HARE: Hardware Accelerator for Regular Expressions

Vaibhav Gogte*, Aasheesh Kolli*, Michael J. Cafarella*, Loris D'Antoni† and Thomas F. Wenisch*
*University of Michigan
{vgogte,akolli,michjc,twenisch}@umich.edu
† University of Wisconsin-Madison
loris@cs.wisc.edu

***Abstract*—Rapidly processing text data is critical for many technical and business applications. Traditional software-based tools for processing large text corpora use memory bandwidth inefficiently due to software overheads and thus fall far short of peak scan rates possible on modern memory systems. Prior hardware designs generally target I/O rather than memory bandwidth. In this paper, we present HARE, a hardware accelerator for matching regular expressions against large in-memory logs. HARE comprises a stall-free hardware pipeline that scans input data at a fixed rate, examining multiple characters from a single input stream in parallel in a single accelerator clock cycle.**

We describe a 1GHz 32-character-wide HARE design targeting ASIC implementation that processes data at 32 GB/s—matching modern memory bandwidths. This ASIC design outperforms software solutions by as much as two orders of magnitude. We further demonstrate a scaled-down FPGA proof-of-concept that operates at 100MHz with 4-wide parallelism (400 MB/s). Even at this reduced rate, the prototype outperforms grep by 1.5-20× on commonly used regular expressions.

***Keywords* —** regular expression matching, text processing, finite automata

I. Introduction

Fast analysis of unstructured textual data, such as system logs, social media posts, emails, or news articles, is growing ever more important in technical and business data analytics applications [26]. Nearly 85% of business data is in the form of unstructured textual logs [5]. Rapidly extracting information from these text sources can be critical for business decision making. For instance, a business might analyze trends in social media posts to better target their advertising budgets.

Regular expressions (regexps) provide a powerful and flexible approach for processing text and unstructured data [3]. Historically, tools for regexp processing have been designed to match disk or network [21] bandwidth. As we will show, the most widely used regexp scanning tool, `grep`, typically achieves at most 100-300 MB/s scanning bandwidth on modern servers—a tiny fraction of available memory bandwidth. However, the wide availability of cheap DRAM and upcoming NVRAM [2] allows many important data corpora to be stored entirely in high-bandwidth memory. Data management systems are being redesigned for in-memory datasets [22], [31]. Text processing solutions, and especially regexp processing, require a similar redesign to match the bandwidth available in modern system architectures.

978-1-5090-3508-3/16/$31.00 © 2016 IEEE

Conventional software solutions for regexp processing are inefficient because they rely on finite automata [18]. The large transition tables of these automata lead to high access latencies to consume an input character and advance to the next state. Moreover, automata are inherently sequential [8]—they are designed to consume only a single input character per step. Straight-forward parallelization to multi-character inputs leads to exponential growth in the state space [19].

A common approach to parallelize regexp scans is to shard the input into multiple streams that are scanned in parallel on different cores [17], [23], [25]. However, the scan rate of each individual core is so poor (especially when scanning for several regexps concurrently) that even the large core counts of upcoming multicore server processors fall short of saturating memory bandwidth [33]. Moreover, such scans are highly energy inefficient. Other work seeks to use SIMD parallelism [11], [27] to accelerate regexp processing, but achieves only modest 2×-3× speedups over non-SIMD software.

Instead, our recent work on the HAWK text scan accelerator [33] has identified a strategy to scan text corpora using finite state automata at the full bandwidth of modern memory systems, and has been demonstrated for scan rates as high as 32 giga-characters per second (GC/s; 256 Gbit/s). HAWK relies on three ideas: (1) a fully-pipelined hardware scan accelerator that does not stall, assuring a fixed scan rate, (2) the use of bit-split finite state automata [32] to compress classic deterministic finite automata for string matching [7] to fit in on-chip lookup tables, and (3) a scheme to efficiently generalize these automata to process a window of characters each step by padding search strings with wildcards. We elaborate on these prior ideas in Section III.

HAWK suffers from two critical deficiencies: (1) it can only scan for *exact* string matches and fixed-length patterns containing single-character (`.`) wildcards, and (2) it is unable to process Kleene operators (+, *), alternation (|,?), and character classes (`[a-z]`), which are ubiquitous in practical text and network packet processing [3], [4]. These restrictions arise because HAWK's strategy for processing multiple input characters in each automaton step cannot cope with variable-length matches.

We propose *HARE*, the Hardware Accelerator for Regular Expressions, which extends the HAWK architecture to a broad class of regexps. HARE maintains HAWK's stall-free pipeline

design, operating at a fixed 32 GC/s scan rate, regardless of the regexps for which it scans or input text it processes. Similar to HAWK, we target a throughput of 32GB/s because it is a convenient power-of-two and representative of future DDR3 or DDR4 memory systems. HARE extends HAWK in two key ways. First, it supports character classes by adding a new pipeline stage that detects in which character classes the input characters lie, extending HAWK's bit-split automata with additional bits to represent these classes. Second, it uses a counter-based mechanism to implement regexp quantifiers, such as the Kleene Star (*), that match repeating characters. The combination of repetition and character classes presents a particular challenge when consecutive classes accept overlapping sets of characters, as some inputs may match an expression in multiple ways.

We evaluate HARE through:

- An ASIC RTL implementation of a stall-free HARE pipeline operating at 1GHz and processing 32 characters per cycle, synthesized using a commercial 45nm design library. We show that HARE can indeed saturate a 32GB/s memory bandwidth—performance far superior to existing software and hardware approaches.
- A scaled-down FPGA prototype operating at 100 MHz processing 4 characters per cycle. We show that even this scaled-down prototype outperforms traditional software solutions like `grep`.

II. Overview

HARE seeks to scan in-memory text corpora for a set of regexps while fully exploiting available memory bandwidth.

A. Preliminaries

HARE builds on the previous HAWK architecture [33], which provides a strategy for processing character windows without an explosion in the size of the required automata. HARE extends this paradigm to support two challenging features of regular expressions: character classes and quantifiers.

HARE is not able to process all regular expressions as no fixed-scan-rate accelerator can do so; some expressions inherently require either backtracking or prohibitive automata constructions, such as determinization. Moreover, when allowing combinations of features, such as Kleene star and bounded repetitions, even building a non-deterministic automaton can incur an exponential blowup [29].

We extend HAWK to support character classes, alternations, Kleene operators, bounded repetitions, and optional quantifiers. HARE allows Kleene (+,*) operators to be applied only to single characters (or classes/wild-cards) and not multi-character sub-expressions. Nevertheless, we demonstrate that this subset of regexps covers the majority of real-world regexp use cases.

B. Design Overview

HARE's design comprises a stall-free hardware pipeline and a software compiler. The compiler transforms a set of regexps into state transition tables for the automata that implement the matching process and configures other aspects of the hardware pipeline, such as look-up tables used for character classes and the configuration of various pipeline stages.

Figure 1 depicts a high-level block diagram of HARE's hardware pipeline. The figure depicts HARE as a six logical stages, where input text originates in main memory and matches are emitted to post-processing software (via a ring-buffer in memory). Note that individual logical stages are pipelined over multiple clock cycles to meet timing constraints. The two stages marked in orange (Character Class Unit, CCU; and Counter-based Reduction Unit, CRU) are newly added in HARE and provide the functionality to support regexps; the remaining stages are similar to units present in the HAWK baseline, which can match only fixed-length strings.

A HARE accelerator instance is parameterized by its width W, the number of input characters it processes per cycle. HARE streams data from main memory, using simple stream buffers to manage contention with other cores/units. W incoming characters are first processed by the CCU, which uses compact look-up tables to determine to which of $|C|$ pre-compiled character classes (those appearing in the input regexp) the input characters belong. The CCU outputs the original input characters (W×8 bits) augmented with additional W×$|C|$ bits indicating if each input character belongs to a particular character class.

The Pattern Automata perform the actual matching, navigating the set of automata constructed by the HARE compiler to match the sub-expressions of the input regexp. To make the state transition tables tractable, the Pattern Automata rely on the concept of *bit-split state machines* [32], wherein each pattern automaton searches for matches using only a subset of the bits of each input character. Bit-split state machines reduce the number of outgoing transition edges (to two in the case of single-bit automata) per state, drastically reducing storage requirements while facilitating fixed-latency lookups. We detail the bit-split concept and how we extend it to handle character classes in Section III-B.

