Picking Pesky Parameters: Optimizing Regular Expression Matching in Practice

September 14, 2016

Outline

- **Introduction to regular expressions**
- Design space exploration
- Results
- Optimal Regular Expression Matching Configuration
- Conclusion

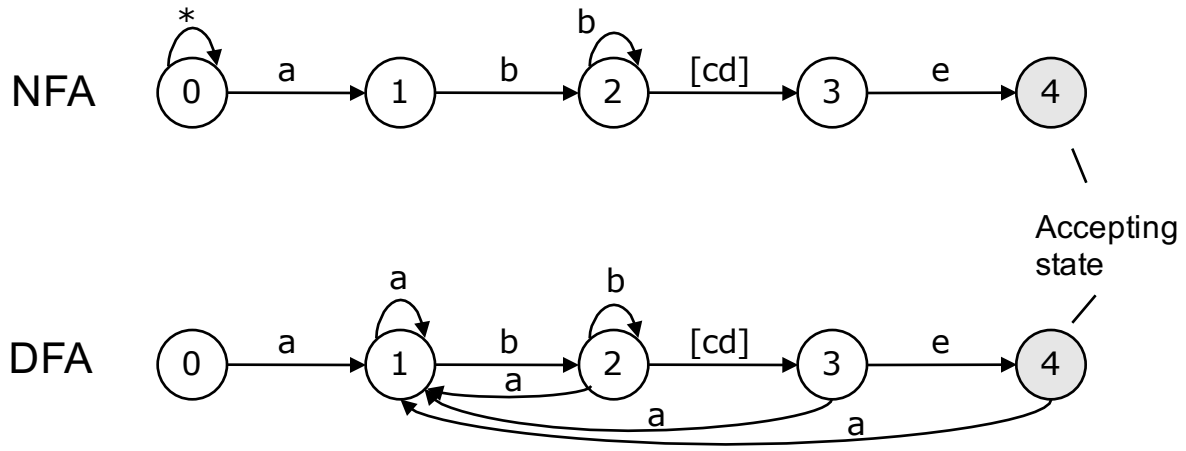
What is regular expression matching

- A regular expression (abbreviated **regex**) patterns a match to a string.
  - E.g. this regex matches a valid IP address:
  - (([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.){3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])
- Application of regular expression matching:
  - bibliographic search
  - Intrusion detection system
  - Protocol identification
  - Content filtering
- Many network security software, such as Snort and Bro, use rule sets of regular expressions that match attacks.
- These software need to operate at multiple to tens of Gigabit per second link rates to meet the performance requirements of the network.

How to implement a regex lookup engine?

1. Transform the rule set into a state machine (finite automaton).
2. packet payloads are scanned by traversing the state machine.

- Automaton can be non-deterministic (NFA) or deterministic (DFA)
- Example: NFA and DFA of .*ab+[cd]e



What is the problem?

- There are too many algorithms proposed to tune regex matching.
- There are too many different systems implementations for regex matching:
  - Different hardware;
  - Different types of processors;
  - Different memory configurations.
- The performance metrics used in previous publications differ:
  - reduce memory requirements;
  - improve the average and worst case throughput;
  - reduce power and energy consumption.
- It is very difficult to determine which technique or system implementation to use.

What does our work do?

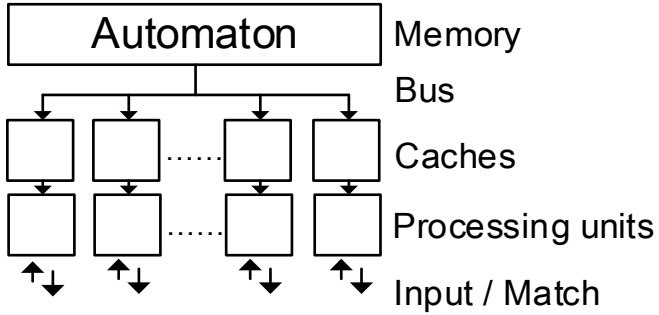
- Our work addresses the problem of choosing which regular expression technique to use for a given system, rule set, and traffic configuration.
- We present a systematic evaluation of many widely used regular expression techniques using real-world rule sets.
- We evaluate the throughput, memory size, energy consumption, and estimated chip area of each configuration.
- We provide a method for choosing the right configuration based on the results from our experiments.

