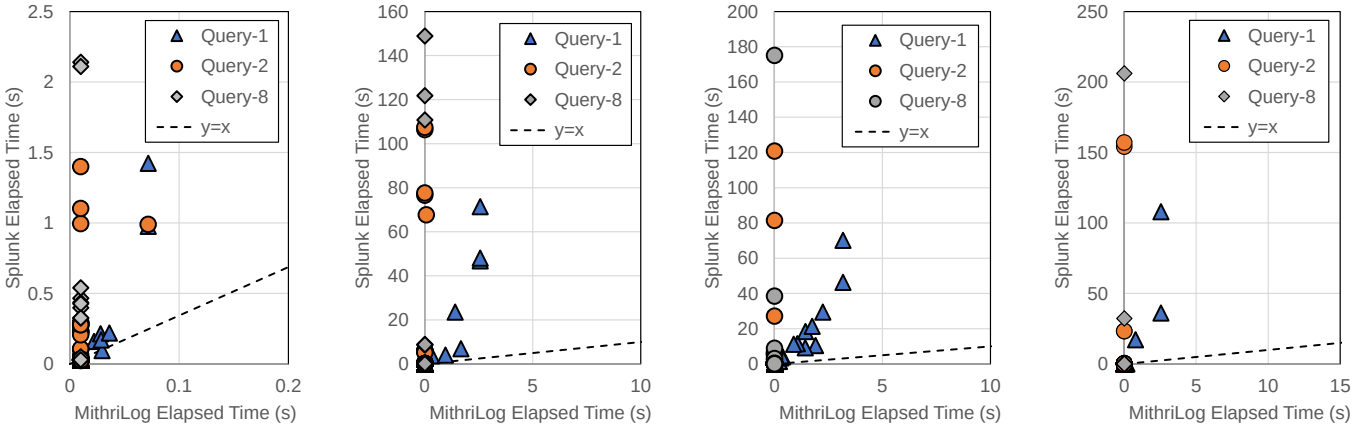
Figure 16 shows the scatter plot of the query time differences between MithriLog and Splunk, for all single and combination queries tested. Thanks to the effectiveness of the inverted index implementations in both Splunk and MithriLog, most of the queries finish in sub-second latency. However, the index structures were not always completely effective, especially with queries with multiple negative terms (e.g., "NOT A"), requiring a large subset of the log file to be loaded and processed. Such queries appear as data points clustered around the left-side edge of the scatter plots, because the increased processing requirements affect the performance of Splunk more than they affect MithriLog. Table 7 presents the average performance improvement of MithriLog over Splunk, calculated by comparing the total execution time for all tested queries per dataset. MithriLog demonstrates an order of magnitude better performance and sometimes even more, on the total set of queries we tested. We further emphasize that the elapsed time for Splunk used for comparison was after dividing the measured time with the number of threads (12).

BGL2	Liberty2	Spirit2	Thunderbird
9.93	352.26	201.20	86.32

Table 7: Average performance improvement over Splunk.

For example, one of the shorter automatically generated queries for the `liberty2` dataset is `("failed" AND NOT "pbs_mom:")`. Our inverted index implementation was only able to reduce the number of storage page reads by 30%. This represents 22 GB of uncompressed data for the token filter to process. We suspect the index of Splunk had similar levels of success. This query took 561 seconds for Splunk, resulting in the plotted elapsed time to be $561 \div 12 = 46$ seconds. Meanwhile, MithriLog was able to complete the query in around two seconds end-to-end, including index structure access. Considering the lower bound performance for MithriLog is an elapsed time of 2.6 seconds — when the index completely fails and the whole file needs to be processed — MithriLog demonstrates over an order of magnitude performance improvement for large search queries.

One relevant observation regarding templates generated by FT-tree, and also how we translate them to queries for Splunk, is that many of the complex queries involving many query terms included a very large number of negative terms and a small number of positive ones. We also tried removing all `NOT` terms from the queries and leaving only the potentially small number of partitive terms, to evaluate the performance impact of such terms on Splunk. This change resulted in at most 2× performance difference, not significantly changing the performance relationship between Splunk and MithriLog even if such a change was allowed.



(a) Query performance with the BGL2 dataset.

(b) Query performance with the Liberty2 dataset.

(c) Query performance with the Spirit2 dataset.

