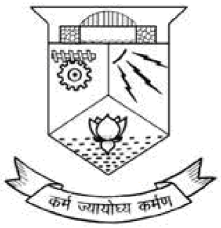
# A Tool For Very Fast Regular Expression Matching

*A Seminar Report*
*Submitted in partial fulfilment of*
*the requirements for the award of the degree of*

*Master of Technology*
*in*
*Computer Science and Engineering*

*by*

**Samyuktha M.**
**M105113**




**Department of Computer Science & Engineering**
**College of Engineering Trivandrum**
**Kerala - 695016**
**2010-11**

**College of Engineering Trivandrum**
**Department of Computer Science & Engineering**

*Certified that this Seminar Report entitled*

# A Tool For Very Fast Regular Expression Matching

*is a bonafide record of the seminar presented by*

**Samyuktha M.**
**M105113**

*in partial fulfilment of*
*the requirements for the award of the degree of*
*Master of Technology*
*in*
*Computer Science & Engineering*

---

**Mr. Rameez Mohammed A**
*Guide*
*Faculty*
*Dept.of Computer Science & Engineering*

**Dr. M.S Rajasree**
*Professor and*
*Head*
*Dept.of Computer Science & Engineering*

# Acknowledgement

I would like to express my sincere gratitude and heartful indebtedness to my guide **Mr Rameez Mohammed A**, Lecturer , Department of Computer Science And Engineering for his valuable guidance and encouragement in pursuing this seminar .

I am thankful to **Dr. M S Rajasree**, Head of the Department , **Shine S** , P.G Coordinator, **Shreelekshmi R**, Asst. Professor and **Anvar A**, Asst. Professor and project coordinator Department of Computer Science and Engineering for their help and support.

I also acknowledge my gratitude to other members of faculty in the Department of Computer Science And Engineering and all my friends for their whole hearted cooperation and encouragement.

Above all I am thankful to the God Almighty.

**Samyuktha M.**

# Contents

# 1 Abstract

DotStar is a tool for very fast regular expression matching. It is an innovative algorithmic solution that compiles user provided regular expressions into a compact automaton using a sequence of more manageable intermediate representations. The resulting automaton can search using a single pass without backtracking and is both space and time efficient. Regular expressions are a common choice for defining configurable rules for data parsing because of their expressiveness in detecting recurrent patterns and information. For many data intensive applications, regular expression matching is the first line of defense in performing online data filtering. Unfortunately, few solutions can keep up with the increasing data rates and the complexity posed by sets with hundreds of expressions. DotStar addresses this problem by providing a complete algorithmic solution and a software tool chain that can compile large sets of user provided regex first into a sequence of intermediate representations.

# 2 Introduction

DotStar is an innovative algorithmic solution that compiles user-provided regular expressions into a compact automaton using a sequence of more manageable intermediate representations. The resulting automaton can search using a single pass without backtracking, and is both space and time efficient. Regular expressions, or regex, are a common choice for defining configurable rules for data parsing because of their expressiveness in detecting recurrent patterns and information. For many data intensive applications, regex matching is the first line of defense in performing online data filtering. Unfortunately, few solutions can keep up with the increasing data rates and the complexity posed by sets with hundreds of expressions. DotStar addresses this problem by providing a complete algorithmic solution and a software tool chain that can compile large sets of user provided regex first into a sequence of intermediate representations and then into an automaton that can search for matches in a single pass without backtracking. The entire software tool chain supports the extended Posix standard syntax for regex.

Regex concisely describe a set of strings without explicitly listing the set content. Each expression consists of one or more strings connected with a set of operators such as alternate (—), which chooses among two strings repetition (*), which repeats a string zero or more times and optional (?). Given an input string, a matching operation determines if that string is a possible pattern instance. An example is ABC*D, which recognizes any string that starts with AB, continues with zero or more C, and finishes with D. Sample matching input strings might be ABD,ABCD,or ABCCCD.

