Memory-Efficient Pipelined Architecture for Large-Scale String Matching *

Yi-Hua E. Yang and Viktor K. Prasanna
Ming Hsieh Department of Electrical Engineering, University of Southern California
{yeyang, prasanna}@usc.edu

***Abstract*—We propose a pipelined *field-merge* architecture for memory-efficient and high-throughput large-scale string matching (LSSM). Our proposed architecture partitions the (8-bit) character input into several bit-field inputs of smaller (usually 2-bit) widths. Each bit-field input is matched in a *partial state machine* (PSM) pipeline constructed from the respective bit-field patterns. The matching results from all the bit-fields in every pipeline stage are then merged with the help of an *auxiliary table* (ATB). This novel architecture essentially divides the LSSM problem with a continuous stream of input characters into two disjoint and simpler sub-problems: 1) $O(\text{character\_bitwidth})$ number of pipeline traversals, and 2) $O(\text{pattern\_length})$ number of table lookups. It is naturally suitable for implementation on FPGA or VLSI with on-chip memory. Compared to the bit-split approach [12], our field-merge implementation on FPGA requires 1/5 to 1/13 the total memory while achieving 25% to 54% higher clock frequencies.**

I. Introduction

Large-scale string matching (LSSM) has applications in many areas including data mining, virus scanning [1], and network intrusion detection [2]. In all these cases, high-throughput input streams are matched against large dictionaries of text or binary patterns. Matches can occur at multiple points within the input stream or overlap with each other. Brute-force approaches for this type of processing are both data and computation intensive.

Due to the rapid growth of network technologies such as the commoditization of Gigabit Ethernet, an LSSM solution needs to meet the following three requirements to be practical and useful:

1) *High per-stream throughput*. The string matching engine needs to handle input streams with *per-stream throughput* in the range of gigabits per second.
2) *Large dictionary size*. The matched dictionary can consist of *tens of thousands* of patterns with *hundreds of thousands* of characters.
3) *Easy dynamic update*. The matched dictionary may need to be *updated frequently and dynamically* while the matching engine is operating.

Recently, tremendous progress has been made on LSSM with respect to each of the three aspects of performance requirements above. The bit-split approach proposed in [12] and studied in [7] achieves per-stream throughputs of 1~2 Gbps over a dictionary of ~1000 patterns on a single FPGA. While a dictionary of 1000 patterns is relatively small (the latest Snort rules contain more than 7000 unique matching patterns), the throughput achieved by the bit-split approach on FPGA [7] is still much lower than that achieved in other approaches such as [10, 6, 4].

In this study, we propose a *field-merge* architecture which aims to improve upon the bit-split approach on all three performance aspects listed above. Our contributions in the field-merge architecture include the following:

1) Convert the variable-length string matching problem to a simple tree traversal followed by a fixed-width pattern lookup.
2) Use less memory and achieves higher clock frequency than the bit-split FPGA implementation [7].
3) Allow the matching of each bit-field to be optimized independently.
4) Allow easy and dynamic dictionary updates.

The rest of the paper is organized as follows. Section II describes existing approaches for large scale string matching. We focus on the different performance bottlenecks of each approach. Section III describes the construction algorithm and the circuit architecture, while Section IV proves the correctness of the proposed field-merge string matching. Section V evaluates a field-merge implementation on FPGA using two different types of dictionaries. Section VI concludes the paper and discusses future work.

II. Related Work

There have been three general approaches toward large-scale and high-throughput string matching: 1) pipelined NFA approach [10, 4, 6], 2) bit-split DFA approach [12, 7], and 3) interleaved DFA approach [8, 9].

A. Pipelined Circuit-Based NFA Approach

In a pipelined NFA string matching architecture, the input stream is matched by multiple string patterns in parallel on an FPGA via a pipelined network. The architectures allow matching $m > 1$ characters per clock cycle either by replicating a match pipeline m times [10] or by using a multi-character decoder [6]. With a multi-character design, matching throughputs of $8 \sim 10$ Gbps were achieved against $210 \sim 360$ string patterns.

While achieving very high throughputs, the main drawbacks of this approach are lack of flexibility and relatively low

* This work was supported by U.S. National Science Foundation under grant CCR-0702784. Equipment grant from Xilinx Inc. is gratefully acknowledged.

pattern capacity. According to [10], a 4-character input circuit takes from 16 to 20 LUTs per dictionary character. This is $5 \sim 8$ times higher than the LUT usage in [5, 13, 14], where the more complex problem of regular expression matching was considered. Furthermore, the logic cell saving in [4] depends highly on the content of dictionary patterns, making it necessary to re-synthesize the entire circuit for any dictionary update. This not only makes on-line dictionary update impossible, but also makes it difficult to predict resulting performance before the circuit place & route is actually performed.

B. Bit-Split DFA Approach

In [3], Aho and Corasick proposed an algorithm to convert a dictionary tree of n patterns to a DFA with $O(n)$ states. In terms of computation complexity, the AC-DFA requires $O(1)$ computations per character irrespective of the size of the dictionary. While both space and time complexities of the AC-DFA are asymptotically optimal, the valid (non-zero) states in the state transition table (STT) are usually very sparsely populated. This sparsity causes large amount of memory wastage [12] and makes memory bandwidth and latency the bottlenecks for string matching based on AC-DFA.