(d) Query performance with the Thunderbird dataset.

Figure 16: MithriLog consistently demonstrates superior performance against Splunk for complex queries. (Lower right is better for Splunk, upper left is better for MithriLog. Note the two axes have different scales.)

7.6 Power Performance Evaluation

Table 8 presents the power consumption breakdown of the two platforms. Each BlueDBM storage is powered via a separate power plug, whose power consumption can be measured using wall port power monitors. We measured each of the two VC707 FPGA boards consuming about 18 Watts during steady state processing, and each of the four BlueDBM storage cards about 6 to 7 Watts. The BlueDBM storage cards are prototypes that are more power-hungry compared to the optimized, off-the-shelf storage devices used by the software systems. MithriLog's power efficiency should improve even further with better-optimized hardware platforms. The accelerated storage devices in total consumes about 60 W of power under load, a similar number as the one published for Samsung's SmartSSD [33]. Since we did not have a way to measure the PCIe-attached SSDs used for the software system, we used the numbers published by Samsung [57], and subtracted them from the total measured power consumption to calculate the CPU and memory power.

The measurements show that by using power-efficient FPGAs for computation, the total power consumption of the system actually decreased. Since performance has improved by over an order of magnitude, resulting power efficiency is also over an order of magnitude higher compared to state-of-the-art software systems.

Component	MithriLog	Software
CPU+Memory (Watt)	90	160
Total Storage (Watt)	24	10
2x FPGA (Watt)	36	0
Total (Watt)	150	170

Table 8: Estimated power consumption breakdown of the two platforms.

8 CONCLUSION AND FUTURE WORK

We present MithriLog, a log analytics platform with near-storage accelerators. MithriLog improves the effective bandwidth of the backing storage to match the accelerator performance by positioning itself near-storage, and also by using a high-throughput compression accelerator. Coupled with an inverted index implementation, MithriLog demonstrates over an order of magnitude performance improvements above state-of-the art log analytics platforms including Splunk, for complex queries such as tree-based template search. MithriLog also reduces the overall power consumption of the system by replacing CPU threads with FPGA accelerators, resulting in a dramatic power efficiency improvement. We project MithriLog can greatly benefit applications such as large-scale system monitoring and intrusion detection in both cloud and resource-restricted edge environments. Furthermore, near-storage acceleration approaches like MithriLog will become more important in the future, as storage performance scaling outpace communication and general-purpose computation.

We are also actively working on building higher-order log analytics accelerators that process the output of the MithriLog system, such as neural networks for intrusion detection, join operations, and data mining. Another ongoing effort is expanding the template-based query capabilities. Targets include matching other template structures such as regular expressions, as well as exploring wire-speed methods for tagging each log line with template IDs.

ACKNOWLEDGMENTS

We thank the anonymous reviewers of MICRO 2021, as well as our shepherd for his/her help improving the presentation of this paper. We also thank Christos Karamanolis, Marc Fleischmann, David Ott and Amy Tai from VMware for their support and industrial insight. This work has been made possible in part by gifts from VMware's University Research Fund, as well as funding from NSF (CNS-1908507).

REFERENCES

[1] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 267–277.

[2] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *Cidr*, Vol. 5. 225–237.

[3] Christopher Dennl, Daniel Ziener, and Jürgen Teich. 2013. Acceleration of SQL restrictions and aggregations through FPGA-based dynamic partial reconfiguration. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 25–28.

[4] Himel Dev and Zhicheng Liu. 2017. Identifying frequent user tasks from application logs. In *Proceedings of the 22nd International Conference on Intelligent User Interfaces*. 263–273.

[5] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. 2013. Query processing on smart ssds: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1221–1230.

[6] Min Du and Feifei Li. 2016. Spell: Streaming parsing of system event logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 859–864.

[7] Facebook Engineering. 2017. LogDevice: a distributed data store for logs. https://engineering.fb.com/2017/08/31/core-data/logdevice-a-distributed-data-store-for-logs/. [Online; accessed 2021-04-14].

[8] Bo Feng, Chentao Wu, and Jie Li. 2016. MLC: an efficient multi-level log compression method for cloud backup systems. In *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE, 1358–1365.

[9] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. 2007. X-trace: A pervasive network tracing framework. In *4th {USENIX} Symposium on Networked Systems Design & Implementation ({NSDI} 07)*.