Very fast regex matching is currently a hot topic in applied research, with more applications searching large pattern sets with increasingly faster data streams. Applications include scanning for signatures as part of antivirus software, selecting the proper tag using path expression in XML applications, pattern matching of DNA sequences in genome research, and traffic flow classification using deep packet inspection at gigabit per second rates. Deep packet inspection is one of the most demanding applications, and regex are a common threat detection mechanism in both commercial and open source NIDSs.

Unfortunately, the automata that medium to large regex sets generate require prohibitive amounts of memory, and the DFA and NFA based regex implementations suffer from amnesia and acalculia.Amnesia is the inability to efficiently follow the progress of multiple partial matches.which forces the use of a separate state for each combination of a partial match. Acalculia is the inability to count subexpression occurrences.DotStar addresses both these weaknesses. It is as fast as a DFA and uses counting and status bits to avoid state explosion.

# 3 Dotstar

At the core of DotStar is a large class of regex (the DotStar class) that DotStar tools can compile into an Aho-Corasick automaton the de facto standard for keyword scanning going through a series of intermediate representations that include the Glushkov automaton. Because Aho-Corasick automata remember all the suffixes of pattern instances at any given state, DotStar can dramatically reduce memory requirements by eliminating one of the major causes of amnesia.

To handle the regex that do not belong to the DotStar class,A set of classification and rewriting rules are developed that automatically fragment more complex regex into a collection of equivalent DotStar expressions. The runtime system connects the recognized fragments using status bits and locations and leverages bit level parallelism and the vector operations available on many commodity processors. Because bits and locations can count the number of specific subexpression repetitions, acalculia is no longer a concern. As Figure (1) shows, the DotStar tool chain comprises five elements.

## 3.1 Elements Of Dotstar

1:classification engine
2:expander
3:compiler
4:compactor
5:runtime engine

## 3.2 Classification Engine

A classification engine/preprocessor discriminates between DotStar expressions and the more complex ones. An expander transforms complex expressions in a collection of DotStar expressions, together with the data structures that will be used at runtime to resolve the dependencies between them. A compiler implements DotStar expression combining and outputs both an NFA description and a list of add on data items. A compactor reads the NFA description and builds a compact DFA representation. Finally, a runtime engine reads the binary representation, selects a specific runtime code depending on the required add on items, and streams input data through it. In Figure 1, ABC*D and BCCE belong to the DotStar class, and the expander must fragment the other two expressions into four subexpressions (B[C-E] C, CE*G, ABC*D, and BCCCE).

## 3.3 Compilation Steps

Figure 2 describes the four compilation steps to transform a set of DotStar regex into an NFA. DotStar is based on an innovative evolution of the Aho-Corasick keyword scanning algorithm. This algorithm operates on a tree containing a composite representation of all keywords; the tree transforms into an NFA through the addition of a fail function, F(), which points to the longest proper suffix already recognized. The execution process for each symbol proceeds at most once across a keyword tree edge or a finite number of times across F()