The bit-split string matching [12] proposed to split a full AC-DFA into several split state machines, each accepting a small portion (1 or 2 bits) of the input as transition labels. A partial match vector (PMV), one bit per pattern in the dictionary, is maintained at every state in the split state machine to map the state to a set of possible matches. At every clock cycle, the PMVs from all split state machines are AND-ed together to produce a full match vector (FMV) to generate the actual matches. A pattern match is found only if its corresponding FMV bit is TRUE.

There are several issues with the bit-split approach. First, the construction of the split state machines from the AC-DFA has a worst-case complexity of $O\left(n^2/g\right)$, where n is the total number of patterns in the dictionary and g is the number of pattern groups.[1] This can be very expensive for large dictionaries with thousands of states. Second, every state in the split state machines needs to maintain a (n/g)-bit PMV, making total memory size also $O\left(n^2/g\right)$. Third, in every clock cycle, g FMV results need to be aggregated, incurring a tree-like structure whose routing delay is likely to be $O(g)$.[2] While it is possible to optimize performance by choosing a good group size (n/g) [7], such an optimization requires resource planning for each group and makes dynamic dictionary updates more difficult.

[1]Although complexity was claimed to be $O(n)$ in [11], we believe this under-estimates the worst-case. In the bit-split algorithm, a pattern group can spend up to $O(n/g)$ steps converting the AC-DFA to the split state machines, while in each step up to $O(n/g)$ states in the AC-DFA must be checked for inclusion. This procedure is performed on each of the g pattern groups.

[2]The performance claim in [11] ignored routing delay and assumed a clock frequency of 1.125 GHz (SRAM speed reported by CACTI 3.2). Their functional prototype on FPGA, however, achieved only 150 MHz over 500 patterns.

C. Interleaved DFA Approach

In [9] the authors adopted an input-parallel approach to perform large-scale string matching based on AC-DFA. All string patterns in a dictionary is compiled together into a single AC-DFA. Up to 256 state machines, 16 per Synergistic Processing Element (SPE), run in parallel on two Cell/B.E. microprocessor. Each state machine updates its own current state number by accessing a shared AC-DFA state transition table (STT) in the main memory. Memory access latency is hidden by the large number of state machines per SPE. A combination of techniques, including state caching, state and alphabet shuffling, and state replication are employed to reduce congestion at the memory subsystem. The result is a bandwidth optimized STT spanning across hundreds of megabytes of memory, achieving between 2 Gbps on 8 SPEs to 3.5 Gbps on 16 SPEs. Alternatively, a number of small-scale but high-throughput designs are also proposed, using only the local store (LS) of the SPEs to hold the STT [8].

The main problem of this approach is the various constraints imposed on the string matching operation. The 256 concurrent state machines help to fully utilize the memory bandwidth and to achieve high aggregated throughput. The per-stream throughput, however, remains much lower (< 20 Mbps) compared to other approaches. It is suggested [9] that a single stream can be dissected into overlapping chunks to be matched in parallel. Such dissection, however, would require a large buffer and incur high processing overhead and matching latency. Furthermore, the performance scaling of this approach relies heavily on the complex transformations performed on the STT, which makes incremental updates very difficult, if not impossible, to perform at run time.

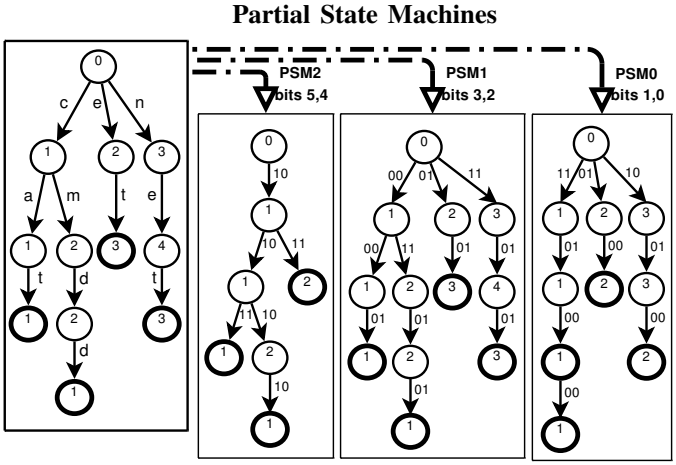
III. Field-Merge String Matching

A. Basic Field-Merge Construction

The field-merge string matching partitions every input character into k bit-fields similar to the bit-split approach. Based on the discussion in Section II-B, we identify the following issues with bit-split string matching:

- The partial match vectors (PMVs) must be maintained at every state in the SSMs to map the SSM matches back to the potential AC-DFA matches.
- The PMVs must have a number of bits equal to the number of patterns in the dictionary.
- The construction, storage and aggregation of the PMVs reduce the bit-split performance with respect to design flexibility, memory size and clock frequency.

To get rid of the PMVs, we do not apply the full Aho-Corasick algorithm in field-merge. Instead, we construct a tree-structured non-deterministic finite automaton (NFA), called the *full state machine* (FSM), based on the directed tree graph of the dictionary [3]. Then we divide the dictionary "vertically" into orthogonal bit-fields and use each bit-field to construct a tree-structured NFA, called the *partial state machine* (PSM), with fewer bits per transition label. Every PSM has the same depth (number of levels) as the FSM. However, due to the



Auxiliary Tables

	FSM	PSM2	PSM1	PSM0
Level 2:	**3**	**2**	**3**	**2**
Level 3:	**1**	**1**	**1**	**1**
	3	**1**	**3**	**2**
Level 4:	**1**	**1**	**1**	**1**

Figure 1. Construction of the PSMs from a 4-pattern, 4-level FSM, and the per-level ATBs mapping full match states to partial match states.

surjective mapping from the FSM states to the PSM states, the size of the PSMs can be considerably less than that of the FSM. Every match state in the FSM, called a *full match state*, corresponds to exactly one *partial match state* in every PSM. In addition, during the construction of the PSMs, a per-level *auxiliary table* (ATB) is maintained to map every full match state uniquely into a set of partial match states on the same level, one from each PSM. Algorithm 1 gives a formal description of the field-merge PSM construction.

