Picking Pesky Parameters: Optimizing Regular Expression Matching in Practice

Xinming Chen
Dept. of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA, USA
xinmingchen@ecs.umass.edu

Brandon Jones
Dept. of Electrical and Computer Engineering
University of Missouri, Columbia, MO, USA
bjjzqd@mail.missouri.edu

Michela Becchi
Dept. of Electrical and Computer Engineering
University of Missouri, Columbia, MO, USA
becchim@missouri.edu

Tilman Wolf
Dept. of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA, USA
wolf@ecs.umass.edu

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 vs. 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. 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 processors and field programmable gate arrays 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 significantly when implementing regular expression matching systems in practice.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*

General Terms

Algorithms, Performance, Design, Security

Keywords

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 [20] and Bro [1], 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 Gigabit 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 is based on 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 systems implementations for regular expression matching using different types of processors, logic components, and memory configurations.

The key problem in regular expression matching is not the lack of innovative techniques, but the difficulty of deciding *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 [12,16,25], some aim to improve the average and worst case throughput [4, 6], and some aim to reduce power and energy consumption [24]. 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

978-1-4799-1640-5/13/$31.00 ©2013 IEEE

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 vs. FPGA). We use real hardware as well as simulation to evaluate the throughput, memory size, energy consumption, 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 prove useful for regular expression matching systems in practice.

The remainder of the paper is organized as follows. In Section 2, we provide some more 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 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 [18,23] and optimized with several state- and transition-reduction techniques [5]. 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 [13], and reduced through the minimization algorithm described in [12]. 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 [4, 15, 22, 25] 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 [6,9,14], run-length encoding [9], default transition compression [6, 16], state merging [12] and delta-FAs [11] are mechanisms falling into the first category; multiple-DFAs [9, 25], hybrid-FAs [4], history-based-FAs [15], and XFAs [22] fall into the second one.

Multi-stride FAs (or k-NFA/k-DFA) [5,9] are 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 Σ, a k-NFA/k-DFA is equivalent to a FA defined on alphabet Σ^k. Multi-stride FA allow achieving 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 [5]. Figures 1(a) and (b) 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. Figure 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 allowing easy scalability to multiple packet flows. Memory-based solutions often use caches to overcome high memory latencies and memory bandwidth limitations. In logic-based solution-

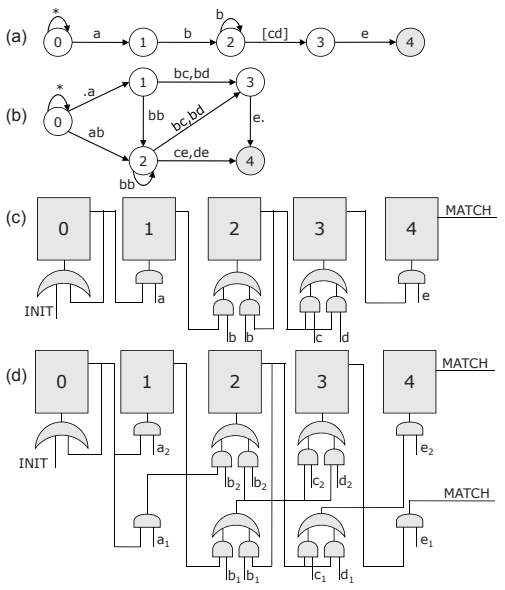

Figure 1: Example regular expression and automata for *.*ab+[cd]e*: (a) NFA, (b) 2-NFA, (c) stride-1 NFA logic design, and (d) hardware stride-2 NFA logic design.

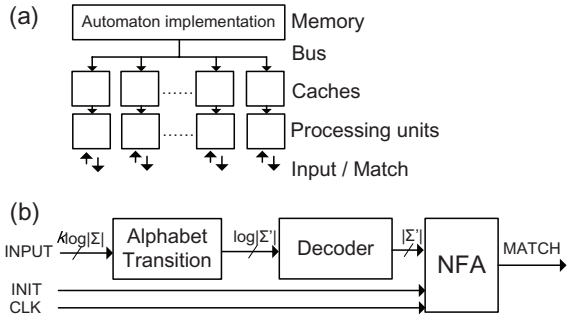

Figure 2: Abstraction of (a) memory-based and (b) logic-based solution.

s, usually implemented on FPGA, the automata are represented as logic circuits. Such solutions, typically based on NFA, have the advantage of algorithmic simplicity (they do not require sophisticated state and transition compression schemes). However, since they require automata replication to handle concurrent packet flows, they are not scalable to multiple flows.