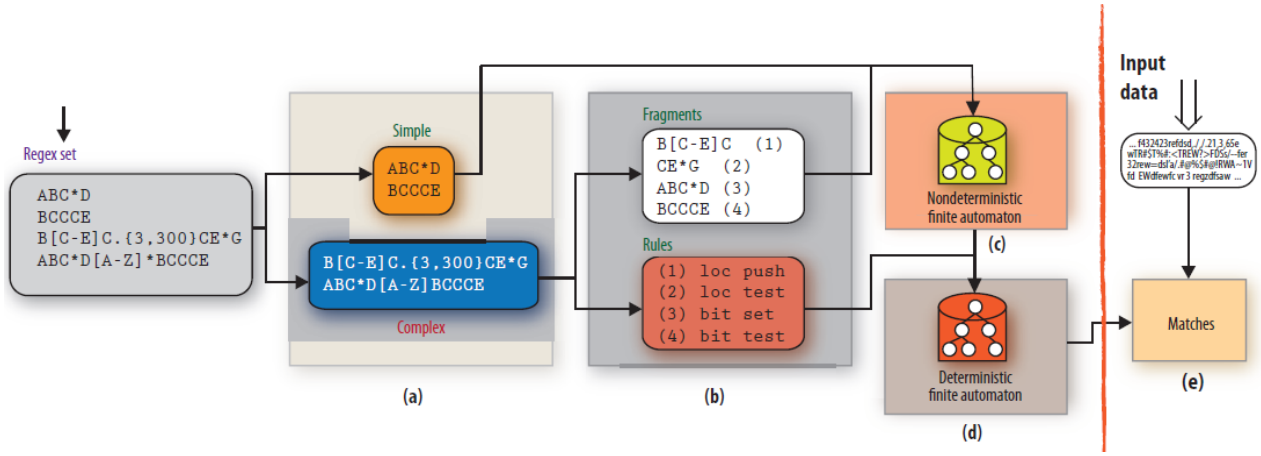



Figure 1: Steps In Dotstar

edges until the algorithm either detects a proper suffix or reaches the root node. The DotStar algorithm is innovative because it extends the keyword tree with a specific kind of loop, turning it into a keyword graph.

## 3.4 Creating a Keyword Graph

Several definitions are key to understanding the DotStar compilation process. A DotStar regex is a string built from a finite symbol alphabet character classes over the alphabet (such as [a-z] or the wildcard .) and the grouping, alternative, and closure operators. The closure operator cannot be applied to character classes.

A keyword graph is a directed graph with the following characteristics:

- It has an initial (root) node.
- Every edge is labeled with a symbol.
- Any two edges leaving a node have different symbols, and all edges that enter a node have the same symbol.
- Any two cycles in the graph do not share edges unless one contains the other.

The concatenation of edge labels along a complex path from the root node to any final node defines an instance of a regex pattern. A Glushkov automaton is an NFA described by the tuple (S, S, i, F, d), where S is the set of states (m + 1 for an m symbols expression) S is the alphabet i is the initial state F is the set of final states and d is the automatons transition function. The compiler uses the Glushkov automaton as an intermediate step in building a keyword graph for a DotStar regex. Figures(3a) and(3b) show a sample Glushkov automaton and a sample keyword graph for ABC*D.

A Glushkov automaton has several interesting properties.No empty transitions, and it is homogeneous.It has two special nodes, the initial node, ni, and the final node, nf, such that no edge enters ni, no edge exits nf, and every other node is in a path from ni to nf .Every maximal orbit in the graph representing the automaton is strongly stable. Thus, an edge exists between any orbit exit node (vertex with outgoing edges not included in the orbit) and any orbit enter node (vertex with incoming edges not included in the orbit).Every maximal orbit in

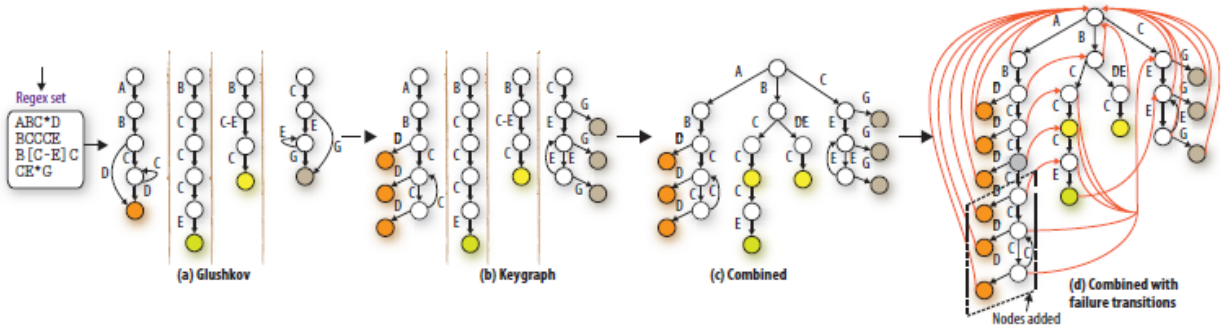



Figure 2: The four main compilation steps for DotStar regex. (a) The compiler transforms regex into Glushkov automata, (b) expands Glushkov automata in keyword graphs, (c) combines keyword graphs in a single graph, and (d) extends the keyword graph with a fail function, F().

the graph representing the automaton is strongly transverse. Consequently, if a node outside the orbit has an edge toward one orbit enter node, it must have edges to every other orbit enter node.