Outline

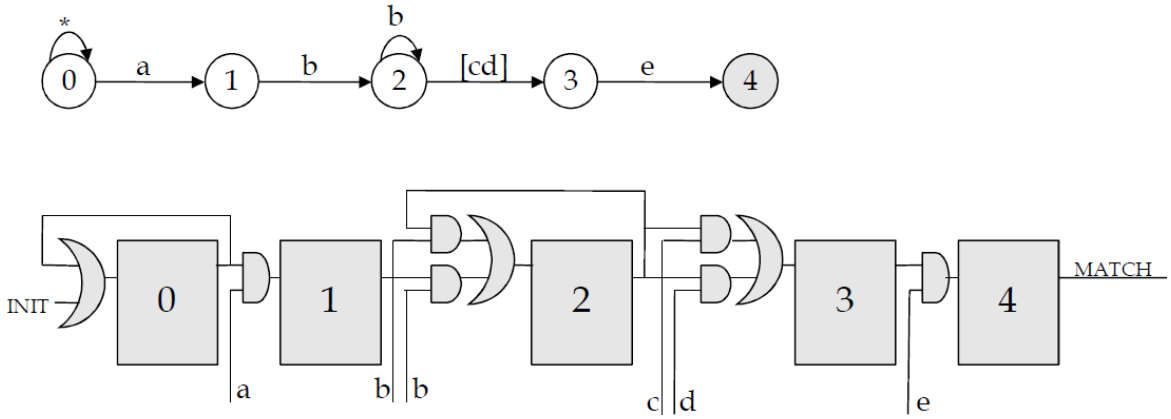
- Introduction to regular expressions
- **Design space exploration**
- Results
- Optimal Regular Expression Matching Configuration
- Conclusion