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 [6, 16]. 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 [7]).

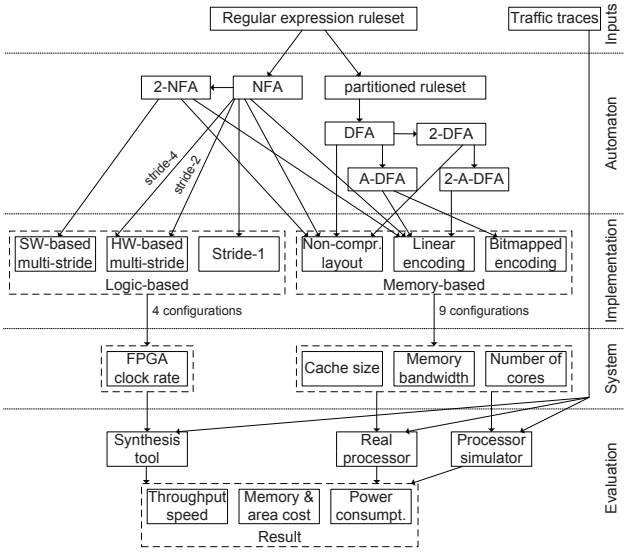

Figure 3: Design space of regular expression matching system.

Logic-based solutions typically encode NFA using the one-hot encoding scheme [21], 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 Figure 1(c), 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 [5]. Figure 2 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 [5]. 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 [5], we first generate a 2-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. Figure 1(d) 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 Figure 3. The core 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 Figure 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 [5].
- DFA: We use subset construction to convert and NFA into a DFA. The procedure described in [12] 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 [25], RECCADR [19], Becchi's [7], and GRELS [17]. We use GRELS as the partition algorithm, because it can generate minimum number of states with reasonable performance, and require least renew time among the available algorithms. GRELS 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 Figure 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, and 2-DFA in memory-based solutions and NFA, 2-NFA, and 4-NFA in logic-based solutions.

3.1.2 Implementation

For memory-based solutions, there are many choices for memory layouts to represent a given automaton. 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):

- Non-compressed layout: This memory encoding uses all $|\Sigma|$ transitions in a state. For NFAs, non-compressed layout uses a linked list to represent multiple transitions on one character. Characters with no corresponding transitions are represented with a NULL pointer. Epsilon transitions are removed to allow for faster speed. For DFAs, non-compressed layout does not use default transition. 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 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 2-level bitmap. The first level is a 32-bit bitmap that addresses the $|\Sigma|$-bits level-2 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 1s. 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

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	

2-NFA. The hardware-based approach, however, is capable to implement both 2-NFA and 4-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 9 configurations for memory-based regular expression matching and 4 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.

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 Figure 2(a)).

Other architecture aspects may also impact the performance. For example, in our experiments, the speed of in-order execution is only 50% to 70% of that of out-of-order execution. Also, there is a ±20% speed difference between a bimodal branch predictor and a 2-level adaptive predictor. However, these parameters are typically not configurable in practice and thus they are not discussed further.

For logic-based solutions, 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 that 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 due to its fast and deterministic matching speed.

In our experiments, we use both real rulesets (from Snort [20], L7-filter [2], and Bro [1]) 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 described in [7]. The characteristics of our rulesets are summarized in Table 2.

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

Real-life traffic traces are not used in our evaluation, because they can not construct the worst case scenario for NFA. In order to quantitatively evaluate the traces' effect to NFA performance, we use traffic traces generated by the traffic generator included in the Regular Expression Processor [3]. The size of a traffic trace is set to 1MB. Since traffic content may affect the performance of an NFA, there are 4 traces with different with p_M values. The p_M parameter is a measure of the probability to move down one character in regular expression. A high p_M value means potentially more active NFA states during matching (e.g., representing malicious traffic aiming to cause denial-of-service). Real-life traffic traces are generally random, and perform like the $p_M = 0.35$ traces in our experiments.

3.3 Evaluation

The various configurations are evaluated with various rulesets and trace configurations with three different techniques. Memory-based configurations are evaluated with real processors (where the number of cores can be varied). Since cache sizes and speed cannot be varied on a real system, we also perform simulations using the gem5 simulator [8]. The logic-based systems are evaluated with the synthesis tool in the FPGA tool chain, which provides information on throughput performance and resource use.

