HAWK: Hardware Support for Unstructured Log Processing

Prateek Tandon prateekt@umich.edu
Faissal M. Sleiman sleimanf@umich.edu
Michael J. Cafarella michjc@umich.edu
Thomas F. Wenisch twenisch@umich.edu

Department of Computer Science and Engineering, University of Michigan

Abstract*—Rapidly processing high-velocity text data is critical for many technical and business applications. Widely used software solutions for processing these large text corpora target disk-resident data and rely on pre-computed indexes and large clusters to achieve high performance. However, greater capacity and falling costs are enabling a shift to RAM-resident data sets. The enormous bandwidth of RAM can facilitate scan operations that are competitive with pre-computed indexes for interactive, ad-hoc queries. However, software approaches for processing these large text corpora fall far short of saturating available bandwidth and meeting peak scan rates possible on modern memory systems. In this paper, we present HAWK, a hardware accelerator for ad hoc queries against large in-memory logs. HAWK comprises a stall-free hardware pipeline that scans input data at a constant rate, examining multiple input characters in parallel during a single accelerator clock cycle. We describe a 1GHz 32-character-wide HAWK design targeting ASIC implementation, designed to process data at 32GB/s (up to two orders of magnitude faster than software solutions), and demonstrate a scaled-down FPGA prototype that operates at 100MHz with 4-wide parallelism, which processes at 400MB/s (13× faster than software grep for large multi-pattern scans).

I. Introduction

High-velocity electronic text log data, such as system logs, social media updates, web documents, blog posts, and news articles, have undergone explosive growth in recent years [24]. These textual logs can hold useful information for time-sensitive domains, such as diagnosing distributed system failures, online ad pricing, and financial intelligence. For example, a system administrator might want to find all HTTP log entries that mention a certain URL. A financial intelligence application might search for spikes in the number of Tweets that contain the phrase *"can't find a job"*. Queries on high-velocity text data are often *ad hoc*, *highly-selective*, and *latency-sensitive*; i.e., the workload is not known beforehand; queries often ignore the vast majority of the corpus; and answers must be generated quickly and reflect up-to-the-second data.

The dominant current-generation tools for management of unstructured data like textual logs, such as Splunk or Hadoop, are designed to process data sets that reside on disk or SSD. They achieve high performance through scale, by sharding data over a large number of disks and servers. However, RAM storage costs have fallen drastically over the past decade, and new storage technologies, such as the recently announced Intel/Micron XPoint 3D memory [2], which promise RAM-like performance at even lower cost per bit, will be available soon. As a result, memory-resident databases are becoming a popular architectural solution, not simply for transactional [17], [27] workloads, but for analytical ones [1], [19], [25], [26], [34] as well. For example, Twitter's own search engine now stores recent data in RAM [8].

The shift to RAM-resident data sets fundamentally alters the performance requirements of a data management engine. Memory bandwidth—the rate at which the architecture supports transfers from RAM to the CPU for processing—is nearly two orders of magnitude higher than bandwidth available over the network or to disk. As a result, latency-sensitive queries, which conventionally could meet performance objectives only via pre-computed indexes, may become achievable with scans, avoiding the compute and storage costs of constructing and maintaining an index.

Unfortunately, existing processing techniques do not come close to saturating available memory bandwidth. For example, using a state-of-the-art in-memory database, we measure a peak scan rate of less than 2GB/s on a dual-socket 16-threaded server—only 15% of available memory bandwidth. Non-database textual tools, such as *grep* and *awk*, perform even worse, sometimes by orders of magnitude. The gap arises because these tools must execute many instructions, on average, for each input character they scan. Thus instruction execution throughput, rather than memory bandwidth, limits performance. With the advent of DDR4, the gap between instruction throughput and memory bandwidth will grow.

System Goal — There are many questions when building an in-memory analytical database, but in this paper we focus on just one: *can we saturate memory bandwidth when processing text log queries?*[1] If so, the resulting system could be used directly in *grep*- and *awk*-style tools, and integrated as a query processing component in memory-resident relational systems.

We are interested in designs that include both software and hardware elements. Although hardware accelerators have had a mixed history in data management systems, there is reason to be newly optimistic about their future. The anticipated end of CMOS voltage scaling (Dennard scaling) has led experts to predict the advent of chips with "dark silicon";

[1]Although we are motivated primarily by text log processing, general streaming data query processing has many of the same requirements.

that is, chips that are designed to have a substantial portion powered off at any given time [5], [11], [30]. This forecast has renewed interest in domain-specific hardware accelerators that can create value from otherwise dark portions of a chip—accelerators powered only when especially needed. Researchers have recently proposed several hardware designs tailored for data management [14], [33]. Further, recently-announced chip designs include *field programmable gate array (FPGA)* elements [7], making domain-specific hardware accelerators—implemented in FPGAs—even more practical and promising. There has also been substantial interest in using FPGAs for database query processing [13], [20], [31], [32].

Technical Challenge — It is not surprising that current software systems on standard cores fall short of saturating memory bandwidth. Most text processing systems use pattern matching state machines as a central abstraction, and standard cores that implement these machines in software can require tens of instructions per character of input. Further, efficiently representing state machines for large alphabets and complex queries is challenging; the resulting transition matrices are sparse, large, and randomly accessed, leading to poor hardware cache performance.

We set an objective of processing in-memory ASCII text at 32 *giga-characters per second* (GC/s), corresponding to 32GB/s from memory—a convenient power of two expected to be within the typical capability of near-future high-end servers incorporating several DDR3 or DDR4 memory channels. We investigate whether a custom hardware component can reach this performance level, and how much power and silicon area it takes. Achieving this processing rate with conventional multicore parallelism (e.g., by sharding the text log data into subsets, one per core) is infeasible; our measurements of a state-of-the-art in-memory database suggest that chips would require nearly 20× more cores than are currently commonplace in order to reach the target level of performance.

Our Approach — We propose a combination of a custom hardware accelerator and an accompanying software query compiler for performing selection queries over in-memory text data. When a user's query arrives, our compiler creates pattern matching finite state automata that encode the query and transmits them to the accelerator; the accelerator then executes the automata, recording the memory addresses of all text elements that satisfy the query. This list of results can then be used by the larger data management software to present results to the user, or as intermediate results in a larger query plan.

We exploit two central observations to obtain fast processing while using a reasonable hardware resource budget. First, our accelerator is designed to operate at a fixed scan rate: it always scans and selects text data at the same rate, regardless of the data or the query, streaming data sequentially from memory at 32GB/s. We can achieve such performance predictability because the scan engine requires no control flow or caches; hence, the hardware scan pipeline never stalls and can operate at a fixed 1GHz frequency, processing 32 input characters per clock cycle. Our approach allows us to avoid the cache misses, branch mispredictions, and other aspects of CPUs that make performance unpredictable and require area-intensive hardware to mitigate. Second, we use a novel formulation of the automata that implement the scan operation, thereby enabling a hardware implementation that can process many characters concurrently while keeping on-chip storage requirements relatively small. We conceptually concatenate 32 consecutive characters into a single symbol, allowing a single state transition to process all 32 characters. Naïvely transforming the input alphabet in this way leads to intractable state machines—the number of outgoing edges from each state is too large to enable fixed-latency transitions. So, we leverage the concept of *bit-split pattern matching automata* [29], wherein the original automaton is replaced with a vector of automata that each process only a bit of input. As a result, each per-bit state requires only two outgoing transitions. Matches are reported when all bit-split automata have recognized the same search pattern.

