# Picking Pesky Parameters: Optimizing Regular Expression Matching in Practice

Xinming Chen, Brandon Jones, Michela Becchi, and Tilman Wolf

**Abstract**—Network security systems inspect packet payloads for signatures of attacks. These systems use regular expression matching at their core. Many techniques for implementing regular expression matching at line rate have been proposed. Solutions differ in the type of automaton used (i.e., deterministic versus non-deterministic) and in the configuration of implementation-specific parameters. While each solution has been shown to perform well on specific rule sets and traffic patterns, there has been no systematic comparison across a large set of solutions, rule sets and traffic patterns. Thus, it is extremely challenging for a practitioner to make an informed decision within the plethora of existing algorithmic and architectural proposals. Moreover, as multi-core processors are becoming popular, many parameters need to be tuned to maximize the multi-core potential. To address this problem, we present a comprehensive evaluation of a broad set of regular expression matching techniques. We consider both algorithmic and architectural aspects. Specifically, we explore the performance, area requirements, and power consumption of implementations targeting multi-core processors and FPGAs using rule sets of practical size and complexity. We present detailed performance results and specific guidelines for determining optimal configurations based on a simple evaluation of the rule set. These guidelines can help significantlywhen implementing regular expression matching systems in practice.

**Index Terms**—Network security, deep packet inspection, deterministic finite automaton, non-deterministic finite automaton, regular expressions, design space exploration

## 1 Introduction

Most attacks on computer systems are currently performed remotely through the Internet. Since existing network protocols do not provide inherent protection from such attacks, intrusion detection systems (IDS) are used throughout the Internet to identify and block malicious network traffic. These IDS scan the payloads of all forwarded traffic to search for patterns that correspond to known attacks. To express these patterns, regular expressions are used and public databases, such as Snort [1] and Bro [2], maintain rule sets of regular expressions that match attacks.

The implementation of IDS presents a set of interesting technical challenges. IDS need to operate at multiple to tens of Gigabits per second link rates to meet the performance requirements of the network. At the same time, inspecting potentially every byte of packet payload requires much more time and energy than simple packet forwarding, which reads packet headers only. To address these system challenges, numerous regular expression matching techniques have been developed.

Regular expression matching is typically implemented in two steps: first, the rule set is transformed into a state machine or automaton; second, packet payloads are scanned by traversing the state machine. Malicious traffic is identified when the state machine reaches an accepting state (indicating that the regular expression of a known attack has been matched). This automaton generated from the rule set can be deterministic or non-deterministic, and can be broken up into multiple components, depending on the specific regular expression matching technique. In addition, there are many different system implementations for regular expression matching using different types of processors, logic circuits, and memory configurations.

Because there are so many regular expression matching techniques, it is difficult to decide *which technique actually works best* in a given system setting. The body of published work in this domain unfortunately does not help identify which techniques are most suitable for use in practice. The performance metrics used in publications differ; some seek to reduce memory requirements [3], [4], [5], some aim to improve the average and worst case throughput [6], [7], and some aim to reduce power and energy consumption [8]. It is very difficult to determine which technique or system implementation to use. In addition, the optimal choice depends on the characteristics of the underlying rule set and on the traffic pattern.

Our work addresses this problem of choosing which regular expression technique to use for a given system, rule set, and traffic configuration. We present a systematic evaluation of many widely used regular expression techniques using real-world rule sets. Our design space exploration encompasses the automaton domain (e.g., NFA, DFA, multi-stride FA), the implementation domain (e.g., memory layouts, ruleset partition algorithms), the system domain (e.g., cache size, memory bandwidth, processor versus FPGA). We use real hardware as well as simulations to evaluate the throughput, memory size, energy consumption,

• *X. Chen and T. Wolf are with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003. E-mail: {xinmingchen, wolf}@ecs.umass.edu.*
• *B. Jones and M. Becchi are with the Department of Electrical and Computer Engineering, University of Missouri, Columbia, MO. E-mail: bjjzqd@mail.missouri.edu, becchim@missouri.edu.*

*Manuscript received 14 Jan. 2015; revised 24 Apr. 2015; accepted 4 June 2015. Date of publication 7 July 2015; date of current version 13 Apr. 2016. Recommended for acceptance by Z. Tari.*
*For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.*
*Digital Object Identifier no. 10.1109/TPDS.2015.2453986*

and estimated chip area of each configuration. Based on the results from our experiments, we provide a method for choosing the right configuration.

The specific contributions of our paper are:

- Definition of the regular expression matching design space. We present a systematic characterization of design space parameters for regular expression matching, which include automaton, implementation, and system aspects.
- Benchmarking of configurations that evaluate the design space both on simulator and on real hardware. We present results for the regular expression matching design space that consider performance (i.e., throughput) and cost (i.e., memory size, chip area, energy consumption), as well as their tradeoffs.
- Analysis of ruleset to obtain optimal configuration. We present a method based on analysis and simple simulations for quickly identifying which particular regular expression matching technique is optimal in a given scenario.

We believe that our contribution of providing a process for determining the most suitable regular expression matching implementation for a given scenario can be useful for regular expression matching systems in practice.

The remainder of the paper is organized as follows. In Section 2, we provide background on regular expression matching and related work. In Section 3, we describe the methodology of evaluation and the parameter range we use. We present experimental results in Section 4 and a method for finding the best configuration of regular expression matching systems in Section 5. Section 6 summarizes and concludes this paper.

## 2 Background and Related Work

The main functionality of a deep packet inspection system is to determine if the network traffic matches any of the patterns in a rule set. Prior work has proposed a variety of algorithmic solutions and system implementations to perform this task. We briefly review some of these techniques before presenting the design space that we evaluate in our work in Section 3.

### 2.1 Matching Algorithms

Regular expression matching can be implemented by representing the rule set using finite automata (FA): the matching operation is then equivalent to an FA traversal guided by the input stream (i.e., the packet's payload). From an algorithmic perspective, finite automata can be classified into two types: non-deterministic and deterministic finite automata (NFA and DFA, respectively). An NFA accepting a given set of regular expressions can be constructed using the well-known Thompson's algorithm [9], [10] and optimized with several state- and transition-reduction techniques [11]. The main advantage of NFA is their limited size (i.e., number of states), which grows linearly with the number of characters in the rule set. The main disadvantage of NFA is that they allow multiple state activations for every input character processed, thus leading to unpredictable performance and potentially high memory bandwidth requirements. This fact makes NFA vulnerable to denial-of-service (DoS) attacks. A worst-case bound on the processing time can be achieved using DFA, which allow a single state traversal per input character and can be constructed from NFA through subset construction [12], and reduced through the minimization algorithm described in [3]. Since each DFA state corresponds to a set of concurrently active NFA states, the number of states in a DFA can be exponentially larger than that in the equivalent NFA. It has been shown [5], [7], [13], [14] that this phenomenon, called *state explosion*, occurs only in the presence of repetitions of wildcards and large character sets within the regular expressions.

Limiting hardware resource requirements is the main design issue when using DFAs to perform regular expression matching. In particular, two aspects have been studied in the literature. First, *compression mechanisms* aimed at minimizing the DFA memory footprint have been designed. Such schemes do not aim at avoiding state explosion. Instead, their goal is to allow an effective representation of feasible DFAs, that is, DFAs that—having a limited number of states—can be generated on reasonable hardware. These DFA compression schemes typically leverage the transition redundancy characterizing practical DFAs. Second, novel automata to be used as an alternative to DFAs in case of state explosion have been proposed. Alphabet reduction[6], [15], [16], run-length encoding [15], default transition compression [4], [6], state merging [3] and delta-FAs [17] are mechanisms falling into the first category; multiple-DFAs [5], [15], hybrid-FAs [7], history-based-FAs [13], and XFAs [14] fall into the second one.

Multi-stride FA (or $k$-NFA/$k$-DFA) [11], [15] is a mechanism to improve performance at the price of increased resource requirements. Such FAs process $k$ input characters at a time. If the initial alphabet is $\Sigma$, a $k$-NFA/$k$-DFA is equivalent to a FA defined on alphabet $\Sigma^k$. Multi-stride FA achieves linear speedup at the cost of exponentially growing resource requirements. In case of memory-centric implementations and large DFAs, any $k$ beyond 2 is not recommended in practice. Even for $k = 2$, alphabet reduction and transition compression should be used to restrict the memory size to an acceptable level [11]. Figs. 1a and 1b show the NFA and 2-NFA accepting regular expression *.*ab$^+$[cd]e*.

### 2.2 Matching Systems

From the implementation perspective, regular expression matching engines can be classified into two categories: memory-based and logic-based solutions. Fig. 2 shows the abstraction of the two categories.

Memory-based solutions include general purpose processors, network processors, ASICs, GPU, and TCAM. In these implementations, the automata are stored in memory and queried by processing units (e.g., processor cores or logic circuits). Each automaton can be shared by multiple processing units, thus utilizing the full potential of multi-core processors, and allowing easy scalability to multiple packet flows. Memory-based solutions often use cache to overcome high memory latencies and memory bandwidth limitations.