In order to match a new character every clock cycle, the PSMs are pipelined along the tree depth, one stage per level. Assume the input character is divided into k bit-fields as $\mathrm{PSM}_0, \ldots, \mathrm{PSM}_{k-1}$. At every clock cycle, stage L of PSM_i receives (1) input bits b_i from bit-field #i, and (2) state number $S_i(L-1)$ from stage $L-1$ of PSM_i. If $S_i(L-1)$ is a valid (non-zero) state, then it is used together with b_i to access a stage-local STT for the next state number, $S_i(L)$, which is forwarded to stage $L+1$ in the next clock cycle. Furthermore, if all $S_i(L)$, $i = 0, \ldots, k-1$, are partial match states, then the vector $\bar{S}(L) \equiv [S_0(L), \ldots, S_{k-1}(L)]$ is looked up in the level-L auxiliary table $\mathrm{ATB}(L)$ for a potential full match state. An actual match is declared at level L if and only if $\bar{S}(L)$ is found as a valid full match state in $\mathrm{ATB}(L)$.

The novelty in this approach is that by using a relatively small ATB, both pipeline traversals and output merging of the PSMs can be made simple. The price to pay is one ATB lookup per level, which can be performed efficiently as a content addressable memory (CAM) access or a hash table lookup.

Figure 1 shows an example where a 4-pattern, 4-level FSM with 6-bit alphabet is converted into 3 mutually independent

Algorithm 1 Construction of Partial State Machines from a Pattern Dictionary.

Global data:

FSM	The full state machine.
PSM_i	The partial state machine for bit-field #i.
$\mathrm{ATB}(L)$	The auxiliary table at level L.

Conventions:

p_i	Bit-field #i of string pattern p.
s.level	Level number of a state s.
s.index	Level-local state number of a state s.
$\mathrm{ATB}(L)$.size	Number of entries in auxiliary table at level L.
$\mathrm{ATB}(L)[m]$	Entry of index m in auxiliary table at level L.

Macros:

$s \leftarrow$ `ADD_PATTERN` (p_i, PSM_i) :
Add a pattern p_i to PSM_i. Return the last (partial match) state.

```
BEGIN
  for each pattern p in dictionary
    f ⇐ ADD_PATTERN(p, FSM);
    L ⇐ f.level;
    if f exists in ATB(L) then
      // found pattern duplicate
      continue;
    else
      m = ATB(L).size++;
      create new entry ATB(L)[m];
      ATB(L)[m].FSM ⇐ f.index;
    end if
    for each bit-field i
      s_i ⇐ ADD_PATTERN(p_i, PSM_i);
      ATB(L)[m].PSM_i ⇐ s_i.index;
    loop
  loop
END
```

PSMs, each corresponding to a bit-field of width 2. Each of the levels 2, 3, and 4 further consists of an auxiliary table mapping the full match states to a set of 3 partial match states, one per PSM. Note that the state numbers in one level can overlap with those in a different level, since a valid state transition can only go from one level to the next in any PSM.

The field-merge approach has several notable differences from the previous bit-split approach:

1) Unlike bit-split, field-merge does not apply the Aho-Corasick algorithm to the partial state machines (PSMs). Instead, the NFA-based PSMs are pipelined to achieve a throughput of one character per clock cycle.
2) There is no need of the *per-state* partial match vectors (PMVs). Instead, a *per-level* auxiliary table (ATB) maps each full match state to a set of partial match states at the same level.
3) Pattern grouping or PMV aggregation is not needed.

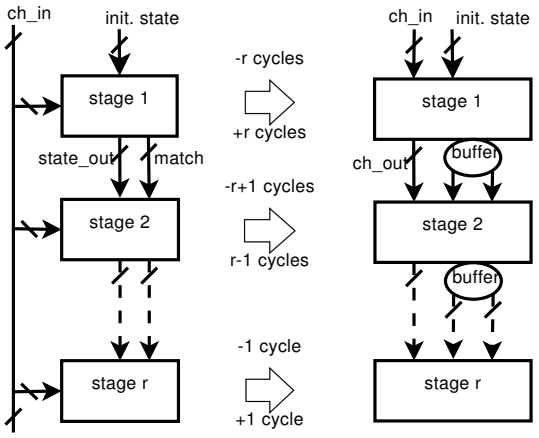


Figure 2. Transform a field-merge pipeline of r stages from a semi-systolic array (left) to a true systolic array (right).

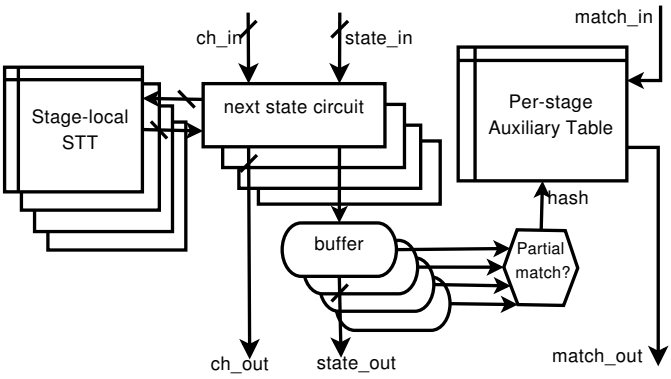


Figure 3. Circuit architecture a field-merge pipeline stage.