Contributions and Outline — The core contributions of this paper are as follows:

1) We describe a typical log processing query workload, describe known possible solutions (that are unsuitable), and provide some background information about conventional approaches (Sections II, III).
2) We propose HAWK, a hardware accelerator design with a fixed scan-and-select processing rate. HAWK employs *automata sharding* to break a user query across many parallel processing elements. The design is orthogonal to standard data sharding (i.e., breaking the dataset into independent parts for parallel processing), and can be combined with that approach if needed (Sections IV, V).
3) We describe a 1GHz 32-character-wide HAWK design targeting ASIC implementation, which can saturate near-future memory bandwidth, outperforming current software solutions by orders of magnitude. Indeed, our scan operations are fast enough that they are often competitive with software solutions that utilize pre-computed indexes.
4) We validate our ASIC design with a scaled-down FPGA prototype. The FPGA prototype is a 4-wide HAWK design and operates at 100MHz. Even at this greatly reduced processing rate, the FPGA design outperforms *grep* by 13× for challenging multi-pattern scans.

Section VII covers related work and Section VIII concludes.

II. Problem Description

```
hadoop.apache.org; 06:32:09; opera;   linux;   131.24.0.7;    279
mahout.apache.org; 06:32:15; safari;  osx;     187.98.32.1;   1729
hadoop.apache.org; 06:32:23; firefox; osx;     243.56.171.53; 583
chukwa.apache.org; 06:32:25; ie;      windows; 54.12.87.10;   9854
...
```

Fig. 1: A sample log file.

We focus on the problem of enabling scans of textual and log-style data to saturate modern memory bandwidth. Figure 1 shows a brief example of such data. The query workload is a mixture of standing queries that can be precompiled, and *ad hoc* ones driven by humans or by automated responses to

previous query results. The actual queries involve primarily field-level tokenization plus string equality tests.

In this section, we cover the user-facing desiderata of such a system, including the data model and query language. Then, we consider traditional software solutions for such queries and why hardware acceleration is desirable.

A. Desiderata for a Log Processing System

Data Characteristics — The text to be queried is log-style information derived from Web servers or other log output from server-style software. We imagine a single textual dataset that represents a set of records, each consisting of one or more explicitly delimited fields. The number of fields per record may vary. For example, a typical record may contain date stamp and optional source identifier information in addition to free-form text.

Standard sharing formats such as JSON are increasingly common but still create non-trivial computational serialization and deserialization overhead when applied at large scale. As a result, they are generally only used for relatively-rare "interface-level" data communications, and are not standard for bulk logs that are intended to be human-readable. However, if the user does want to process JSON with our proposed hardware, doing so is possible using the filter-style deployment described in Section II-B.

In contrast to existing systems like *Splunk*, we do not construct an index when ingesting data or as logs arrive, thereby avoiding both the processing and storage cost of indexing, and instead execute queries exclusively via scan operators. Whereas an inverted index might accelerate string-equality queries, index construction can take hours and consume a majority of available memory capacity, increasing the number of servers required to manage a given data set. Furthermore, joins seeking n-grams with frequent terms can remain expensive even with indexes. Our objective is to demonstrate that index-free selection queries are viable with hardware acceleration.

Query Language — The data processing system must answer selection and projection queries over the aforementioned data. Fields are simply referred to by their field number. For example, for the data in Figure 1, we might want to ask:

```
SELECT $3,$5 WHERE $6=200 AND
  ($5="132.99.20.201" OR $5="100.202.44.1")
```

The system must support Boolean predicates on numeric (=, <>, >, <, <=, =<) and textual fields (equality and LIKE).

Query Workload — We assume queries that have four salient characteristics. *1)* They are *ad hoc*, possibly written in response to ongoing shifts in the incoming log data, such as in financial trading, social media intelligence, or network log analysis. *2)* Queries are *time-sensitive*: the user expects an answer as soon as possible, perhaps so she can exploit the quick-moving logged phenomenon that caused her to write the query in the first place. *3)* Queries are *highly selective*: the vast majority of the log data will be irrelevant to the user. The user is primarily interested in a small number of very relevant rows in the log. Thus, although our system offers projections, it is *not* designed primarily for the large aggregations that motivate columnar storage systems. *4)* Queries may *entail many equality tests*: we believe that when querying logs, it will be especially useful for query authors to search a field for a large number of constants. For example, imagine the user wants to see all log entries from a list of suspicious users:

```
SELECT $1,$2,$3 WHERE $3='user1'
  OR $3='user2' OR $3='user3' OR ...
```

or imagine a website administrator wants to examine latency statistics from a handful of "problem URLs":

```
SELECT $1,$4, WHERE $1='/foo.html'
  OR $1='/bar.html' OR ...
```

If we assume the list of string constants—the set of usernames or the set of problematic URLs—is derived from a relation, these queries can be thought of as implementing a semijoin between a column of data in the log and a notional relation from elsewhere [10]. This use case is so common that we have explicit support for it in both the query language and the execution runtime. For example, for a query logically equivalent to the one above, a user can more compactly write:

```
SELECT $1,$4 WHERE $1={"problemurls.txt"}
```

When integrating HAWK with the software stack and interacting with the user, we envision at least two possible scenarios. The first usage scenario involves close integration with a data management tool. When the database engine encounters an ad hoc query, the query is handed off to the accelerator for processing, potentially freeing up the server cores for other processing tasks. Once the accelerator has completed execution, it returns pointers in memory to the concrete results. The database then retakes control and examines the results either for further processing (such as aggregation) or to return to the user. This scenario can be generalized to include non-database text processing software, such as *grep* and *awk*.

The second usage scenario involves a stand-alone deployment, in which a user submits queries directly to the accelerator (via a minimal systems software interface) and the accelerator returns responses directly to the user. In either case, the RDBMS software and the user do not interact directly with the hardware. Rather, they use the hardware-specific query compiler we describe in Section V-A.

B. Regular Expression Parsing

Processing regular expressions is not a core goal for our design: regular expressions may not be required for many log processing tasks, and our hardware-based approach does not lend itself to the potentially deep stacks that regex repetitions enable. The hardware natively supports exact string comparisons including an arbitrary number of single-character wildcards. However, it is possible to build a complete regular expression processing system on top of our proposed mechanism—HAWK can be used to implement all of the equality testing driven components of the regular expression, and strings that pass this "prefilter" can then be examined with a more traditional software stack for full regex processing.

C. Conventional Solutions

Today, scan operations like those we consider are typically processed entirely in software, and often using inverted indexes. Simple text processing is often performed with command-line tools like *grep* and *awk*, while more complex scan predicates are more efficiently processed in column-store relational databases, such as *MonetDB* [17] and *Vertica* [15]. Keyword search is typically performed using specialized tools with pre-computed indexes, such as *Lucene* [18] or the *Yahoo S4* framework [21].

However, software-implemented scans fall well short of the theoretical peak memory bandwidth available on modern hardware because scan algorithms must execute numerous instructions (typically tens, and sometimes hundreds) per byte scanned. Furthermore, conventional text scanning algorithms require large state transition table data structures that cause many cache misses. For our design goal of 32GC/s, and a target accelerator clock frequency of 1GHz, our system must process 32 characters each clock cycle. Given a conventional core's typical processing rates of at most a few instructions per cycle, and many stalls due to cache misses, we would potentially require hundreds of cores to reach our desired level of performance. Indexes are clearly effective, but are also time-consuming and burdensome to compute. Traditional index generation is expensive in time and memory.

Hardware-based solutions have been marketed for related applications, for example, IBM Netezza's data analytics appliances, which make use of FPGAs alongside traditional compute cores to speed up data analytics [13]. Our accelerator design could be deployed on such an integrated FPGA system. Some data management systems have turned to graphics processing units (GPUs) to accelerate scans. However, prior work has shown that GPUs are ill-suited for string matching problems [35], as these algorithms do not map well to the *single instruction multiple thread (SIMT)* parallelism offered by GPUs. Rather than rely on SIMT parallelism, our accelerator, instead, is designed to efficiently implement the finite state automata that underlie text scans; in particular, our accelerator incurs no stalls and avoids cache misses.