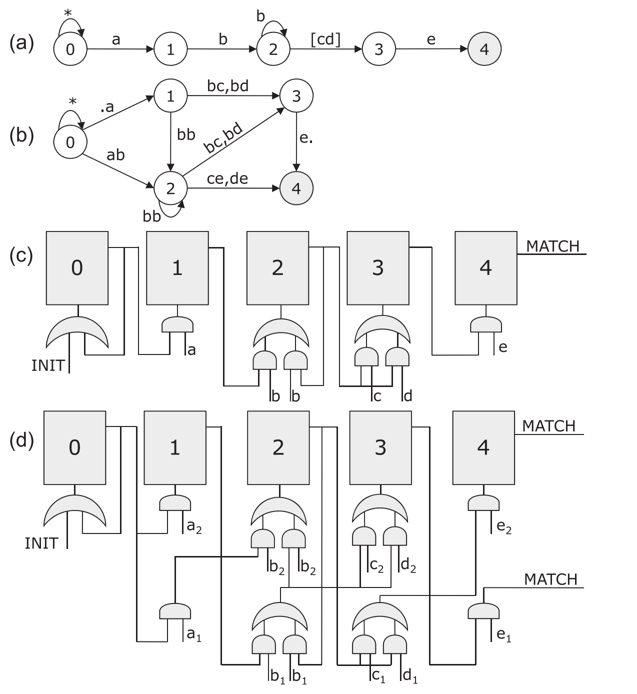


Fig. 1. Example regular expression and automata for *.*ab+[cd]e*: (a) NFA, (b) 2-NFA, (c) stride-1 NFA logic design, and (d) hardware stride-2 NFA logic design.

In memory-based solutions, the selection of a memory layout to store the automaton is an important design decision because it provides a tradeoff between speed and memory size. We consider three memory layouts: non-compressed layout, linear encoding and bitmapped encoding. In the non-compressed layout, each state has as many entries as characters in the alphabet (in the NFA case, unnecessary entries are introduced for missing transitions). In the linear and bitmapped encoding case, DFAs are first compressed using default transitions [4], [6]. In addition to entries associated to existing transitions, bitmap encoding adds to each state a bitmap that allows direct indexing to its transitions. We provide more detail on these memory layouts in Section 3.1.2 (the interested reader can find a complete discussion in the work of Becchi et al. [18]).

In logic-based solutions, the automata are represented as logic circuits (usually implemented on FPGA). Such solutions, typically based on NFA, have the advantage of algorithmic simplicity (they do not require sophisticated state and transition compression schemes). The advantage of logic-based NFA solutions is that they allow a worst-case 1 clock cycle per input character. However, since they require automata replication to handle concurrent packet flows, they are not scalable to multiple flows.

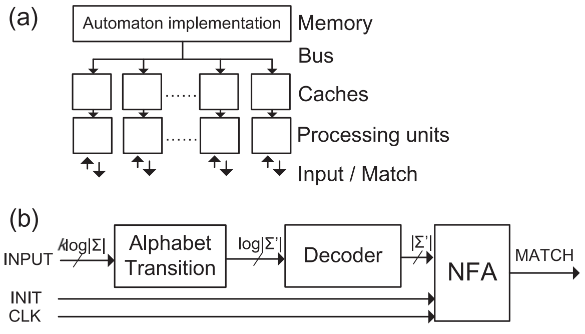


Fig. 2. Abstraction of (a) memory-based parallel solution and (b) logic-based solution.

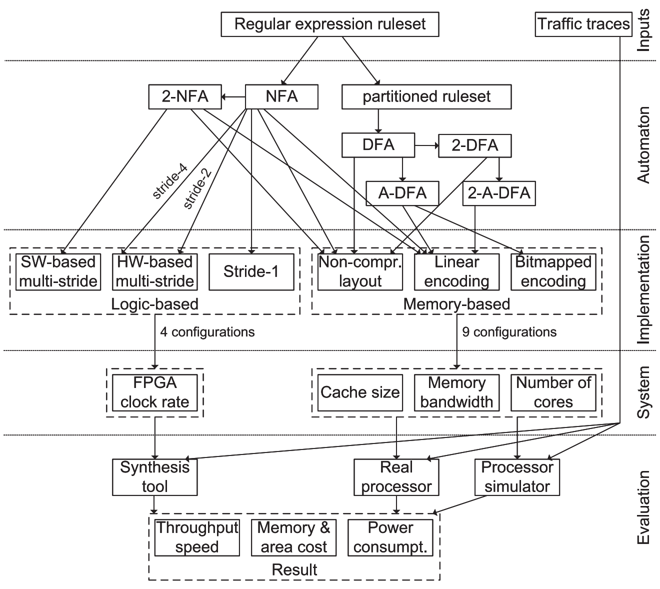


Fig. 3. Design space of regular expression matching system.

Logic-based solutions typically encode NFA using the one-hot encoding scheme [19], which allows processing one input character per clock cycle independently of the number of concurrently active states in the NFA. In the one-hot encoding scheme, exemplified in Fig. 1c, each NFA state is represented by a flip-flop and each symbol by a bit that is set when the input character matches the symbol. State transitions are encoded using combinatorial logic: to reduce the amount of such logic and simplify the design, the number of transitions is minimized using alphabet compression [11]. Fig. 2b shows three basic blocks of the logic-based design: the alphabet translation module, the alphabet decoder and the NFA module. The first block performs alphabet translation. The output of this block must be decoded to produce a one-hot encoding of the processed character, which is input to the NFA block. This operation is performed by the alphabet decoder. Finally, the NFA block encodes the NFA. In our implementation, we include all the optimizations described in the work of Becchi et al. [11] To allow processing multiple characters per clock cycle, we consider two ways of performing stride doubling: using either a software-based or a hardware-based approach. In the software-based approach [11], we first generate a two-NFA, and then we encode it in logic (as if it was a simple NFA). In the hardware-based approach, the stride-one NFA and the corresponding alphabet translation table are deployed in hardware. However, the portion of logic implementing the state transitions is duplicated. Note that the alphabet translation module and the decoder are duplicated as well, since the two input characters need to be separately translated and decoded. Fig. 1d exemplifies hardware-based stride doubling on the NFA block.

## 3 Design Space and Evaluation Methodology

The overall methodology of our evaluation process is shown in Fig. 3. The key components of this process are the various configurations for the regular expression matching automaton, implementation, and system. The cross-product of these configurations make up the design space of regular expression matching as we explain in more detail below. The inputs to our evaluation methodology are various regular expression rule sets and traffic traces. These are processed by different regular expression matching configurations using real processors, processor simulators, and—for logic-based implementations—FPGA synthesis tools. The output of our evaluation are detailed quantitative measures for throughput performance, memory and chip area requirements, as well as power consumption.

## 3.1 Design Space

To explain the design space for regular expression matching in more detail, we discuss each of the three aspects of regular expression matching (automaton, implementation, and system) in more detail and highlight the available configuration choices in each domain. These choices are all represented in Fig. 3.

### *3.1.1 Automaton*

The automaton is the representation of the rule set as a state machine. There are two fundamentally different approaches: NFA and DFA with the tradeoffs discussed in Section 2:

- NFA: The NFA generation is a straightforward translation of the ruleset as described in the work of Becchi et al.[11]. Before exporting to memory layout, the epsilon transitions are usually removed for faster transition.
- DFA: We use subset construction to convert an NFA into a DFA. The procedure described in the work of Hopcroft [3] is used to minimize the number of DFA states. We also use Becchi et al.'s A-DFA [6] to reduce the transitions in DFA states. Such transition reduction is only meaningful with compressed memory layout, so A-DFA is only used with linear and bitmapped encoding.

  When using DFAs, complex ruleset must be partitioned to prevent state explosion. A variety of partitioning algorithms have been proposed, such as Yu's [5], RECCADR [20], Becchi's [18], and RegexGrouper [21]. We use RegexGrouper as the partition algorithm, because it can generate minimum number of states with reasonable performance, and require least renew time among the available algorithms. RegexGrouper takes the desired group number $n$ as input. Therefore, in our experiments, the threshold of the number of states in single DFA $th_{DFA}$ is obtained from the resulting DFAs. The selection of parameter $th_{DFA}$ is studied in Section 3.3.1.

In addition, each automaton can operate in multi-stride mode as discussed in Section 2 and illustrated in Fig. 1. For memory-based multi-stride automata, the largest stride number we use is $k = 2$, because a stride beyond 2 would be slow to generate and require extraordinary amounts of memory. Thus, the overall set of automaton considered in our design space are NFA, DFA, 2-NFA, 2-DFA in memory-based solutions, and NFA, two-NFA, four-NFA in logic-based solutions.