Instead, the ATB lookup is localized at each stage and can be performed efficiently.

4) The complexity of constructing the PSMs is only $O(n)$ where n is the total number of patterns in the dictionary.
5) Field-merge is a fully memory-based approach where dynamic dictionary updates can be performed easily by injecting special "update bubble" into the input stream.

B. FPGA Circuit Architecture

The basic architecture of the field-merge circuit is a sequence of stages, one per partial state machine (PSM) level, as shown on the left of Figure 2. Suppose the input character is partitioned into k bit-fields ($k = 3$ in the example in Figure 1). Each stage in the field-merge pipeline consists of k PSMs and $k+1$ local memory modules accessed in parallel. A valid state number produced at level L should match a length-L suffix of the input stream.

The problem with this basic structure is that in every clock cycle, the input character needs to be broadcast to all PSMs of every pipeline stage. Depending on the length of the patterns in the dictionary, this can result in a fan-out of hundreds. To localize the signal paths, we apply a semi-systolic-to-systolic transformation shown in Figure 2. As discussed below, the extra buffering of `state_out` and `match_out` can be absorbed into the stage to shorten the critical path of ATB lookup.

All stages in the pipeline have the same circuit architecture shown in Figure 3. Within a stage, there are k sets of circuits, one for each PSM. Each set of circuit receives one bit-field of the input character (`ch_in`) and the corresponding state input (`state_in`) from the previous stage. The character and state inputs (of a particular bit-field) are combined to form an address to the local state transition table (STT), which returns the state number (`state_out`) for the PSM at the next stage. While the input character is forwarded to the next stage immediately, the state numbers are buffered an extra clock cycle during which their "match flags" are checked. If all k state numbers (one from each PSM) are partial match states, then the k state numbers are hashed (XOR-ed) to access the per-stage auxiliary table (ATB) for a potential full match state (`match_out`), which is pipelined to the next stage in the same way as the state number.

At design time, the STT and ATB sizes of each stage can be configured individually. Depending on the pattern lengths and the common-prefix properties of the dictionary, each stage or PSM can require a different number of STT and ATB entries. We note that although it is more space-efficient to make the table sizes as small as possible, doing so would also impact the ability to add new match patterns dynamically. In practice, memory sizes are usually multiples of a power of two, thus in almost all cases there will be some head room for adding match patterns dynamically.

The architecture allows multiple and overlapping matches because every character in the input stream initiates a new matching sequence, independent of the matching sequences initiated by other characters. A subtlety arises when aggregating the match states across stages, where it is possible for a stage to both generate and receive a valid match state in the same clock cycle. However, due to the fact that the match states are pipelined in the same way as the state numbers, it can be shown that the *received* match state will always represent a proper prefix of the *generated* match state. When two match states "collide" in the same stage, only the match state generated at the higher (current) level needs to be reported, since it can *imply* the lower-level (previous) match state. This common-prefix implication can be derived statically from the dictionary. Note that if two pattern matches overlap with each other but do *not* have common prefix, the field-merge architecture will report both in different clock cycles.

IV. Analysis

In this section we prove the correctness of the field-merge string matching. Our goal is to show that a pattern of length L occurs in the input stream if and only if the field-merge pipeline reports a match to that pattern in the level L stage.

A. Mapping Strings to Coordinates

Definition 1: A *bit-field* of a character is a fixed selection of bits in the binary representation of the character. A string $\bar{s}$ is

	V	I	K	T	O	R
b2	3	2	2	3	2	3
b1	1	2	2	1	3	0
b0	2	1	3	0	3	2
v2	0.750	0.81250	0.8281250	0.839843750	0.84082031250	**0.8415527343750**
v1	-0.250	-0.18750	-0.1718750	-0.175781250	-0.17285156250	**-0.1735839843750**
v0	0.250	0.18750	0.2343750	0.222656250	0.22558593750	**0.2258300781250**

	T	R	O	J	A	N
b2	1	1	0	0	2	2
b1	1	0	3	2	0	3
b0	0	2	3	2	1	2
v2	-0.250	-0.31250	-0.3593750	-0.371093750	-0.37011718750	**-0.3698730468750**
v1	-0.250	-0.43750	-0.3906250	-0.386718750	-0.38964843750	**-0.3889160156250**
v0	-0.750	-0.68750	-0.6406250	-0.636718750	-0.63769531250	**-0.6374511718750**

Figure 4. String-to-coordinates conversion for "VIKTOR" and "TROJAN".

said to be *partitioned into k bit-fields* if each of its characters is partitioned into k bit-fields in the same way.
In general, a character in an alphabet of size 2^W can have at least one bit-field (the entire binary representation) and at most W bit-fields (one bit per bit-field).

Definition 2: Let a string $\bar{s} = \{s(1), \ldots, s(L)\}$ of finite length $L \geq 1$ be partitioned into k bit-fields. The i-th *bit sequence* of $\bar{s}$, $\bar{s_i}$, $1 \leq i \leq k$, is obtained by concatenating the i-th bit-field of $s(j)$, $1 \leq j \leq L$, in increasing order of j.

Theorem 1: Every string $\bar{s}$ of finite length $L \geq 1$ over an alphabet of size 2^W can be represented uniquely by a k-dimensional point $v = \langle v_1 \ldots v_k \rangle$, $k \leq W$, in the Cartesian system. The point v falls within the box bounded by $(-1, +1)$ along each of the k dimensions.