In short, existing software and hardware solutions are unlikely to reach our goal of fully saturating memory bandwidths during scan—the most promising extant solution is perhaps the FPGA-driven technique. Therefore, the main topic of this paper is how we can use dedicated hardware to support the aforementioned query language at our target processing rate.

III. Background

We briefly describe the classical algorithm for scanning text corpora, on which HAWK is based. The **Aho-Corasick algorithm** [4] is a widely used approach for scanning a text corpus for multiple search terms or *patterns* (denoted by the set S). Its asymptotic running time is linear in the sum of the searched text and pattern lengths. The algorithm encodes all the search patterns in a finite automaton that consumes the input text one character at a time.

The Aho-Corasick automaton *M* is a 5-tuple $(Q, \alpha, \delta, q_0, A)$ comprising:

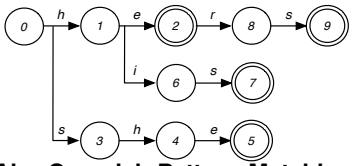


Fig. 2: An Aho-Corasick Pattern Matching Automaton. Search patterns are *he*, *she*, *his*, and *hers*. States *2*, *5*, *7*, and *9* are accepting states.

1) A finite set of states Q: Each state q in the automaton represents the longest prefix of patterns that match the recently consumed input characters.
2) A finite alphabet α.
3) A transition function ($\delta : Q \times \alpha \rightarrow Q$): The automaton's transition matrix comprises two sets of edges, which, together, are closed over α. The *goto function* $g(q, \alpha_i)$ encodes transition edges from state q for input characters α_i, thereby extending the length of the matching prefix. These edges form a trie (prefix tree) of all patterns accepted by the automaton. The *failure function* $f(q, \alpha_i)$ encodes transition edges for input characters that do not extend a match.
4) A start state $q_0 \in Q$, or the *root node*.
5) A set of accepting states A: A state is accepting if it consumes the last character of a pattern. An *output function output(q)* associates matching patterns with every state q. Note that an accepting state may emit multiple matches if several patterns share a common suffix.

Figure 2 shows an example of an Aho-Corasick trie for the patterns 'he', 'she', 'his' and 'hers' (failure edges are not shown for simplicity).

Two challenges arise when seeking to use classical Aho-Corasick automata to meet our performance objective: (1) achieving deterministic lookup time, and (2) consuming input fast enough. To aid in our description of these challenges, we leverage the notation in Table I.

Parameter	*Symbol*
Alphabet	α
Set of search patterns	S
Set of states in pattern matching automaton	Q
Characters evaluated per cycle (accelerator width)	W

TABLE I: Notation.

Deterministic lookup time — A key challenge in implementing Aho-Corasick automata lies in the representation of the state transition functions, as various representations trade off space for time. The transition functions can be compactly represented using various tree data structures, resulting in lookup time logarithmic in the number of edges that do not point to the root node (which do not need to be explicitly represented). Alternatively, the entire transition matrix can be encoded in a hash table, achieving amortized constant lookup time with a roughly constant space overhead relative to the most compact tree. However, recall that our objective is to process input characters at a constant rate, without any possibility of stalls in the hardware pipeline. We require deterministic time per state transition to allow multiple automata to operate in lockstep on the same input stream. (As will become clear later, operating

multiple automata in lockstep on the same input is central to our design). Hence, neither logarithmic nor amortized constant transition time are sufficient.

Deterministic transition time is easily achieved if the transition function for each state is fully enumerated as a lookup table, provided the resulting lookup table is small enough to be accessed with constant latency (e.g., by loading it into an on-chip scratchpad memory). However, this representation results in an explosion in the space requirement; the required memory grows with $O(|\alpha| \cdot |Q| \cdot log(|Q|))$. This storage requirement rapidly outstrips what is feasible in dedicated on-chip storage; to achieve a 1GHz access frequency, transition tables must fit within tens of kilobytes of storage (comparable to a core's L1 cache capacity). Storing transition tables in cacheable memory, as in a software implementation, again leads to non-deterministic access time.

Consuming multiple characters — A second challenge arises in consuming input characters fast enough to match our design target of 32GC/s. If only one character is processed per state transition, then the automaton must process state transitions at 32GHz. However, no feasible memory structure can be randomly accessed to determine the next state at this rate. Instead, the automaton must consume multiple characters in a single transition. The automaton can be reformulated to consume the input W characters at a time, resulting in an input alphabet size of $|\alpha|^W$. However, this larger alphabet size leads to intractable hardware—storage requirements grow due to an increase in the number of outgoing transitions per state on the order of $O(|\alpha|^W \cdot \log_2 |Q|)$. Moreover, the automaton must still accept patterns that are arbitrarily aligned with respect to the window of W bytes consumed in each transition. Accounting for these alignments leads to $|Q| = O(|S| \cdot W)$ states. Hence, storage scales exponentially with W as $O(|S| \cdot W \cdot |\alpha|^W \cdot \log_2(|S| \cdot W))$.

HAWK uses a representation of Aho-Corasick automata that addresses the aforementioned challenges. In the next section, we discuss the principle of HAWK's operation, and detail the corresponding hardware design.

IV. HAWK in Principle

We now describe our proposed system for scanning text at rates that meet or exceed memory bandwidth.

A. Preliminaries

Recall that we propose a *fixed scan rate* system, meaning that the amount of input processed each clock cycle is constant: HAWK has no pipeline stalls or variable-time operations. Since semiconductor manufacturing technology will limit our clock frequency (we target a 1GHz clock), the only way to obtain arbitrary scanning capacity with our design is to increase the number of characters processed each clock cycle.

There are multiple possible deployment settings for our architecture: integrating into existing server systems as an on-chip accelerator (like integrated GPUs), or as a plug-in replacement for a CPU chip, or "programmed" into reconfigurable logic in a CPU-FPGA hybrid [7]. The most appropriate packaging depends on workload and manufacturing technology details that are outside the scope of this paper.

An *accelerator instance* is a sub-system of on-chip components that processes a compiled query on a single text stream. It is possible to build a system comprising multiple accelerator instances to scale processing capability; we explore this design space. We define an accelerator instance's *width* W as the number of characters processed per cycle; an instance that processes one character per cycle is called *1-wide*, and an instance that processes 32 characters per cycle is called *32-wide*. Thus, for a target scan rate of 32GB/s, and a 1GHz clock, we could deploy either a single *32-wide* accelerator instance, or 32 *1-wide* accelerator instances. When deploying HAWK, an architect must decide *how many* accelerator instances should be manufactured, and of *what width*.

A common technique in data management systems is *data sharding*, in which the target data (in this case, the log text) is split over many processing elements and processed in parallel. Our architecture allows for data sharding—in which each accelerator instance independently processes a separate shard of the log text, sharing available memory bandwidth—but it is not the primary contribution of our work. More interestingly, our architecture enables *automata sharding*, in which the user's query is split over multiple accelerator instances processing a single input text stream in lockstep. Automata sharding enables HAWK to process queries of increasing complexity (i.e., increasing numbers of distinct search patterns) despite fixed hardware resources in each accelerator instance. HAWK is designed to make automata sharding possible.

B. Key Idea

The key idea that enables HAWK to achieve wide, fixed-rate scanning is our reformulation of the classic Aho-Corasick automaton to process W characters per step with tractable storage. As previously explained, simply increasing the input alphabet to $|\alpha|^W$ rapidly leads to intractable automata. Instead, we extend the concept of *bit-split pattern matching automata* [29] to reduce total storage requirements and partition large automata across multiple, small hardware units. Tan and Sherwood propose splitting a byte-based ($|\alpha| = 2^8 = 256$) Aho-Corasick automaton into a vector of eight automata that each process a single bit of the input character. Each state in the original automaton thus corresponds to a vector of states in the bit-split automata. Similarly, each bit-split state maps to a set of patterns accepted in that state. When all eight automata accept the same pattern, a match is emitted.