Each pattern automaton outputs a bit vector indicating strings that *may* have matched at each input position, for the subset of bits examined by that automaton in the present cycle. These bit vectors are called *partial match vectors* or PMVs. A sub-expression of the regexp matches in the input text only if it is matched in *all* partial match vectors. The Intermediate Match Unit computes the intersection of all PMVs, called the intermediate match vector or IMV, using a tree of AND gates.

HAWK is only able to match fixed-length strings. Variable length matches pose a problem because they thwart HAWK's strategy for addressing the multiple possible alignments of each search string with respect to the window of W characters processed in each cycle. The central innovation of HARE is to split each regexp into multiple fixed-length sub-expressions called *components* and match the components separately using the pattern automata and intermediate match unit. The next stage, the Counter-based Reduction Unit, combines separate matches of the components and resolves ambiguities that arise due to concatenated character classes to determine a

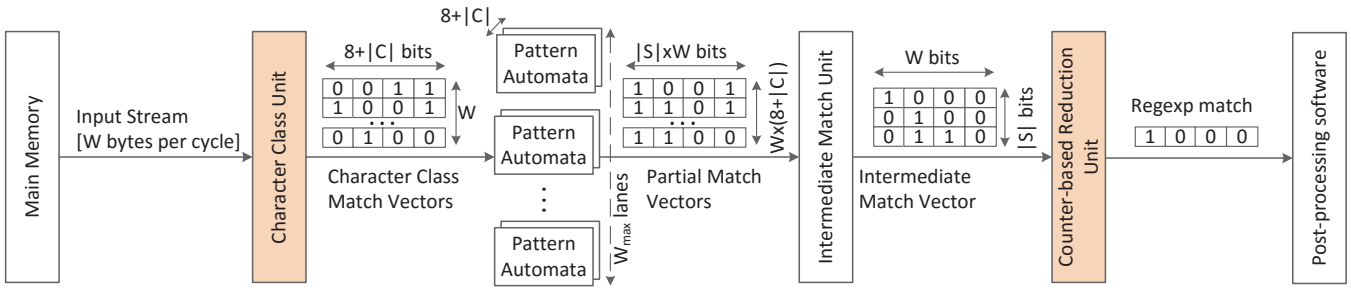

Fig. 1: **HARE block diagram.** The hardware pipeline enables stall-free processing of regexps. Shaded components are newly added relative to the baseline HAWK design.

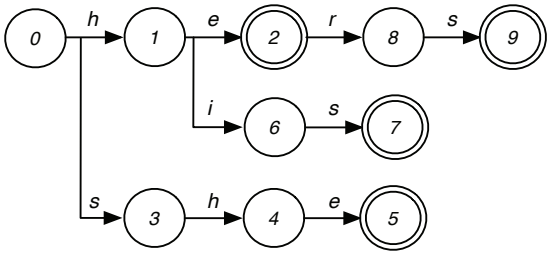

Fig. 2: **An Aho-Corasick Pattern Matching Automaton.** Automaton for search patterns *he*, *hers*, *his*, and *she*. States *2*, *5*, *7*, and *9* are accepting.

final match. This stage also allows it to handle Kleene (+,*), and bounded repetition ({a,b}) quantifiers in the presence of (potentially overlapping) character classes. Quantifiers pose a challenge because they can match a variable number of input characters. We elaborate on these issues in in Section III-D

III. From HAWK to HARE

HARE builds on HAWK [33], which itself builds on the Aho-Corasick algorithm [7] for matching strings.

A. Aho-Corasick algorithm

The Aho-Corasick algorithm [7] is widely used for locating multiple strings (denoted by the set S) in a single scan of a text corpus. The algorithm centers around constructing a deterministic finite automaton for matching S. Each state in the automaton represents the longest prefix of strings in S that match the recently consumed characters in the input text. The state transitions that extend a match form a trie (prefix tree) of all strings accepted by the automaton. The automaton also has a set of accepting states that consume the last character of a string; an accepting state may emit multiple matches if several strings share a common suffix. Figure 2 illustrates an Aho-Corasick automaton that accepts the strings {he,she,his,hers} (transitions that do not extend a match are omitted).

The classic Aho-Corasick automaton is a poor match for hardware acceleration, due to two key flaws:

• **High storage requirement**: The storage requirements of the state transitions overwhelm on-chip resources. To facilitate fixed-latency next-state lookup (essential to achieve a stall-free hardware pipeline), transitions must be encoded in a lookup table. The size of the required lookup table is the product of the number of states $|S|$ and the alphabet size $|\alpha|$, which rapidly becomes prohibitive for an ASCII text.

• **One character per step**: In the classic formulation, the Aho-Corasick automaton consumes only a single character per step. Hence, meeting our performance goal of saturating memory bandwidth (32GBps) either requires an infeasible 32-GHz clock frequency or consuming multiple characters per step. One can scale the classic algorithm by building an automaton that processes digrams, trigrams, W-grams, etc. However, the number of outgoing transition edges from an automaton grows exponentially in the width W, yielding $|\alpha|^W$ transition edges per state. Constructing and storing such an automaton for even modest W is not feasible.

B. Bit-split Automata

HAWK overcomes the storage challenge of the classic Aho-Corasick automaton using *bit-split automata* [32]. This method splits an Aho-Corasick automaton that consumes one character per step into an array of automata that operate in parallel and each consume only a subset of the bit positions of each input character. The state of each bit-split automaton now represents the longest matching prefix for its assigned bit positions, and its output function indicates the set of possibly matching strings; HAWK represents this set as a bit vector called a *partial match vector* (PMV). The output function of the original Aho-Corasick automaton is the disjunction of these PMVs, which HAWK implements via a tree of AND gates in its Intermediate Match Unit.

The bit-split technique reduces the number of outgoing edges per state. In HAWK, each automaton examines only a single input bit, hence, there are only two transition edges per state, which are easy to store in a deterministic-latency lookup table.

C. Scaling to $W > 1$

The bit-split technique drastically reduces storage, but still consumes only a single character per machine step. The primary contribution of HAWK is to extend this concept to consume a window of $W = 32$ characters per step, to search for $|S|$ strings using an array of $|S|\times W$ 1-bit automata operating in lock-step.

The key challenge to processing W characters per step is to account for the arbitrary alignment of each search string with

respect to the window of W positions. For example, consider an input search string *he* in input text *heatthen*, processed four characters at a time. While *he* begins at the first position in first four-character window (*heat*), it begins at the second position in the second window (*then*).

HAWK addresses this challenge by rewriting each search string into W strings corresponding to the W possible alignments of the original string with respect to the window, padding each possible alignment with wildcard (`.`) characters to a length that is a multiple of W. For example, for the string *he* and $W = 4$, HAWK will configure the hardware to search concurrently for <*he..*>,<*.he.*>, <*..he*>, and <*...h e...*>.

D. Challenges of Regexps

HAWK's hardware is sufficient to search for exact string matches and single-character (`.`) wildcards. However, HAWK's alignment/padding strategy is thwarted by regular expression quantifiers, because quantifiers may match a variable number of characters. To generalize HAWK's padding strategy in a straight-forward way, we must rewrite a single regexp containing a quantifier (e.g., `ab*c`) to consider all possible alignments of the prefix and all possible widths of the quantifier sub-expression, which rapidly leads to an infeasible combinatorial explosion.

HAWK's approach is further confounded by character classes, especially in cases involving multiple character classes. Consider, for example, the regular expression [a-f][o-r]ray can match six characters in the first position (characters *a* to *f*) and four characters in the second position (characters *o* to *r*). HAWK needs to enumerate the characters within the range of a character class to create all possible strings the character class can potentially match—24 patterns in the above example.

IV. HARE Design

We now describe the details of HARE's compilation steps and hardware units. We refer readers to [33] for the details of constructing bit-split automata and microarchitectural details of the pattern automata and intermediate match unit, which we only summarize here.