Proof: We prove the theorem by construction. Let $\bar{s}$ be partitioned into k bit-fields. Let $\bar{s_i} \triangleq \{s_i(1), \ldots, s_i(L)\}$, $1 \leq i \leq k$, be the i-th bit sequence of $\bar{s}$, where $s_i(j)$, $1 \leq j \leq L$, is the i-the bit-field of the j-th character of $\bar{s}$. Assume $s_i(j)$ has w_i bits, $\sum_{i=1}^{k} w_i = W$. It follows that $0 \leq s_i(j) \leq 2^{w_i} - 1$ for all i, j. Define the coordinate value v_i as

$$v_i \triangleq \sum_{j=1}^{L} [2 \cdot s_i(j) - (2^{w_i} - 1)] \times 2^{-w_i \cdot j}. \quad (1)$$

It can be shown from Eq. 1 that (1) the value of any v_i falls within $(-1, +1)$, and (2) $v_i = v_i'$ if and only if s_i and s_i' have the same length L and $s_i(j) = s_i'(j)$ for all $1 \leq j \leq L$. Then $v \triangleq \langle v_1 \ldots v_k \rangle$ is the coordinate of the k-dimensional point which uniquely represents $\bar{s}$. ■
Figure 4 shows two examples of string-to-coordinate conversion. In each example, the lower 6 bits of the characters' ASCII codes are partitioned into bit-fields b0, b1, and b2, each being 2-bit wide. Both the bit-field values (between 0 and 3) and the bit-field coordinates corresponding to the string prefix are shown.

B. Matching Strings versus Matching Bit-fields

Theorem 2: Two strings $\bar{s}$ and $\bar{s'}$ are identical if and only if they have the same length L and $\bar{s_i}$ and $\bar{s_i'}$ are identical for all $1 \leq i \leq k$.

Proof: According to Theorem 1, $\bar{s}$ and $\bar{s'}$ are identical if and only if $v = v'$ and $s_i(j) = s_i'(j)$ for all $1 \leq i \leq k$, $1 \leq j \leq L$. According to Definition 2, this is true if and only if $\bar{s_i}$ and $\bar{s_i'}$ are identical for all $1 \leq i \leq k$. ■
Having developed the equivalence between strings and bit sequences and that between strings and coordinates, we can now prove the correctness of the field-merge architecture for large-scale string matching (LSSM).

Theorem 3: The field-merge architecture described in Figure 2 and 3 correctly performs LSSM.

Proof: First we show that the field-merge architecture matches any fixed-length input string against the dictionary. Then we show that field-merge also matches a continuous stream of input characters against the dictionary to complete the proof.

According to Theorem 2, an input string and a dictionary pattern match each if and only if their corresponding bit sequences match each other in all bit-fields. This is precisely performed by the field-merge stage architecture (Figure 3), where each PSM (*i.e.*, next-stage circuit + stage-local STT) matches the bit sequence of the input string against the bit sequences of the dictionary patterns in one bit-field. The matching results from all PSMs are then combined to search the ATB for a full pattern match, which will be found if and only if corresponding bit sequences of the input string and the dictionary pattern match each other in all bit-fields.

The field-merge pipeline architecture (Figure 2) initiates a matching sequence starting at every input character. Suppose the maximum pattern length of the dictionary is L_{max}. Then each stage at level L, $1 \leq L \leq L_{max}$, matches every length-L substring in the input against all length-L patterns in the dictionary. Together all the pipeline stages of field-merge match substrings of all lengths (between 1 and L_{max}) in the input stream against the entire dictionary. This completes the proof. ■

V. Implementation and Performance Evaluation

A. Memory Size Evaluation

We evaluate the total number of states and memory usage per level using two dictionaries: (A) Roget's English dictionary, containing 160,652 characters in 21,667 patterns, and (B) Snort patterns of lengths 63 or less,[3] containing 103,329 characters in 6944 binary (8-bit) strings.

Figures 5 and 6 show the number of states in field-merge pipelines for the two types of dictionaries, Roget's and Snort, both partitiond into bit-fields $\langle 2\ 2\ 2\ 2 \rangle$. For Roget's, the highest bit-field 0x`c0` can be ignored because it always has bit values "`01`". We notice that the total number of states needed for Roget's dictionary is significantly less than the total number of characters in the dictionary, even though we have three state transition tables (one for each PSM) instead of one. This is because English words have a lot of information redundancy, which are greatly reduced by compiling the words

[3]We limit the pattern length from Snort because it is impractical to build a very long field-merge pipeline on a single chip due to the limitation of number of memory ports. The number 63 is chosen so that the level number can be encoded in 6 bits. More than 93% of the 7455 patterns are 63 characters or shorter, while the longest Snort pattern has 232 characters.

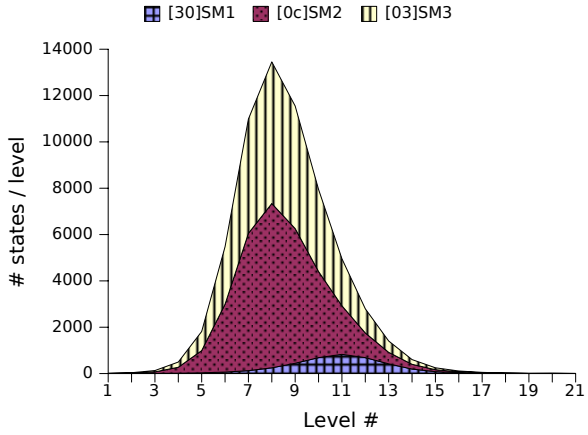


Figure 5. Distribution of number of states for Roget's English dictionary. Total 62,156 states matching 21,667 patterns with 160,652 characters.