Bit-split automata conserve storage in three ways. First, the number of transitions per state is drastically reduced to two, making it trivial to store the transition matrix in a lookup table. Second, reduced fan-out from each state and skew in the input alphabet (ASCII text has little variation in high-order bit positions) results in increased prefix overlap. Third, the transition function of each automaton is distinct. Hence, the automata can be partitioned in separate storage and state IDs can be reused across automata, reducing the number of bits required to distinguish states.

Our contribution is to extend the bit-split automata to process W characters per step. Instead of the eight automata

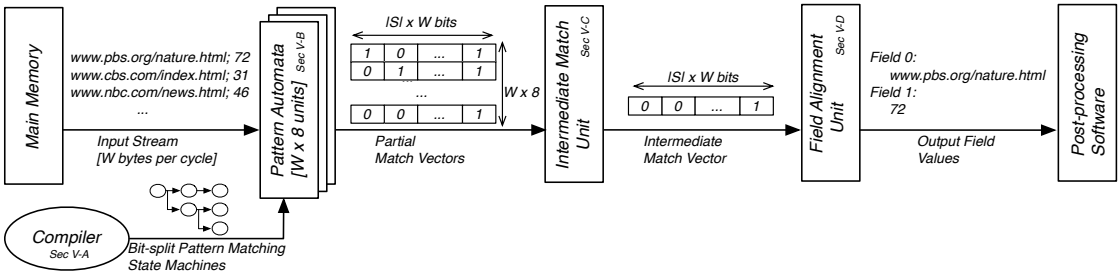


Fig. 3: Block diagram of the accelerator architecture.

that would be used in the bit-split setting (one automaton per bit in a byte), our formulation requires $W \times 8$ automata to process W characters per step. Increasing W introduces the new challenge of addressing the alignment of patterns with respect to the W-character window scanned at each step; we cover this issue in detail in later sections. Extending the bit-split approach to $W > 1$ results in exponential storage savings relative to widening conventional byte-based automata. The number of states in a single-bit machine is bounded in the length of the longest search term L_{max}. Since the automaton is a binary tree, the total number of nodes cannot exceed $2^{L_{max}+1} - 1$. The key observation we make is that the length of the longest search pattern is divided by W, so each bit-split automaton sees a pattern no longer than $\frac{L_{max}}{W} + P$, with P being at most two characters added for alignment of the search term in the W-character window. We find $|Q|$ for a single bit machine scales as $O(2^{[\frac{L_{max}}{W}+P+1]}) = O(1)$ in W. The storage in the bit-split automata grows as $O(|S| \cdot W)$ to overcome the aforementioned alignment issue (reasons for this storage increase will become clear in subsequent sections). With $W \times 8$ bit-split machines, the total storage scales as $O(8 \cdot |S| \cdot W^2)$, thereby effecting exponential storage savings compared to the byte-based automaton.

C. Design Overview

We now describe HAWK in detail. Figure 3 shows a high-level block diagram of a HAWK system. At query time, the system compiles the user's query and sends the compiled query description to each accelerator instance. Each instance then scans the in-memory text log as a stream, constantly outputting matches that should be sent to higher-level software components for further processing (say, to display on the screen or to add to an aggregate computation).

The major components of our design are:

- A *compiler* that transforms the user's query into a form the hardware expects for query processing—a set of *bit-split pattern matching automata*. These automata reflect the predicates in the user's query.
- *Pattern automaton* hardware units that maintain and advance the bit-split automata. At each cycle, each pattern automaton unit consumes a single bit of in-memory text input. Because each automaton consumes only one bit at a time, it cannot tell by itself whether a pattern has matched. After consuming a bit, each automaton emits a *partial match vector (PMV)* representing the set of patterns that *might* have matched, based on the bit and the automaton's current state. For an accelerator instance of width *W*, there are *W*×*8* pattern automaton units. For a query of $|S|$ patterns, the PMV requires $|S|$×*W* bits.
- The *intermediate match* hardware unit consumes PMVs from the pattern automata processing each bit position to determine their intersection. At each clock cycle, the intermediate match unit consumes *W*×*8* PMVs, performing a logical AND operation over the bit-vectors to produce a single *intermediate match vector (IMV)* output. The *IMV* is the same length as the PMVs: $|S|$×*W*.
- Finally, the *field alignment* unit determines the field within which each match indicated by the IMV is located. Pattern matching in all of the preceding steps takes place without regard to delimiter locations, and therefore, of fields and records in the input log file. This *after-the-fact mapping of match locations to fields*, which is a novel feature of our design, allows us to avoid testing on field identity during pattern matching, and thereby avoids the conditionals and branch behavior that would undermine our fixed-rate scan design. If the field alignment unit finds that the IMV indicates a match for a field number that the user's query requested, then it returns the resulting *final match vector* (FMV) to the database software for post-processing. To simplify our design, we cap the number of fields allowed in any record to 32—a number sufficient for most real-world log datasets.

Note that each accelerator instance supports searching for 128 distinct patterns. Therefore, a device that has 32 *1-wide* accelerator instances can process up to 32×128 patterns, a device with 16 *2-wide* instances can process up to 16×128 distinct patterns, and a device with a single *32-wide* instance can process up to 1×128 distinct patterns. By varying the number of instances and their width, the designer can trade off pattern constraints, per-stream processing rate, and, as we shall see later, area and power requirements (see Section VI-C).

V. HAWK Architecture

We now describe the four elements of HAWK highlighted in Figure 3 in detail.

A. Compiler

HAWK first compiles the user's query into pattern-matching automata. Figure 4 conceptually depicts compilation for a 4-wide accelerator. Algorithm 1 lists the compilation algorithm. The compiler's input is a query in the form described in

Algorithm 1 The multicharacter bit-split pattern matching automata compilation algorithm.

Input: Query $\mathbf{K}$ and architecture width W
Output: Bit split automata set M.

```
1: S = shard( sort( ⋃ predicates(K) ) )
2: S' = []
3: for each s ∈ S do
4:     for i = 1 to W do
5:         S'.append(pad(s, i, W))
6:     end for
7: end for
8:
9: Automata set M = {}
10: for each s ∈ S' do
11:     for i = 0 to len(s) do
12:         for bit b ∈ s[i] do
13:             M[i MOD W].addNode(b)
14:         end for
15:     end for
16: end for
17:
18: for each m ∈ M do
19:     makeDFA(m)
20:     for each q ∈ M.states do
21:         makePMV(q)
22:     end for
23: end for
```

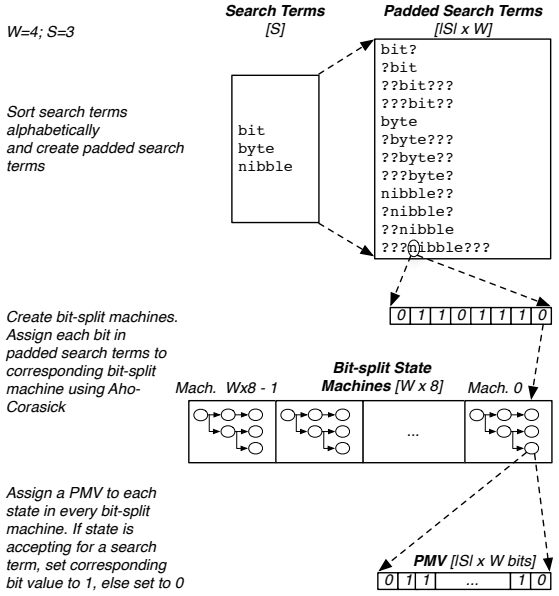


Fig. 4: Three-step compiler operation for a 4-wide accelerator and three search terms (W=4, S=3).

Section II. After parsing the query, the compiler determines the set of all patterns S, which is the union of the patterns sought across all fields in the WHERE clause. S is sorted lexicographically and then sharded across accelerator instances (Line 1). Sharding S lexicographically maximizes prefix sharing within each bit-split automaton, reducing their sizes.