A. HARE Compiler

HARE's compiler translates a set of regexps into configurations for each of its stages. The compilation process proceeds in four steps: (1) split components, (2) compute precedence vectors and repetition bounds, (3) compile character classes, and (4) generate bit-split machines. Then, HARE invokes HAWK's existing compilation steps to construct bit-split automata and generate a bit stream to load into the accelerator. We describe the new compilation steps for regular expressions.

1) Component splitting: As previously noted, HAWK's string padding solution, which enables it to recognize matches that are arbitrarily aligned to the W character window scanned in each cycle, does not generalize to sub-expressions of a regexp that may match a variable number of characters.

Instead of pre-constructing an exponential number of pattern alignments, a key idea in HARE is to instead search for smaller, fixed-length sub-expressions of a regexp separately (and concurrently) and then confirm if the partial matches are concatenated (and possibly repeated) in a sequence that comprises a complete match. So, the first step of compilation is to split a regexp into a sequence of such sub-expressions, which we call *components*. The baseline HAWK is already able to scan for multiple fixed-length strings at arbitrary alignments; HARE configures it to search concurrently for all components comprising a regexp. The HARE compiler splits a regexp at the start and end of the operand of every quantifier (`?`, `*`, `+`, `{a,b}`) and alternation (`|`). (As previously noted, HARE does not support repetition operators applied to multi-character sequences).

Consider the example regexp `abc+de`, containing a Kleene Plus operator. The compiler splits the regexp at the operand of the Kleene Plus, `c`, resulting in three components `ab`, `c`, and `de`. The pattern automata are configured to search separately for these components (at all alignments). After reduction in the intermediate match unit, each IMV bit corresponds to a particular component detected at a particular alignment. These IMV bits are then processed in the counter-based reduction unit to identify matches of the full expression.

2) Compute precedence vectors: To locate a complete regexp match, HARE checks that components occur in the input stream in a sequence accepted by the regexp. As a regexp is split into multiple components, the compiler maintains a *precedence vector* that indicates which components may precede a given component in a valid match. The precedence vector for the first component is the empty set. Subsequent components include in their precedence vector all components that may precede them in a legal match. For example, a component following an optional (`?`) operator includes both the optional component and its predecessor in its precedence vector. We enumerate the rules for computing precedence vectors for each operator below. Along with the precedence vector, the compiler also records an upper and lower repetition bound for each component. For literal components (i.e., not a quantifier operand), the bounds are simply [1,1], otherwise, the bounds are determined by the quantifier.

Together, the precedence vectors and repetition bounds are used by the CRU to determine if a sequence of components (represented in the stream of IMVs consumed by the unit) constitutes a match. We next outline how to compute precedence vectors and repetition bounds for each operator.

• **Alternation** – An alternation operator (`|`) indicates that multiple components may occur at the same position in a matching input. The precedence vector for a component following an alternation includes all alternatives. For instance, for a regexp `gr(e|a)y` consisting of components `gr`, `e`, `a` and `y`, either component `e` or component `a` can appear after component `gr`. So, the precedence vectors for components `e` and `a` include component `gr`, while the vector for component `y` includes both components `e` and `a`. The lower and upper bounds for each alternative are determined by their sub-expressions (e.g., [1,1] for literals).

• **Optional quantifier** – A component followed by an optional

quantifier can appear zero or one time. The successor of an optional component includes the optional component and its predecessor in its precedence vector. For example, for regexp `ab?c`, consisting of components `a`, `b`, and `c`, the precedence vector for `b` includes only `a`. However, the precedence vector for `c` includes both `a` and `b`. The bounds for optional components are [1,1]. Note that the minimum bound for component `b` is not zero; if the component appears, it must appear at least once. The possibility that the component `b` may not appear is reflected in the precedence vector of component `c`.

• **Bounded repetition quantifier** – A bounded repetition quantifier sets a range of allowed consecutive occurrences of a component. For instance, the expression `ab{2,4}c` matches an input text starting with `a` followed by two, three, or four consecutive occurrences of `b` and finally terminating with `c`. Since all the components must appear at least once in the sequence, the precedence vector for each component includes only its immediate predecessor. The min and max bounds of component `b` are configured to match the bounds of the repetition quantifier *i.e.* [2,4]. Our implementation constrains bounds to a maximum of 256 to limit the width of the counters in the counter-based reduction unit.

• **Kleene Plus** – The operand of a Kleene Plus must appear one or more times in a match. Hence, each component's precedence vector includes only its immediate antecedent. For the earlier example `abc+de`, the precedence vector of `c` includes only `ab` and `de` includes only `c`. The max bound of a Kleene Plus operand is set to a special value indicating an unbounded number of repetitions. So, the min and max bound on components `ab` and `de` are [1,1], whereas, for `c` the bounds are [1,inf].

• **Kleene Star** – A Kleene Star (*), which matches a component zero or more times, is handled as if it were a Kleene Plus followed by an optional quantifier (`(+)?`). So, the precedence vector of its successor component includes it and its predecessor. In a regexp `ab*c`, the component `c` can either follow one or more repetitions of component `b` or a single instance of component `a`. Its precedence vector thus includes both the components `a` and `b`. Like the Kleene Plus, the bounds for the operand of a Kleene Star are set to [1,inf]. As with optional components, the minimum bound of component `b` is not zero; if the component appears, it appears at least once.

3) Compiling character classes: Character classes define sets of characters that may match at a particular input position. For instance, the regexp `tr[a-u]ck` matches ASCII characters between `a` and `u` at the third position, including strings `track` and `truck`. The naive approach of expanding character classes by enumerating all the characters in the character class range and matching all such patterns separately rapidly leads to blowup in the size of the automata. Bit-split automata, as used in HAWK, provide no direct support for character classes and must resort to such alternation.

We observe that we can augment the eight bit-split automata that process a single character with additional automata that process arbitrary Boolean conditions, for example, whether a character belongs to a particular character class. We determine

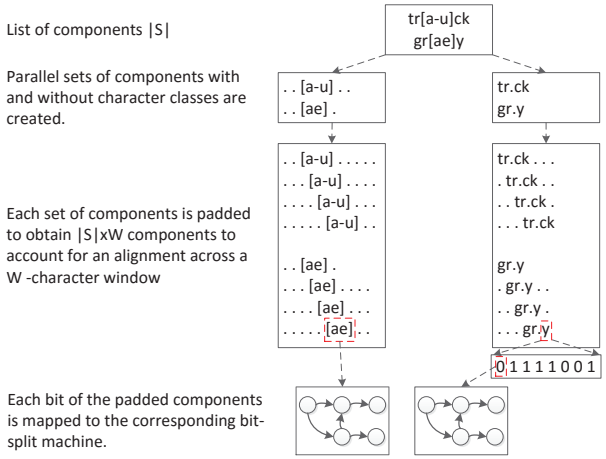

Fig. 3: **Compiling components containing character classes.** The components containing character classes are split in two, separating character classes from literals. These sets are separately padded and compiled to create bit-split automata.

if an ASCII character belongs to a class using a simple lookup table in HARE's CCU. For each character class in the regexp, the compiler emits a 256-bit vector, wherein a given bit is set if the corresponding ASCII character belongs to the class. For instance, for the character class `[a-u]`, bits 97 (corresponding to `a`) through 117 (corresponding to `u`) are set. These vectors are programmed into HARE's CCU, which outputs a `one` when a character falls within the class. Note that our scheme can be readily extended to Unicode character ranges by replacing the lookup table with range comparators.

Next, HARE breaks components containing character classes into two separate components, one comprising only literal characters, where character classes are replaced with single-character (`.`) wildcards, and the second comprising only character classes, with literals replaced by wild cards. Figure 3 illustrates the process of breaking and padding (for a 4-wide accelerator) these components for two example regexps including character classes. The regexps `tr[a-u]ck` and `gr[ae]y` consist of only a single component as they do not have any operators. The literal components are encoded in pattern automata exactly as in HAWK. The character class component uses the additional pattern automata that receive the output of the CCU. Both patterns are then padded for all possible alignments, as in the HAWK baseline.

Note that the main complexity of character classes arises in regexps where classes with overlapping character sets may occur at the same position in matching inputs (e.g., due to an alternation or Kleene operator). Placing classes into separate components facilitates their handling in the reduction stage.