3.3.1 Parameters

There are a number of parameters that need to be set during the evaluation process. Table 3 lists these parameters, their notations, and default values. We briefly explain how we have determined values for ruleset partitioning and transition thresholds.

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 tradeoff between th_{DFA}, DFA size, and the resulting number of DFAs. Figure 4 shows that the total number of DFA states grows exponentially as 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 10000 and 20000. In

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	
	τ_{L1}	L1 latency	4 cycles
	τ_{L2}	L2 latency	21 cycles
	m_{D1}, m_{D2}	L1/L2 D-cache miss rate	
	$linesize$	cache line size	64 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	τ_{mem}	first DRAM read latency	100 cycles
	τ_{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
	ρ_{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

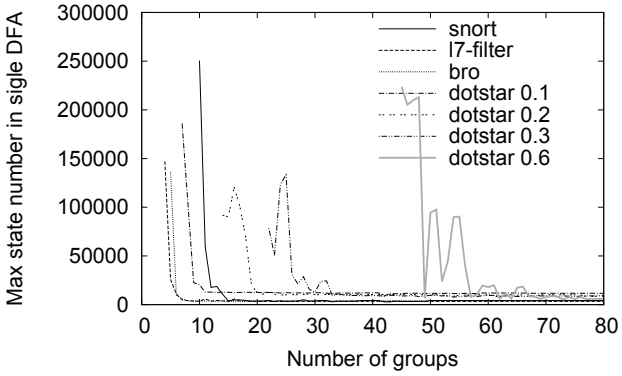

Figure 4: Maximum and total DFA states for different number of groups.

this paper, we select $th_{DFA} = 15000$. 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 less states are represented in full and linear encoding has to traverse more transitions to find the correct one. Figure 5 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 $th_{tx} = 4$.

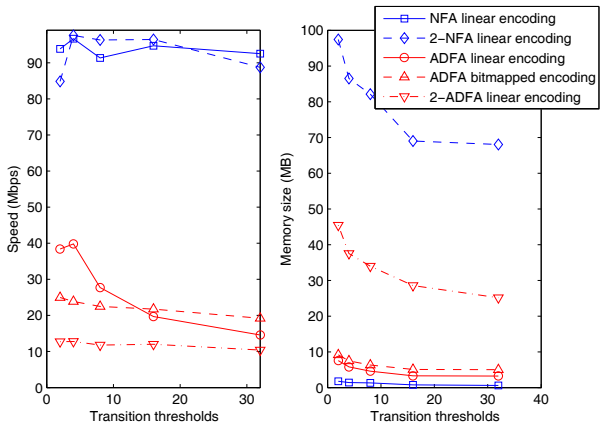

Figure 5: Speed and memory size of different transition thresholds (TI OMAP 4460 ARM processor, Snort ruleset, p_M=0.35).

3.3.2 Metrics

The metrics used in our work are matching speed (in Mbps), cost (in mm^2 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, consider tradeoffs between speed and resource cost for each configuration to identify optimal configurations.

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

For memory-based solutions, we used a TI OMAP 4460 ARM processor. This processor uses Cortex-A9 architecture and 45nm technology. The parameters of this processor are listed in Table 3. We chose an ARM processor because it has a good performance to power ratio. For logic-based solutions, we synthesized the designs on a Xilinx Virtex 5 device (XC5VLX50). We used the Xilinx Design Suite 10.1 for synthesis and the Xilinx Power Estimator tool for power consumption estimation.

4.1.1 Results of Memory-Based Implementations

Figure 6 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 a experimental run. Therefore, a ruleset with very high m_{NFA} and very low m_{DFA} should use DFA, and