Two types of solutions

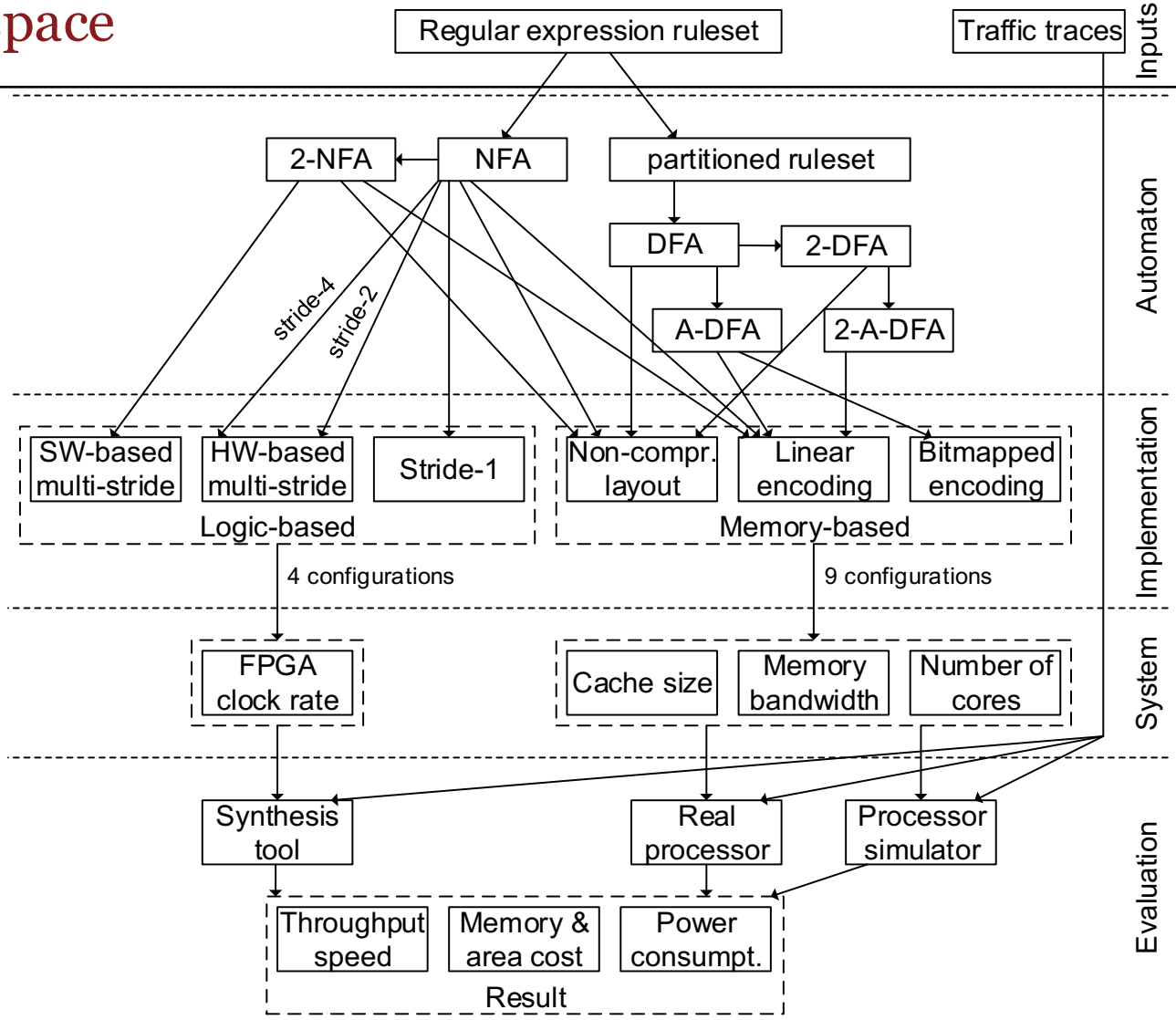
- Memory based solution



- Logic based solution



Design space



Automaton domain

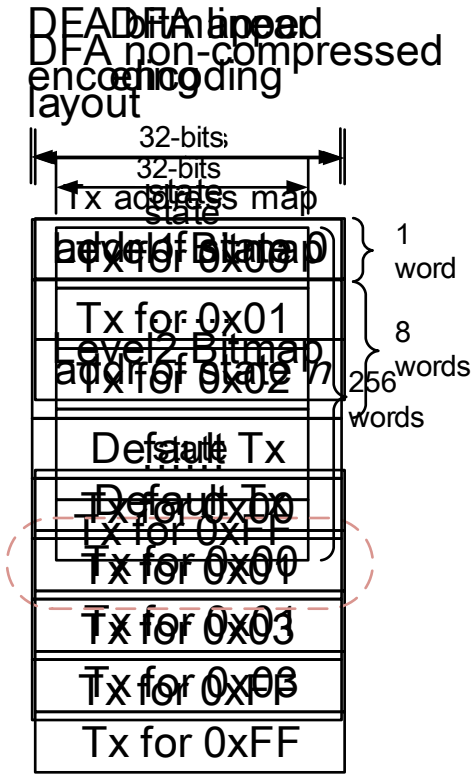
- NFA (Nondeterministic Finite Automaton)
  - Generated from regex ruleset.
  - The number of states is small, but it allows multiple state activations at the same time.
- DFA (Deterministic Finite Automaton)
  - Generated from NFA.
  - Allows only one active state at the same time: stable performance.
  - Size could grow exponentially if some complex patterns exist (called state explosion).
  - Large rulesets need to be partitioned into several parts, and generate multiple DFAs.
  - A-DFA: a compression technique that allow a DFA state use less than 256 transitions. Should use with a compressed memory layout.
- Multi-stride NFA/DFA (or k-NFA/k-DFA)
  - 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.

Implementation domain -- Memory based solution

- Three memory layouts:

1. Non-compressed layout
   - Uses all |Σ| transitions in a state.
2. Linear encoding
   - Only encodes the existing transitions in an NFA, or default transition and other transitions in an A-DFA.
   - Linear search is performed until a transition matching the input character is found or its absence is verified.
3. Bitmapped encoding
   - Similar to linear encoding, but use a bitmap to avoid linear search.
   - Only apply to stride-1 DFA

- 9 configurations in total
  - Non-compressed – NFA, DFA, 2-NFA, 2-DFA
  - Linear encoding – NFA, A-DFA, 2-NFA, 2-A-DFA
  - Bitmapped encoding – A-DFA