## 3.5 Combining Keyword Graphs

Figure 3c shows a sample combined keyword graph. Combining starts with an empty keyword graph, which contains only the root node, and then adds one keyword graph at a time. The procedure is designed to maintain the basic keyword graph properties, eventually replicating nodes and subtrees to disambiguate partially overlapping patterns that contain closures.

## 3.6 Computing Fail-Function

In computing F(), the compiler tries to identify the longest proper suffix of another pattern instance already recognized while matching the current one. The basic Aho Corasick algorithm visits the graph breadth first and, for each node, computes the F() of that node's children using the parent node's F(). The breadth first visit ensures that, if n is the length of the path from the radix to the current node, all patterns with length n - 1 have a correct F() defined. With modifications, this algorithm can deal with loops and edges that have character class labels

When the F() computation reaches a loop's backward edge, the target node will already have an F(). The compiler computes a new F() using the backward edge source node and compares it with the one already present. If the recognized pattern length for the new F() is greater than the old length, the compiler must unroll the loop and continue processing down the path. When the F() computation reaches a node using a character set and discovers that it should have different F() targets depending on the input symbol, it duplicates the node, splitting the character class edge into nonintersecting subsets, and sets a proper F() for each one.

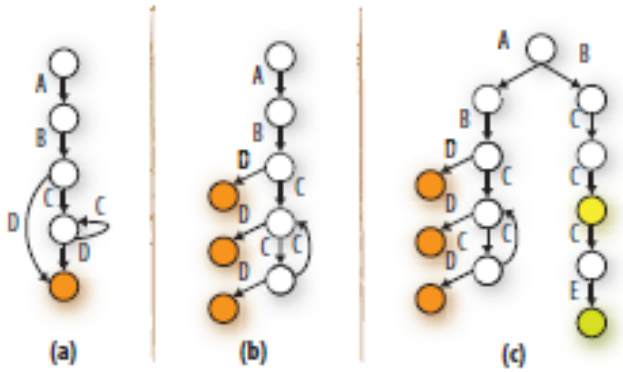



Figure 3: Building a keyword graph for a DotStar regex ABC*D. (a) Compilation starts with the creation of a Glushkov automaton and continues with a breadth-first visit of the NFA. (b) The result is a keyword graph that represents all the possible complex paths from the start node to every final node. (c) Combining merges multiple keyword graphs into a single one.

### 3.7 Handling Complex Expressions

DotStar can also accommodate the complex regex of the extended Posix standard. A complex regex is a string built from a finite symbol alphabet character classes over the alphabet and the grouping, alternative, closure, optional, and repetition operators. Examples of complex regex are A[a-z]*B and START.*MID.3,300END.The DotStar algorithm described so far will experience state explosion if an expression contains a character class inside a closure or a repetition operator. The problem arises while computing F(), since the process unrolls loops in the graph when two suffixes overlap if the loop label is a character class or wild card, the number of unrolls could become infinite. For example ".*" would appear in the keyword graph as a loop with a wild card label, making any other regex pattern instance the longest proper suffix. The result will be a complete replication of the graph.

Figure 4 shows the resulting automaton after the compiler computes the F() of ABC*D and BCCCCE.

To address this problem,solution modifies the Aho-Corasick runtime system by compressing sections of the automaton. Specifically, the expander annotates NFA states with actions and tests to be performed when the runtime system visits the state. All changes are modular, and our runtime system can use a minimal set of add on items, depending on the regex set's specifics. The add on data that compresses sections of the automaton consists of four elements. The status bit is a Boolean value associated with a closure over a wild card. The runtime system sets the bit when it recognizes the prefix and tests the bit when it detects the postfix to validate the match.