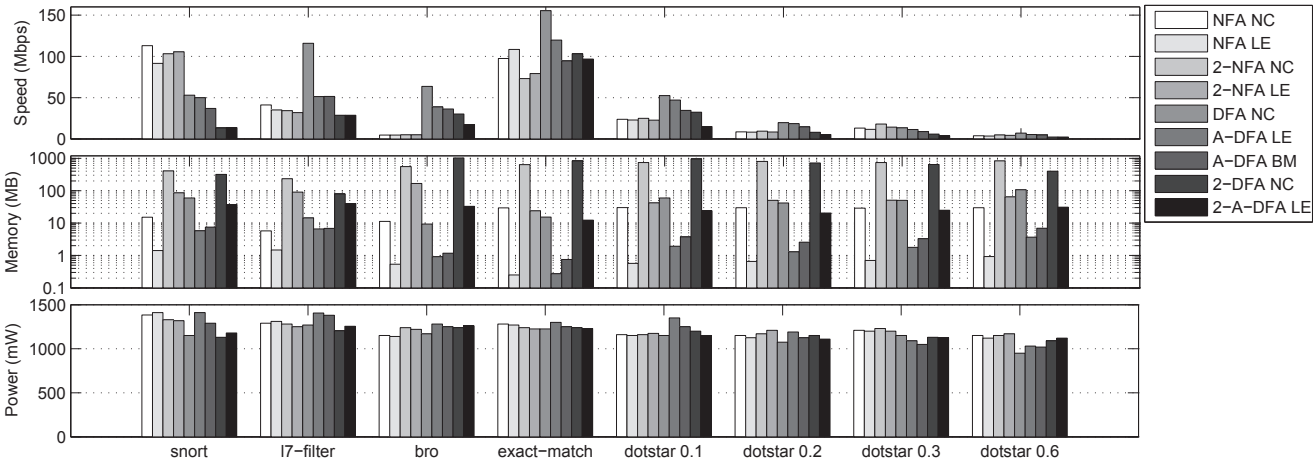

Figure 6: Speed, memory and power of different rulesets, algorithms, and memory encodings. (TI OMAP 4460 ARM processor, $p_M = 0.35$ trace; power data include CPU and memory system power.)

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.

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% 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% that of the non-compressed layout. For some implementations using large memory size (e.g., 2-NFA and 2-DFA), the compressed layouts even improve the processing speed.

4.1.2 Results of Logic-Based Implementations

Figure 7 shows the speed, cost and power of the four logic-based configurations listed in Table 1. The cost is expressed 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 3k (l7-filter) to about 8k (snort) in our real rulesets, and is about 15k 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 6746 (snort) to 11731 (bro) on our real rulesets, and from 5502 to 7096 (depending on the fraction of ".*" terms) on or 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 15k to 56k on synthetic rulesets with increasing complexity. For (software) stride-2 NFA, these numbers jump to 983k (bro) to 4.8M (l7-filter) on real rulesets, and to 69k to 2.4M 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% of the LUT available on the FPGA). The LUT utilization increases with the stride (up to 50-100% the FPGA availability). In software-stride implementations, about half 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 only related to 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 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 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 calcu-

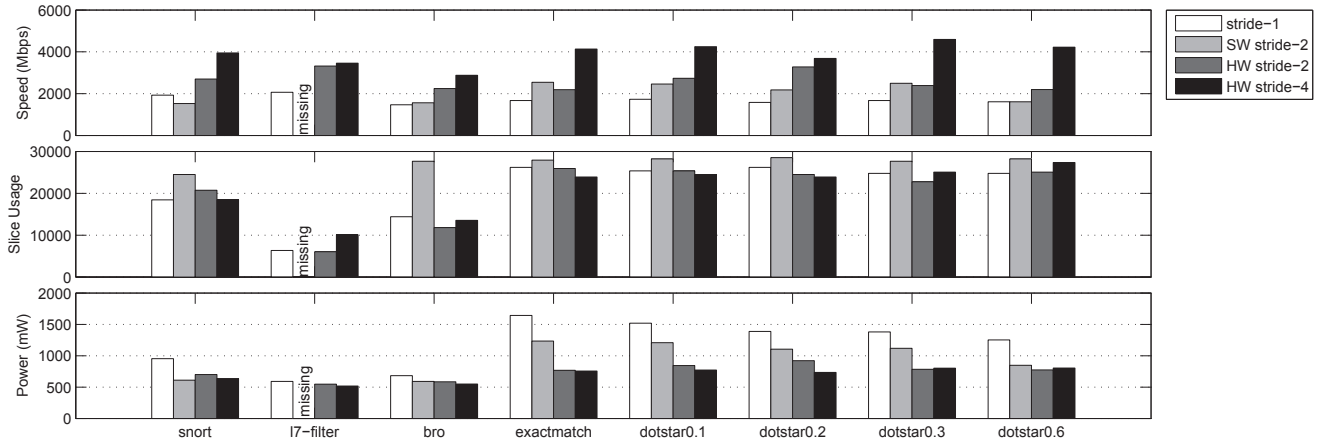

Figure 7: Speed, slice usage and power of logic-based implementations. (l7-filter's software stride-2 implementation is missing because it exceeds the available slices)