[10] Apache Software Foundation. 2017. Apache Kafka. https://kafka.apache.org/. [Online; accessed 2021-04-14].

[11] Apache Software Foundation. 2021. Apache Datadog. https://www.datadoghq.com/blog/tag/apache/. [Online; accessed 2021-04-14].

[12] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 ninth IEEE international conference on data mining*. IEEE, 149–158.

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

[14] Ceki Gülcü. 2003. *The complete log4j manual*. QOS. ch.

[15] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. 2016. Logmine: Fast pattern recognition for log analytics. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 1573–1582.

[16] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining frequent patterns without candidate generation. *ACM sigmod record* 29, 2 (2000), 1–12.

[17] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. 2017. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 33–40.

[18] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. 2016. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 207–218.

[19] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. 2020. Loghub: a large collection of system log datasets towards automated log analytics. *arXiv preprint arXiv:2008.06448* (2020).

[20] Helion. 2008. LZRW Compression cores. https://www.heliontech.com/comp_lzrw.htm. [Online; accessed 2021-08-17].

[21] S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. 2012. Monetdb: Two decades of research in column-oriented database. *IEEE Data Engineering Bulletin* (2012).

[22] Intel. 2011. How to Determine the Effectiveness of Hyper-Threading Technology with an Application. https://software.intel.com/content/www/us/en/develop/articles/how-to-determine-the-effectiveness-of-hyper-threading-technology-with-an-application.html. [Online; accessed 2021-04-14].

[23] Zsolt István, David Sidler, and Gustavo Alonso. 2016. Runtime parameterizable regular expression operators for databases. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 204–211.

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

[25] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, et al. 2018. GraFBoost: Using accelerated flash storage for external graph analytics. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 411–424.

[26] Wonmook Jung, Hongchan Roh, Mincheol Shin, and Sanghyun Park. 2015. Inverted index maintenance strategy for flashSSDs: Revitalization of in-place index update strategy. *Information Systems* 49 (2015), 25–39.

[27] Yangwook Kang, Yang-suk Kee, Ethan L Miller, and Chanik Park. 2013. Enabling cost-effective data processing with smart SSD. In *2013 IEEE 29th symposium on mass storage systems and technologies (MSST)*. IEEE, 1–12.

[28] Roman Kaplan, Leonid Yavits, and Ran Ginosar. 2018. Prins: Processing-in-storage acceleration of machine learning. *IEEE Transactions on Nanotechnology* 17, 5 (2018), 889–896.

[29] Byungseok Kim, Jaeho Kim, and Sam H Noh. 2017. Managing array of ssds when the storage device is no longer the performance bottleneck. In *9th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*.

[30] Gunjae Koo, Kiran Kumar Matam, I Te, HV Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. 2017. Summarizer: trading communication with computing near storage. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 219–231.

[31] Reinhard Kutzelnigg. 2006. Bipartite random graphs and cuckoo hashing. In *Discrete Mathematics and Theoretical Computer Science*. Discrete Mathematics and Theoretical Computer Science, 403–406.

[32] Jinho Lee, Heesu Kim, Sungjoo Yoo, Kiyoung Choi, H Peter Hofstee, Gi-Joon Nam, Mark R Nutter, and Damir Jamsek. 2017. Extrav: boosting graph processing near storage with a coherent accelerator. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1706–1717.

[33] Joo Hwan Lee, Hui Zhang, Veronica Lagrange, Praveen Krishnamoorthy, Xiaodong Zhao, and Yang Seok Ki. 2020. SmartSSD: FPGA Accelerated Near-Storage Data Analytics on SSD. *IEEE Computer Architecture Letters* 19, 2 (2020), 110–113.

[34] Zhou Li and Alina Oprea. 2016. Operational security log analytics for enterprise breach detection. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 15–22.

[35] Hao Lin, Jingyu Zhou, Bin Yao, Minyi Guo, and Jie Li. 2015. Cowic: A column-wise independent compression for log stream analysis. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 21–30.

[36] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. 2016. Log clustering based problem identification for online service systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 102–111.

[37] Jinyang Liu, Jieming Zhu, Shilin He, Pinjia He, Zibin Zheng, and Michael R Lyu. 2019. Logzip: Extracting hidden structures via iterative clustering for log compression. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 863–873.