4) Generate bit-split state machines: Once the two sets of components (one comprising only literal characters, the other comprising character classes) are generated, HARE's compiler invokes HAWK's algorithm to generate the bit-split machines processing W-characters per clock cycle. As illustrated in Figure 3, the two sets of components are padded front and back with wildcard characters to account for their alignment within a W-character window. The compiler then generates

bit-split automata for the padded components according to the algorithm proposed by Tan and Sherwood [32].

B. HARE Hardware Units

We next describe the microarchitecture of HARE's hardware pipeline, as depicted in Figure 1 and Figure 4.

1) Character Class Unit: Figure 4 (top) illustrates the character class unit (CCU). For each character class used in a regexp, the HARE compiler emits a 256-bit vector indicating which characters belong to the class. These vectors are programmed into a W-ported lookup table in the CCU. We denote the number of classes supported by the unit as $|C|$. Each of the W characters that enter the accelerator pipeline each clock cycle probes the lookup table and reads a $|C|$-bit vector indicating to which classes, if any, that character belongs. These $|C|$-bit vectors augment the 8-bit ASCII encodings of each character and all are passed to the pattern automata units.

2) Pattern Automata: As described in Section III-C, HAWK provisions W×8 bit-split automata to process a W-wide window of 8-bit ASCII characters each clock cycle. These automata emit W×8 partial match vectors indicating which components may match at that input position. The PMVs are each $|S|$×W bits long, where $|S|$ represents the number of distinct components the accelerator can simultaneously match (our implementations use $|S|$=64). The PMVs are then output to the intermediate match unit.

HARE adds W×$|C|$ automata units to process the output of the CCU. These automata store the transition tables for character class components constructed as described in Section IV-A3, emitting additional PMVs representing the potential character class matches to the intermediate match unit. The (8+$|C|$)×W bit-split automata operate in lock-step, consuming the same window of W characters, and emit (8+$|C|$)×W PMVs comprising $|S|$×W bits each. Figure 4 (middle) illustrates the pattern automata. Each cycle, an automaton consults the transition table stored in its local memory to compute the next state and corresponding PMV to emit, based on whether it consumed a zero or one. We refer readers to [33] for additional microarchitectural details of the pattern automata, which are unchanged in HARE.

3) Intermediate Match Unit: The intermediate match unit (IMU), as illustrated in Figure 4 (bottom), combines partial matches produced by the W lanes of the pattern automata to produce a final match. The W×(8+$|C|$) PMVs are intersected (bitwise AND) to yield an intermediate match vector (IMV) of $|S|$×W bits. Each bit in the IMV indicates that a particular component has been matched by all automata at a specific location within the W-character window.

4) Counter-based reduction unit: The counter-based reduction unit (CRU): (1) determines if components appear in a sequence accepted by the regexp, (2) counts consecutive repetitions of a component, (3) resolves ambiguities among consecutive character classes that accept overlapping sets of characters, and (4) determines if the repetition counts for the components fall within the bounds set by the HARE compiler.

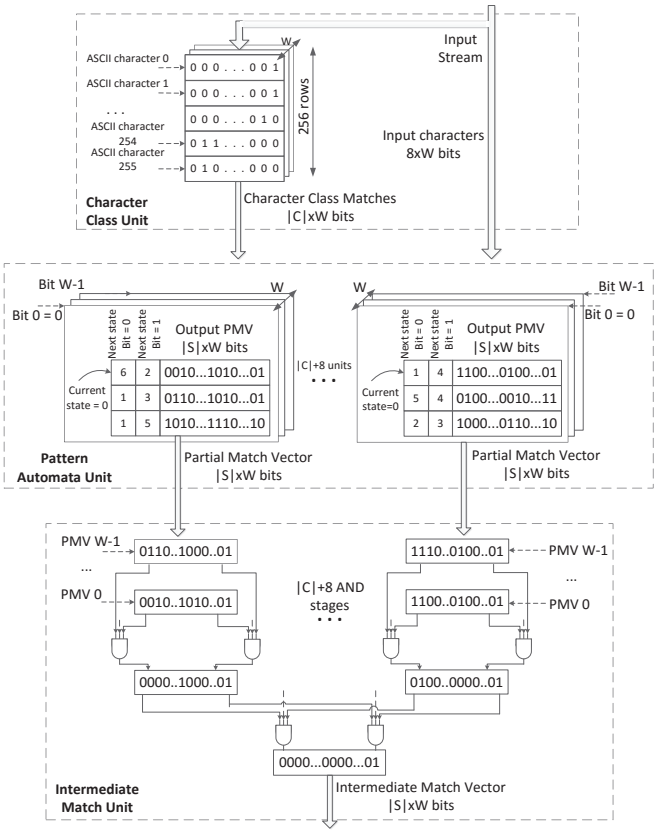

Fig. 4: **Accelerator sub-units.** The character class unit compares the input characters to the pre-compiled character classes, pattern automata processes the bit streams to generate PMVs which are later reduced by IMU to compute component match.

Our CRU design leverages the min-max counter-based algorithm proposed by Wang et al [35], which was designed to address character class ambiguities (3). Their algorithm consumes a single input character per step; we extend it to accept W-character windows per step and handle alternation operators and multi-character components. Throughout our discussion, we refer to Figure 5, which depicts the unit and an example of a complex expression that includes several of the subtle issues the CRU must address.

The input to the CRU in each clock cycle is the intermediate match vector produced by the intermediate match unit. $IMV_{i,j}$ is a bit matrix comprising $|S|$ rows, one per component j in the regexp, and W columns, one per position i in the input window. $IMV_{i,j}$ is set if a component has been detected to end at that input position. A new IMV matrix arrives each clock cycle. Figure 5 (top) illustrates arriving IMVs for $|S|$=5, W=4, and two clock cycles.

Internally, the CRU maintains three kinds of state, depicted in the remaining parts of Figure 5.

Two matrices of counter-enable signals $MAX\_EN_{i,j}$ and $MIN\_EN_{i,j}$ account for the relationship between consecutive components. They track whether component j respectively may or must consume input character i to extend a match, based on the input consumed by preceding components. Loosely, if component $j-1$ matches at position $i-1$, or

component j consumed character $i-1$, then these signals indicate that component j may consume character i. In our initial explanation, we assume that the precedence vector for component j includes only component $j-1$, and relax this restriction later.

The two matrices $\{MIN_{i,j}, MAX_{i,j}\}$ of counters indicate respectively the minimum number of repetitions that `must` be consumed and the maximum number of repetitions that `may` be consumed by component j to extend a match to position i. These repetition counts must be represented as a range, rather than an exact count, to handle adjacent character classes that accept overlapping character sets. In general, it is not known which input characters correspond to which components until a match is complete. Indeed, the CRU does not actually assign input characters to particular components as some regexps can match a given pattern in multiple ways. Rather, it determines if any match is possible.

Finally, a set of regexp match vectors $RMV_{i,j}$ track if the regexp matches up to and including component j at position i. $RMV_{i,j}$ is set if $MAX_{i,j}$ is above the lower repetition bound for component j and $MIN_{i,j}$ is below the upper bound, indicating that there is a feasible mapping of the input to components up to the ith character. A regexp matches at position i when the $RMV_{i,j}$ for the final component j is set.

Min-max matching for $W > 1$. We first describe our generalization of Wang's algorithm for min-max matching for $W > 1$, with reference to Algorithm 1. The min-max matching algorithm can match regexps containing a sequence of consecutive character classes when the character classes accept overlapping character sets. We describe the algorithm assuming precedence vectors form a strict chain (i.e., no `*`,`|`,`?` operators), and with only single-character components. We then remove these restrictions.

Consider a sequence of (potentially repeated) character classes $CC_1 ... CC_n$, such as `[a-d]{2,4}[abe]{2,3}`. This expression is challenging because some input texts can match the expression in multiple ways and it is generally impossible to assign input characters to specific components incrementally as the input is consumed. For example, the input `adbceb` can be matched by assigning `adbc` to CC_1 and `eb` to CC_2. However, a scheme that incrementally assigns characters might match `ad` to CC_1 and attempt to match `bce` to CC_2, at which point the match cannot be extended. The min-max algorithm resolves such ambiguous matches.