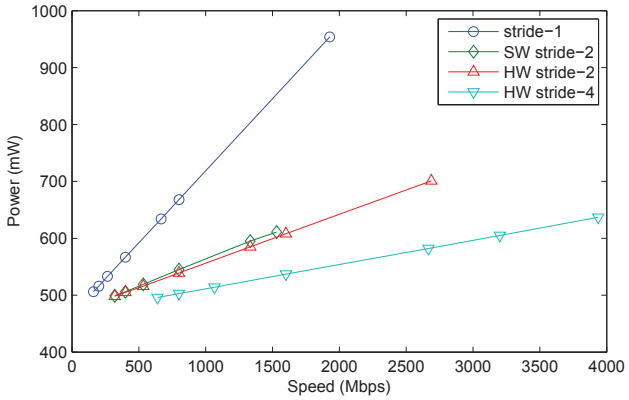

Figure 8: Speed vs. power trade-off of different logic-based implementations (Snort ruleset).

lated by the following equation:

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

where α 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 (Figure 7). Figure 8 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.

4.2.1 Cache Size

In [7], Becchi et al. observed that both NFA and DFA traversals exhibit a high degree of locality. 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 sizes are usually larger than D-cache size, while the instruction size are small enough to fit into I-cache.

The cache replacement policy is configured as least-recently-used (LRU). L1 cache is 2-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% (the exceptions are 2-DFA and 2-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.

An analytical model can be employed to estimate the effect of the latency of different parts. Assuming τ_{D1}, τ_{D2},

Table 4: Cache miss rate of different algorithms (Snort ruleset, 32KB L1 cache, 1MB 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

and τ_{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 τ_{load} can be calculated as:

$$\begin{aligned}\tau_{load} &= \tau_{D1}(1-m_{D1})+\tau_{D2}m_{D1}(1-m_{D2}) \\ &\quad +\tau_{DRAM}m_{D1}m_{D2}. \end{aligned} \tag{3}$$

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

To determine suitable L1 and L2 cache sizes for practice, we evaluate different combinations of L1 and L2 cache sizes. 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_{D1} and c_{D2} 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 Parallel Processing

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

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

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%. Because of the low utilization of memory bandwidth, the read-only automaton data can be easily shared by multiple processing units (as illustrated in Figure 2(a)). To determine the maximum parallelism n, we can set a threshold th_{mem} for the maximum allowed memory bandwidth utilization. David et al. [10] measured that when $\rho_{mem} = 0.2$, the memory latency only increases by about 6% compared to the memory latency while $\rho_{mem} = 0$. Therefore, it is reasonable to set $th_{mem} = 0.2$. So n can be calculated by:

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

Table 5 lists the maximum parallelism of each algorithm.

To illustrate that high levels of parallelism are feasible in practice, Figure 9 shows the speedup of different algorithms. The experiment is run on a 32-core x86 HPC system (four

Table 5: Best cache size of different algorithms (Snort ruleset).

	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

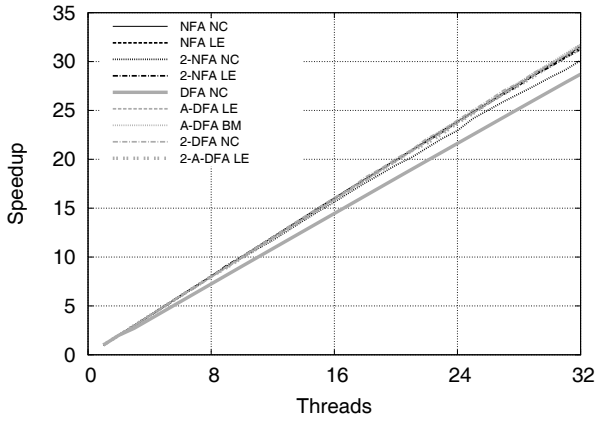

Figure 9: Scalability of different memory based implementations on Intel x86 CPU.

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

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

For the optimization of memory-based configurations, 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/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:

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

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.)