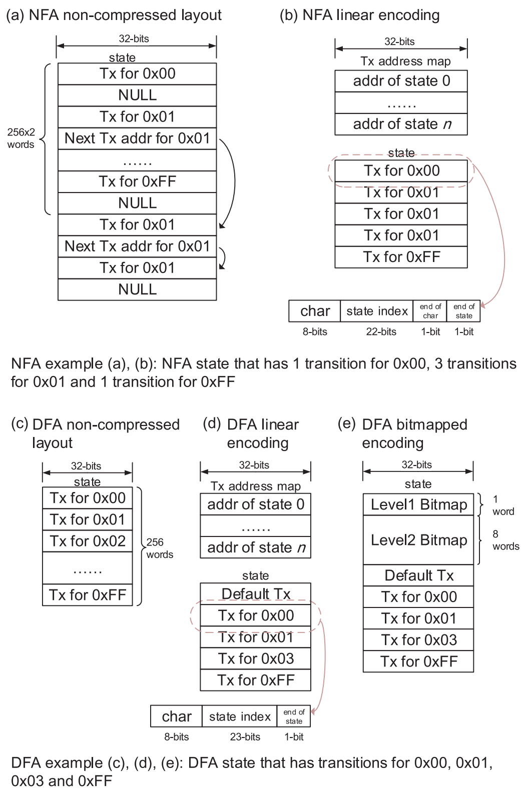


Fig. 4. Memory layouts for NFA and DFA.

### *3.1.2 Implementation*

For memory-based solutions, the data structures used in the automaton generation is usually big and inefficient. To use the automaton for lookup, it is common to encode the automaton in a more compact and efficient memory layout. Three memory layouts are considered in this paper: 1) non-compressed layout, 2) linear encoding and 3) bitmapped encoding (the first two can be used with NFAs and all three can be used with DFAs). The data structure of these memory layouts are illustrated as Fig. 4.

- Non-compressed layout: This memory encoding uses all $|\Sigma|$ transitions in a state. For NFAs, non-compressed layout uses a linked-list-like structure to represent multiple transitions on one character. Characters with no corresponding transitions are represented with a NULL pointer. The goal of non-compressed layout is to have the fastest speed possible—with the trade-off of largest memory space. So the direct address is used as the transition, and there is no default transition for DFAs. The non-compressed layout of DFAs requires only one memory access for a transition and thus has the fastest speed.
- Linear encoding: This encoding reduces the required memory size by only encoding the existing transitions in an NFA, or default transition and other transitions in an A-DFA. Each 32-bit transition entry encodes the input character, transition index, and end of character/state indication. While looking up a linearly encoded state, linear search is performed until a transition matching the input character is found or its absence is verified. Because 22 or 23 bits are not

TABLE 1
Combinations of Design Space

| | Name | Automaton | Implementation | System |
|---|---|---|---|---|
| memory based | NFA NC | NFA | non-compressed layout | varying cache size, memory frequency, number of cores |
| | NFA LE | NFA | linear encoding | |
| | 2-NFA NC | 2-NFA | non-compressed layout | |
| | 2-NFA NC | 2-NFA | linear encoding | |
| | DFA NC | DFA | non-compressed layout | |
| | A-DFA LE | A-DFA | linear encoding | |
| | A-DFA BM | A-DFA | bitmapped encoding | |
| | 2-DFA NC | 2-DFA | non-compressed layout | |
| | 2-A-DFA LE | 2-A-DFA | linear encoding | |
| logic based | 1-NFA | NFA | stride-1 implementation | varying FPGA frequency |
| | 2-NFA software | 2-NFA | software based multi-stride | |
| | 2-NFA hardware | NFA | hardware based multi-stride | |
| | 4-NFA hardware | NFA | hardware based multi-stride | |

enough to represent the full address of a state, a transition address map is used. Linear encoding can effectively reduce memory size, but it is slower than non-compressed layout due to the linear search and the extra memory access to the address map.

- Bitmapped encoding: This memory encoding extends linear encoding by using bitmaps to avoid linear search. Each state has an array of $|\Sigma|$ bits indicating the existence of the corresponding transitions. While looking up a bitmapped encoded state, the bitmap is first analyzed. If it contains a 0 in the position of the input character, default transition is followed. Otherwise, a pop-count of 1s preceding the current position is performed, and the next transition can be accessed based on this information. Because the input character is not encoded in the transitions, the transition can hold the full address of a state and the address map is not needed. In this paper, we use a two-level bitmap. The first level is a 32-bit bitmap that addresses the $|\Sigma|$-bits level-two bitmap.

For both linear and bitmapped encoding, states having more than $th_{tx}$ outgoing transitions are still represented in non-compressed layout to allow fast access. This threshold parameter $th_{tx}$ is studied in Section 3.3.1.

Not all automata can use all three layouts. To use linear encoding and bitmapped encoding, a DFA must be converted into an A-DFA first. Bitmapped encoding is only useful for stride-1 DFAs. (For NFA states, the number of transitions is non-deterministic, which means we cannot locate the correct transition by pop-counting of 1 s. For multi-stride FAs, the alphabet size is too large, so bitmaps would require too much memory, and pop-counting would require too much time.)

For logic-based solutions, we use both software-based approach and hardware-based approach to multi-stride NFAs, as explained in Section 2.2. The software-based approach requires many resources, so it is only feasible to be used with two-NFA. The hardware-based approach, however, is capable to implement both two-NFA and four-NFA.

Given the various constraints of which automaton can be used with which memory layout, we only have a total 13 different possible automaton-implementation combinations. These are listed in Table 1. There are nine configurations for memory-based regular expression matching and four configurations for logic-based solutions. While this may seem like a small design space, there are many system parameters that also need to be considered.

TABLE 2
Characteristics of Rulesets

| Ruleset | #reg-ex | Length | | | # DFA |
|---|---|---|---|---|---|
| | | min | max | avg | |
| snort | 462 | 10 | 202 | 44.1 | 12 |
| l7-filter | 111 | 6 | 438 | 63.2 | 7 |
| bro | 782 | 5 | 211 | 34.8 | 8 |
| exact-match | 500 | 10 | 256 | 49.2 | 2 |
| dotstar 0.1 | 500 | 10 | 243 | 49.6 | 11 |
| dotstar 0.2 | 500 | 11 | 212 | 49.0 | 24 |
| dotstar 0.3 | 500 | 11 | 251 | 47.1 | 33 |
| dotstar 0.6 | 500 | 11 | 274 | 50.3 | 49 |

### 3.1.3 System

For memory-based solutions, the architecture of the memory subsystem has a large impact on matching performance. In this paper, we explore different cache sizes for level-1 and level-2 cache ($c_{D1}$ and $c_{D2}$) and different memory frequencies ($clk_{mem}$). In addition, we explore multi-core configurations, where memory is shared between multiple matching engines (see Fig. 2a).

Other architecture aspects may also impact the performance. For example, in our experiments, the speed of in-order execution is only 50 to 70 percent of that of out-of-order execution. Also, there is a $\pm 20$ percent speed difference between a bimodal branch predictor and a two-level adaptive predictor. However, these parameters are typically not configurable in practice and thus they are not discussed further.

For logic-based designs, once the FPGA model is determined, the major trade-off is between speed and power. As system parameter, we study the effect of different FPGA clock rates.

## 3.2 Inputs

To make our exploration representative of common practice, we focus on complex rulesets that have hundreds to thousands of entries and contain complex patterns, such as ranges and wildcards. There are two reasons to do this: first, rulesets used for network security purpose have increased both in size and complexity in the past several years; second, regular expression matching on small rulesets is a trivial problem: if a single DFA can be generated, then DFA is simply preferable over other automatons, because of its fast and deterministic matching speed.

In our experiments, we use both real rulesets (extracted from Snort [1], L7-filter [22], and Bro [2]) and some synthetic rulesets with different characteristics ("exact-match" and "dotstar-$x$" rulesets). Each dotstar-$x$ ruleset includes a fraction $x$ of regular expressions containing ".*" terms, which will lead to state explosion during DFA generation. This allows the exploration of the performance for different ruleset complexities. The synthetic rulesets are generated using the tool of Becchi et al. [18]. The characteristics of our rulesets are summarized in Table 2.

Traffic traces are generated by the traffic generator included in the regular expression processor [23]. Traffic size

TABLE 3
Parameter Set

| | Symbol | Description | Default Value |
|---|---|---|---|
| CPU | $clk_{CPU}$ | CPU frequency | 1.2GHz |
| | $a_{ALU}$ | area per ALU | 4.66 mm$^2$ |
| | $n$ | number of CPU cores | |
| cache | $c_{D1}$, $c_{D2}$ | L1/L2 D-cache size | |
| | $\tau_{L1}$ | L1 latency | 4 cycles |
| | $\tau_{L2}$ | L2 latency | 21 cycles |
| | $m_{D1}$, $m_{D2}$ | L1/L2 D-cache miss rate | |
| | *linesize* | cache line size | 32 B |
| | $a_{L1}$ | area of L1 cache per KB | 0.0052 mm$^2$/KB |
| | $a_{L2}$ | area of L2 cache per KB | 0.0014 mm$^2$/KB |
| off-chip memory | $\tau_{mem}$ | first DRAM read latency | 100 cycles |
| | $\tau_{mem2}$ | additional DRAM read latency | 10 cycles |
| | $bw_{mem}$ | DRAM bandwidth | 12.8 GB/s |
| | $clk_{mem}$ | DRAM bus frequency | 800 MHz |
| | $width_{mem}$ | DRAM bus width | 32 bit |
| | $channel_{mem}$ | DRAM channel | 2 |
| | $a_{mem}$ | area of DRAM per MB | 0.1367 mm$^2$/MB |
| | $\rho_{mem}$ | utilization of $bw_{mem}$ | |
| | $th_{mem}$ | maximum allowed memory bandwidth utilization | 20% |
| FPGA | $clk_{FPGA}$ | FPGA frequency | |
| automaton | $th_{DFA}$ | maximum number of states in single DFA | 15000 |
| | $th_{tx}$ | threshold of transitions with non-compressed layout | 4 |

significantly affects NFA's speed, because NFA accumulates more dead states during the matching if the traffic size is large. Since regular expression matching are usually applied to flows (i.e. packets with the same source IP, destination IP, source port, destination port, and protocol), the length of the trace should be equal to the average flow length in the network. In this paper, we select 1 MB trace size as a (nearly) worst case for NFA. According to Gebert et al.'s work [24], as of year 2010, 1 MB is longer than 95 percent of the multimedia flows and 99 percent of all other flow types.