Initialization. (Lines 1-4). All counters, counter-enable, and RMV are initialized to zero, and the lower and upper bounds BL and BU for each component are initialized based on the bounds emitted by the HARE compiler. Each clock cycle, $IMV_{i,j}$ arrives from the intermediate match unit indicating components $0 <= j < |S|$ ending at positions $0 <= i < W$.

Determine counter-enables. (Lines 5-10). The counter-enable step captures the relationship between consecutive components and determines if the character at position i can potentially extend a match. More precisely, it determines if character at position i may potentially be consumed by component j based on whether the preceding input through $i-1$ matches the preceding regexp components up to (and possibly including) j. If $RMV_{i-1,j-1}$ is set, then component $j-1$ matches through position $i-1$, hence, character i may be the first occurrence of component j. Alternatively, if character at position $i-1$ was consumed by j, then i may be an additional repetition extending the match of component j. Note that the RMV for $j = -1$ is considered to be set at all positions in the input, meaning that first component $j = 0$ may begin at any position.

Update minimum counts. (Lines 12-18). The minimum counts $MIN_{i,j}$ reflect the count of characters that must be consumed by component j because they cannot be consumed by the preceding component $j-1$. If $MIN\_EN_{i,j}$ is set, then character i may be consumed by j. If $IMV_{i,j}$ is set, then i belongs to the character class of component j. However, if character i may also be consumed by the preceding component $j-1$, as reflected by $RMV_{i,j-1}$, then it is not *necessary* for component j to consume the character and $MIN_{i,j}$ is reset, else it is incremented. The min counter, therefore, always reflects the fewest characters that can be accounted for by repetitions of component j.

Update maximum counts. (Lines 20-26) The max counters, on the other hand, reflect the largest number of characters that could be consumed by component j. As above, $MAX\_EN_{i,j}$ indicates if character i may be consumed by j, and $IMV_{i,j}$ indicates if the character matches component j. If both conditions hold, the maximum counter is incremented.

Update RMVs. (Lines 28-32). Once MIN and MAX

Algorithm 1 Algorithm for computing regexp match using counter-based reduction unit.

Input: Intermediate Match Vector $\mathbf{IMV}$, number of components $|\mathbf{S}|$, architecture width $\mathbf{W}$, lower bounds $\mathbf{BL}$, and upper bounds $\mathbf{BU}$
Output: Regexp match vector RMV.

```
 1: MIN_EN = [[0 from 0 to |S|-1] from 0 to W-1]
 2: MAX_EN = [[0 from 0 to |S|-1] from 0 to W-1]
 3: MIN = [[0 from 0 to |S|-1] from 0 to W-1]
 4: MAX = [[0 from 0 to |S|-1] from 0 to W-1]
 5: for i = 1 to W-1 do
 6:     for j = 1 to |S|-1 do
 7:         MIN_EN[i][j] = RMV[i-1][j-1] || MIN[i-1][j] > 0
 8:         MAX_EN[i][j] = RMV[i-1][j-1] || MAX[i-1][j] > 0
 9:     end for
10: end for
11:
12: for i = 1 to W-1 do
13:     for j = 1 to |S|-1 do
14:         if MIN_EN[i][j] & IMV[i][j] then
15:             MIN[i][j] = RMV[i][j-1] ? MIN[i-1][j] + 1 : 0
16:         end if
17:     end for
18: end for
19:
20: for i = 1 to W-1 do
21:     for j = 1 to |S|-1 do
22:         if MAX_EN[i][j] & IMV[i][j] then
23:             MAX[i][j] = MAX[i-1][j] + 1
24:         end if
25:     end for
26: end for
27:
28: for i = 1 to W-1 do
29:     for j = 1 to |S| do
30:         RMV[i][j] = MAX[i][j] >= BL[i][j] & MIN[i][j] <= BU[i][j]
31:     end for
32: end for
```

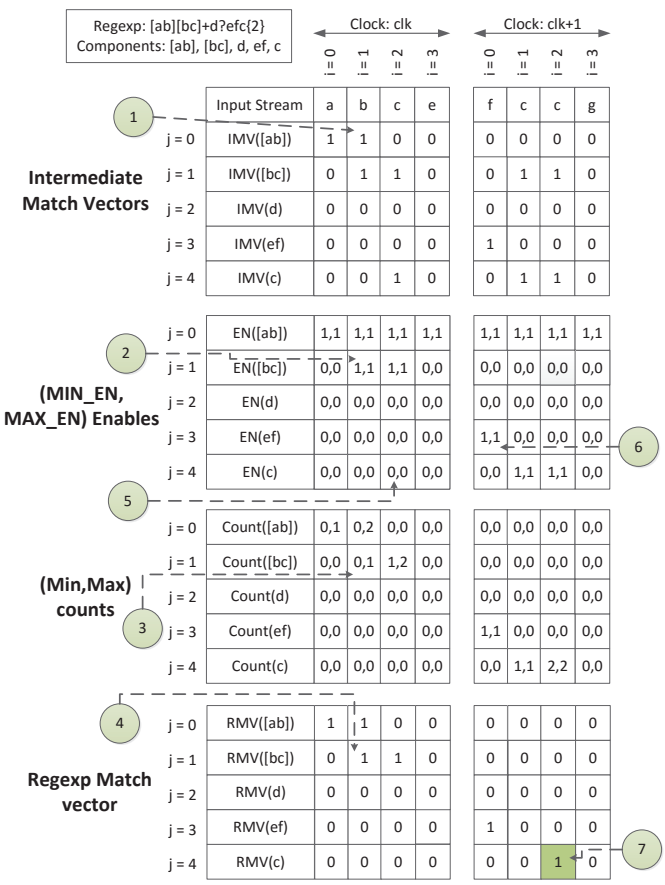

Fig. 5: **Counter-based reduction unit pipeline** - CRU combines the separate matches of the components generated by IMU. It maintains three states, namely counter enables, counters, and RMV to determine whether components of a regexp occur in a desired order.

are computed, RMV is computed as previously described; $RMV_{i,j}$ is true if MIN and MAX fall within BU and BL, respectively. A full regexp matches when $RMV_{i,j}$ for final component is set.

Example. Figure 5 illustrates how the CRU processes the regexp `[ab][bc]+d?efc{2}` consisting of components `[ab]`, `[bc]`, `d`, `ef`, and `c`. The figure illustrates the matching process for the input string `abcefccg`. The figure shows IMVs for two clock cycles, indicating where each component has matched in the input. ① indicates where two different character classes, corresponding to components `[ab]` and `[bc]` can match input character `b` at $i = 1$. Note that the counter-enables for component `[ab]` ($j = 0$) are always enabled and the minimum counter is always reset to zero, as a match of the regexp may begin at any point in the input. Component `[ab]` ($j = 0$) matches character `a` at $i = 0$ and increments $MAX_{0,0}$ to 1. Hence, $RMV_{0,0}$ is set, since $MIN_{0,0}$ is below upper bound BU_0 = 1 and $MAX_{0,0}$ equals lower bound BL_0 = 1.

The second character `b` is then processed and the counters $MIN_{1,1}$ and $MAX_{1,1}$ are enabled, since $RMV_{0,0}$ is set, enabling $MIN\_EN_{1,1}$ and $MAX\_EN_{1,1}$, indicated by ②. Furthermore, the counters $MAX_{1,0}$ and $MAX_{1,1}$ are both incremented as $IMV_{1,0}$ and $IMV_{1,1}$ are set. In other words, `b` can be consumed by either of the first two components.

Note that $MIN_{1,1}$ is not incremented, since `b` may be consumed by component $j = 0$, as indicated by ③. Since both counters for $j = 1$ satisfy the component's repetition bounds, $RMV_{1,1}$ is set, indicated by ④. When the third character is consumed, the counters $MIN_{2,4}$ and $MAX_{2,4}$ are not enabled as the preceding components did not match, indicated by ⑤.

Handling optional/alternative components. We next generalize the min-max algorithm to handle optional and alternative components. Recall that HARE's compiler emits, for each component, a precedence vector indicating the components that may precede it (see Section IV-A2). Rather than calculate MIN_EN and MAX_EN based solely on the immediately preceding component $j-1$, they are calculated as the logical-OR of all components in j's precedence vector. In words, component j may consume character i if any of its possible predecessors can consume character $i-1$.

