RC24645 (W0809-080) September 22, 2008
Computer Science

IBM Research Report

DotStar: Breaking the Scalability and Performance Barriers in Regular Expression Set Matching

Davide Pasetto
IBM Technology Campus
Dublin Software Lab
Mulhuddart
Dublin 15
Ireland

Fabrizio Petrini
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
USA

DotStar: Breaking the Scalability and Performance Barriers in Regular Expression Set Matching

Davide Pasetto[1] and Fabrizio Petrini[2]

[1]IBM Technology Campus
Dublin Software Lab
Mulhuddart, Dublin 15, Ireland
pasetto_davide@ie.ibm.com

[2] IBM TJ Watson Research Center
Cell Solution Department
Yorktown Heights, NY 10598 USA
fpetrin@us.ibm.com

Abstract

Regular expressions are widely used to parse data and to detect recurrent patterns and information: they are a common choice for defining configurable rules for a variety of systems. In fact, many data-intensive applications rely on regular expression parsing as the first line of defense to perform on-line data filtering. Unfortunately, few solutions can keep up with the increasing data rates and the complexity posed by several hundreds of regular expressions.

In this paper we present DotStar (.), a complete software tool-chain that can compile a set of regular expressions into an automaton that is highly optimized to run on multi-core processors with vector/SIMD extensions. DotStar relies on several algorithmic breakthroughs, to transform the user-provided regular expressions into a sequence of more manageable intermediate representations. The resulting automaton is both space and time efficient, and can perform the search in a single pass, without backtracking.*

The experimental evaluation, performed on a state-of-the-art Intel quad-core processor, shows that DotStar can efficiently handle both small sets of regular expressions, such as those used in protocol parsing, and much larger sets like the ones designed for Network Intrusion Detection Systems (NIDS). The experimental results show that we can achieve processing rates ranging from 2.2 Gbits/sec with the more demanding sets of NIDS expressions, to 5 Gbits/sec with XML parsing, with a performance speedup of almost two orders of magnitude when compared to popular libraries such as Boost, reaching, and in some case exceeding, the performance of specialized ASICs and FPGAs.

1 Introduction and Related Work

Very fast regular expression matching is currently a “hot topic” in applied research with more and more applications searching large set of patterns in increasingly fast data streams. Regular expression applicability is very broad. For example anti-virus software use regular expression to scan for virus signatures in files and data; XML parsing and rewrite applications are based on selecting the proper tag in the hierarchy using a path expression; genome researchers need to match DNA sequences and patterns in their data; modern network devices must accurately classify traffic flows at Gigabit rates using deep packet inspection.

Due to the current growth rate in network threats and network speeds, deep packet inspection is probably the application whose needs are growing faster, since every modern NIDS systems switched from using a string based dictionary to a regular expression engine to be able to accurately detect threats. Regular expressions are used in commercial NIDS, such as Cisco [2] and TippingPoint [15] as well as open source NIDS, like Snort [13], Bro [6], and Linux Application Protocol Classifiers (l7-filters) [7].

Very fast parsing of XML documents and protocols is essential for improving the performance of current and future network services. Although protocol parsing can be expressed as a sub-case of regular expression matching, existing parsers are often tailored to custom-designed accelerators, due to the poor performance of general-purpose regular expression engines [18] [8]. Several other data intensive problems can be easily expressed by sets of regular expressions, but again the performance of existing solutions limits their applicability.

Matching input data against a set of regular expressions can be a very demanding task and greatly depends on the complexity of the regular expressions. The classical approach is to build either a Non-deterministic Finite Automaton (NFA) or a Deterministic Finite Automaton (DFA) that encode the expression set. Both solutions suffer from structural problems that limit their potential in systems that want to recognize large sets of complex regular expressions at line-speed.

NFAs can potentially require exponential time to simulate the automaton using backtrack, and also generate *branchy* code that seriously affect the performance of multi-core processors. The main problems of DFAs are the inability of the automaton to remember that it is currently matching a specific pattern (which forces a complete state expansion, leading to exponential memory requirements) and the inability to count transitions (which, again, forces a complete expansion of every alternative, with analogous exponential space requirements). To the best of our knowledge, there are no theoretical results or practical algorithms that can attack the critical issues –space and time, in a unified framework; for example PERL regular expressions matching is proven to be NP-hard when using back-references [9].

To overcome these difficulties, and enhance matching speed, several researchers have attacked the problem using an arsenal of heuristics and optimizations. Solutions available in literature cover a wide range of techniques, ranging from memory compression, to reduce the memory footprint, to bit-level parallelism when simulating NFAs or DFAs, to speed-up run-time processing. The complexity can also be reduced restricting the available operators or modifying the match semantic (for example matching shortest strings only, or avoiding matching expressions inside Kleene closures). Another option is to partition the expressions into sub-sets, running multiple parallel automata, to reduce the number of states [17]. Or we can modify the DFA to encode more information in the graph and reduce space requirements (e.g. Delayed Input DFA) [5]. Or partition the DFA into a fast portion and a slow portion, where the fast portion matches the beginning of a pattern instance and eventually triggers the slow portion (bifurcated pattern matching); add a match history to a DFA, which will allow the use of conditions on DFA edges (History Based DFA - H-FA) or add counters to a history based DFA to allow conditions based on number of symbols recognized (History based counting DFA - H-cFA) [4].