Next, the compiler must transform S to account for all possible alignments of each pattern within the W-character window processed each cycle. The compiler forms a new set S' wherein each pattern in S is padded on the front and back with wildcard characters to a length that is a multiple of W, forming W patterns for all possible alignments with respect to the W-character window (Lines 2-7). Figure 4 shows an example of this padding for S={bit, byte, nibble} and W=4. For a machine where W=1, there is just one possible pattern alignment in the window; no padding is required.

The compiler then generates bit-split automata for the padded search patterns in S' according to the algorithm proposed by Tan and Sherwood [29] (Lines 9-16). A total of W×8 such automata are generated, one per input stream bit processed each cycle. Since each state in these automata has only two outgoing edges, the transition matrix is easy to represent in hardware. Automata are encoded as transition tables indexed by the state number. Each entry is a 3-tuple comprising the next state for input bits of zero and one and the PMV for the state. Each state's PMV represents the set of padded patterns in S' that are accepted by that automaton in that state. The compiler assigns each alignment of each pattern a distinct bit position in the PMV (Line 21). It is important to note that the hardware does not store S' directly. Rather, patterns are represented solely as bits in the PMV.

Accelerator Width (W)	*1*	*2*	*4*	*8*	*16*	*32*
Per Bit-split Machine Storage (KB)	74.8	69.6	33.5	16.5	16.4	32.8
Total Storage (MB)	0.6	1.11	1.07	1.06	2.1	8.4

TABLE II: Provisioned storage.

B. Pattern Automata

The *pattern automata* (first panel of Figure 5) each process a single bit-split automaton. Each cycle, they each consume one bit from the input stream, determine the next state, and output one PMV indicating possible matches at that bit position. Consider the pattern automaton responsible for bit 0 of the W×8-bit input stream (from Figure 5). In *cycle 0*, the automaton's current state is 0. The combination of the current state and the incoming bit value indicates a lookup table entry; in this case, the incoming bit value is 0, so the lookup table indicates a next state of 1. The pattern automaton advances to this state and emits its associated PMV to the intermediate match unit for processing in the next cycle.

The transition table and PMV associated with each state are held in dedicated on-chip storage. We use dedicated storage to ensure each pattern automaton can determine its next state and output PMV at a 1GHz frequency. (Accesses may be pipelined over several clock cycles, but, our implementation requires only a single cycle at 1GHz). We determine storage requirements for pattern automata empirically. We select 128 search terms at random from an English dictionary and observe the number of states generated per automaton. We then round the maximum number of states required by any automaton to the next power of 2, and provision this storage for all automata. (Note that if the query workload were to systematically include longer strings, such as e-commerce URLs, then storage requirements would be correspondingly higher.)

Table II shows the *per-automaton* and *total* storage allocation for a range of accelerator widths. Importantly, the storage per pattern automaton is comparable to a first-level data cache of a conventional CPU, which must support a similar access frequency. We observe a few interesting trends. First, the per-automaton-storage is minimal for W=8 and W=16. Whereas the number of patterns grows with W (a consequence of our

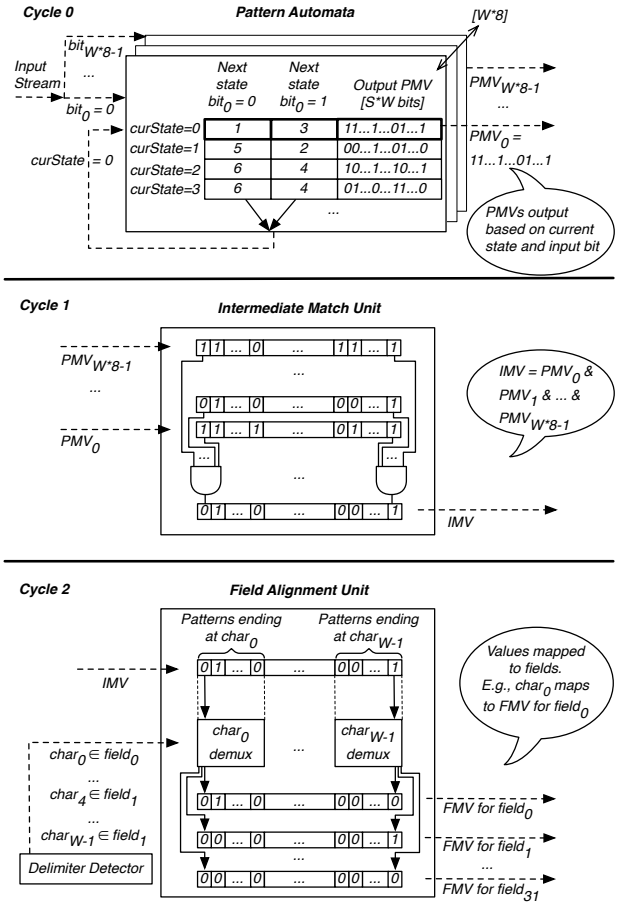


Fig. 5: Operation of the major string matching subunits.

padding scheme), the number of states in each automaton shrinks due to an effective reduction in pattern length (a consequence of processing multiple characters simultaneously). Simultaneously, as the number of patterns grows, the PMV width increases. The reduction in states dominates the larger PMV widths until W=16, after which the impact of increased PMV widths starts to dominate. Note that we conservatively provision the same storage for all automata, despite the fact that ASCII is highly skewed and results in far more prefix sharing in high-order bit positions. This decision allows HAWK to support non-ASCII representations and ensures symmetry in the hardware, which facilitates layout.

C. Intermediate Match Unit

The *intermediate match* unit (middle panel of Figure 5) calculates the intersection of the PMVs. A pattern is present at a particular location in the input stream only if it is reported in the PMVs of *all* pattern automata. The intermediate match unit is a wide and deep network of AND gates that computes the conjunction of the W×8 $|S| \times W$-bit PMVs. The result of this operation is a $|S| \times W$-bit wide intermediate match vector (IMV). As with the pattern automata, the intermediate match unit's execution can be pipelined over an arbitrary number of clock cycles without impacting the throughput of the accelerator, but our 32-wide ASIC implementation requires only a single cycle. In our FPGA prototype, we integrate the pattern automata and intermediate match unit and pipeline them over 32 cycles; this simplifies delay balancing across pipeline stages. Figure 5 shows that the PMVs generated by the pattern automata in *cycle 0* are visible to the intermediate match unit in *cycle 1*. The intermediate match unit performs a bitwise AND operation on all W×8 $|S| \times W$-bit PMVs and yields an IMV. In our example, the second and last bits of all PMVs are set, indicating that the padded patterns corresponding to these entries have been matched by all bit-split automata (i.e., true matches). The intermediate match unit, therefore, outputs an IMV with these bits set.

D. Field Alignment Unit

HAWK's operation so far has ignored the *locations* of matches between the log text and the user's query; it can detect a match, but cannot tell whether the match is in the correct field. The *field alignment* unit (bottom panel of Figure 5) reconstructs the association between pattern matches and fields. The output of the field alignment unit is an array of field match vectors (FMVs), one per field. Each FMV has a bit per padded pattern that allows the determination of the exact location of the matching pattern within the input stream; i.e., bit i in FMV j indicates whether pattern i matches field j and the pattern's location within the input stream.

The field alignment unit receives two inputs. The first input is the $|S| \times W$-bit IMV output from the intermediate match unit. This vector represents the patterns identified as true matches. The second input comes from a specialized *delimiter detector* that is preloaded with user-specified delimiter characters. (The hardware design for the delimiter detector is straightforward and is not detailed here for brevity. It is essentially a simple single-character version of pattern matching.) Each cycle, the delimiter detector emits a field ID for every character in the W-character window corresponding to the current IMV (overall, W field IDs). Search patterns that end at a particular character location belong to the field indicated by the delimiter detector. Recall that bit positions in the PMVs (and hence, the IMV) identify the end-location of each padded search pattern within the current W-character window (see Section V-A). Thus for every end-location, the field alignment unit maps corresponding IMV bits to the correct field ID, and the respective FMV. The operation of the field alignment unit is a demultiplexing operation (see Figure 5).