Another element is the masked status bit a Boolean value associated with a closure over a character set. Again, the runtime system sets the bit when it recognizes the prefix and tests the bit when it detects the postfix. It can also clear the bit while examining the input data if the symbol does not belong to the character set. The other two elements are storage items. Location contains an offset in the input stream and is associated with a bounded or unbounded repetition operator over a wild card the runtime system stores the current input stream offset when it detects the prefix and checks the value when it recognizes the postfix. Masked location contains an offset in the input stream and is associated with a bounded or unbounded repetition

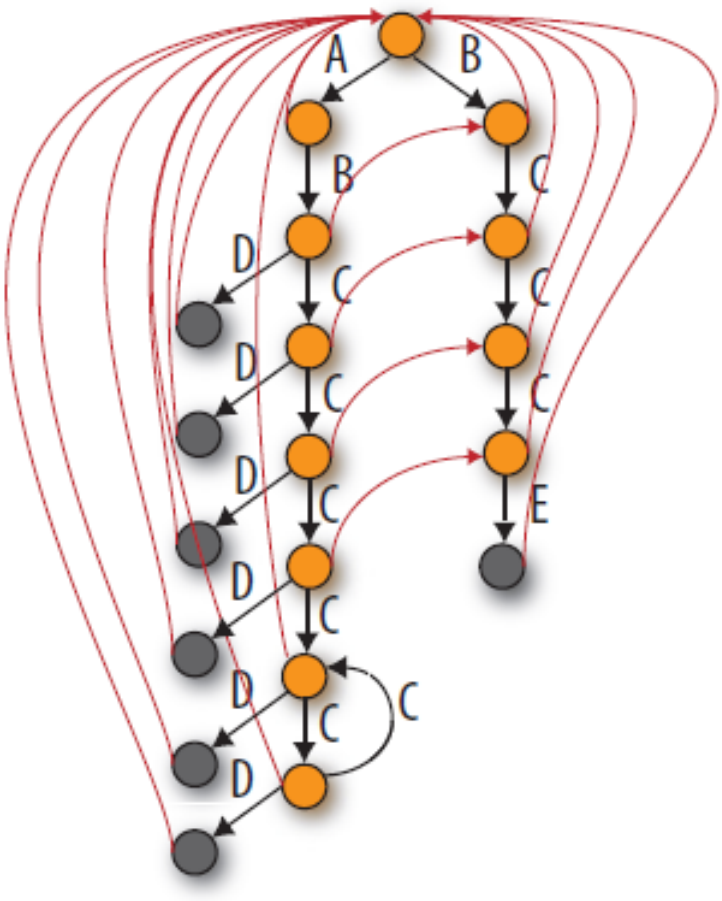



Figure 4: Finding Fail-Function

operator over a character set. Again, the runtime system stores the current input stream offset when it detects the prefix, checks the value when it recognizes the postfix, and can clear the value while examining the input data if the symbol does not belong to the character set.

The DotStar preprocessor examines the regex set, recognizing the problematic parts, and the expander fragments them into multiple DotStar expressions. The expander allocates a minimal set of add on data items and annotates expressions with set and test operations. Expressions that contain both a status bit set and a test or compare annotation become conditional set expressions. If they contain location push and a test or compare annotation, they become conditional push expressions.

Figure 5 shows the DotStar NFA for the regex set in Figures 1 and 2. Specific nodes in the automaton are marked with operations or tests to be executed at runtime when the automaton reaches a certain state.