2 Contribution

In this paper we present DotStar, a complete tool-chain that builds a deterministic finite automaton for recognizing regular expressions. The automaton that is generated at the end of the compilation process is an extension of the Aho-Corasick, the *de facto* standard for keyword scanning algorithms. DotStar

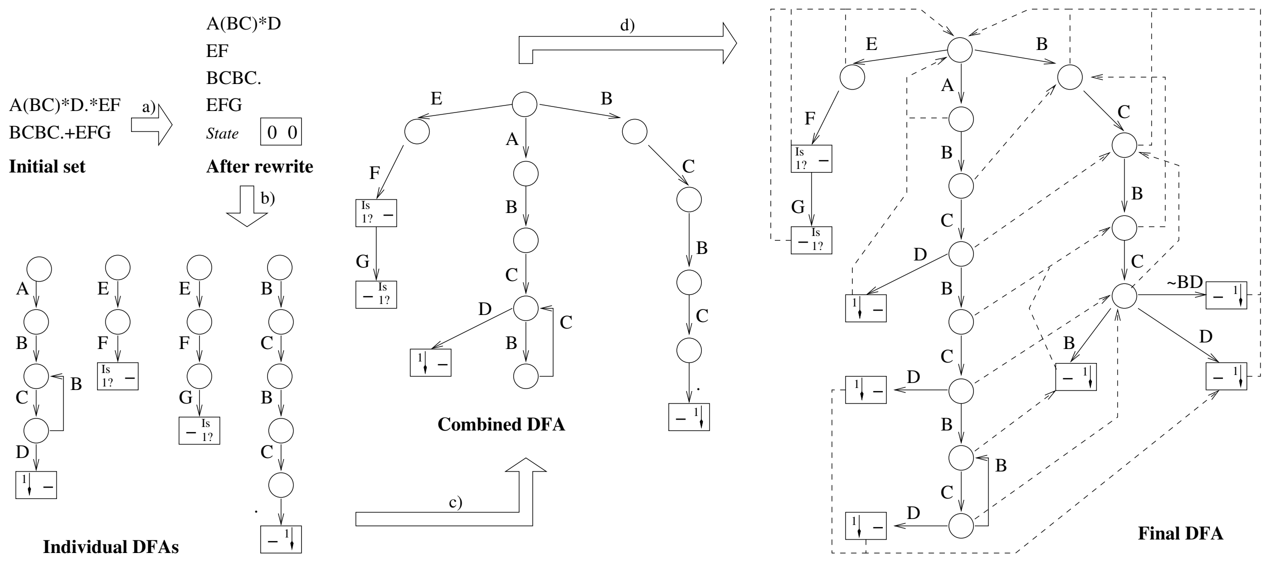


Figure 1: A graphical representation of DFA building steps. Step a) is the pre-processor (expander), which minimizes the state explosion and is described in section 4, step b) builds individual automata for each regular expression, c) combines them in a unique automaton and d) computes the failure function: they are described in section 3. The process marks some states with test ("Is 1?") and set ("1↓") actions, described in section 5.

is based on a sophisticated compile time analysis of the input set of regular expression: the compiler implements several automata transformations, that are outlined in Figure 1.

Each regular expression is first simplified in a normal form, rewriting and eventually splitting it into sub-expressions, and then transformed into a Glushkov NFA; we selected the Glushkov automaton because it has a number of interesting properties that we can use to obtain a specific set of structures in the graph that represents the automaton. The Glushkov NFA is then turned into a DFA which fulfills a specific set of properties which again determine the shape of its underlying graph. These automata are then combined together by an algorithm which operates on their topology; the combining algorithm is possible only because the graph structure obtained by the previous steps contains only specific shapes. The resulting graph is then extended, as in the Aho-Corasick algorithm, by a "failure" function, whose computation can further modify the graph structure. The result is a DFA that:

- groups every regular expression in a single automaton,
- reports exhaustive and complete matches, including every overlapping pattern,
- in most practical cases is as memory efficient as an NFA and
- operates on single pass, since just one pass in the automaton is able to identify all matches.

Exhaustive and complete matches condition can be relaxed to simplify the automaton and generate less states. Our runtime system leverages the memory hierarchy and multiple cores using caching, partitioning, data replication and mapping techniques, and uses bit level parallelism and vector instructions to operate on the automata data structures.

The experimental results of Section 6 show that the performance of DotStar is comparable with hardwired FPGA implementations [3] since most of the algorithmic complexity is addressed at compile

	Single automaton	Overlapping matches	Compress # of states	Handle memory h.	Use bit parallelism	Use vector instr.	Unlimited operators	No backtrack	Insensitive to DoS	No cond. on transition	No cond. on states
Bit parallel Thompson	✗	✓	✗	✗	✓	✗	✓	✗	✗	✓	✓
Bit parallel Glushkov	✗	✓	✗	✗	✓	✗	✓	✗	✓	✓	✓
Classical NFA to DFA	✗	✓	✗	✗	✓	✗	✓	✓	✓	✓	✓
Filtration (gnu grep)	✗	✗	✓	✗	✓	✗	✗	✗	✓	✓	✓
Flex	✓	✗	✗	✗	✓	✗	✗	✓	✓	✓	✓
Bifurcated matching [4]	✓	✓	✗	✓	✗	✗	✓	✓	✗	✓	✗
H-FA and Hc-FA [4]	✗	✓	✓	✗	✗	✗	✓	✓	✓	✗	✓
Partitioned [17]	✗	✗	✓	✗	✗	✗	✗	✓	✓	✓	✓
Delayed input DFA [5]	✗	✗	✓	✗	✗	✗	✗	✓	✓	✓	✓
DotStar	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗

Table 1: Comparison between DotStar and other solutions.

time. Moreover the availability of large memory in commercial off the shelf multi-core systems allows implementation of very large regular expression sets (which are difficult to map on FPGAs due to space constraints) and will ultimately allow a large cost reduction.