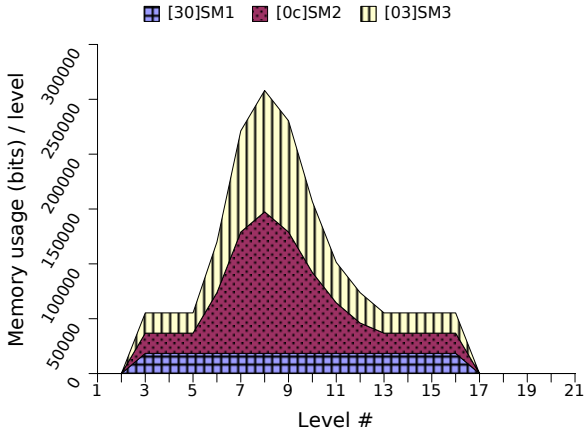


Figure 7. Distribution of STT memory for Roget's English dictionary with bit-fields $\langle 2\ 2\ 2\ 2 \rangle$. Total 1,548,288 bits BRAM and 150 LUTs dist. RAMs.

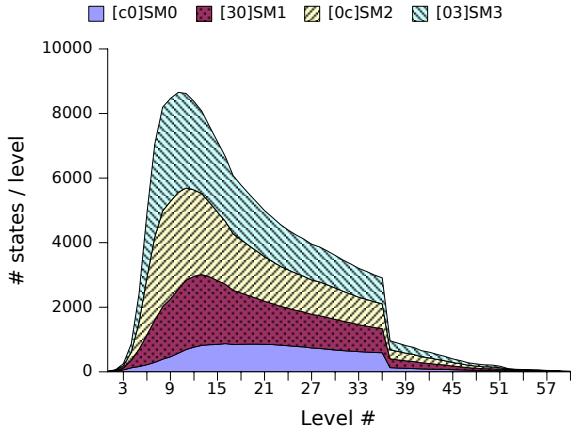


Figure 6. Distribution of number of states for Snort patterns. Total 179,306 states matching 6,944 patterns with 103,329 characters.

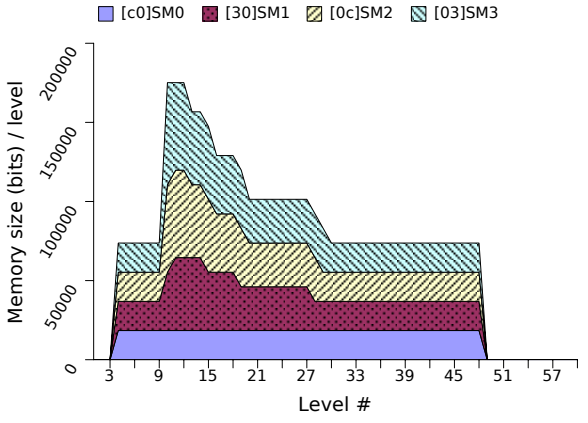


Figure 8. Distribution of STT memory for Snort patterns with bit-fields $\langle 2\ 2\ 2\ 2 \rangle$. Total 4,322,304 bits BRAMs and 1150 LUTs dist. RAMs.

into bit-fields. Note that in the previous bit-split approach, the whole dictionary is divided into small groups of words (usually from 16 to 32). This makes it less effective in removing the redundancy across different tiles and produces a lot more states in total.

For Snort patterns, we notice that the total number of states in the field-merge pipeline is ~ 1.74x higher than the total number of characters in the dictionary. There are two reasons for this phenomenon. Firstly, the patterns defined by Snort rules consist of characters with higher degree of randomness and less information redundancy. Secondly, the number of states is summed across all four PSMs, each with a different tree structure. While this may seem counter-productive in terms of compressing the state machine, we note that each state in the field-merge pipeline has $2^2 = 4$ next-state entries, compared to $2^8 = 256$ next-state entries in the original dictionary tree. Thus a 74% increase in the number of PSM states can still result in more than 30x memory saving.

Figures 7 and 8 show the total amount of memory used for the field-merge state transition tables (STT), implemented as both block RAM (BRAM) and distributed RAM on a Virtex 5 LX devices. The STT memory size is roughly proportional to the total number of states, subject to the following two exceptions:

1) The BRAMs are allocated in units of 18k bits. Thus the STTs have sizes in units of 18k bits.
2) STTs with less than 64 states (256 entries) are implemented as distributed RAM rather than BRAM.

There are 21 STTs for Roget's dictionary and 60 STTs for Snort patterns implemented as distributed RAMs, occupying about 150 and 1150 LUTs, respectively.