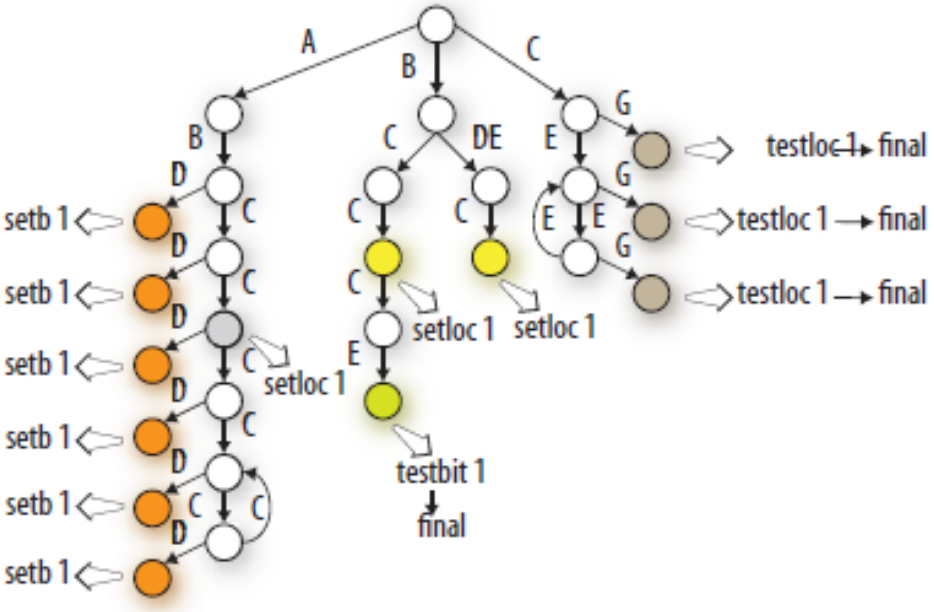



Figure 5: DotStar NFA

# 4 Runtime Behaviour

Figure 6 shows the DotStar runtime behavior during a matching operation. Specific states in the automaton trigger runtime operations on the add on data structure, which stores status bits and location values. The figure shows the parsing of an 18-byte input string using the sample NFA built for recognizing B[C-E]C.3,300CE*G, BCCCE, ABC*D[A-Z]*BCCCE, and ABC*D. The leftmost graph shows the automaton transitions that occur when the runtime system parses the first four characters; at the fourth byte, it reaches a setb, which tells it to mark a status bit because it detected the prefix for ABC*D[A-Z]*BCCCE.