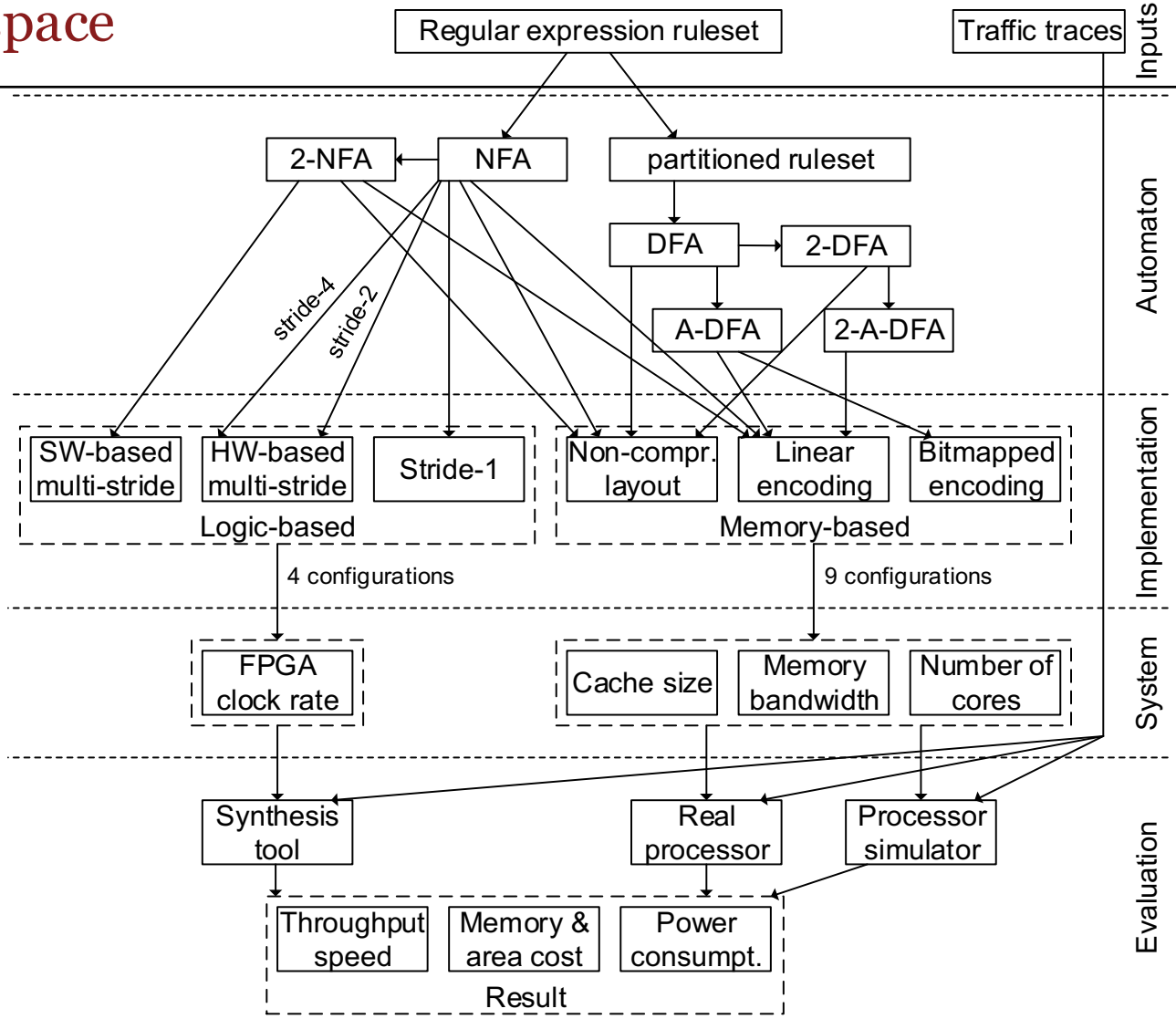
Implementation domain -- Logic based solution

- Logic based solutions only use NFA
  - Stride-1 implementation
  - Software-based multi-stride approach
    - First generate a k-NFA, then encode it in logic.
    - Resource costly, can only support stride-2
  - Hardware-based multi-stride approach
    - Have a stride-one NFA and the corresponding alphabet translation table
    - Resource efficient, can support up to stride-4
- 4 configurations in total
  - Stride-1 implementation
  - Software-based -- 2-NFA
  - Hardware-based -- 2-NFA, 4-NFA

System domain

- Memory based solution
  - Different cache sizes for level-1 and level-2 cache
  - Memory bandwidth
  - Different number of cores
- Logic based solution
  - Different FPGA clock rates