A comparison between DotStar and some of the available solutions is shown in Table 1. A single automaton allows a smaller per-flow state and speeds up parsing examining a single transition per input character; a system that reports overlapping matches can be applied in a wider range of application domain; compressing the number of states reduces memory requirements while handling memory hierarchy provides a performance gain on general purpose processors. Bit parallelism and vector operations maps directly on large registers and vector units. Supporting every operator will enhance the expressiveness of the system; avoiding backtrack will reduce running time and being insensitive to Denial of Service (DOS) attacks is an essential feature of many networking and real time applications. Conditions on transitions will blow up the number of transitions per state as well as increasing the required operations per input symbol, while conditions on states will increase only the operation count per input symbol.

3 Keyword Graphs: Building the DotStar DFA

Finding every instance of a regular expression pattern, including overlaps, is a "complex" problem either in space or time. Matching a complete set of regular expressions adds another level of complexity: we need a novel mechanism for combining several regular expressions into a single engine while keeping the complexity of the problem under control.

DotStar uses a modified version of the well known Aho-Corasick algorithm. Aho-Corasick operates on a keyword tree, that is a tree containing a combined representation of all keywords. The keyword tree is transformed into a DFA by using a failure function $F()$, which is followed when no direct transition is available at the current state. $F()$ target 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 $F()$ edges till either a proper suffix is detected or the root node is reached. A linear time DFA is obtained by pre-computing every $F()$; space complexity is also linear with the sum of keyword lengths.

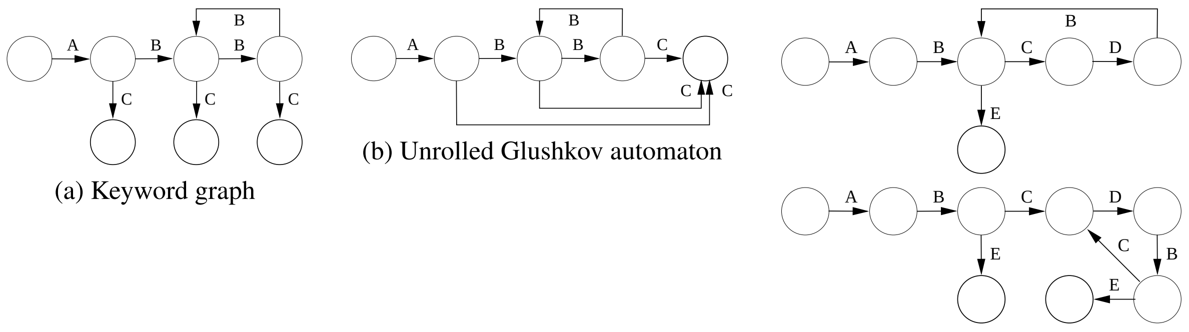
Several slight modification of Aho-Corasick implementation are available in literature, to efficiently encode states and transitions to execute them either on FPGA [12] or custom ASIC [14] or generic multicore architectures [16] [11].

Definition 1 *a* keyword tree *is a tree where every edge is labeled by a symbol; any two edges out of a node have different symbols and any path in the tree defines a unique keyword by concatenating edge labels.*

DotStar extends the keyword tree data structure into a graph with a "tree like" structure: we still have a high level tree that combines all pattern instances but we allow for specific kind of loops to appear inside tree branches, thus transforming it into a graph.

Definition 2 *a* keyword graph *is a directed graph that has an initial (root) node; every edge is labeled by a symbol; any two edges out of 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; every path from the root node to any node marked "final" defines a match of a regular expression pattern instance by concatenating edge labels.*

A sample keyword graph is shown in figure 2a. We start describing the DotStar compilation process by considering only basic regular expressions.



(a) Keyword graph

(b) Unrolled Glushkov automaton

(c) Loop unrolling

Figure 2: Sample graphs and operations

Definition 3 *a* basic regular expression *is a string built from a finite alphabet of symbols, the grouping "()", alternative "|" and closure "$*$" operator ($*$ repeats the previous element zero or more times).*

In order to build a keyword graph for a basic regular expression we use a Glushkov automaton [10] as an intermediate step.

Definition 4 *a* Glushkov automaton *is a NFA described by the tuple (S,Σ,i,F,δ) where:* S *is the set of states ($m + 1$ for an m symbols expression); Σ is the alphabet;* i *is the initial state;* F *is the set of final states; initial state belongs to* F *only if the empty string is in the recognized language; δ is the transition function of the automaton.*

We transform the basic automaton by unrolling edges that start and end at the same node (figure 2b).

Definition 5 *an* unrolled Glushkov automaton *is a Glushkov automaton where every transition from a node to itself is unrolled, introducing a new identical node and a number of transitions to recognize the same language.*

We are using Glushkov automata because they have a number of interesting properties [1]:

- The NFA is ϵ-free, which means that has no empty transition.
- The NFA is homogeneous: all transitions leading to a given state y are labeled by the same symbol.
- The graph representing the automaton is a hammock. This means that has 2 special nodes: the initial node n_i and the final node n_f such as no edge enters n_i, no edge exits n_f and every other node of the graph is in a path from n_i to n_f.
- Every maximal orbit in the graph representing the automaton is strongly stable. This means that there exist an edge between any orbit exit node (vertex that have outgoing edges not included in the orbit) and any orbit enter node (vertex that have incoming edges not included in the orbit).
- Every maximal orbit in the graph representing the automaton is strongly transverse, which means that if a node outside the orbit has an edge towards one orbit enter node it must have edges to every other orbit enter nodes.

To construct a keyword graph from a simple regular expression we start building the Glushkov automaton, we transform it in an unrolled Glushkov automaton, we then transform it into a DFA using any NFA to DFA method and we breadth first visit the DFA turning it into a keyword graph. The output is, by construction, a keyword graph which represents every pattern instance of a single basic regular expression.