In *cycle 2*, the field alignment unit evaluates the window previously processed by the pattern automata and the intermediate match unit. In our example, the IMV's second and last bits are set; i.e., the patterns ending at character$_0$ and character$_{W-1}$ have matched in *some* fields. The delimiter detector indicates that character$_0$ is in field$_0$, and character$_{W-1}$ is in field$_1$. Thus, the patterns ending at character$_0$ are mapped to the FMV for field$_0$, and the patterns ending at character$_{W-1}$ are mapped to the FMV for field$_1$. The mapped FMVs are subsequently sent to the post-processing software.

The field alignment unit hardware entails 32 AND operations for each bit of the IMV. Compared to the pattern matching automata, area and power overheads are minor.

VI. Experimental Results

Processor	Dual socket Intel E5630
	16 threads @ 2.53 GHz
Caches	256 KB L1, 1 MB L2, 12 MB L3
Memory Capacity	128 GB
Memory Type	Dual-channel DDR3-800
Max. Mem. Bandwidth	12.8 GB/s

TABLE III: Server specifications.

We utilize three evaluation metrics for HAWK. The most straightforward is query processing performance when compared to conventional solutions on a modern server. The other metrics are HAWK's area and power requirements, constraints extremely important to chip designers. We will show that when given hardware resources that are a fraction of those used by a Xeon chip, an ASIC HAWK can reach its goal of 32GC/s and can comfortably beat conventional query processing times, sometimes by multiple orders of magnitude. Furthermore, we validate the HAWK design through proof-of-concept implementation in an FPGA prototype with scaled down frequency and width and demonstrate that even this drastically downscaled design still can outperform software.

A. Experimental Setup

We compare HAWK's performance against four traditional text querying tools: *awk*, *grep*, *MonetDB* [17], and *Lucene* [18]. We run all conventional software on a Xeon-class server (see Table III). We preload datasets into memory, running an initial throwaway experiment to ensure data is hot. We repeat all experiments five times and report average performance.

We implement a HAWK ASIC in the Verilog hardware description language. Fabricating an actual ASIC is beyond the scope of a single paper; instead, we estimate performance, area, and power of the ASIC design using Synopsys' DesignWare IP suite [28], which includes tools that give timing, area, and power estimates. (Synthesis estimates of area and power are part of conventional hardware design practice). Synthesizing an ASIC design entails choosing a target manufacturing technology for the device. We target a commercial 45nm manufacturing technology with a nominal operating voltage of 0.72V, and design for a clock frequency of 1GHz. The details are less important than the observation that this technology is somewhat out of date; it is two generations behind the manufacturing technology used in the state-of-the-art Xeon chip for our conventional software performance measurements. However, the 45nm technology is the newest ASIC process to which we have access. Since power and area scale with the manufacturing technology, we compare HAWK's power and area against a prior-generation Intel processor manufactured at the same 45nm technology node as HAWK[2].

The FPGA HAWK prototype is tested on an Altera Arria V development platform. Due to FPGA resource constraints, we build a single 4-wide HAWK accelerator instance. We use the block RAMs available on the FPGA to store the state transition tables and PMVs of the pattern matching automata. In the aggregate, the automata use roughly half of these RAMs; there are insufficient RAMs for an 8-wide accelerator instance. Because of global wiring required to operate the distributed RAMs, we restrict clock frequency to 100MHz. Thus, the prototype achieves a scan rate of 400MB/s. Because of limited memory capacity and overheads in accessing off-chip memory on our FPGA platform, we instead generate synthetic log files directly on the FPGA. Our log generator produces a random byte-stream (via a linear feedback shift register) and periodically inserts a randomly selected search term from a lookup table[3]. We validate that the accelerator correctly locates all matches.

The HAWK compiler is written in C. Relative to query execution time, compilation time is negligible. Since the primary focus of this paper is on string pattern matching, our compiler software does not currently handle numeric fields automatically; we compile numeric queries by hand.

Our evaluation considers three example use cases for HAWK that stress various aspects of its functionality. In each case, we compare to the relevant software alternatives.

1) Single Pattern Search: We first consider the simplest possible task: a scan through the input text for a single, fixed string. We generate a synthetic 64GB dataset comprising 100-byte lines using the text log synthesis method described by Pavlo [23]. We formulate the synthetic data to include target strings that match a notional user query with selectivities of 10%, 1%, 0.1%, 0.01%, and 0.001%. We time the queries needed to search for each of these strings and report matching lines. We compare HAWK against a relational column-store database (*MonetDB*) and the UNIX *grep* tool. For *MonetDB*, we load the data into the database prior to query execution.

2) Multiple Pattern Search: Next, we consider a semijoin-like task, wherein HAWK searches for multiple patterns in a real-world dataset, namely, the Wikipedia data dump (49 GB). We select patterns at random from an English dictionary; we vary their number from one to 128. We compare against an inverted text index query processor (*Lucene*) and *grep*. For *Lucene*, we create the inverted index prior to query execution; indexing time is not included in the performance comparison. *Lucene* and *grep* handle certain small tokenization issues differently; to ensure they yield identical search results, we make some small formatting changes to the input Wikipedia text. We execute *grep* with the *-Fw* option, which optimizes its execution for patterns that contain no wildcards.

3) Complex Predicates: Finally, we consider queries on a webserver-like log of the form <*Source IP, Destination URL, Date, Ad Revenue, User Agent, Country, Language, Search Word, Duration*>. This dataset is also based on a format proposed by Pavlo [23]. A *complex query* has selection criteria for multiple columns in the log. It takes the following form[4]:

```
SELECT COUNT(*) FROM dataset WHERE (
  (Date in specified range)
  AND (Ad Revenue within range)
  AND (User Agent LIKE value2 OR ...)
```

[2]We measure software performance on the more recent Xeon E5630 chip listed in Table III; the 45nm Xeon W5590 is used only for area and power comparisons.

[3]We pursue this methodology since HAWK's performance is independent of the characteristics of both the input stream and search patterns.

[4]We use COUNT so *MonetDB* does not incur extra overhead in returning concrete result tuples, but rather incurs only trivial aggregation costs.

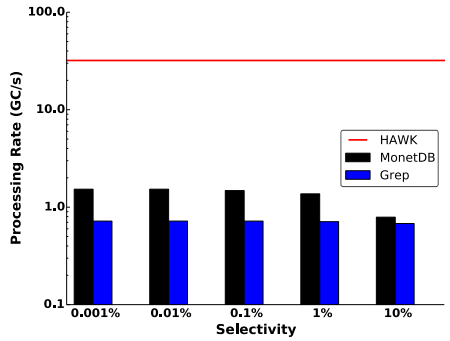


Fig. 6: Query performance for the single pattern search task on synthetic data, across varying selectivities.

```
AND (Country LIKE value4 OR Country LIKE ...)
AND (Language LIKE value6 OR Language LIKE ...)
AND (Search Word LIKE value8 ...)
AND (Duration within range)).
```

We tune the various query parameters to achieve selectivities of 10%, 1%, 0.1%, 0.01%, and 0.001%. We compare against equivalent queries executed with the relational column-store (*MonetDB*) and the UNIX tool *awk*.

B. Performance

We contrast the performance of HAWK to various software tools in GC/s. By design, the HAWK ASIC always achieves a performance of 32GC/s (0.4GC/s for the FPGA); due to conscious design choices, there is no sensitivity to query selectivity or the number of patterns with HAWK, provided the query fits within the available automaton state and PMV capacity. In contrast, software tools show sensitivity to both these parameters, so we vary them in our experiments.