Traffic content will also affect the performance of automatons. For each ruleset, we generate 4 traces with different $p_M$ values. The $p_M$ parameter is described in detail in Becchi et al.'s work[18]. It is the probability to move down one character in regular expression. Higher $p_M$ trace will traverse deeper on both NFA and DFA, causing more active NFA states, and less cache locality on both NFA and DFA. Therefore, lower $p_M$ trace represents normal traffic, and higher $p_M$ trace represents malicious traffic aiming to overload the matching algorithm.

If users want to use any NFA performance number in this paper (e.g., the numbers in Fig. 9), they should be aware that these are dependent on the trace size. If users want to determine whether to use NFA or DFA, they should follow the guideline in Section 5.2, to trail-run a sample trace with the NFA and get the $m_{NFA}$ value.

## 3.3 Evaluation

The memory-based solutions are evaluated on a TI OMAP 4460 ARM processor. This processor uses Cortex-A9 architecture and 45 nm technology. The parameters of this processor are listed in Table 3. We chose an ARM processor

Fig. 5. Power monitor used for power measurement.

because it has RISC architecture like most of the network processors do, and the simulators of ARM are more mature and precise than other RISC processors. Because cache size and speed cannot be changed on a real system, we also perform simulations using the gem5 simulator [25]. The parameters of simulator is calibrated against the real processor so that their results are comparable, as shown in Fig. 6.

The logic-based solutions are evaluated with the synthesis tool in the FPGA tool chain. We synthesized the designs on a Xilinx Virtex 5 device (XC5VLX50). We performed synthesis, translation, map, place and route and bit file generation using Xilinx's ISE Design Suite 10.1.

The power consumption usually is not one of the major concerns for a security oriented system. But since we are comparing two different hardware platform (CPU and FPGA), measuring the difference in power will help us to choose. For the ARM platform, the power is measured by a Monsoon power monitor as shown in Fig. 5. Because it is difficult to measure the CPU power consumption alone, we measure the power that incurred by the work load. That is, we measure the entire system power consumption under full work load, then subtract the idle system power from it, this will show the power difference incurred by algorithms and memory layout more clearly. For the FPGA platform, we use the Xilinx Power Estimator software for power consumption estimation.

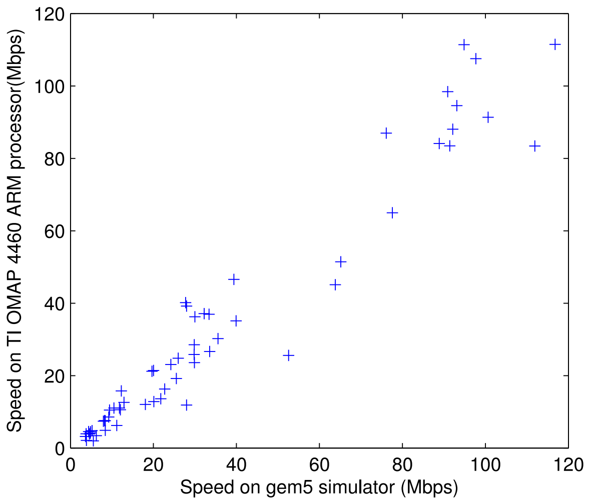


Fig. 6. Calibration of gem5 simulator and real processor. Each point represents the speed of a particular combination of parameters. The *x* and *y* axis's Pearson product-moment correlation coefficient r = 0.9675, meaning the correlation is very significant.

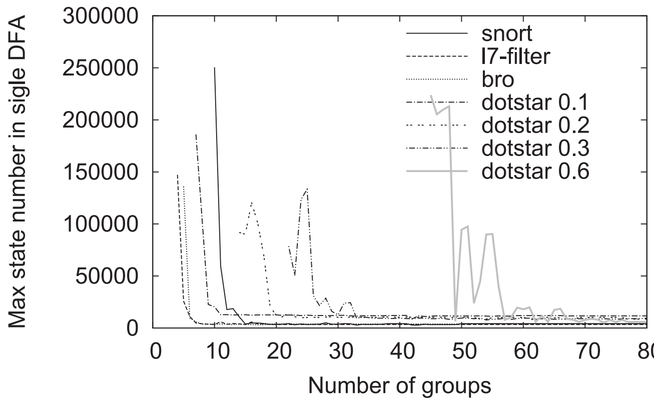


Fig. 7. Maximum and total DFA states for different number of groups.

### 3.3.1 Parameters

For memory-based solutions, Table 3 lists the parameters used in the simulation, including their notations and default values. Some parameters are not considered as design dimensions, but they still lead to trade-offs between speed and memory size, and need to be tuned before the evaluation process. We briefly explain how we have derived the default values for such parameters about ruleset partition and transition threshold:

*Maximum DFA states*. Ruleset partitioning for DFA is necessary to avoid state explosion. When partitioning, the maximum number of states $th_{DFA}$ in single DFA must be selected. There is a trade-off between $th_{DFA}$, DFA size, and the resulting number of DFAs. Fig. 7 shows that the total number of DFA states grows exponentially when the number of groups decreases. Once the state explosion starts, further decreasing the group number would be very difficult. A good $th_{DFA}$ for the rule sets we consider lies between 10,000 and 20,000. In this paper, we select $th_{DFA} = 15{,}000$. A much higher value would require much more memory and time to generate the automaton and would not significantly decrease the number of groups.

*Transition threshold*. For both linear and bitmapped encoding, states that have more than $th_{tx}$ outgoing transitions will be represented in non-compressed layout to allow fast access. A larger threshold $th_{tx}$ results in smaller memory size, but also slows down the speed because fewer states are represented in full and linear encoding has to traverse more transitions to find the correct one. Fig. 8 shows the trade-off between speed and memory size for different threshold values. It can be observed that for most of the algorithms, $th_{tx} = 4$ has the peak speed with reasonable memory size. Based on our results, we select $th_{tx} = 4$.

### 3.3.2 Metrics

The metrics used in our work are matching speed (in Mbps), resource cost (in MB for memory-based solutions, or FPGA slices for logic-based solutions), and power consumption (in mW). In Section 4, we present results for each of these metrics. In Section 5, we consider trade-offs between speed and resource cost for each configuration to identify optimal configurations.

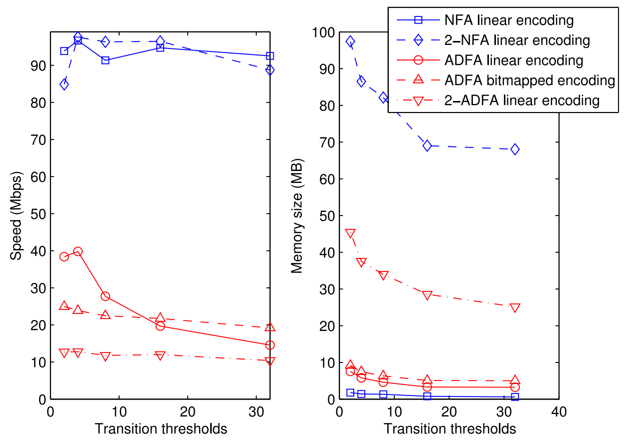


Fig. 8. Speed and memory size of different transition thresholds (TI OMAP 4460 ARM processor, Snort ruleset, $p_M = 0.35$).

## 4 Results

Based on the evaluation methodology described in Section 3, we present results obtained using real hardware as well as processor simulation.

## 4.1 Results from Real Hardware

### 4.1.1 Results of Memory-Based Implementations

Fig. 9 shows the throughput, memory size, and power consumption of the nine memory-based configurations listed in Table 1. We can observe that for some of the rulesets, DFA based implementations are faster, while some other rulesets have faster NFA based implementations. This is because the speed of DFA based implementations is proportional to the number of DFAs (i.e., $m_{DFA}$). NFA based implementations' speed is less predictable: the speed is positively correlated to the average number of active states (i.e., $m_{NFA}$), which is not known a priori, and can only be retrieved from experiments. Therefore, a ruleset with very high $m_{NFA}$ and very low $m_{DFA}$ should use DFA, and a ruleset with very high $m_{DFA}$ and very low $m_{NFA}$ should use NFA. The quantitative boundary between these two situations is analyzed in Section 5.1.2.

We can also observe that the performance does not double with the stride. This is because the increased memory size leads to higher cache miss rate. On processors with small caches, stride-2 FAs can be slower than stride-1 FAs.