In addition to STTs, the field-merge pipeline also requires per-level auxiliary tables (ATBs). Figure 9 and 10 show the distributions of ATB entries across different levels. The size of an ATB is exactly proportional to the number of full match states in the level. When storing in memory, we also treat ATBs with 32 or fewer entries differently by implementing them as distributed RAM rather than BRAM. This avoids wasting a full chunk of 18k bits BRAM for small tables. On the other hand, we always allocate at least 50% more entries than needed

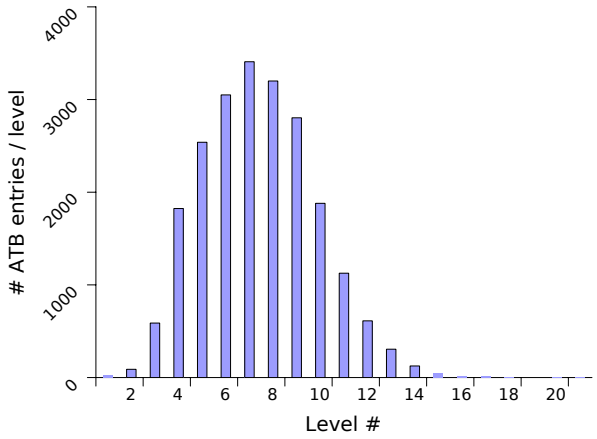


Figure 9. Distribution of ATB entries for Roget's dictionary, taking 1,225,728 bits BRAMs and 180 LUTs distributed RAMs.

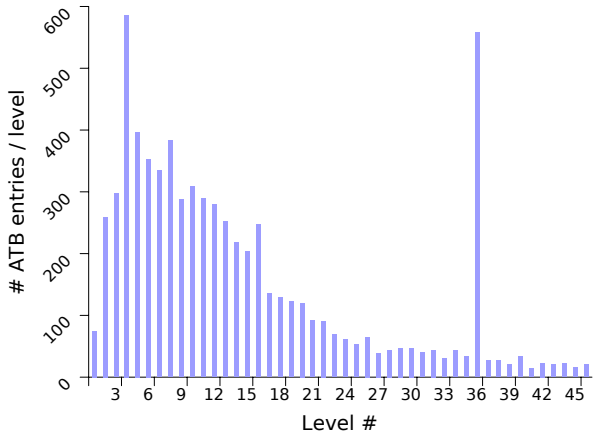


Figure 10. Distribution of ATB entries for Snort patterns, taking 912,384 bits BRAMs and 960 LUTs distributed RAMs.

for each ATB to avoid any potential conflict in our simple XOR hashing scheme (Section III-B). Overall the ATBs use 1,225,728 bits BRAM for Roget's dictionary and 912,384 bits for Snort patterns. The amount of distributed RAM used by ATBs for Roget's dictionary is about 5k bits (in ~180 LUTs), while that for Snort patterns is about 20k bits (in ~960 LUTs).

We observe that the memory requirement of the STTs and that of the ATBs can be quite different. The former is affected by the length and complexity of the patterns, while the latter by the number of unique patterns in the dictionary:[4]

- The Roget's dictionary consists of a larger number (~21k) of English words, which are relatively short and simple. This results in relatively larger ATBs (~1.2 Mb) and smaller STTs (~1.5 Mb).
- The Snort dictionary consists of a smaller number (~7k) of network traffic signatures, which are often long and complex. This results in smaller ATBs (~0.9 Mb) and larger STTs (~4.2 Mb).

[4]Both are also proportional to the logarithm of the number of partial match states, which dictates the number of bits needed to encode each entry.

Table II
PERFORMANCE COMPARISON OF FIELD-MERGE (F.M.) AND BIT-SPLIT (B.S.) STRING MATCHING APPROACHES.

Scenario	BRAM kb	Dist. RAM	B/ch	LUTs/ch	MHz
F.M./Snort	5,112	2100 LUTs	6.33	0.27	285
F.M./Roget's	2,709	320 LUTs	2.16	0.15	285
B.S./Snort	27,504	N.A.	34.1	(0.27)	(226.2)
B.S./Roget's	35,712	N.A.	28.5	(0.27)	(226.2)

Number inside parenthesis are taken directly from the base-case result in [7] and may be too optimistic for the scenarios here.

Overall the field-merge pipeline for Roget's dictionary requires 2,774,016 bits of BRAM and ~320 LUTs as distributed RAMs. The pipeline for Snort patterns requires 5,234,688 bits of BRAM and ~2100 LUTs as distributed RAMs. Both are well within the capacity of modern FPGA devices.

Note that the minimum memory requirement of field-merge is actually $5 \sim 15\%$ less than the values reported above. The difference includes the overheads due to both bit-width (multiples of 36) and block size (multiples of 512) constraints on FPGA and the extra space needed in the ATBs to avoid hash conflicts. We believe the inclusion of such overheads is necessary for a practical evaluation.

B. Implementation on FPGA

We implemented a 56-stage field-merge pipeline on a Virtex 5 LX330 device, using the circuit architecture of Figure 3. We choose 56 to be the number of pipeline stages due to the BRAM I/O constraint on the Virtex 5 LX device.[5] We allocate as much resource as possible for the state transition tables (STTs) and the auxiliary tables (ATBs). The goal is to design a field-merge circuit that not only can match the known dictionaries but also have head room for future pattern additions. The size of each STT and ATB is scaled up according to the distributions in Figure 8 and 10, respectively.

To get the highest clock frequency, we interleave two input streams in one pipeline to allow BRAM I/O to be fully buffered. Table I shows the statistics reported by the place and route routines. Utilizing the dual-port feature of the BRAM (as well as the distributed RAM) on Virtex 5 FPGAs, we are able to achieve a throughput of 4.56 Gbps over four input streams, or a per-stream throughput of 1.14 Gbps. If we reduce the circuit to a 24-stage pipelining using only 512k bits of BRAM, a clock rate of 347 MHz can be achieved. Such a smaller design will be able to match 5.55 Gbps input (dual input streams) against a dictionary of about 3,000 English words or 650 binary IDS patterns.