The next step is to combine keyword graphs in a single DFA able to recognize every expression at once. Several actions in the combining process need to transform a loop in a graph by unrolling it to disambiguate between two partially overlapping patterns containing closures. The unroll procedure is always applied at the topmost node (the one nearest to the root node) of a loop in the graph and will create a copy of that node, along with every connected subtrees, and will move the loop backward edge; the process is shown in figure 2c.

The DotStar combining algorithm starts from an empty keyword graph, which contains only the root node, and adds one keyword graph at a time; the algorithm is designed to maintain the basic keyword graph properties, eventually replicating nodes and subtrees. To add a keyword graph to an existing one we perform a simultaneous breadth first visit of both graphs, maintaining a mapping between a node in the combined graph and a node in the combining; the procedure starts mapping the two root nodes. For every node in the combining graph, the algorithm considers the outgoing edges and tries to match them with an existing edge in the combined graph. Several cases are possible: the edge may be a new one or may already exist and it may or may not enter a loop. The combining algorithm handles every combination transforming either graph and progressing the visit; loops will be unrolled when partially overlapping prefixes are detected.

The final step for building a DFA is the computation of the correct $F()$. The purpose of the $F()$ function is to identify the longest proper suffix of another keyword (pattern in our case) already recognized while matching the current one. The basic Aho-Corasick algorithm to compute it performs a breadth first visit of the graph and for each node it computes the $F()$ of child nodes using the $F()$ of the parent node. 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 $F()$ function defined.

This algorithm must be modified for handling loops in the keyword graph: when the $F()$ computation reaches the backward edge of a loop in the graph the target node will already have an $F()$ defined. In this case we must compute a new $F()$ function using the backward edge source node and compare it with the $F()$ already present. If the length of recognized pattern instance for the new one is strictly longer than the old one then the loop must be unrolled and processing continued down the path.

An example where this happens is when computing the $F()$ function of the combination of the two expressions

```
A(BC)*D          BCBCBCE
```

The loop defined by the "(BC)*" closure will not be unrolled during combining (since it is after the "A" symbol) but it will be unrolled twice while computing $F()$ to detect the overlapping pattern. The third time the suffix length of the new $F()$ computed on the backward edge will be equal to the length of the one already present since there are no longer patterns. The unrolling procedure will always terminate because two (or more) loops cannot force each other to unroll, since they have a different suffix: if they had the same suffix they would have been combined by the previous compiler step.

The resulting DFA (composed by a keyword graph extended with a failure function) can be executed on a standard Aho-Corasick implementation and is able to report every matching instance of any pattern of a basic regular expression set, including overlapping patterns, using in a single pass. We now describe how DotStar handles character classes and the wildcard when not used inside a closure operator.

Definition 6 *an* extended regular expression *is a string built from a finite alphabet of symbols; character classes over the alphabet of symbols (e.g. [a-f0-9] which can assume one value among the set of choices); the wildcard (which represents any symbol); the grouping "()", alternative "|" and closure "$*$" operators. In extended regular expressions, character classes and wildcards are prohibited inside the closure operator.*

It is straightforward to handle extended regular expressions since we can consider a character class and a wildcard as a primitive symbols during most compilation steps. Care must be taken when turning the NFA into a DFA, when combining keyword graphs and while computing the failure function: partial overlaps may occur while examining edges that are labeled with character class or wildcard and a number of intersection tests must be handled. During combining these tests may further modify the combined graph and/or the combining graph, splitting edges with a character class labels into non intersecting subsets, in order to perform suffix disambiguation and properly combine keyword graphs. Failure function computation must be modified to consider character classes again to obtain a proper edge disambiguation. The rationale is that, while visiting the graph, we can find a node which is reached using a character set and that has different $F()$ targets depending on the input symbol. In this case we must duplicate the node, splitting the character class edge into non intersecting subsets, to be able to specify different $F()$.

The operations introduced to handle character classes in expressions can create a number of similar subtrees when replicating portions of expressions; consider for example the two expressions:

```
PRE[a-d]POST        PRE[c-f]SECOND
```

The combining algorithm logically transforms them into this set of expressions while computing the overall keyword graph:

```
PRE[a-b]POST        PRE[c-d]POST
PRE[c-d]SECOND      PRE[e-f]SECOND
```

The “POST” and “SECOND” subtrees are replicated by the procedure, creating identical section of the keyword graph where the last node is a final node that recognizes the same expression. We easily can detect these structures and collapse them into a single copy starting by the leaf nodes of the graph and working backwards.

4 Complex Regular Expressions and State Explosion

In this section we show how DotStar can handle a larger class of regular expressions: we determine which kind of expressions will lead to state explosion and we will show how to alleviate the problem. The first step is including support for a number of commonly used operators by specifying a set of rewrite rules that transform the regular expression into one or more expressions that use only the basic set of operators.

Rewrite Rule 1 Regular expression with positive closure operator *Implementing the “+” operator is straightforward since it can be rewritten using the “*” operator: “AB+C” becomes “ABB*C”*

Rewrite Rule 2 regular expression with operator “?” *In order to implement the optional operator we rely on the fact that we can handle DFA with millions of states using techniques described in [10] and that we combine regular expressions into a single keyword graph automatically reusing similar subtrees. We can then transform the expression containing one or more optional operator into a set of regular expressions by combinatorial expansion of the “?” operators and then pruning all redundant expressions.*

Note that in real world sets the “?” operator is often used in place of the bounded repetition operator (for example “AB?B?B?B?B?C”), which allows smarter rewrite rules.