Linear encoding reduces memory size by 80 to 90 percent compared to the non-compressed layout. Bitmapped encoding's compression ratio is about the same as linear encoding. Linear encoding and bitmapped encoding both introduce some extra processing, but the reduced memory size also reduces the cache miss rate. As a consequence, the processing speed decreases only slightly with compression: the throughput of linear and bitmapped encoding is about 70 to 90 percent of the non-compressed layout. For some implementations using large memory size (e.g., two-NFA and two-DFA), the compressed layouts even improve the processing speed.

### 4.1.2 Results of Logic-Based Implementations

Fig. 10 shows the speed, cost and power of the four logic-based configurations listed in Table 1. The cost is expressed

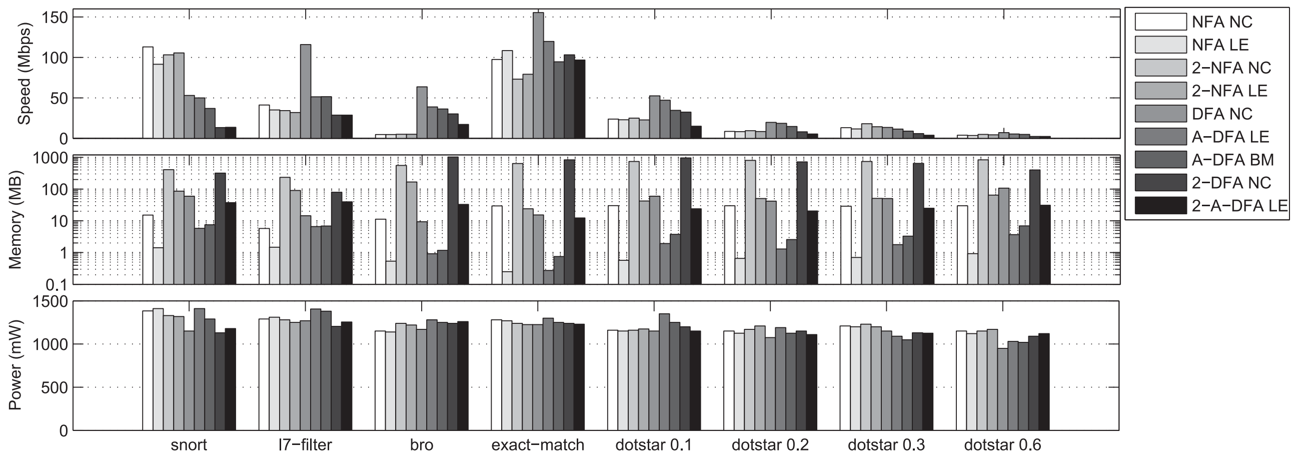


Fig. 9. Speed, memory and power of different rulesets, algorithms, and memory encodings. (TI OMAP 4460 ARM processor, $p_M = 0.35$ trace; power data include CPU and memory system power.)

in terms of slice usage. The Xilinx Virtex 5 device used in our experiments has 28,800 slices (our design does not make use of block memory).

The slice usage depends on the number of flip-flops (FF) and look-up tables (LUT) required by each FPGA design (the LUT implement combinatorial logic). To understand the slice usage data, we have to consider the characteristics of the NFA corresponding to our rulesets. In fact, the FF utilization depends solely on the number of NFA states (recall that each state is encoded in a FF); LUT are used to encode alphabet translation, character decoding, and state transitions. The number of NFA states varies from about 3 k (l7-filter) to about 8k (Snort) in our real rulesets, and is about 15 k across all our synthetic rulesets. The number of transitions depends on the reduced alphabet size. For stride-1 NFA, the alphabet size varies from 115 (Snort) to 197 (Bro) on our real rulesets, is 112 for exact-match and 116 for all dotstar-$x$ rulesets. In case of software stride-2 NFA, the alphabet size varies from 6,746 (Snort) to 11,731 (Bro) on our real rulesets, and from 5,502 to 7,096 (depending on the fraction of ".*" terms) on our synthetic rulesets. On stride-1 NFA, the number of transitions varies from about 23k (bro) to 90k (l7-filter) on real rulesets, and from about 15 to 56 k on synthetic rulesets with increasing complexity. For (software) stride-2 NFA, these numbers jump to 983 k (bro) to 4.8 M (l7-filter) on real rulesets, and to 69 k to 2.4 M on synthetic ones. Recall that hardware stride implementations are based on stride-1 automata.

For stride-1 NFA, the slice utilization is dominated by FF usage (the designs use only 1-8 percent of the LUT available on the FPGA). The LUT utilization increases with the stride (up to 50-100 percent the FPGA availability). In software-stride implementations, about half of the LUT are used for alphabet compression and input decoding; in hardware-based solutions, the LUT are mostly dedicated to encode state transitions. In the latter case (especially for stride 4), the Xilinx synthesizer is able to make better usage of the slice resources (by using all LUT and FF on the each slice).

The speed of logic-based solutions is determined by the clock rate, and is independent of the input trace:

$$speed = clk_{FPGA} \times stride \times 8\,bits. \quad (1)$$

The maximum speed depends on the maximum achievable clock rate ($clk_{FPGA\_MAX}$) of the circuit, which is related to the complexity of the combinational logic used (specifically, to the maximum number of logic gates that a signal must traverse in a clock cycle). The lowest clock rates are achieved using the software-based stride-2 implementation, because of its significantly increased alphabet. The stride-1 implementation, which is the least complicated of the four, leads to the fastest achievable clock rate. The hardware-based multiple stride implementations fall in the

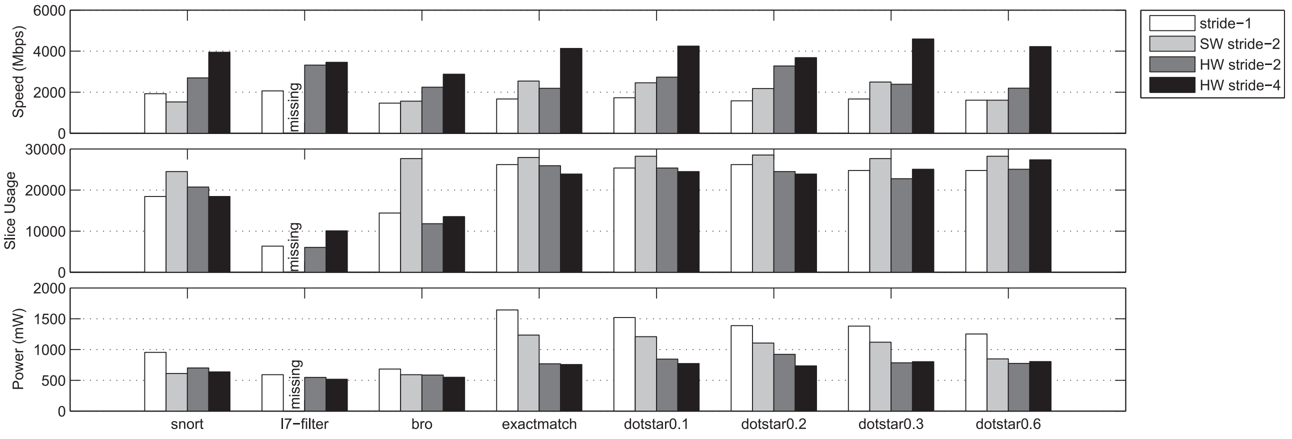


Fig. 10. Speed, slice usage and power of logic-based implementations. (l7-filter's software stride-2 implementation is missing because it exceeds the available slices).

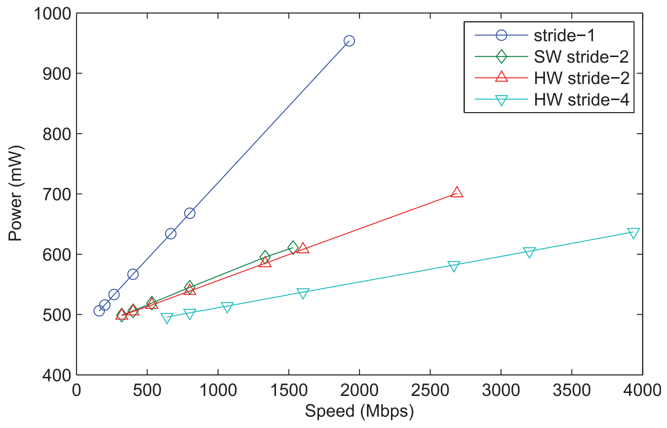


Fig. 11. Speed versus power trade-off of different logic-based implementations (Snort ruleset).

middle of the stride-1 and software based stride-2 implementations, due to the extra logic needed to match multiple characters at a time. Despite their reduced clock rate, multi-stride implementations achieve better speed due to their ability to process multiple input characters per clock cycle.