Multi-character components. As originally proposed, Wang's min-max algorithm assumed the input would be consumed a single character at a time and had no need to handle multi-character components. Because PMV bits are a limited resource, it is critical for HARE to match multi-character sub-strings with a single component where possible, since HAWK provides that capability. We support multi-character components by storing the length of each component in a vector LEN_j. When indexing $RMV_{i,j}$ for a multi-character component, we right-shift the vector (in i) by $LEN_j - 1$ positions. That is, we ignore the columns of $RMV_{i,j}$ that fall within component j, and instead reference the last character of the preceding component.

We complete the preceding example to illustrate these extensions. In Figure 5, component `ef` may be preceded by either `[bc]` or `d`. Hence, in the second clock cycle, when computing $MIN\_EN_{0,3}$ and $MAX\_EN_{0,3}$ for component `ef` as indicated by ⑥, both possible predecessors `[bc]` ($j = 1$) and `d` ($j = 2$) are considered. Moreover, since the length of `ef` is two, count-enables, MIN, and MAX are calculated by referring to $RMV_{2,j}$ rather than $RMV_{3,j}$. Ultimately as illustrated by ⑦, the expression is matched when $RMV_{2,4} = 1$ in the second cycle (indicated by the green cell), when the MIN and MAX counts for component `c` ($j = 4$) match its bound of exactly 2 repetitions.

V. Evaluation

We evaluate two implementations of HARE, an RTL-level design targeting an ASIC process and a scaled-down FPGA prototype to validate feasibility and correctness. We study a suite of over 5500 real-world and synthetically generated regexps. We first contrast HARE against conventional software solutions and then evaluate area and power of the ASIC implementation of HARE for different processing widths.

A. Experimental Setup

We compare HARE's performance against software baselines on an Intel Xeon class server with the specifications listed

Processor	Dual socket Intel E5645 12 threads @ 2.40 GHz
Caches	192 KB L1, 1 MB L2, 12 MB L3
Memory Capacity	128 GB
Memory Type	Dual-channel DDR3-1333
Max. Mem. Bandwidth	21.3 GB/s

TABLE I: Server specifications.

in Table I. We select three software baselines: *grep* version 2.10, the Lucene search-engine *lucene* [16] version 5.5.0, and the Postgres relational database *postgres* [30] version 9.5.1.

We generate input text using Becchi's traffic generator [9]. The traffic generator is parameterized by the probability of a match *pM*; that is, the probability that each character it emits extends a match. For instance, for a *pM*=0.75, the traffic generator extends the preceding match with probability 0.75 and emits a random character with probability 0.25.

We implement the HARE ASIC design in Verilog and synthesize it for varying widths *W* of 2, 4, 8, 16, and 32. In our ASIC implementation, we configure HARE to match at most 64 components in a single pass. We target a commercial 45nm standard cell library operating at 1.1V and clock the design at 1GHz. Although this library is two generations behind currently shipping technology, it is the latest commercial process to which we have access. We synthesize the complete design using Synopsys Design-Ware IP Suite and report the timing, area and power estimates from Design Compiler.

To validate feasibility and correctness, we implement a scaled-down design on the Altera Arria V SoC development platform. Due to FPGA limitations, we implement a 4-wide HARE design. We use the FPGA's block RAMs to store pattern automata transition tables and PMVs; the available block RAMs limit the scale of the HARE design. Due to the overheads of global wiring to far-flung block RAMs, we limit clock frequency to 100MHz. Our software compiler generates pattern automata transition tables, PMVs, and reducer unit configurations, which we load into the block RAMs.

Because of the limited on-board memory capacity and poor bandwidth to host system memory available on our platform, we synthetically generate input text on the fly on the FPGA to test the functionality of the HARE FPGA. We tested 300 synthetic and hand-written regular expressions that stress various regexp features. We generate random text using linear feedback shift registers and then use a table-driven approach to periodically insert pre-generated matches into the synthetic text and confirm that all matches are found.

B. Regexp Workloads

We evaluate the capability and performance of HARE using a combination of human-written and automatically generated regexps from a variety of sources. Our human-written regexps are drawn from the online repository RegExLib [3] and the Snort [4] network intrusion detection library. Moreover, we derive synthetically generated regexps from the libraries provided by Becchi [9]. Table II shows the characteristics of each workload, indicating the number of expressions, the fraction HARE can support, the average number of components, and the average length of components. Several regexps on RegExLib are syntactically incorrect and we therefore discard them. HARE can support up to 99% of regexps in the workloads proposed by Becchi and around 86% of the regexps in the Snort library. In addition, despite the complexity of many of the expressions on RegExLib (some involving more than 50 components), HARE can support over 56% of them. Moreover, of the regexps we do not support, 83% of the Snort regexps and 45% of the RegExLib regexps contain non-regular operators, such as back references and look-ahead; when allowing these operators the matching problem is NP-complete [6]. The remaining unsupported expressions either contain nested repetitions or apply repetition operator to multi-character sub-strings. The HARE compiler detects unsupported regexps, reports a detailed error, and does not produce false negatives. Table II was derived from regexps flagged as unsupported by the compiler.

Workload	Regexps	Supported	Comp.	Comp. Len
dotstar0.3	300	99.0%	3.8	14.6
dotstar0.6	300	99.0%	4.4	12.5
dotstar0.9	300	99.0%	4.9	9.9
exact-match	300	99.6%	2.1	23.4
range05	300	99.6%	2.9	18.9
range1	300	99.3%	3.4	15.2
snort	1053	85.6%	4.6	5.5
RegExLib	2673	56.4%	12.3	1.7

TABLE II: Characteristics of regexp workloads.

HARE resource constraints. A HARE hardware implementation imposes two fundamental resource constraints: the number of supported character classes ($|C|$), which is constrained in the CCU and by the number of pattern automata, and the number of components in a regular expression ($|S|$), which is restricted by the number of PMV and IMV bits. Regular expressions that exceed these constraints cannot be processed in a single pass without additional software support.

Other implementation constraints, such as the maximum component length (equal to W), or the maximum precedence vector length (four per component) are automatically handled by the HARE compiler by splitting a component that exceeds the constraints into multiple components. All the workloads proposed by Becchi lie under these constraints. For Snort and RegExLib, the maximum precedence lengths of 9 and 59, respectively, exceed the hardware limit. The HARE compiler splits these components, increasing PMV utilization.

C. Performance - Scanning single regexp

We first contrast HARE's ASIC and FPGA performance with software baselines while scanning an input text for a single regular expression. We generate several 1GB inputs while varying pM. To exclude any time the software solutions spend materializing output, we execute queries that count the number of matches and report the count. We randomly select 100 regexps from each of the eight workloads for performance tests, and report average performance over these 100 runs. In the interest of space, we report results for only three of Becchi's six benchmarks, as the remaining benchmarks show similar trends in the performance. For Lucene, we first create

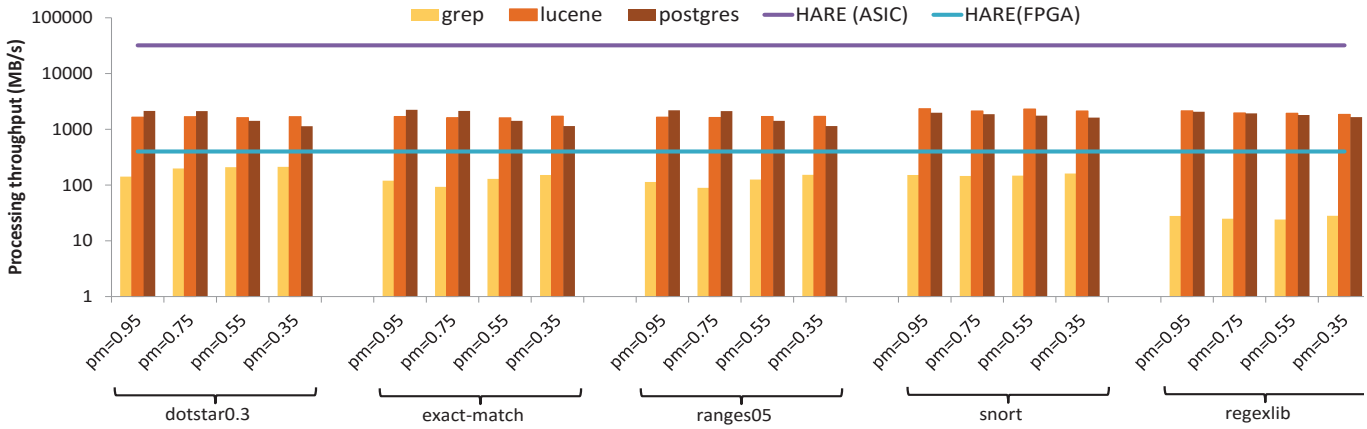

Fig. 6: **Single Regexp Performance Comparison.** We contrast HARE's fixed 32GB/s ASIC and 400 MB/s FPGA performance against software solutions. ASIC implementation of HARE performs two order of magnitude better than the software solutions.