C. Performance Comparison

Table II compares the performance of field-merge and bit-split on FPGAs with respect to memory usage, resource efficiency, and clock frequency. We did not compare the

[5]A Virtex 5 LX 330 has 288 BRAM I/O ports. A 56-stage $\langle 2\,2\,2\,2 \rangle$ field-merge pipeline would require $56 \times 5 = 280$ BRAM I/O ports, just under the 288 limit. A 57-stage field-merge pipeline is also feasible, but can severely hurt the achievable clock frequency of the pipeline.

Table I
STATISTICS OF THE LARGEST FIELD-MERGE IMPLEMENTATION ON VIRTEX 5 LX330 FPGA.

# LUTs total	# LUTs dist. RAM	# slices used	# 18k BRAM	Freq. MHz	Tput Gbps
30,616 (14%)	5,840	12,027 (23%)	560 (97%)	285	4×1.14

field-merge directly with other bloom filter or hashed AC-DFA approaches. While both are intellectually challenging and academically interesting, the bloom filter approach can have false positive whose effects to real-world applications (*e.g.*, intrusion detection) is difficult to quantify. The hashed AC-DFA approach usually has a memory performance highly dependent on the dictionary structure, while its throughput estimate often disregards the implementation complexity of the hash functions. Both make meaningful comparison with field-merge very difficult. The memory usages of field-merge are from the calculations in Section V-A, while the clock rates are those of the largest implementation in Section V-B. For bit-split, we follow the methodology used in [7] and compile each dictionary into small tiles of 28 patterns. The memory usage includes both state transition entries and partial match vectors, while the clock frequency is the highest found in [7].

The comparison shows that for the same dictionary, field-merge requires only 1/5 to 1/13 the memory of bit-split while achieving 25% higher clock frequency. If we use the smaller field-merge pipeline (347 MHz) for comparison, then the clock rate advantage of field-merge is over 50%. Note that the bytes per character (B/ch) values that we calculate for bit-split is slightly lower than the result in [7]. This is likely due to the different dictionaries used in the two implementations. In particular, in our evaluation of bit-split, the dictionary patterns are sorted before added to a tile, resulting in the most common prefixes. Note that for field-merge, all patterns in the dictionary are compiled into the same set of PSMs and the order in which patterns are added makes no difference.

VI. CONCLUSION AND FUTURE WORK

We proposed the field-merge architecture for large-scale string matching. While it is similar to the bit-split approach, the proposed architecture is easier to construct, achieves higher clock rates, and uses much less on-chip memory. By *not* using the Aho-Corasick algorithm, the aggregation of the partial match results can be separated from the traversal of the partial state machines, allowing both parts to be implemented by simpler circuits. Our prototype circuit constructed on the Virtex 5 LX330 FPGA can hold more than 7,000 binary patterns from Snort or several copies of the 21k words from Roget's English dictionary.

For future work, we are looking at techniques to match multiple characters per clock cycle and to reduce the total number of stages in the pipeline. We are also looking at techniques to allow more power-efficient operations, for example, to turn off a later stage conditionally when it does not hold a valid (non-zero) state. Since statistically a longer pattern would be matched less frequently, higher level stages can have more power saving opportunities.

REFERENCES

[1] Clam AntiVirus. http://www.clamav.net/.

[2] SNORT. http://www.snort.org/.

[3] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[4] Zachary K. Baker and Viktor K. Prasanna. A Methodology for Synthesis of Efficient Intrusion Detection Systems on FPGAs. In *Proc. of 12th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, April 2004.

[5] João Bispo, Ioannis Sourdis, João M. P. Cardoso, and Stamatis Vassiliadis. Regular expression matching for reconfigurable packet inspection. In *Proc. of IEEE International Conference on Field Programmable Technology (FPT)*, pages 119–126, December 2006.

[6] C.R. Clark and D.E. Schimmel. Scalable pattern matching for high speed networks. In *Proc. of 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 249–257, April 2004.

[7] Hong-Jip Jung, Z.K. Baker, and V.K. Prasanna. Performance of FPGA Implementation of Bit-Split Architecture for Intrusion Detection Systems. In *Proc. of Int. Parallel and Distributed Proc. Sym. (IPDPS)*, April 2006.

[8] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. Exact multi-pattern string matching on the cell/b.e. processor. In *Proc. of 2008 Conf. on Computing Frontiers*, pages 33–42, 2008.

[9] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. High-Speed String Searching against Large Dictionaries on the Cell/B.E. Processor. In *Proc. of IEEE Int. Parallel & Distributed Proc. Sym. (IPDPS)*, 2008.

[10] Ioannis Sourdis and Dionisios Pnevmatikatos. Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System. *Lector Notes in Computer Science*, 2778:880–889, 2003.

[11] Lin Tan, Brett Brotherton, and Timothy Sherwood. Bit-Split String-Matching Engines for Intrusion Detection and Prevention. *ACM Trans. on Architecture and Code Optimization*, 3(1):3–34, 2006.

[12] Lin Tan and Timothy Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *Proc. of International Symposium on Computer Architecture*, pages 112–122, 2005.

[13] Norio Yamagaki, Reetinder Sidhu, and Satoshi Kamiya. High-Speed Regular Expression Matching Engine Using Multi-Character NFA. In *Proc. of International Conference on Field Programmable Logic and Applications (FPL)*, pages 697–701, Aug. 2008.

[14] Yi-Hua E. Yang, Weirong Jiang, and Viktor K. Prasanna. Compact Architecture for High-Throughput Regular Expression Matching on FPGA. In *Proc. of 2008 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, November 2008.