Rewrite Rule 3 regular expression with bounded and unbounded repetitions *The repetition operator “$\{n;m\}$” specifies that a sub-expression must be present at least “n” times and at most “m” times. Both numbers can be omitted, with “n” defaulting to 0 and “m” default meaning unbounded. To implement this operator again we expand the regular expression into a set by replacing all bounded repetitions and eventually inserting a closure operator at the end.*

```
AB{3;5}C    ->   ABBBC  ABBBBC  ABBBBBC
AB{3;}C     ->   ABBBB*C
```

When we combine the resulting keyword graphs most nodes will be re-used. The outlined rules are only examples: we designed an extensive set of rewrite rules that speed up handling of real world regular expression sets by recognizing common patterns in operator usage and applying proper combined transformations.

In the outlined approach, state explosion happens when the regular expression set uses character classes or wildcard inside a closure, bounded or unbounded repetition operators. The problem is that when we are computing the $F()$ function we are looking for the longest prefix already matched; our algorithm "unrolls" loops in the graph when a prefix overlaps with another one that contains a closure or repetition operator to be able to properly detect the longest prefix. When the loop contains a symbol this unroll operation happens a finite number of times, but if it contains a character class then a very large (or infinite number) of unrolls may happen. For example the regular expression component ".*" will appear in the graph as a loop with the wildcard as label. Any other regular expression pattern instance is a longest proper prefix we should track while handling the ".*".

To solve this problem we designed a set of changes to Aho-Corasick runtime using specific "add-on data" to compress sections of the automaton. Our toolchain annotates the DFA states with actions and tests to be performed when the state is entered. The "add-on data" are tailored at supporting specific regular expression constructs that lead to state explosion. All the changes are modular and our toolchain is able to use a minimal set of add-on items depending on the specifics of a regular expression set.

Definition 7 *a* complex regular expression *is a string built from a finite alphabet of symbols; character classes over the alphabet of symbols (e.g. [a-f0-9] which can assume one value among the set of choices); the wildcard (which represents any symbol); the grouping "()", alternative "|", closure "$*$", positive closure "$+$", optional "?" and repetition "$n;m$" operators. Examples are:*

```
A[a-z]*B        START.*MIDDLE.*END
```

DotStar uses the following add-on data items to compress the automaton:

Status bit Is a boolean value associated with a closure operator over a wildcard; the value is set when the prefix is recognized and is tested when the postfix is detected to validate the complete match.

Masked status bit Is a boolean value associated with a closure operator over a character set; the value is set when the prefix is recognized and is tested when the postfix is detected to validate the complete match. The bit may be cleared while examining input data if the symbol does not belong to the character set.

Location memory Is a storage item containing a position in the input stream and is associated with a bounded or unbounded range operator over a wildcard; the value is pushed when the prefix is detected and is compared when the postfix is recognized.

Masked location memory Is a storage item containing a position in the input stream and associated with a bounded or unbounded range operator over a character set; the value is pushed when the prefix is detected and is compared when the postfix is recognized. The valued may be cleared while examining input data if the symbol does not belong to the character set.

Tail counter Is a counter initialized at an instant during the matching operation and is decremented after each input character; when the counter reaches 0 a regular expression pattern instance is detected.

Masked tail counter Is a counter initialized at an instant during the matching operation and is decremented after each input character; when the counter reaches 0 a regular expression pattern instance is detected. The counter may be reset while examining input data if the symbol does not belong to the character set.

DotStar contains a pre-processor step that examines the set of regular expressions, recognizes the problematic parts and expands them into multiple simpler expressions. The tool allocates add-on data items and annotates expressions with set and test operations. The pre-processor starts by expanding the expression into top level alternatives, reducing to simpler disjoint expressions. It then examines each regular expression for the presence of "bad" components, that is closure or range operators over character classes. If it finds any bad component it splits the regular expression into a sequence of subsets; even list elements are always "good" sub-expressions, while odd elements contain "bad" sub-expressions. The pre-processor then allocates a status bit or a location memory for each bad component and annotates their prefix and postfix components, adding them to the output regular expression set. Tail counters are used if the last element of the regular expression is a "bad" one, such as a bounded repetition that would need a long sequence of states in the automaton. We developed a number of transformation heuristics able to cope with any sequence of sub-expressions, for example by transforming consecutive "bad" parts into equivalent expressions with a single "bad" part.

Status bits, location memory and tail counters may be shared among non overlapping expressions and our pre-processor contains a number of heuristics to minimize the number of add-on data items generated in the process. Expressions which contain both 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.

An example transformation is the following: we start from the expression

```
User:.*Password|User:[a-z]{0,25}Login
```

After the first step we obtain the following set:

```
User:.*Password        User[a-z]{0,25}Login
```

After the second step and uniquifying the expressions:

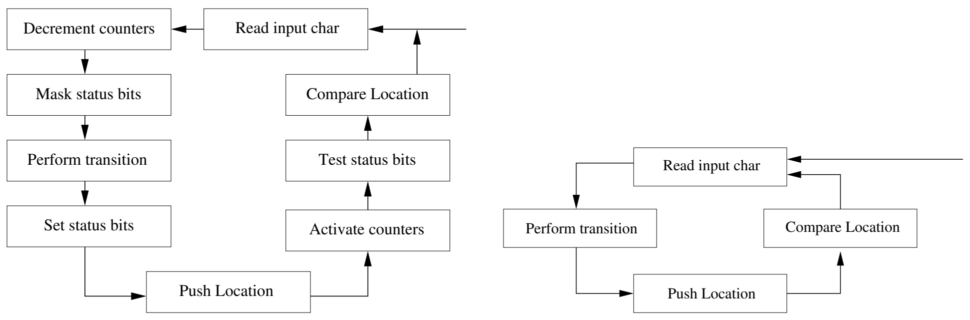
```
User:        Password        Login
```

The pre-processor allocates one status bit (b0) and one location memory (l0); the expressions are annotated as follows: after detecting "User" set b0 and push l0; after detecting "Password" test b0; after detecting "Login" check if l0 is between 5 and 30. The final output of the compiling process is a generic keyword graph.