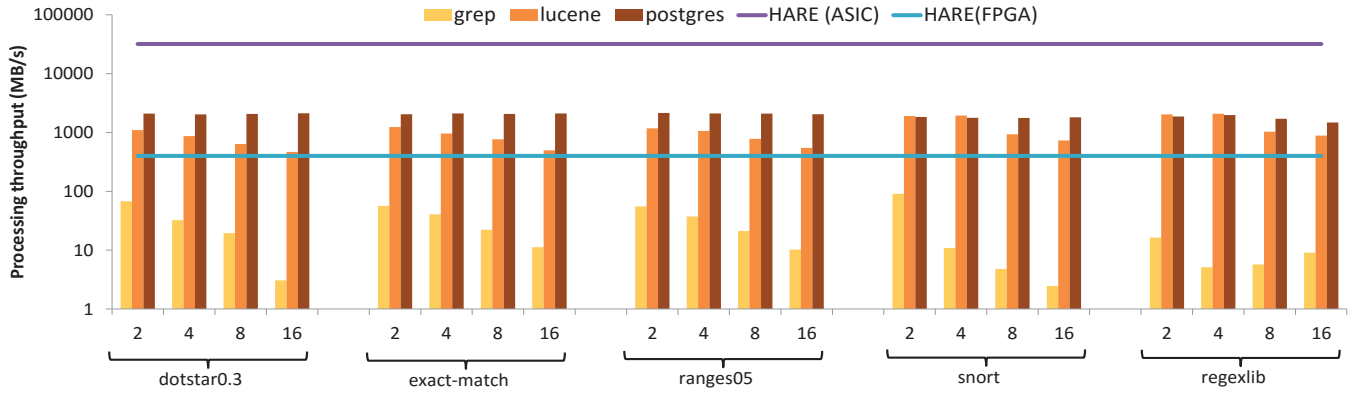

Fig. 7: **Multiple Regexp Performance Comparison.** The software solutions generally slow down as they search for more expressions concurrently. HARE's performance is insensitive to the number of expressions, provided the aggregate resource requirements of the expressions fit within HARE's implementation limits.

an inverted index of the input and do not include index creation time in the reported performance results. Similarly for Postgres, we first load the input into the database, excluding the load time from the results. We report their throughput by dividing the query execution time by the number of characters in the input text.

Figure 6 compares the throughput of grep, Lucene, and Postgres to the fixed scan rates of the HARE designs. The software systems are configured to use all 12 hardware threads of the Xeon E5645. The 32GB/s constant processing throughput of ASIC HARE is an order of magnitude higher than the software solutions. While HARE can saturate memory bandwidth, none of the other solutions come close. Even the scaled-down FPGA HARE implementation outperforms grep, which can only process at a maximum throughput of 300MB/s. Lucene and Postgres perform consistently above 1GB/s but fall considerably short of HARE's processing throughput.

D. Performance - Scanning multiple regexps

Figure 7 compares the performance of HARE and the software systems when scanning for multiple regexps concurrently (by separating a list of patterns with alternation operators). We randomly choose regexps from the workloads and vary their number from two to 16. We concatenate portions of the input text produced for each regexp (with pM=0.75) to ensure that all occur within the combined 1GB input text.

As expected, as the software systems search for more regexps, their throughput decreases. The performance of grep drops precipitously to 5MB/s when processing 16 regexps simultaneously; in practice, it is often better to perform multi-regexp searches consecutively rather than concurrently with grep. Postgres and Lucene still maintain a processing throughput of above 1GB/s even while scanning for 16 regexps. Again, note that we do not include the time Lucene and Postgres take to precompute indexes and load the input. On the contrary, HARE can still process the regexps simultaneously at constant throughput of 32GB/s.

E. ASIC Power and Area

We report the area and power requirement of ASIC HARE and its sub-units when synthesized for 45nm technology. We synthesize the HARE design for widths varying from two to 32 characters. As per our goal, we pipeline each design to meet a 1GHz clock frequency.

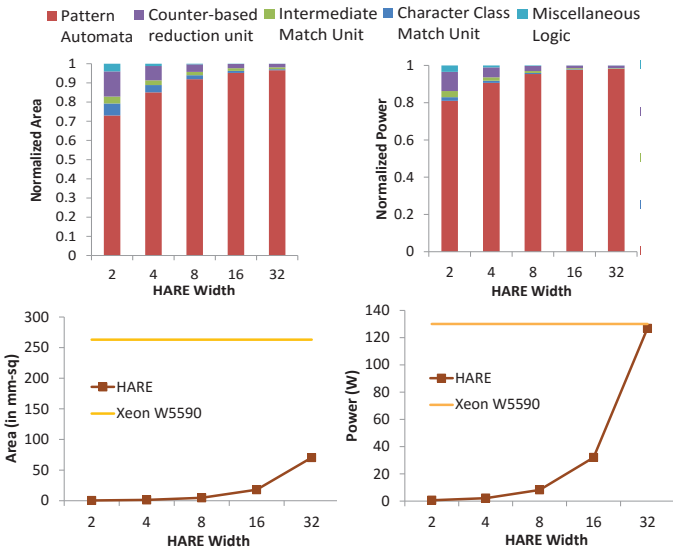

Fig. 8: **ASIC HARE area and power.** Pattern automata dominates area and power consumption of HARE due to the storage for bit-split machines. Overall, all the implementations of HARE consumes lower power than Xeon W5590.

As shown in Figure 8 (top), we find that the area and power requirement of HARE is dominated by the storage for state transition tables and PMVs in the pattern automata unit. Moreover, the contribution of pattern automata units to the total HARE area and power increases as the width of HARE grows, because the storage required for the bit-split machines grows quadratically with the accelerator width.

In Figure 8 (bottom), we compare the total area and power of HARE to an Intel W5510 processor. We select this processor for comparison because it is implemented in the same technology generation as our ASIC process. We see that the 8-wide and 16-wide instances of HARE require just 1.8% and 6.8% of the area of a W5510 chip. Moreover, the 8-wide and 16-wide HARE consumes only 6.3% and 24.6% of the power of our baseline processor. Even the 32-wide instance of HARE can be implemented in 26.7% of the area while consuming lower power than the W5510. Note that the 45nm technology used in our evaluation is two generations behind the state of the art. As the area and power requirements scale with technology, HARE would occupy a much smaller fraction of chip area relative to current state-of-the-art processors.

F. FPGA prototype

We validate the HARE design by implementing a scaled-down version on the Altera Arria V FPGA. We implement a 4-wide instance of HARE provisioning 64 components at 100MHz. The scaled-down HARE design uses 12% of the logic and 14% of the block memory capacity of the FPGA. Since we generate input text synthetically on the FPGA, HARE scans the input at a constant throughput of 400MB/s. Even when scaled down, HARE still scans the input text 1.9x faster than grep when scanning for a single regexp and this gap widens when processing multiple regexps.

VI. Related Work

Parallel regexp matching. Several works seek to parallelize matching by running the regexp automaton separately on separate substrings of the input and combining the results obtained on each part of the text [17]. Since each substring may start at an arbitrary point in the input, the automaton must consider all states as start states, which is problematic for large automata. PaREM [23] tries to minimize the number of states on which the automaton runs by exploiting the structure of automata that have sparse transition tables. Mytkowicz et al. [25] further optimize this concept by representing transitions as matrices and combining multiple automata executions using matrix multiplication. They also use SIMD to perform multiple lookups for different sections of the input text at once.