1) Single Pattern Search: Figure 6 compares HAWK's single pattern search performance against *MonetDB* and *grep*. HAWK's constant 32GC/s performance is over an order of magnitude better than either software tool, and neither comes close to saturating memory bandwidth. *MonetDB*'s performance suffers somewhat when selectivity is high (above 1%), but neither *grep* nor *MonetDB* exhibit much sensitivity at lower selectivities.

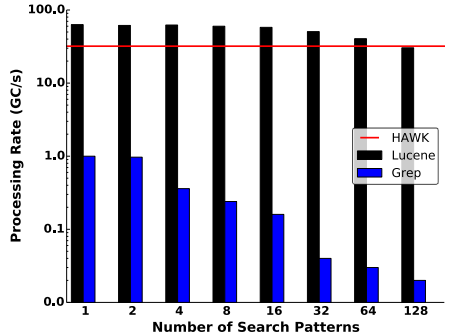


Fig. 7: Query performance on real-world text data, for varying numbers of search patterns.

2) Multiple Pattern Search: Figure 7 compares HAWK against *Lucene* and *grep* when searching for multiple randomly-chosen words in the Wikipedia dataset. For *Lucene*, we explore query formulations that search for multiple patterns in a single query or execute separate queries in parallel and report the best result.

Grep's performance is poor: its already poor performance for single-pattern search (1GC/s) drops precipitously as the number of patterns increases, to as little as 20 megacharacters/s in the 128-word case. Unsurprisingly, because it uses an index and does not actually scan the input text, *Lucene* provides the highest performance. We report its performance by dividing query execution time by the size of the data set to obtain an equivalent GC/s scan rate. Note that this equivalent scan rate exceeds available memory bandwidth in many cases (i.e., no scan-based approach can reach this performance).

Remarkably, however, our results show that, when the number of patterns is large, a HAWK ASIC is competitive with Lucene *even though HAWK does not have access to a precomputed inverted index*. In the 128-pattern case, *Lucene's* performance of 30.4GC/s falls short of the 32GC/s performance of HAWK. At best, *Lucene* outperforms HAWK by a factor of two for this data set size (its advantage may grow for larger data sets, since HAWK's runtime is linear in the dataset size). Of course, these measurements do not include the 30 minutes of pre-query processing time that *Lucene* requires to build the index. Our result demonstrates that scan-based query execution can be performance-competitive with pre-computed indexes for RAM-resident text corpora.

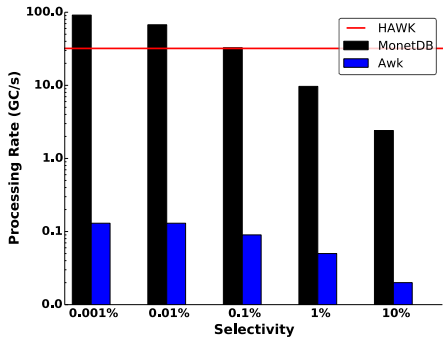
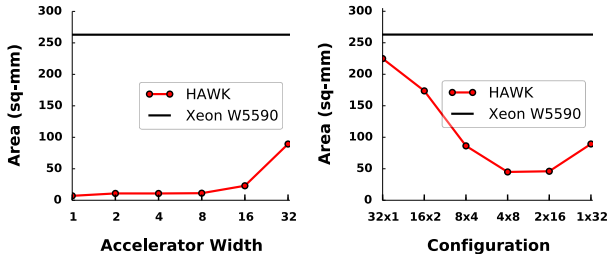


Fig. 8: Query performance for complex predicates task, across varying selectivities.

3) Complex Predicates: Figure 8 compares HAWK, *MonetDB*, and *awk* on the complex queries described in Section VI-A3. *MonetDB* performance spans a 45× range as selectivity changes from 10% to 0.001%. When selectivity is low, *MonetDB* can order the evaluation of query predicates to rapidly rule out tuples, avoiding the need to access most data. For 0.001% selectivity, it outperforms HAWK by 3×. However, for queries that admit more tuples (i.e., where *MonetDB* must more frequently examine large text fields), HAWK provides superior performance, with more than 10× advantage at 10% selectivity. The performance of *awk* is not competitive.

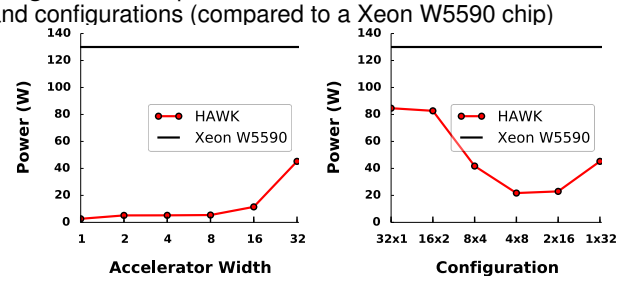
C. ASIC Area and Power

We report a breakdown of an ASIC HAWK instance's per-sub-component area and power estimates for two extreme



(a) Single-Unit (b) Multi-Unit, 32GC/s

Fig. 9: Area requirements for various accelerator widths and configurations (compared to a Xeon W5590 chip)



(a) Single Unit (b) Multi-Unit 32GC/s

Fig. 10: Power requirements for various accelerator widths and configurations (compared to a Xeon W5590 chip).

design points, *1-wide* and *32-wide*, in Table IV. For both designs, the pattern automata account for the vast majority of area and power consumption. Pattern automata area and power are dominated by the large storage structures required for the state transition matrix and PMVs. We can see here the impact that state machine size has on the implementation. Even with the drastic savings afforded by the bit-split technique, the automata storage requirements are still large; without the technique, they would render the accelerator impractical.

	1-wide		*32-wide*	
Unit	*Area (mm^2)*	*Power (mW)*	*Area (mm^2)*	*Power (mW)*
Pattern Automata	5.7	2602	86	44,563
Intermediate Match Unit	< 0.1	< 1	< 1	35
Field Alignment Unit	< 1	14	1	448
Delimiter Detector	1.1	< 1	< 1	< 1
Numeric Units	< 0.1	1	< 1	39
Other Control Logic	0.2	26	1	146
Total	7.1	2644	89	45,231

TABLE IV: Component area and power needs for 1-wide and 32-wide configurations.

Figures 9 and 10 compare the area and power requirements of ASIC HAWK to an Intel Xeon W5590 chip [3], a chip in the same generation manufacturing technology as our synthesized design (45nm). We find that a 1-wide HAWK instance requires only 3% of the area and 2% of the power of the Xeon chip. A 32-wide HAWK requires 42% of the area and 35% of the power of the Xeon processor. Although these values are high, they would improve when using more modern manufacturing technology; a 32-wide HAWK instance might occupy roughly one-sixth the area of a modern server-class chip.

Figures 9 and 10 also reveal an interesting trend. The 8-wide (*4×8*) and 16-wide (*2×16*) HAWK configurations utilize resources more efficiently (better performance per area or watt) than other configurations. This saddle point arises due to two opposing trends. Initially, as width W increases from 1, the maximum padded pattern length (L_{max}) per bit-split automaton decreases rapidly. Since each bit-split automaton is a binary tree, lower L_{max} yields a shallower tree (i.e., fewer states) with more prefix sharing across patterns. Overall, the reduced number of states translates into reduced storage costs. However, as W continues to grow, L_{max} saturates at a minimum while the set of padded patterns, S', grows proportionally to $|S| \times W$. Each pattern requires a distinct bit in the PMV, which increases the storage cost per state. Above $W = 16$, the increased area and power requirements of the wide match vectors outweigh the savings from reduced L_{max}, and total resource requirements increase.