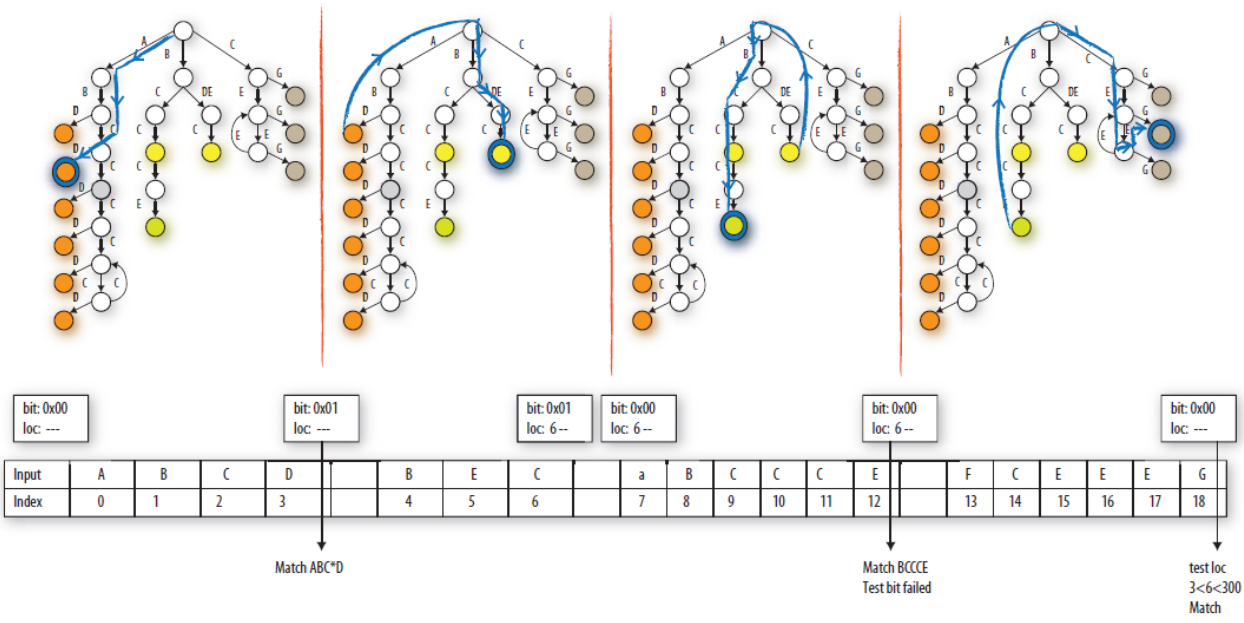



Figure 6: Runtime Behavior

This is also a final state for ABC*D, and the match is reported. Processing resumes as the runtime system follows the failure transition to the root node down the middle branch (second graph from the left). After three more bytes, a setloc state is reached, which detects a prefix for the fourth expression and stores the current input stream position inside location memory. The next character is an "a" which does not belong to the character set used in the third expression thus, the masked status bit is cleared. Processing continues across the automaton to reach yet another final state (third graph from left), which detects both BCCCE (whose match is reported) and a potential postfix for the third expression. This potential match is invalid, since the status bit is not set and the match is not reported. Finally, processing resumes (rightmost graph) and reaches a testloc state, which validates a match for the first expression.

In real-world regex sets, only a small number of automaton states require add-on actions relative to the overall number of states (less than 7 percent on Linux NIDS, for example), and the number of executions at runtime is very small (less than 1 percent with heavy hitting test data). Moreover, typical regex sets require multiple operations for each marked state As part of manipulating the regex set, the expander transforms operators and creates identical fragment prefixes. When combined, these fragments create hot spot states that will require several operations at runtime. As a result, the runtime system can exploit bit level parallelism and vector operations to execute the add-on actions in parallel.As Figure 7 shows, the system stores status bits and location values in vector registers. After every input character, it ANDs each vector register to implement masked status bits and locations.

The extended automata states contain a bit set mask, which the runtime system ORs with the current value, and a location value push mask, which the runtime system uses along with splat and mask operations to set all the required locations at once. Extended states also contain bit test registers, which the runtime system ANDs with current status bit values, as well as location compare vectors, which the runtime system handles using vector splat, arithmetic, compare, and gather instructions.The runtime system can speculatively or conditionally execute all these operations, depending on the target architecture support for branch prediction. With this data layout, the system can exploit all the available parallelism in each transition's bookkeeping operations.

# 5 Experimental Result

In the experimental evaluation,applied DotStar to regex in three categories:

- XML tokenization, which uses a compact regex set to extract tags, attribute names and values, cdata chunks, and so on. A standard XML compliance test suite served as the input data.
- SMTP parsing, which uses a very small regex set to capture subpatterns and extract protocol components. Test input data was the SMTP streams of the Berkeley dumps.