[38] Weiqiang Liu, Faqiang Mei, Chenghua Wang, Maire O'Neill, and Earl E Swartzlander. 2018. Data compression device based on modified LZ4 algorithm. *IEEE Transactions on Consumer Electronics* 64, 1 (2018), 110–117.

[39] Khalid Mahmood, Kjell Orsborn, and Tore Risch. 2019. Comparison of nosql datastores for large scale data stream log analytics. In *2019 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE, 478–480.

[40] Khalid Mahmood, Tore Risch, and Minpeng Zhu. 2015. Utilizing a nosql data store for scalable log analysis. In *Proceedings of the 19th International Database Engineering & Applications Symposium*. 49–55.

[41] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. 2019. LogAnomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs.. In *IJCAI*, Vol. 7. 4739–4745.

[42] Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel Briand, and Raimondas Sasnauskas. 2018. A search-based approach for accurate identification of log message formats. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 167–16710.

[43] Abhishek Mitra, Marcos R Vieira, Petko Bakalov, Vassilis J Tsotras, and Walid A Najjar. 2009. Boosting XML filtering through a scalable FPGA-based architecture.. In *CIDR*.

[44] MonetDB. April 2014 (Accessed Sep 25, 2019). *Citus Data cstor_fdw (PostgreSQL Column Store) vs. MonetDB TPC-H Shootout*. https://www.monetdb.org/content/citusdb-postgresql-column-store-vs-monetdb-tpc-h-shootout.

[45] Roger Moussalli, Mariam Salloum, Walid Najjar, and Vassilis J Tsotras. 2011. Massively parallel XML twig filtering using dynamic programming on FPGAs. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 948–959.

[46] Adam Oliner, Archana Ganapathi, and Wei Xu. 2012. Advances and challenges in log analysis. *Commun. ACM* 55, 2 (2012), 55–61.

[47] Adam Oliner and Jon Stearley. 2007. What supercomputers say: A study of five system logs. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE, 575–584.

[48] Adam J Oliner, Alex Aiken, and Jon Stearley. 2008. Alert detection in system logs. In *2008 Eighth IEEE International Conference on Data Mining*. IEEE, 959–964.

[49] Savan Oswal, Anjali Singh, and Kirthi Kumari. 2016. Deflate compression algorithm. *International Journal of Engineering Research and General Science* 4, 1 (2016), 430–436.

[50] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. 2017. Centaur: A framework for hybrid CPU-FPGA databases. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 211–218.

[51] Rasmus Pagh and Flemming F1riche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.

[52] Byung H Park, Saurabh Hukerikar, Ryan Adamson, and Christian Engelmann. 2017. Big data meets hpc log analytics: Scalable approach to understanding systems at extreme scale. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 758–765.

[53] Raphael Polig, Kubilay Atasu, Laura Chiticariu, Christoph Hagleitner, H Peter Hofstee, Frederick R Reiss, Huaiyu Zhu, and Eva Sitaridi. 2014. Giving text analytics a boost. *IEEE Micro* 34, 4 (2014), 6–14.

[54] Raphael Polig, Kubilay Atasu, Heiner Giefers, and Laura Chiticariu. 2014. Compiling text analytics queries to FPGAs. In *2014 24th international conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–6.

[55] Balázs Rácz and András Lukács. 2004. High density compression of log files. In *Data Compression Conference, 2004. Proceedings. DCC 2004.* IEEE, 557.

[56] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A seven-dimensional analysis of hashing methods and its implications on query processing. *PVLDB* 9, 3 (2015), 96–107.

[57] Samsung Semiconductor. 2021. Samsung SSD 970 EVO. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/970evo/. [Online; accessed 2021-04-14].

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

[59] 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*. 403–415.

[60] Przemysław Skibiński and Jakub Swacha. 2007. Fast and efficient log file compression. In *proceedings of 11th east-European conference on advances in databases and information systems (ADBIS)*. 330–342.

[61] Jon Stearley and Adam J Oliner. 2008. Bad words: Finding faults in spirit's syslogs. In *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*. IEEE, 765–770.

[62] Karun Subramanian. 2020. Introducing the Splunk Platform. In *Practical Splunk Search Processing Language*. Springer, 1–38.

[63] Candace Suh-Lee, Ju-Yeon Jo, and Yoohwan Kim. 2016. Text mining for security threat detection discovering hidden information in unstructured log messages. In *2016 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 252–260.