The power consumption of a CMOS circuit can be calculated by the following equation:

$$P = P_{static} + P_{dynamic} = P_{static} + \alpha C V^2 clk_{FPGA}, \tag{2}$$

where $\alpha$ is the activity factor (i.e., the fraction of the circuit that is switching), $C$ is the dynamic capacitance, $V$ is the supply voltage, and $clk_{FPGA}$ is the clock rate. Since multi-stride implementations can run at lower clock frequencies, they result in a lower power consumption when running at peak speed (Fig. 10). Fig. 11 shows each implementation's speed and power consumption under different clock rates. It can be observed that the dynamic power grows linearly with the speed (and the frequency). When also considering the added static power, higher frequency implementations have a higher speed/power ratio. Therefore, practitioners should choose the maximum achievable clock rate for a maximum speed/power ratio.

In general, it can be observed that the hardware-based stride 4 implementation leads to the best results, and that the processing speed of logic-based solutions far exceeds that of memory-based ones. It must be noted that these results are limited to processing a single packet flow. Multiple flow handling requires logic-replication on FPGA (and is therefore expensive). On the other hand, multiple flow handling on memory-based implementations requires duplicating only the flow-specific active state information (and not the automata). Therefore, a practitioner may prefer to adopt low-stride logic-based implementations (which have lower slice usage), and use the available FPGA resources to process multiple flows (rather than to achieve higher peak performance on a single packet stream).

## 4.2 Results from Processor Simulation

To explore the effects of parameters that can not be changed on a real processor, we use the gem5 simulator. The default configuration parameters of the simulator match those of ARM processor.

TABLE 4
Cache Miss Rate of Different Algorithms (Snort Ruleset, 32 KB L1 Cache, 1 MB L2 Cache)

| algorithm | D1 miss rate (%) | D2 miss rate (%) | Total miss rate (%) | D1 miss rate (%) | D2 miss rate (%) | Total miss rate (%) |
|---|---|---|---|---|---|---|
| | $p_M$=0.35 | | | $p_M$=0.55 | | |
| NFA NC | 1.72 | 1.95 | 0.03 | 2.04 | 1.12 | 0.02 |
| NFA LE | 0.33 | 6.09 | 0.02 | 0.24 | 7.75 | 0.02 |
| 2-NFA NC | 10.36 | 3.42 | 0.35 | 8.59 | 3.41 | 0.29 |
| 2-NFA LE | 6.59 | 1.26 | 0.08 | 4.58 | 1.41 | 0.06 |
| DFA NC | 1.43 | 0.57 | 0.01 | 1.34 | 0.47 | 0.01 |
| A-DFA LE | 1.39 | 0.15 | 0.00 | 1.06 | 0.18 | 0.00 |
| A-DFA BM | 0.31 | 0.56 | 0.00 | 0.30 | 0.56 | 0.00 |
| 2-DFA NC | 16.59 | 31.19 | 5.18 | 14.39 | 17.04 | 2.45 |
| 2-A-DFA LE | 11.46 | 32.60 | 3.74 | 9.93 | 17.87 | 1.78 |
| | $p_M$=0.75 | | | $p_M$=0.95 | | |
| NFA NC | 2.86 | 0.31 | 0.01 | 2.23 | 1.53 | 0.03 |
| NFA LE | 0.95 | 0.32 | 0.00 | 1.37 | 0.30 | 0.00 |
| 2-NFA NC | 5.01 | 3.44 | 0.17 | 7.64 | 13.46 | 1.03 |
| 2-NFA LE | 3.72 | 1.21 | 0.05 | 5.98 | 6.59 | 0.39 |
| DFA NC | 1.89 | 1.05 | 0.02 | 4.13 | 2.95 | 0.12 |
| A-DFA LE | 1.07 | 0.42 | 0.00 | 1.72 | 0.85 | 0.01 |
| A-DFA BM | 0.85 | 0.57 | 0.00 | 1.73 | 0.93 | 0.02 |
| 2-DFA NC | 13.46 | 6.91 | 0.93 | 17.80 | 9.38 | 1.67 |
| 2-A-DFA LE | 9.30 | 7.13 | 0.66 | 13.38 | 6.11 | 0.82 |

### *4.2.1 Cache Size*

Becchi et al. have observed that both NFA and DFA traversals exhibit a high degree of locality [18]. This implies the importance of having a cache system. Here, we explore the impact of L1/L2 cache, and what is the best cache size for regular expression matching. We focus on data cache (D-cache) instead of instruction cache (I-cache), because the FA size is usually larger than D-cache size, while the instruction size is small enough to fit into I-cache.

The cache replacement policy is configured as least-recently-used (LRU). L1 cache is two-way set associative, L2 cache is 16-way set associative. Other parameters are shown in Table 3.

Table 4 shows the L1 and L2 D-cache miss rate for different algorithms and traces. The result is based on Snort ruleset, but other rulesets show similar results. The total miss rate is the number of L2 miss divided by total memory read requests. Most total miss rates are below 1 percent (the exceptions are two-DFA and two-A-DFA LE, whose memory footprint is too large to fit L2). This means the cache system can effectively exploit the traversal locality for various algorithms and memory layouts. It also implies that cache latency actually determines the matching speed, while off-chip memory speed has very limited effect on it.

We employ an analytical model to estimate the effect of the latency of different parts. Assuming $\tau_{D1}$, $\tau_{D2}$, and $\tau_{DRAM}$ are latencies of L1, L2 D-cache, and DRAM, $m_{D1}$ and $m_{D2}$ are D1 and D2 miss rate, the average load instruction latency $\tau_{load}$ can be calculated as:

$$\tau_{load} = \tau_{D1}(1 - m_{D1}) + \tau_{D2} m_{D1}(1 - m_{D2}) + \tau_{DRAM} m_{D1} m_{D2}. \tag{3}$$

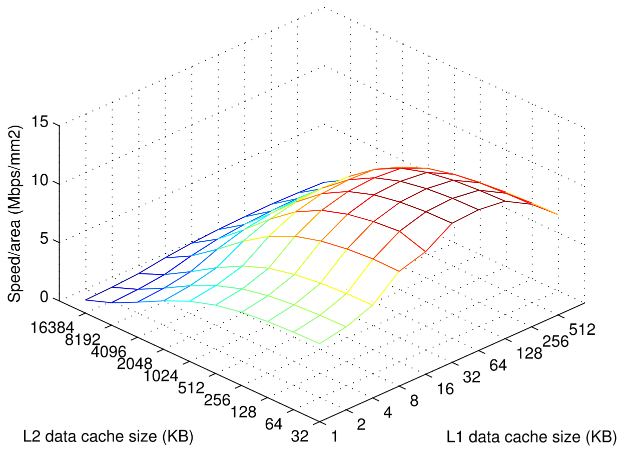


Fig. 12. Speed per area of different L1/L2 sizes (Snort ruleset, DFA NC algorithm).

Based on the data in Table 4, even if the DRAM were as fast as the L2 cache, $\tau_{load}$ only improves 5 percent on average.

To determine suitable L1 and L2 cache sizes for practice, we evaluate different combinations of L1 and L2 cache sizes, as shown in Fig. 12. We aim to find the maximum throughput per implementation area. The area of the entire matching system can be calculated by

$$A_{CPU} = A_{ALU} + a_{L1}(c_{I1} + c_{D1}) + a_{L2}c_{D2}, \tag{4}$$

where $A_{ALU}$ is the area of ALU and peripheral circuits, $a_{L1}$ and $a_{L2}$ are the area per KB of L1 and L2 cache. Parameters $c_{I1}$, $c_{D1}$ and $c_{D2}$ are the sizes of I1, D1 and D2 cache. The values for $c_{I1}$ and $c_{D1}$ are typically the same. We have simulated the speed per die area for each possible combination, and get the best cache size for each algorithm and memory layout in Table 5.

The cache performance shown in this section is for the pure regular expression matching task. If the processor were to do other tasks at the same time, such as packet classification and flow reassembly, the miss rate would increase. So it is recommended to allocate a dedicated core for each processing unit.

### 4.2.2 Memory Bandwidth and the Maximum Degree of Parallelism

In the previous section, we observed that memory bandwidth is not a bottleneck of a system with cache. The memory bandwidth utilization can be calculated as:

$$\begin{aligned}\rho_{mem} &= \frac{\frac{total\ cache\ miss}{total\ execution\ time} \times linesize}{bw_{mem}} \\ &= \frac{\frac{total\ cache\ miss}{total\ execution\ time} \times linesize}{2 \times clk_{mem} \times width_{mem} \times channel_{mem}}.\end{aligned} \tag{5}$$

For a 12.8 GB/s memory bandwidth (DDR3-1600 memory, dual channel, 32-bit width), the memory bandwidth utilization of each algorithm with optimal cache size is listed in Table 5. They are all less than 1 percent. Because of the low utilization of memory bandwidth, the read-only automaton data can be easily shared by multiple processing units (as illustrated in Fig. 2a). To determine the maximum parallelism $n$, we can set a threshold $th_{mem}$ for the

TABLE 5
Best Cache Size of Different Algorithms (Snort Ruleset, $p_M = 0.35$ Trace)

| | Optimal L1 size (KB) | Optimal L2 size (KB) | Utilization of $bw_{mem}$ (%) | Max threads supported |
|---|---|---|---|---|
| NFA NC | 64 | 128 | 0.19 | 107 |
| NFA LE | 32 | 32 | 0.11 | 185 |
| 2-NFA NC | 256 | 1024 | 0.40 | 50 |
| 2-NFA LE | 256 | 512 | 0.23 | 86 |
| DFA NC | 64 | 128 | 0.11 | 187 |
| A-DFA LE | 64 | 64 | 0.04 | 536 |
| A-DFA BM | 32 | 32 | 0.05 | 406 |
| 2-DFA NC | 256 | 4096 | 0.47 | 43 |
| 2-A-DFA LE | 128 | 4096 | 0.26 | 77 |

maximum allowed memory bandwidth utilization. David et al. [26] measured that when $\rho_{mem} = 0.2$, the memory latency only increases by about 6 percent compared to the memory latency while $\rho_{mem} = 0$. Therefore, it is reasonable to set $th_{mem} = 0.2$. So $n$ can be calculated as:

$$n = \frac{th_{mem}}{\rho_{mem}}. \tag{6}$$

Table 5 lists the maximum parallelism of each algorithm.