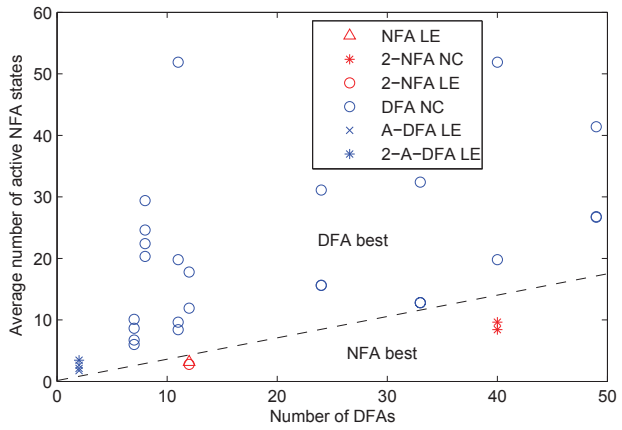

Figure 10: Optimal algorithm selected by speed/area. (Multiple cores allowed.)

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.

For logic-based configurations, the optimization for logic-based implementations is not really an issue 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.

5.2 Optimization Guidelines

As a guideline for which implementation to use in which case, Figure 10 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 can be observed that when $m_{NFA}/m_{DFA} < 0.35$, an NFA-based implementation is preferable; otherwise DFA-based implementations are preferable. The speed per area difference between NFA NC and NFA LE 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 2-DFA is not the best option in most cases, primarily because of its large memory size and low cache locality. Only for some simple rulesets, 2-DFA LE is faster than DFA.

Based on our results in Figure 10 together with Table 5, it is thus easily possible for a practitioner to determine the best (NFA-based or DFA-based) implementation by merely determining m_{DFA} and m_{NFA} for a given ruleset and traffic trace.

6. CONCLUSION

The many choices of automata, implementations, and system parameters makes 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 and 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 National Science Foundation under Grant No. 1115999 and No. 1216756. The authors gratefully acknowledge support form Intel Corporation.

7. REFERENCES

[1] Bro intrusion detection systems. `http://www.bro.org/`.

[2] L7-filter. `http://l7-filter.sourceforge.net/`.

[3] Regular expression processor. `http://regex.wustl.edu/`.

[4] M. Becchi and P. Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM CoNEXT conference*, CoNEXT '07, pages 1:1–1:12, New York, NY, USA, 2007. ACM.

[5] M. Becchi and P. Crowley. Efficient regular expression evaluation: theory to practice. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 50–59, New York, NY, USA, 2008. ACM.

[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.*, 10(1):4:1–4:26, Apr. 2013.

[7] M. Becchi, M. Franklin, and P. Crowley. A workload for evaluating deep packet inspection architectures. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 79–89, Sept.

[8] 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*, 39(2):1–7, Aug. 2011.

[9] B. C. Brodie, D. E. Taylor, and R. K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, pages 191–202, Washington, DC, USA, 2006. IEEE Computer Society.

[10] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, pages 31–40, New York, NY, USA, 2011. ACM.

[11] 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.*, 38(5):29–40, Sept. 2008.

[12] J. E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.

[13] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[14] S. Kong, R. Smith, and C. Estan. Efficient signature matching with multiple alphabet compression tables. In *Proceedings of the 4th international conference on Security and privacy in communication netowrks*, SecureComm '08, pages 1:1–1:10, New York, NY, USA, 2008. ACM.

[15] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, ANCS '07, pages 155–164, New York, NY, USA, 2007. ACM.

[16] 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, Aug. 2006.

[17] T. Liu, Y. Sun, D. Bu, L. Guo, and B. Fang. Grels: Regular expression grouping algorithm based on local searching. *Journal of Software*, 23(9):2261–2272, Sept. 2012.

[18] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *Electronic Computers, IRE Transactions on*, EC-9(1):39–47, 1960.

[19] X. Qian, E. Yue-Peng, J.-G. Ge, and H.-L. Qian. Efficient regular expression compression algorithm for deep packet inspection. *Journal of Software*, 20(8):2261–2272, Aug. 2009.

[20] M. Roesch. Snort Íc lightweight intrusion detection for networks. In *the 13th USENIX Conference on System Administration*, 1999.

[21] R. Sidhu and V. Prasanna. Fast regular expression matching using fpgas. In *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, pages 227–238, 2001.

[22] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 207–218, New York, NY, USA, 2008. ACM.

[23] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[24] Y. Wen, X. Tang, L. Ju, and T. Chen. Perex: A power efficient fpga-based architecture for regular expression matching. In *Green Computing and Communications (GreenCom), 2011 IEEE/ACM International Conference on*, pages 188–193, 2011.

[25] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Architecture for Networking and Communications systems, 2006. ANCS 2006. ACM/IEEE Symposium on*, 2006.