Definition 8 *a* generic keyword graph *is a keyword graph whose states can be optionally "annotated" with a set of status bit to mark, a set of locations to push, a set of tail counters to activate, a set of conditional matches, a set of conditional status bit set, a set of conditional locations to push, a set of conditional tail counters. The graph is also augmented with a mask per character of the input alphabet that specifies when a status bit or a location or a tail counter must be cleared.*

5 Runtime for General Purpose Multicore Processors with Vector Operators

The compilation process builds a generic keyword graph which is essentially a standard Aho-Corasick DFA modularly augmented with new per state items operations and per input symbol masking operations. The high level runtime algorithm is shown in figure 3a. We developed a modular implementation with specific optimizations for general purpose multicore with vector extensions and the output of the compiler dictates the minimum amount of runtime characteristics needed.



(a) Full runtime (b) Protocol parser runtime

Figure 3: The modular runtime loop.

We borrow the basic automaton implementation (namely the handling of transition tables with a CAM, cache and memory pressure handling) from the work described in [11].We then augment the per automaton state information with zero or more of the following data:

- A state bit vector. We need one bit for every different status bit used by the compiler and we pack all bits in one or more words of suitable size, such as 64 bit words or 128 bit SSE words.
- A location memory vector. We need one 16 bit index for every different location memory used by the compiler and we pack them in a vector representation using SSE data types.
- A tail counter vector, with one entry for each counter used by the compiler. We pack them in a vector representation using SSE data types.

The actual add-on data requirement depend on the mix of regular expressions to handle. Our runtime prepare a set of masks for fast zero-ing status bits, locations or counters after every symbol. When the alphabet is small (e.g. 8 bit) we use a vector to store the masks, if it is larger (e.g. UTF16) we use a CAM to compress its size.

We pack the DFA per state actions and tests in separate data structures, each containing one entry per DFA state; we store 3 kind of data: conditional match reporting (which contains a test mask and if the test succeeds a match is reported), unconditional set (which contain new values to be set or pushed) and conditional set (which contain both a condition to evaluate and new values to set or push).

On real world regular expression sets, the number of DFA states that require add-on actions is small compared to the overall number of states (for example is less than 7% on Linux NDIS) and the number of executions at runtime is very small (less than 1% of transitions on Linux NDIS with heavy hitting test data). To leverage this we exploit unused transition target address bits, storing information that specify which add-on action(s) are needed for the state. All these data structures are optional: the mix of regular expressions in the set dictates whether a specific add on table is required; for example the runtime loop

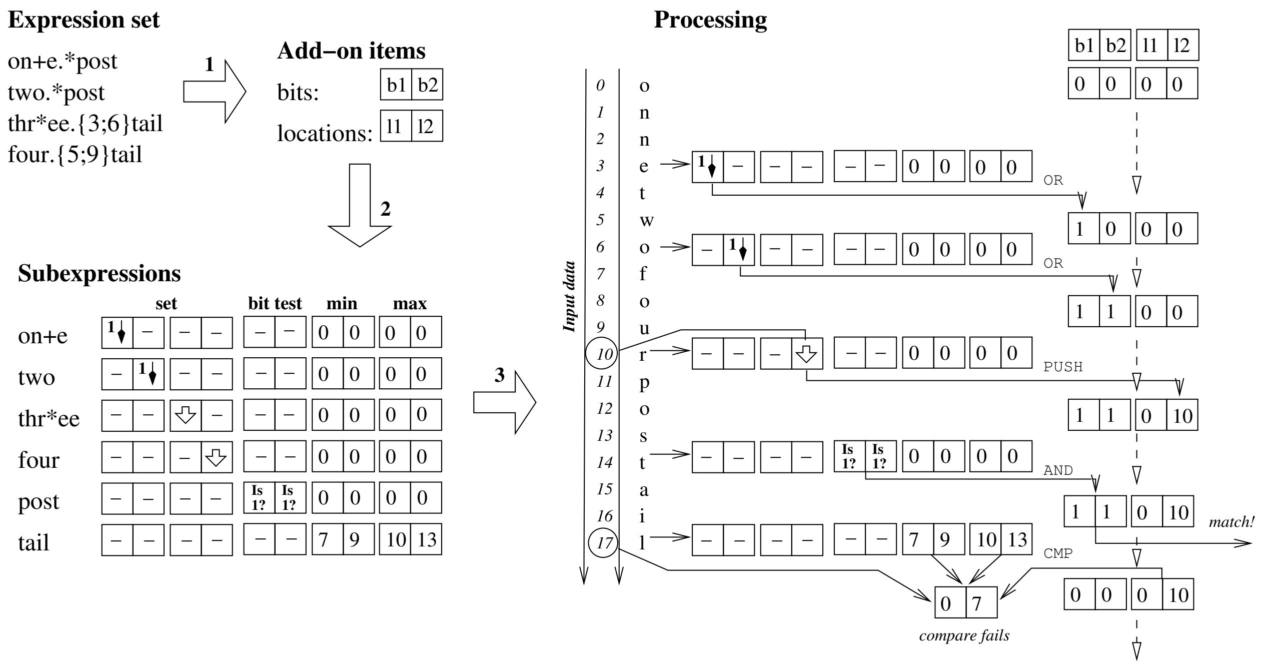


Figure 4: Exploiting bit level parallelism and vector data types. Step 1 shows the pre-processor allocating 2 bits and 2 location. Step 2 describes how sub-expressions are marked for set and test actions. Step 3 shows an evolution of status bit and memory location while analyzing input data.

for a very simple set of regular expressions (enough to parse SMTP or HTML syntax) is shown in picture 3b.