To illustrate that high levels of parallelism are feasible in practice, Fig. 13 shows the speedup of different algorithms. The experiment is run on a 32-core x86 HPC system (four Intel Xeon E7-4820 CPUs with 8 cores each). The regular expression matching software uses multithreading to utilize multiple cores. Each thread is bound to its dedicated core. The threads share the same automaton data. The result shows a linear speedup and corroborates the data in Table 5.

Thus, we have quantitative results for speed, size, and power and an understanding of the high levels of parallelism that are possible.

## 5 Optimal Regular Expression Matching Configuration

The results in Section 4 show the performance of different configurations and their related tradeoffs. We now turn to the main question raised in this paper: Which is the best

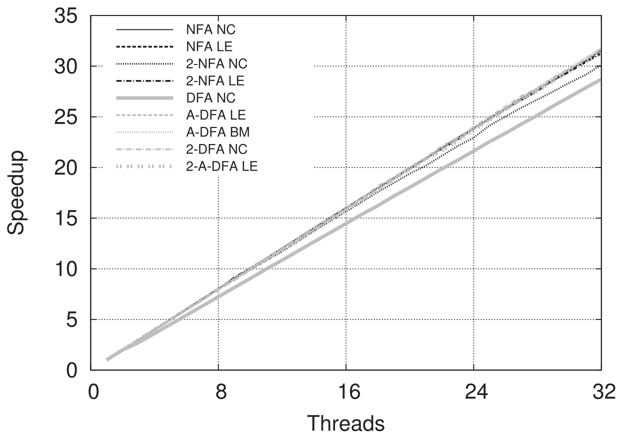


Fig. 13. Scalability of different memory based implementations on Intel x86 CPU (Snort ruleset, $p_M = 0.35$ trace).

TABLE 6
Highest Speed/Area for Different Cache Size, Implementations and Rulesets (in Mbps/mm$^2$, $p_M = 0.35$ Trace)

| | | snort | l7-filter | bro | exact match | dotstar 0.1 | dotstar 0.2 | dotstar 0.3 | dotstar 0.6 |
|---|---|---|---|---|---|---|---|---|---|
| single core | NFA NC | 11.06 | 4.84 | 0.52 | 4.54 | 1.66 | 0.70 | 1.01 | 0.29 |
| | NFA LE | 17.83 | 5.47 | 0.53 | 15.65 | 3.28 | 1.08 | 1.76 | 0.42 |
| | 2-NFA NC | 1.66 | 0.97 | 0.07 | 1.04 | 0.30 | 0.12 | 0.18 | 0.05 |
| | 2-NFA LE | 5.41 | 1.46 | 0.13 | 10.35 | 2.04 | 0.69 | 0.99 | 0.25 |
| | DFA NC | 5.49 | 17.40 | 9.94 | 7.70 | 4.71 | 2.38 | 1.65 | 0.54 |
| | A-DFA LE | 4.99 | 6.59 | 5.69 | 20.54 | 5.76 | 2.73 | 2.11 | 0.96 |
| | A-DFA BM | 4.99 | 6.76 | 5.02 | 15.50 | 4.75 | 2.24 | 1.31 | 0.62 |
| | 2-DFA NC | 0.93 | 4.09 | 0.40 | 0.94 | 0.39 | 0.16 | 0.12 | 0.11 |
| | 2-A-DFA LE | 2.24 | 3.18 | 2.70 | 9.79 | 2.54 | 0.93 | 0.61 | 0.33 |
| multiple cores | NFA NC | 15.94 | 5.66 | 0.69 | 7.84 | 3.09 | 1.31 | 1.85 | 0.53 |
| | NFA LE | 18.58 | 5.71 | 0.53 | 15.76 | 3.33 | 1.10 | 1.80 | 0.43 |
| | 2-NFA NC | 14.64 | 5.38 | 0.55 | 9.21 | 4.21 | 1.75 | 2.56 | 0.80 |
| | 2-NFA LE | 16.83 | 4.37 | 0.57 | 16.33 | 4.13 | 1.52 | 2.21 | 0.62 |
| | DFA NC | 14.38 | 24.28 | 12.30 | 10.49 | 11.97 | 5.02 | 3.86 | 1.97 |
| | A-DFA LE | 5.80 | 7.81 | 5.83 | 20.70 | 6.07 | 2.83 | 2.22 | 1.05 |
| | A-DFA BM | 6.07 | 8.10 | 5.19 | 15.83 | 5.27 | 2.41 | 1.43 | 0.74 |
| | 2-DFA NC | 4.34 | 8.15 | 4.37 | 8.37 | 4.25 | 0.76 | 0.46 | 0.33 |
| | 2-A-DFA LE | 3.30 | 4.78 | 3.82 | 11.32 | 3.29 | 1.15 | 0.77 | 0.44 |

overall configuration to use for regular expression matching? In particular, we use matching speed per implementation area as the key metric for this optimization. As we see, there is no single best configuration for all rule sets. Therefore, we also provide guidelines for determining the best configuration based on specific ruleset characteristics.

## 5.1 Optimal Configurations

### 5.1.1 Memory-Based or Logic-Based Solution

If users want to choose from the broad set of regular expression matching techniques, the first question would be whether to use memory-based or logic-based solutions. Comparing Fig. 9 with Fig. 10, we can see the FPGA implementation has higher speed and lower power consumption compared to CPU solution, this advantage even holds for multi-core processors. However, there are other factors that can dictate the decision, such as the price of the chips, the development difficulty, the ability to run other tasks, etc. These factors cannot be easily quantified. Therefore, the choice really depends on the application requirements, and it is up to the user to decide.

After the choice of memory-based and logic-based is made, the following two subsections describe the optimization within each design space.

### 5.1.2 Optimal Memory-Based Configurations

For the optimization of memory-based solutions, we assume that parallel processing is used, each processing unit has its own L1 and L2 cache but they share one automaton in main memory, and the best solution is selected based on speed per die area. (While power is important, the matching speed per power consumption of memory-based solutions is almost constant across configurations and thus not interesting for optimization.) Based on the analysis in Section 4, speed/area can be calculated as:

$$\frac{s}{A} = \frac{n * s}{(n * A_{cpu} + A_{DRAM})}, \tag{7}$$

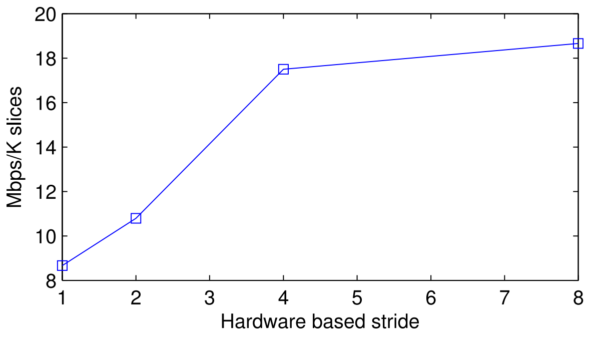


Fig. 14. Logic based solution's speed/utilization of different strides ( Hardware based multi-stride, Snort ruleset).

where $n$ is the maximum parallelism calculated by Equation (6). To avoid impractically large levels of parallelism, we limit $n$ to be no greater than 128. (Processors with more than 128 cores are rare in the market, and with so many parallel cores, the cost of processor cores dominates the cost of memory and other sub-systems and a further increase of $n$ does not significantly reduce speed/area/power.)

Table 6 shows the highest speed/area value for different configurations and rulesets. (The system parameters, such as cache sizes, are set to the best configuration, which may differ between implementations and rulesets.) The implementations with the optimal speed/area values are shaded in the table. Interestingly, a few simple configurations (NFA, DFA, 2-DFA) dominate other, more complex configurations.

### 5.1.3 Optimal Logic-Based Configurations

For logic-based configurations, the optimization is straightforward because of its much smaller design space. Based on the observations from Section 4.1.2, the hardware-based stride-4 implementation is the most efficient in terms of speed/area/power across all rulesets.

We also studied higher strides. Fig. 14 shows the speed per utilization of different strides with Snort ruleset. As the stride increases, the increment of speed gets smaller, and the cost of doubling stride is higher. It is possible that there would be a peak speed per utilization at some stride, but it is beyond this FPGA's resource to validate.