Overall, the 8-wide and 16-wide configurations strike the best balance between these opposing phenomena. It is more efficient to replace one 32-wide accelerator with four 8-wide accelerators or two 16-wide accelerators. The *4×8* configuration, which exhibits the lowest area and power costs, requires approximately 0.5× area and 0.48× power compared to the 32-wide accelerator, while maintaining the same performance. Compared to the W5590, the *4×8* configuration requires 0.21× the area and 0.17× the power. Four 8-wide accelerator instances (*4×8*) provide the best performance-efficiency tradeoff.

D. FPGA Prototype

We validate the HAWK hardware design through our FPGA prototype. As previously noted, the prototype is restricted to 4-wide accelerator instance operating at a 100MHz clock frequency, providing a fixed scan rate of 400MB/s. As with the ASIC design, the storage requirements of pattern automata dominate resource requirements on the FPGA. We program the accelerator instance to search for the same 64 search terms as in the multiple pattern search task described in Section VI-A2. Although it is 80× slower than our ASIC design, the FPGA prototype nevertheless remains faster than *grep* for this search task by 13×, as *grep* slows drastically when searching for multiple patterns. Whereas *grep* achieves nearly a 1GB/s scan rate for a single pattern, it slows to 30MB/s when searching for 64 terms. (Note that this is still faster than searching for the terms sequentially in multiple passes, but only by a small factor). With better provisioning of on-chip block RAMs, both the width and clock frequency of the FPGA prototype could be improved, increasing its advantage over scanning in software.

VII. Related Work

There are several areas of work relevant to HAWK.

String Matching — Multiple hardware-based designs have been proposed to accomplish multicharacter Aho-Corasick processing. Chen and Wang [9] propose a multicharacter transition Aho-Corasick string matching architecture using non-deterministic finite automata (NFA). Pao and co-authors [22] propose a memory-efficient pipelined implementation of the Aho-Corasick algorithm. However, neither work aims to meet or exceed available memory bandwidth. Some elements of our approach have been used in the past. Hua et al. [12] present a string matching algorithm that operates on variable-stride

blocks instead of single bytes; their work is inspired in part by how humans read text as patterns. van Lunteren et al. [16] use transition rules stored using balanced routing tables; this technique provides a fast hash lookup to determine next states. Bremler-Barr and co-authors [6] encode states such that all transitions to a specific state are represented by a single prefix that defines a set of current states. However, we are unaware of previous work that uses our approach of combining bit-split automata with multiple-character-width processing.

Processing Logs — Processing text logs is an important workload that has dedicated commercial data tools and is a common use case for distributed data platforms such as Hadoop and Spark. In-memory data management systems have also become quite popular [17], [25], [27], [34].

Databases and FPGAs — A large amount of research has focused on using FPGAs to improve database and text processing. Mueller et al. explore general query compilation and processing with FPGAs [20]. Teubner et al. propose *skeleton automata* for avoiding expensive FPGA compilation costs [31]. The project with goals most similar to our own is probably that of Woods et al. [32], who examine the use of FPGAs for detecting network events at gigabit speeds. Although this project also focuses on the problem of string matching, it has a lower performance target, does not have our fixed-processing rate design goal, and is technically distinct. IBM Netezza [13] is the best-known commercial project in this area.

VIII. Conclusion

High-velocity text log data have undergone explosive growth in recent years. Rapid improvement in RAM cost and capacity now make it feasible for large text corpora to reside entirely in memory, opening the possibility of scan-based query processing that is performance-competitive with pre-computed indexes. Conventional software scan mechanisms cannot fully exploit available memory bandwidth. We show that our HAWK accelerator can process data at a constant rate of 32GB/s, outperforming state-of-the-art solutions for text processing.

References

[1] Deloitte University Press: In-Memory Revolution. http://dupress.com/articles/2014-tech-trends-in-memory-revolution/.

[2] Intel: A Revolutionary Breakthrough in Memory Technology. http://www.intel.com/newsroom/kits/nvm/3dxpoint.

[3] Intel W5590 Specifications. http://ark.intel.com/products/41643.

[4] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*, 18(6), June 1975.

[5] S. Borkar and A. A. Chien. The Future of Microprocessors. *Communications of the ACM*, 54(5):67–77, May 2011.

[6] A. Bremler-Barr, D. Hay, and Y. Koral. CompactDFA: Generic State Machine Compression for Scalable Pattern Matching. *INFOCOM*, 2010.

[7] D. Bryant. *Disrupting the Data Center to Create the Digital Services Economy*. Intel Corporation, 2014.

[8] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-Time Search at Twitter. In *ICDE*, 2012.

[9] C.-C. Chen and S.-D. Wang. An Efficient Multicharacter Transition String-matching Engine Based on the Aho-corasick Algorithm. *ACM Transactions on Architecture and Code Optimization*, 2013.

[10] N. Doshi. *Using File Contents as Input for Search*. Splunk Blogs, 2009.

[11] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Intl. Symp. on Computer Architecture*, 2011.

[12] N. Hua, H. Song, and T. Lakshman. Variable-Stride Multi-Pattern Matching For Scalable Deep Packet Inspection. In *INFOCOM*, 2009.

[13] IBM Corporation. *IBM PureData System for Analytics Architecture: A Platform for High Performance Data Warehousing and Analytics*. 2010.

[14] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan. Meet the Walkers: Accelerating Index Traversals for In-memory Databases. In *Intl. Symp. on Microarchitecture*, 2013.

[15] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica Analytic Database: C-store 7 Years Later. *Proc. VLDB Endow.*, 2012.

[16] J. Lunteren, C. Hagleitner, T. Heil, G. Biran, U. Shvadron, and K. Atasu. Designing a Programmable Wire-Speed Regular-Expression Matching Accelerator. In *Intl. Symp. on Microarchitecture*, 2012.

[17] S. Manegold, M. L. Kersten, and P. Boncz. Database Architecture Evolution: Mammals Flourished Long Before Dinosaurs Became Extinct. *PVLDB*, 2009.

[18] M. McCandless, E. Hatcher, and O. Gospodnetic. *Lucene in Action*. Manning Publications, 2010.

[19] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. In *PVLDB*, 2010.

[20] R. Müller, J. Teubner, and G. Alonso. Data Processing on FPGAs. *PVLDB*, 2(1), 2009.

[21] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *ICDM Workshops*, 2010.

[22] D. Pao, W. Lin, and B. Liu. A Memory-Efficient Pipelined Implementation of the Aho-Corasick String-Matching Algorithm. *ACM Transactions on Architecture and Code Optimization*, 2010.

[23] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-scale Data Analysis. In *ACM SIGMOD*, 2009.

[24] M. E. Richard L. Villars, Carl W. Olofson. *Big Data: What It Is and Why You Should Care*. IDC, 2011.

[25] V. Sikka, F. Färber, A. K. Goel, and W. Lehner. SAP HANA: The Evolution from a Modern Main-Memory Data Platform to an Enterprise Application Platform. *PVLDB*, 6(11), 2013.

[26] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 Requirements of Real-time Stream Processing. *ACM SIGMOD Record*, 2005.

[27] M. Stonebraker and A. Weisberg. The VoltDB Main Memory DBMS. In *TCDE*, 2013.

[28] Synopsys. *DesignWare Building Blocks*. 2011.

[29] L. Tan and T. Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In *Intl. Symp. on Computer Architecture*, 2005.

[30] M. Taylor. Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. *DAC*, 2012.

[31] J. Teubner, L. Woods, and C. Nie. Skeleton Automata for FPGAs: Reconfiguring without Reconstructing. In *ACM SIGMOD*, 2012.

[32] L. Woods, J. Teubner, and G. Alonso. Complex Event Detection at Wire Speed with FPGAs. *PVLDB*, 2010.

[33] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. Q100: The Architecture and Design of a Database Processing Unit. *ASPLOS*, 2014.

[34] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.

[35] X. Zha and S. Sahni. GPU-to-GPU and Host-to-Host Multipattern String Matching on a GPU. *Computers, IEEE Transactions on*, 2013.