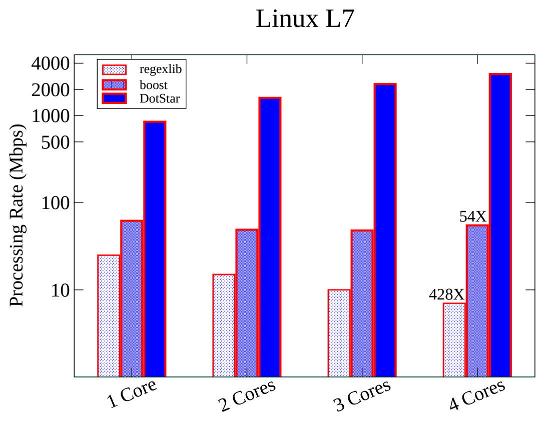
6 Experimental Results

The DotStar tool-chain is composed by:

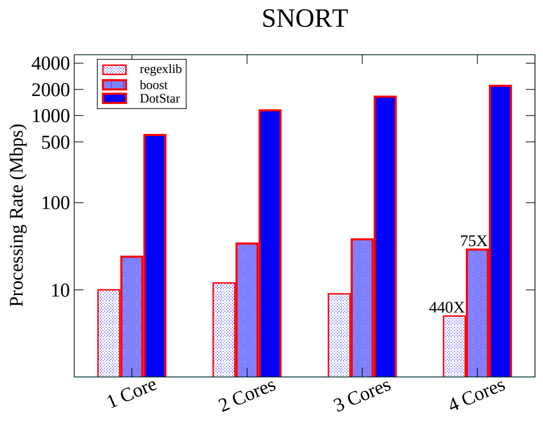
- A preprocessor, which reads the regular expression set and applies the transformations. The output is a new set of regular expressions annotated with all the required add-on components.
- A compiler, that implements the keyword graph combining algorithm. The compiler outputs a description of the DFA and a list of add-on data items.
- A DFA compactor, which reads DFA description and builds a compact binary representation.
- A runtime engine, which reads the DFA binary representation, selects a specific runtime code depending on the required add-on items and streams input data through it.

The set of tools supports most of the PCRE standard syntax for regular expressions. We tested the framework applying it to different application domains: SMTP parsing, XML protocol parsing and network intrusion detection.

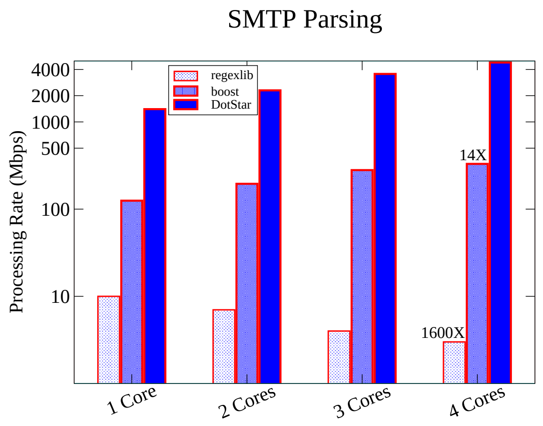
SMTP parsing uses a very small set of regular expression to capture sub-pattern and extract all components of the SMTP protocol; this set of regular expressions is able to parse the full protocol, also validating addresses and domain names. Test input data come from the SMTP dumps from Berkeley



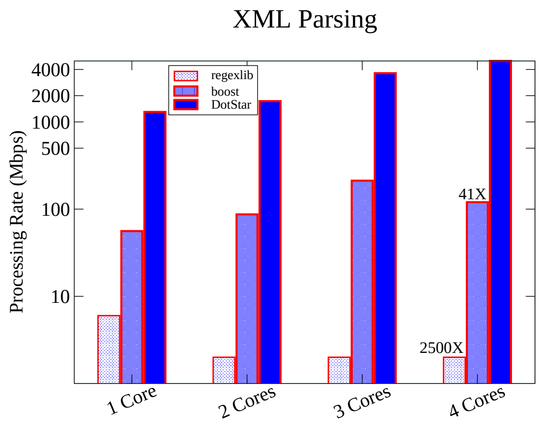
(a) Linux L7 filtering performance



(b) SNORT filtering performance



(c) SMTP parsing performance



(d) XML parsing performance

Figure 5: Performance comparison

NIDS test data. XML parsing can be expressed as a set of regular expressions able to recognize and classify the various components in an XML document. The set of regular expressions we designed comprises few tens of expressions that is able to tokenize all XML version 1.1 grammar; test data come from standard XML compliance test suite. Intrusion detection requires analyzing input data stream and matching them against a large set of complex patterns working on 8 bit binary data. We used Linux L7 traffic classification patterns and the SNORT intrusion detection patterns; input data are the Berkeley test traffic dump

All the tests were run on a state-of-the-art quad core system (2 x dual core Intel Xeon CPU, clocked at 2.6 GHz). Our runtime engine reports which expression matches and at which point in the traffic flow; independent flows can be matched in parallel by a multi threaded version of the runtime engine.

We compared performance with other existing solutions: a state-of-the-art NFA based regular expression engine, used in Linux L7 and Snort and the boost regular expression library. In Figure 5d and 5c we can see the performance comparison for protocol parser like applications. In this case DotStar produces a small automata that uses a streamlined runtime algorithm which is able to achieve very high performance and almost linear scalability. In contrast the NFA library performance degrades when parallelizing it on multiple cores; boost regular expression library is able to obtain a slightly better performance and some benefits from multiple cores.