## 5.2 Optimization Guidelines

To propose a guideline for which implementation to use in which case, we should find the priority for each dimension first. According to our statistics, the better choice between NFA and DFA can be 270 percent better than the other one on speed/area on average. Optimal strides can be 139 percent better, and optimal encodings can be 50 percent better than non-optimal encoding. Therefore, we propose guidelines which considers these three dimensions in turn.

### 5.2.1 NFA or DFA

We can select the preference of NFA versus DFA by $m_{DFA}$ and $m_{NFA}$, because these two parameters determine the speed of these two types of automaton. Fig. 15 shows the optimal algorithms based on speed/area for different rulesets and traces. The axes of the plot are $m_{DFA}$ and $m_{NFA}$. It

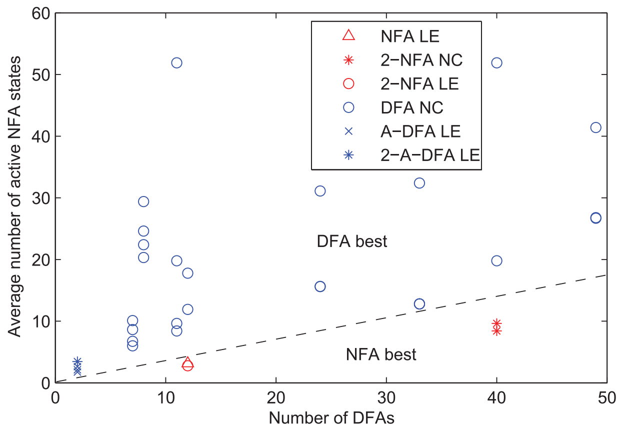


Fig. 15. Optimal algorithm selected by highest speed/area. Each point represents a ruleset-trace combination. The shape of point represents the optimal algorithm for this ruleset and trace. (Assume maximum parallel cores are used.)

can be observed that when $m_{NFA}/m_{DFA} < 0.35$, an NFA-based implementation is preferable; otherwise DFA-based implementations are preferable.

### 5.2.2 *Stride 1 or Stride 2*

Compared with multiple CPU method, increasing the stride is not a good method to boost performance because of its exponential memory requirement. So for large rulesets, stride-1 implementation usually have a better speed/area value. But for small rulesets, it can be used along with multiple CPU to further boost the throughput. According to our experiments, if the number of NFA states is below 7,000, or the total number of DFAs states is below 4,000, stride-2 can be used; otherwise stride-1 is better.

### 5.2.3 *Non-Compressed or Compressed Layouts*

The speed per area difference between non-compressed layouts and compressed layouts is not significant, so both of them can be used. For highly parallel configurations, memory cost can be amortized, so the implementation with highest speed achieves the best speed per area. Therefore, non-compressed layouts are usually better than compressed layouts. It is notable that bitmapped encoding is always worse than linear encoding in our experiments, thus it is not recommended.

When the automaton, the stride and the memory layout are decided, the best cache size can be found in Table 5, and the maximum number of parallel threads can be calculated by Equation (6). It is thus easily possible for a practitioner to determine the best implementation by merely determining $m_{DFA}$ and $m_{NFA}$, and number of NFA/DFA states for a given ruleset and traffic trace.

## 6 Conclusion

The many choices of automata, implementations, and system parameters make it difficult to determine a suitable regular expression matching configuration for a given rule set and traffic. In our work, we present a design space exploration of regular expression matching. We show experimental and simulation results to evaluate the performance and implementation cost of these choices. Our results provide guidelines for practitioners about which implementation to choose to achieve highest matching speed per implementation area. We believe that these results provide important insights for practical implementations of regular expression matching systems.

## Acknowledgments

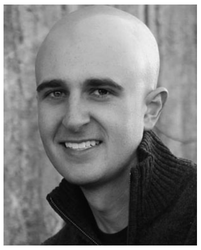
This material is based upon work supported by the US National Science Foundation (NSF) under Grant No. 1115999. The authors gratefully acknowledge support from Intel Corporation.

## References

[1] M. Roesch, "Snort-lightweight intrusion detection for networks," in *Proc. 13th USENIX Conf. Syst. Administ.*, 1999, pp. 229–238.
[2] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Comput. Netw.*, vol. 31, no. 23–24, pp. 2435–2463, 1999.
[3] J. E. Hopcroft, "An n log n algorithm for minimizing states in a finite automaton," Stanford Univ., Stanford, CA, USA, Tech. Rep. CS-TR-71-190, 1971.
[4] 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.*, vol. 36, no. 4, pp. 339–350, Aug. 2006.
[5] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst*, 2006, pp. 93–102.
[6] M. Becchi and P. Crowley, "A-DFA: A time- and space-efficient DFA compression algorithm for fast regular expression evaluation," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 1, pp. 4:1–4:26, Apr. 2013.
[7] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proc. ACM CoNEXT Conf.*, 2007, pp. 1:1–1:12.
[8] Y. Wen, X. Tang, L. Ju, and T. Chen, "Perex: A power efficient FPGA-based architecture for regular expression matching," in *Proc. IEEE/ACM Int. Conf. Green Comput. Commun*, 2011, pp. 188–193.
[9] K. Thompson, "Programming techniques: Regular expression search algorithm," *Commun. ACM*, vol. 11, no. 6, pp. 419–422, Jun. 1968.
[10] R. McNaughton and H. Yamada, "Regular expressions and state graphs for automata," *IRE Trans. Electron. Comput.*, vol. EC-9, no. 1, pp. 39–47, 1960.
[11] M. Becchi and P. Crowley, "Efficient regular expression evaluation: Theory to practice," in *Proc. 4th ACM/IEEE Symp. Archit. Netw. Commun. Syst.*, 2008, pp. 50–59.
[12] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2006.
[13] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in *Proc. 3rd ACM/IEEE Symp. Archit. Netw. Commun. Syst.*, 2007, pp. 155–164.
[14] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata," in *Proc. ACM SIGCOMM Conf. Data Commun.*, 2008, pp. 207–218.
[15] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," in *Proc. 33rd Annu. Int. Symp. Comput. Archit.*, 2006, pp. 191–202.
[16] S. Kong, R. Smith, and C. Estan, "Efficient signature matching with multiple alphabet compression tables," in *Proc. 4th Int. Conf. Security Privacy Commun. Netw.*, 2008, pp. 1:1–1:10.
[17] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. Di Pietro, "An improved DFA for fast regular expression matching," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 5, pp. 29–40, Sep. 2008.

[18] M. Becchi, M. Franklin, and P. Crowley, "A workload for evaluating deep packet inspection architectures," in *Proc. IEEE Int. Symp. Workload Characterization*, Sep. 2008, pp. 79–89.
[19] R. Sidhu and V. Prasanna, "Fast regular expression matching using FPGAS," in *Proc. 9th Annu. IEEE Symp. Field-Programmable Custom Comput. Mach.*, 2001, pp. 227–238.
[20] X. Qian, E. Yue-Peng, J.-G. Ge, and H.-L. Qian, "Efficient regular expression compression algorithm for deep packet inspection," *J. Softw.*, vol. 20, no. 8, pp. 2261–2272, Aug. 2009.
[21] T. Liu, A. Liu, J. Shi, Y. Sun, and L. Guo, "Towards fast and optimal grouping of regular expressions via DFA size estimation," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 10, pp. 1797–1809, Oct. 2014.
[22] L7-filter [Online]. Available: http://l7-filter.sourceforge.net/, 2013.
[23] Regular expression processor [Online]. Available: http://regex.wustl.edu/, 2011.
[24] S. Gebert, R. Pries, D. Schlosser, and K. Heck, *Internet Access Traffic Measurement and Analysis*. New York, NY, USA: Springer, 2012.
[25] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.
[26] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, "Memory power management via dynamic voltage/frequency scaling," in *Proc. 8th ACM Int. Conf. Autonomic Comput.*, 2011, pp. 31–40.

**Xinming Chen** received the BE degree in automation from Tsinghua University, China, in 2009, and the MS degree in automation from Tsinghua University, in 2011. He is currently working toward the PhD degree in the Department of Electrical and Computer Engineering, University of Massachusetts Amherst. His research interests include Internet architecture, software-defined networking, and deep packet inspection.



**Brandon Jones** received the BSc degree in computer engineering from the University of Missouri in 2014. He is currently a software engineer within Intel's Windows OS Division. His interests include software performance and development of new web technologies.

**Michela Becchi** received the MS and PhD degrees in computer engineering from Washington University in St. Louis, and the Laurea degree from Politecnico di Milano. She is currently an assistant professor in the Department of Electrical and Computer Engineering with joint appointments in Computer Science and the Informatics Institute, University of Missouri-Columbia. Previously, he worked at NEC Laboratories in Princeton, NJ; earlier, she spent a few years at IBM Research and Development GmbH in Boeblingen, Germany. Her research interests are in networking systems, high-performance computing, parallel computer architecture, and algorithm acceleration.

**Tilman Wolf** received the DSc degree in computer science from Washington University in St. Louis in 2002. He is a professor in the Department of Electrical and Computer Engineering, University of Massachusetts Amherst. His research interests include network processors, their application in next-generation Internet architectures, and embedded system security.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.