Parabix [20] introduces the idea of processing character bits in parallel and combining the results using Boolean operations. This design allows Parabix to exploit SIMD instructions. Cameron [12] extends the design of Parabix to directly handle non-determinism and provides a tool chain to generate marker streams, the bit-stream that mark the matches in the input text. For different regexp operations, the tool manipulates the marker stream to update the regexp matches.

The Unified Automata Processor (UAP) [14] implements specialized software and hardware support for different automata models e.g., DFAs, NFAs, and A-DFAs. This framework proposes new instructions to configure the transition states, perform finite automata transitions and synchronize the operations of parallel execution lanes. HARE's approach of using a stall-free scan pipeline with parallel bit-split automata and min-max matching bears little similarity to UAP's implementation approach. The UAP relies on parallel processing of multiple input streams to achieve its peak bandwidth of 295 Gbit/sec, but achieves at most a 1.13 GC/s scan rate per stream. In contrast, HARE saturates a memory bandwidth of 32 GC/s (256Gbit/s) when scanning a single input stream.

ASIC and FPGA based solutions. Micron's Automata Processor [13] implements NFAs at the architecture level. Transition tables are stored as 256-bit vectors, which are then connected over a routing matrix. Counting and boolean operations are then used to count the matches of sub-expressions and combine sub-expression results. The processor can consume input strings at a line rate of 1Gbit/sec per chip.

IBM PowerEN SoC integrates RegX, an accelerator for regular expressions [21]. RegX splits regexps into multiple sub-patterns, implements separate DFAs and configures the transition tables using programmable state machines called B-FSMs [34], and finally combines the sub-results in the local result processor. RegX runs at a frequency of 2.3 GHz and achieves a peak scan rate of 9.2Gbit/sec.

A Micron Automata Processor processing 1 character/cycle consumes around 4W [13], while the IBM PowerEn RegX accelerator consumes around 2W [15]. In comparison, a 1-wide HARE implementation consumes less than 1W.

Helios [1] is another accelerator that processes regexps for network packet inspection at line rate. In addition, several

works [10], [24], [28], [36], [37] propose mechanisms to match regexps on FPGAs. They focus on building a finite automaton and encode it in the logic of the FPGA. HARE's 32GB/sec (256Gbit/sec) scan rate is much more ambitious than these prior ASIC or FPGA designs.

VII. Conclusion

Rapid processing of high-velocity text data is necessary for many technical and business applications. Conventional regular expression matching mechanisms do not come close to exploiting the full capacity of modern memory bandwidth. We showed that our HARE accelerator can process data at a constant rate of 32 GB/s and that HARE is often better than state-of-the-art software solutions for regular expression matching. We evaluate HARE through a 1GHz ASIC RTL implementation processing 32 characters of an input text per clock cycle. Our ASIC implementation can thus match modern memory bandwidth of 32GB/s, outperforming software solutions by two orders of magnitude. We also demonstrate a scaled-down FPGA prototype processing 4 characters per clock cycle at a frequency of 100MHz (400 MB/s). Even at this reduced rate, the prototype outperforms grep by 1.5-20× on commonly used regular expressions.

VIII. Acknowledgements

We would like to thank anonymous reviewers for their valuable feedback that helped us improve the paper. This work was supported by grants from ARM, Ltd.

References

[1] "Helios Regular Expression Processor." [Online]. Available: http://titanicsystems.com/Products/Regular-eXpression-Processor-(RXP)

[2] "Intel and Micron Produce Breakthrough Memory Technology." [Online]. Available: https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/

[3] "Regular expression library." [Online]. Available: http://regexlib.com/

[4] "Snort." [Online]. Available: http://snort.org/

[5] "Structuring Unstructured Data." [Online]. Available: www.forbes.com/2007/04/04/teradata-solution-software-biz-logistics-cx_rm_0405data.html/

[6] A. V. Aho, "Handbook of theoretical computer science (vol. a)," J. van Leeuwen, Ed. Cambridge, MA, USA: MIT Press, 1990, ch. Algorithms for Finding Patterns in Strings, pp. 255–300.

[7] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, 1975.

[8] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek *et al.*, "A view of the parallel computing landscape," *Communications of the ACM*, 2009.

[9] M. Becchi, M. Franklin, and P. Crowley, "A workload for evaluating deep packet inspection architectures," in *IEEE International Symposium on Workload Characterization*, 2008.

[10] J. Bispo, I. Sourdis, J. M. P. Cardoso, and S. Vassiliadis, "Regular expression matching for reconfigurable packet inspection," in *IEEE International Conference on Field Programmable Technology*, 2006.

[11] R. D. Cameron and D. Lin, "Architectural support for swar text processing with parallel bit streams: The inductive doubling principle," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[12] R. D. Cameron, T. C. Shermer, A. Shriraman, K. S. Herdy, D. Lin, B. R. Hull, and M. Lin, "Bitwise data parallelism in regular expression matching," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, 2014.

[13] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3088–3098, 2014.

[14] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien, "Fast support for unstructured data processing: The unified automata processor," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015.

[15] H. Franke, C. Johnson, and J. Brown, "The ibm power edge of network processor," 2010.

[16] E. Hatcher and O. Gospodnetic, "Lucene in action," 2004.

[17] J. Holub and S. Štekr, "On parallel implementations of deterministic finite automata," in *Proceedings of the 14th International Conference on Implementation and Application of Automata*, 2009.

[18] J. E. Hopcroft and J. D. Ullman, *Formal Languages and Their Relation to Automata*. Addison-Wesley Longman Publishing Co., Inc., 1969.

[19] N. Hua, H. Song, and T. Lakshman, "Variable-stride multi-pattern matching for scalable deep packet inspection," in *INFOCOM 2009, IEEE*, 2009.

[20] D. Lin, N. Medforth, K. S. Herdy, A. Shriraman, and R. Cameron, "Parabix: Boosting the efficiency of text processing on commodity processors," in *Proceedings of the 18th International Symposium on High Performance Computer Architecture*, 2012.

[21] J. V. Lunteren, C. Hagleitner, T. Heil, G. Biran, U. Shvadron, and K. Atasu, "Designing a programmable wire-speed regular-expression matching accelerator," in *Proceedings of the 45th Annual International Symposium on Microarchitecture*, 2012.

[22] S. Manegold, M. L. Kersten, and P. Boncz, "Database architecture evolution: Mammals flourished long before dinosaurs became extinct," *Proc. VLDB Endow.*, 2009.

[23] S. Memeti and S. Pllana, "Parem: A novel approach for parallel regular expression matching," *CoRR*, 2014.

[24] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling pcre to fpga for accelerating snort ids," in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, 2007.

[25] T. Mytkowicz, M. Musuvathi, and W. Schulte, "Data-parallel finite-state machines," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

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

[27] V. Salapura, T. Karkhanis, P. Nagpurkar, and J. Moreira, "Accelerating business analytics applications," in *Proceedings of the 18th International Symposium on High Performance Computer Architecture*, 2012.

[28] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using fpgas," in *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

[29] M. Sipser, *Introduction to the Theory of Computation*, 1st ed. International Thomson Publishing, 1996.

[30] M. Stonebraker and L. A. Rowe, "The design of postgres," in *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, 1986.

[31] M. Stonebraker and A. Weisberg, "The voltdb main memory dbms." *IEEE Data Eng. Bull.*, 2013.

[32] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," in *Proc. ISCA*, 2005.

[33] P. Tandon, F. M. Sleiman, M. Cafarella, and T. F. Wenisch, "Hawk: Hardware support for unstructured log processing," in *International Conference on Data Engineering*, 2016.

[34] J. van Lunteren, "High-performance pattern-matching for intrusion detection," in *25th IEEE International Conference on Computer Communications*, 2006.

[35] H. Wang, S. Pu, G. Knezek, and J. C. Liu, "Min-max: A counter-based algorithm for regular expression matching," *IEEE Transactions on Parallel and Distributed Systems*, 2013.

[36] Y. H. Yang and V. Prasanna, "High-performance and compact architecture for regular expression matching on fpga," *IEEE Transactions on Computers*, 2012.

[37] Y.-H. E. Yang, W. Jiang, and V. K. Prasanna, "Compact architecture for high-throughput regular expression matching on fpga," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008.