[64] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. 2012. Database analytics acceleration using FPGAs. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 411–420.

[65] Gongjin Sun and Sang-Woo Jun. 2020. ColumnBurst: a near-storage accelerator for memory-efficient database join queries. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*. 9–16.

[66] Haoyu Tan, Wuman Luo, and Lionel M Ni. 2012. Clost: a hadoop-based storage system for big spatio-temporal data analytics. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. 2139–2143.

[67] Haoyu Tan, Wuman Luo, and Lionel M Ni. 2012. Clost: a hadoop-based storage system for big spatio-temporal data analytics. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. 2139–2143.

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

[69] Mahdi Torabzadehkashi, Siavash Rezaei, Ali Heydarigorji, Hosein Bobarshad, Vladimir Alves, and Nader Bagherzadeh. 2019. Catalina: in-storage processing acceleration for scalable big data analytics. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 430–437.

[70] Tal Wagner, Eric Schkufza, and Udi Wieder. 2016. A sampling-based approach to accelerating queries in log management systems. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. 37–38.

[71] Chao Wang, Xi Li, and Xuehai Zhou. 2015. SODA: Software defined FPGA based accelerators for big data. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 884–887.

[72] Teng Wang, Shane Snyder, Glenn Lockwood, Philip Carns, Nicholas Wright, and Suren Byna. 2018. Iominer: Large-scale analytics framework for gaining knowledge from i/o logs. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 466–476.

[73] Kyu-Young Whang, Il-Yeol Song, Taek-Yoon Kim, and Ki-Hoon Lee. 2010. The ubiquitous DBMS. *ACM SIGMOD Record* 38, 4 (2010), 14–22.

[74] Ross N Williams. 1991. An extremely fast Ziv-Lempel data compression algorithm. In *1991 Data Compression Conference*. IEEE Computer Society, 362–363.

[75] Louis Woods, Jens Teubner, and Gustavo Alonso. 2010. Complex event detection at wire speed with FPGAs. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 660–669.

[76] Xilinx. 2018. Xilinx/Applications/data_compression/xil_lz4/. https://github.com/Xilinx/Applications/tree/master/data_compression/xil_lz4. [Online; accessed 2021-08-17].

[77] Xilinx. 2018. Xilinx/Applications/data_compression/xil_snappy/. https://github.com/Xilinx/Applications/tree/master/data_compression/xil_snappy. [Online; accessed 2021-08-17].

[78] Shuotao Xu, Thomas Bourgeat, Tianhao Huang, Hojun Kim, Sungjin Lee, and Arvind Arvind. 2020. AQUOMAN: An Analytic-Query Offloading Machine. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 386–399.

[79] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 117–132.

[80] Yi-Hua Yang and Viktor Prasanna. 2011. High-performance and compact architecture for regular expression matching on FPGA. *IEEE Trans. Comput.* 61, 7 (2011), 1013–1025.

[81] Jongseong Yoon, Doowon Jeong, Chul-hoon Kang, and Sangjin Lee. 2016. Forensic investigation framework for the document store NoSQL DBMS: MongoDB as a case study. *Digital Investigation* 17 (2016), 53–65.

[82] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. 2016. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. *ACM SIGARCH Computer Architecture News* 44, 2 (2016), 489–502.

[83] Vlad-Andrei Zamfir, Mihai Carabas, Costin Carabas, and Nicolae Tapus. 2019. Systems monitoring and big data analysis using the elasticsearch system. In *2019 22nd International Conference on Control Systems and Computer Science (CSCS)*. IEEE, 188–193.

[84] Shenglin Zhang, Ying Liu, Weibin Meng, Jiahao Bu, Sen Yang, Yongqian Sun, Dan Pei, Jun Xu, Yuzhi Zhang, Lei Song, et al. 2020. Efficient and robust syslog parsing for network devices in datacenter networks. *IEEE Access* 8 (2020), 30245–30261.

[85] Shenglin Zhang, Weibin Meng, Jiahao Bu, Sen Yang, Ying Liu, Dan Pei, Jun Xu, Yu Chen, Hui Dong, Xianping Qu, et al. 2017. Syslog processing for switch failure diagnosis and prediction in datacenter networks. In *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*. IEEE, 1–10.

[86] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. 2019. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 121–130.