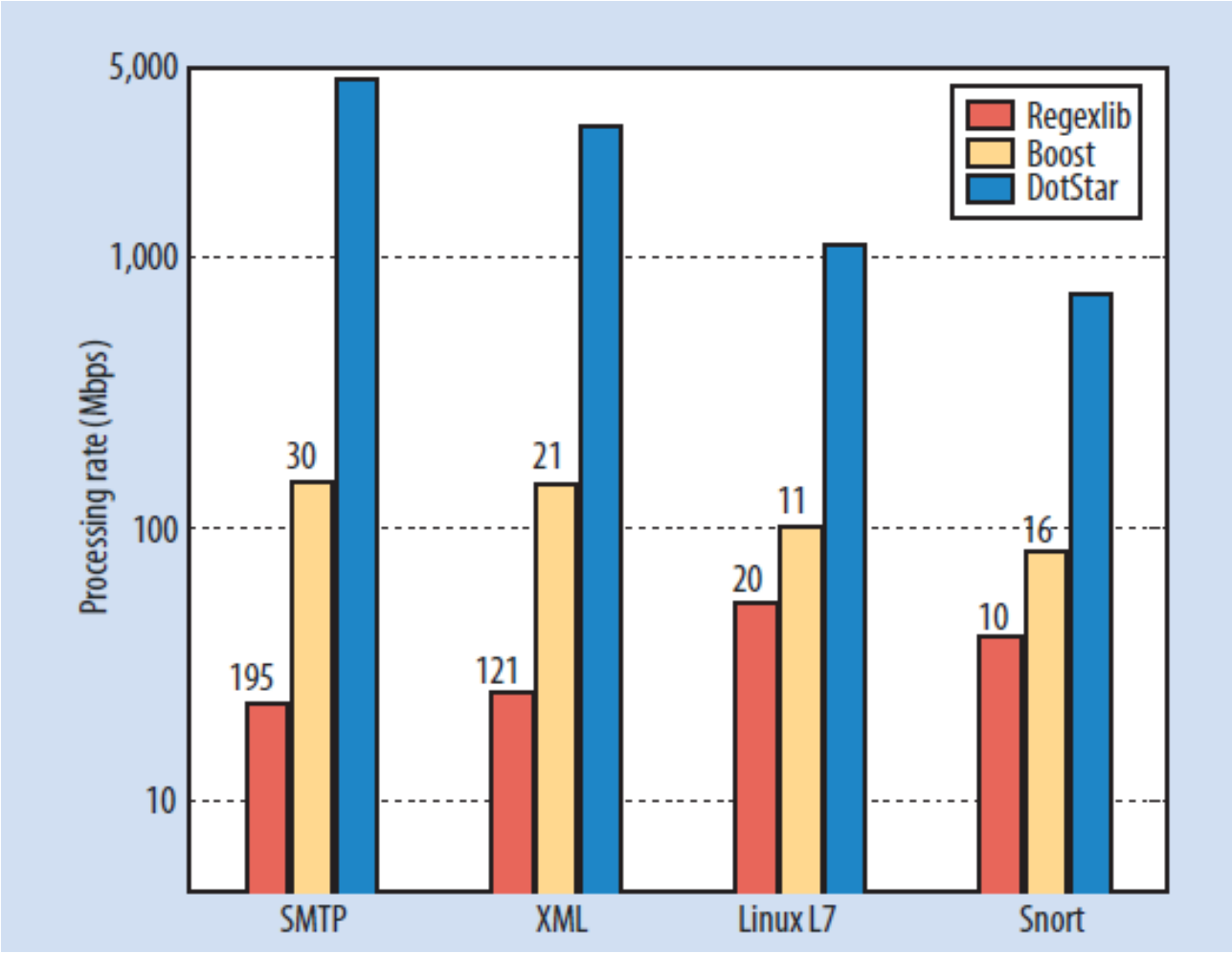



Figure 7: Experimental Result

- Network intrusion detection, which matches the input data stream against a large set of complex patterns, working with 8-bit binary data.Used the Linux L7 traffic classification and the Snort intrusion-detection patterns. The Berkeley traffic dump served as input data.

  DotStar also compares well with the FPGA/accelerator implementations for the NIDS regex sets. Results for these implementations, which had only 200 regex, were from1 Gbps7 to 4.4 Gbps8 and 8.06 Gbps.9 In contrast,Snort rule set has 500 regex, and the Linux L7 implementation has 150 regex.

# 6 Conclusion

DotStar offers a novel approach for building an automaton to recognize both small and large regex sets. With a structure similar to that of the familiar Aho-Corasick automaton, it can detect in a single pass the exact and exhaustive matching of every regex, including overlapping matches. The mechanisms designed effectively cope with state explosion.Experimental evaluation shows that DotStar efficiently parses both small and large regex, reaching a very high matching speed for regex sets with a range of characteristics.Results are significantly better than other state of the art NFA and DFA based systems, such as Regexlib and Boost, and DotStar offers a more flexible solution to acceleration. With many core processors, such as the Intel Larrabee on the horizon, Serves as a data point in the lively debate between special and general purpose acceleration

# References

[1] An Improved DFA for Fast Regular Expression Matching Domenico Ficara, Stefano Giordano, Gregorio Procissi, Fabio Vitucci, Gianni Antichi, and Andrea Di Pietro

[2] Davide Pasetto, IBM Computational Science Center, Ireland Fabrizio Petrini and Virat Agarwal, IBM T.J. Watson Research Center, Yorktown,Fast Regular Expression Matching(dotstar)

[3] G. Navarro and M. Raffinot, Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences, Cambridge University Press, 2002.