	Snort	Linux L7	XML parser	SMTP parser
# of regular expressions	500	150	15	20
% patterns with closures	75%	75%	100%	100%
% patterns with bounded repetition	50%	25%	0%	0%
% patterns with character classes	30%	50%	100%	100%
# of DotStar states	550k	400k	1500	1000
# of DotStar status bits	113	53	0	0
# of DotStar location memory	20	16	1	1
# of DotStar counters	14	3	0	0

Table 2: Test set characteristics and compiler results.

Figure 5a and 5b show performance comparison for typical NDIS application using Linux L7 patterns and snort patterns. The input data is searched for every pattern (in a real world application specific connection ports will be examined for a subset of patterns) to examine the performance over a large regular expression set. Again our DFA obtains orders of magnitude better performance and almost linear scalability even when the automaton contains hundred thousands states.

The overall performance, when using all 4 cores, ranges from 2.2 Gbits/sec, with SNORT to 5 Gbit-s/sec with the XML parsing. The numbers on the two leftmost bars in the 4-core configuration indicate the perfomance speed-up of StarDot when compared to the other two algorithms: DotStar is between 14 and 75 times faster than Boost, with the performance gap that increases when we have more complex sets or regular expressions. Regexlib suffer from scalability problems –we need to replicate all his data structures in its Pthreads parallelization, leading to some undesired cache conflicts: DotStar is at least three order of magnitude faster in all four configurations.

7 Conclusion

In this paper we described DotStar, a novel approach to iteratively build a single combined DFA for a set of regular expressions. This automaton has a structure similar to the Aho-Corasick DFA and can detect in a single pass exact and exhaustive matching of every regular expression pattern instance, including overlapping matches. The objective of our approach is leveraging modern general purpose multicore architecture, which can address large memories and obtain very high performance when correctly tuned. These systems usually contain vector units to speed-up specific kind of computations, and we want to use these units to accelerate the DFA handling.

To build this automaton we implemented sophisticated compile time analysis of the regular expression set and a large number of automatic transformations to rewrite the set into a common form. We use Glushkov non deterministic automaton as an intermediate compiler step in order to obtain a DFA whose graph representation has specific topology properties. We designed a mechanism to turn Glushkov NFA into a DFA, building on ϵ-free, hammock and orbit strong stability properties to obtain a graphs that contains only non intersecting loops. We then combine all these automata, each relative to a single regular expression, using a graph based algorithm that operates on the topology.

Th experimental evaluation of the approach shows that it is applicable to both small regular expression sets, such as those used for protocol parsing, and large regular expression sets, such as those used in network intrusion detection applications. In both cases we were able to reach very high matching speed, between 2.2 and 5 Gbits/sec, which is orders of magnitude better than other state-of-the-art NFA based systems, and achieved linear scalability on a commodity multicore architecture.

Since most of the complexity is addressed and removed at compile time, the performance of our approach is comparable with hardwired FPGA implementations with the added benefits of being able to support a much larger regular expression set and does not to require specialized hardware or accelerator.

References

[1] P. Caron and D. Ziadi. Characterization of Glushkov Automata. *Theoretical Compututer Science*, 233(1-2):75–90, 2000.

[2] Cisco. Cisco IOS IPS Deployment Guide.

[3] M. Faezipour and M. Nourani. Constraint Repetition Inspection for Regular Expression on FPGA. In *16th Annual IEEE Symposium on High-Performance Interconnect (Hot Interconnects 16)*, Stanford University, Palo, Alto, CA, August 2008.

[4] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, pages 155–164, New York, NY, USA, 2007. ACM.

[5] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. *SIGCOMM Comput. Commun. Rev.*, 36(4):339–350, 2006.

[6] Lawrence Berkeley National Laboratory. Bro Intrusion Detection System.

[7] J. Levandoski, E. Sommer, and M. Strait. Application Layer Packet Classifier for Linux.

[8] W. Lu, K. Chiu, and Y. Pan. A parallel approach to xml parsing. *The 7th IEEE/ACM International Conference on Grid Computing (Grid2006)*, Barcelona, Spain, September 28-29, 2006.

[9] W. Martens, F. Neven, and T. Schwentick. Complexity of Decision Problems for Simple Regular Expressions. In *MFCS*, pages 889–900, 2004.

[10] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, New York, NY, USA, 2002.

[11] D. P. Scarpazza, O. Villa, and F. Petrini. Peak-performance DFA-based string matching on the Cell processor. In *In Proc. SMTPS '07*, 2007.

[12] R. Sidhu and V. K. Prasanna. Fast Regular Expression Matching Using FPGAs. In *FCCM '01: Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 227–238, Washington, DC, USA, 2001. IEEE Computer Society.

[13] Sourcefire Inc. SNORT Network Intrusion Detection System.

[14] L. Tan and T. Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 112–122, Washington, DC, USA, 2005. IEEE Computer Society.

[15] TippingPoint. TippingPoint X505.

[16] O. Villa, D. Scarpazza, and F. Petrini. Accelerating Real-Time String Searching with Multicore Processors. *IEEE Computer*, 41(4):42–50, April 2008.

[17] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and Memory-efficient Regular Expression Matching for Deep Packet Inspection. In *ANCS '06: Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, pages 93–102, New York, NY, USA, 2006. ACM.

[18] W. Zhang and R. van Engelen. A table-driven streaming xml parsing methodology for high-performance web services. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*, pages 197–204, Washington, DC, USA, 2006. IEEE Computer Society.