Design space

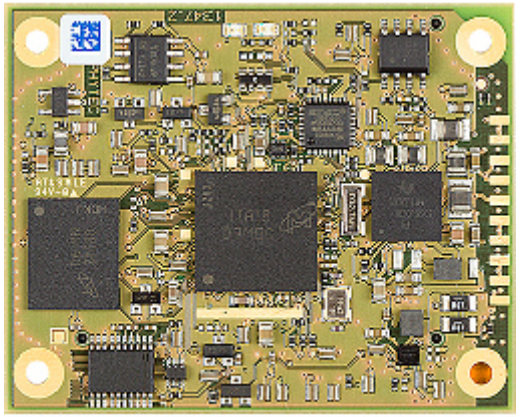


Outline

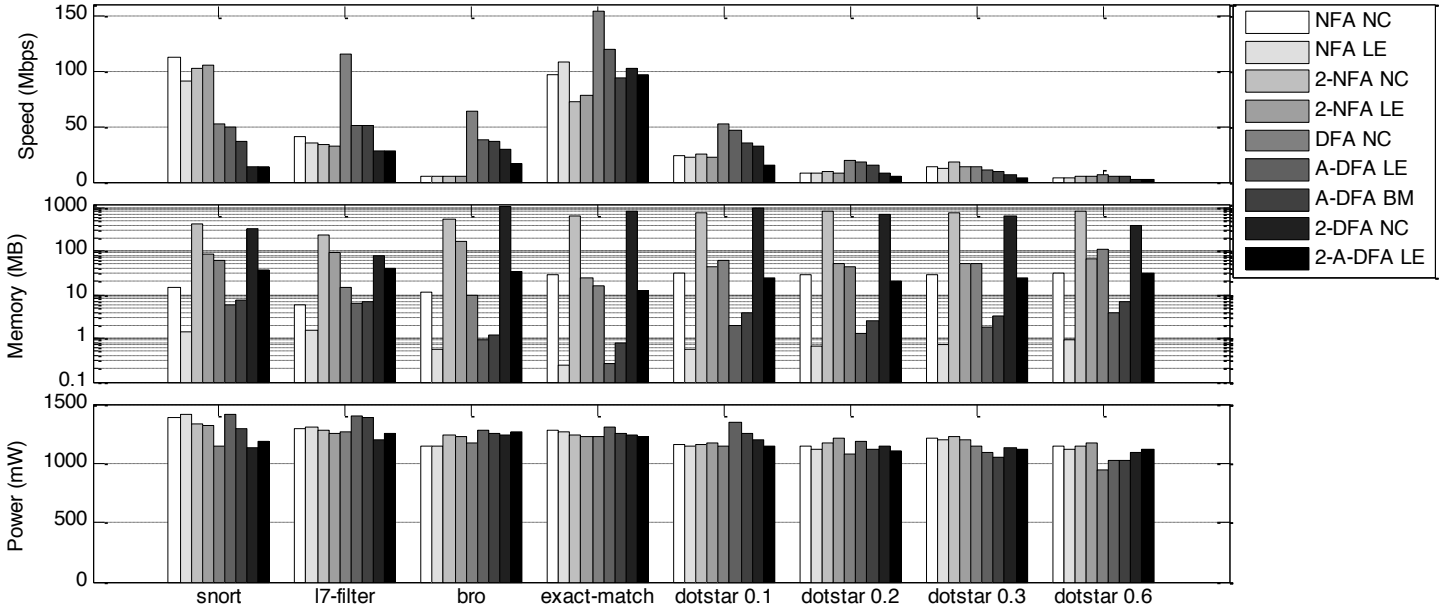
- Introduction to regular expressions
- Design space exploration
- **Results**
- Optimal Regular Expression Matching Configuration
- Conclusion

Evaluation Methodology

- Real hardware
  - TI OMAP 4460 ARM processor
  - Xilinx Virtex 5 FPGA (XC5VLX50)
  - Speed, memory usage/slice usage and power are measured
- Simulator
  - SimpleScalar simulator, calibrated with real hardware.
  - To study the parameters which can not be changed on real hardware
    - Cache size
    - Memory bandwidth
- Inputs
  - We use both real rulesets (from Snort, L7-filter, and Bro) and some synthetic rulesets with different characteristics.
  - Traffic traces are generated by the traffic generator (written by Becchi et.al.)



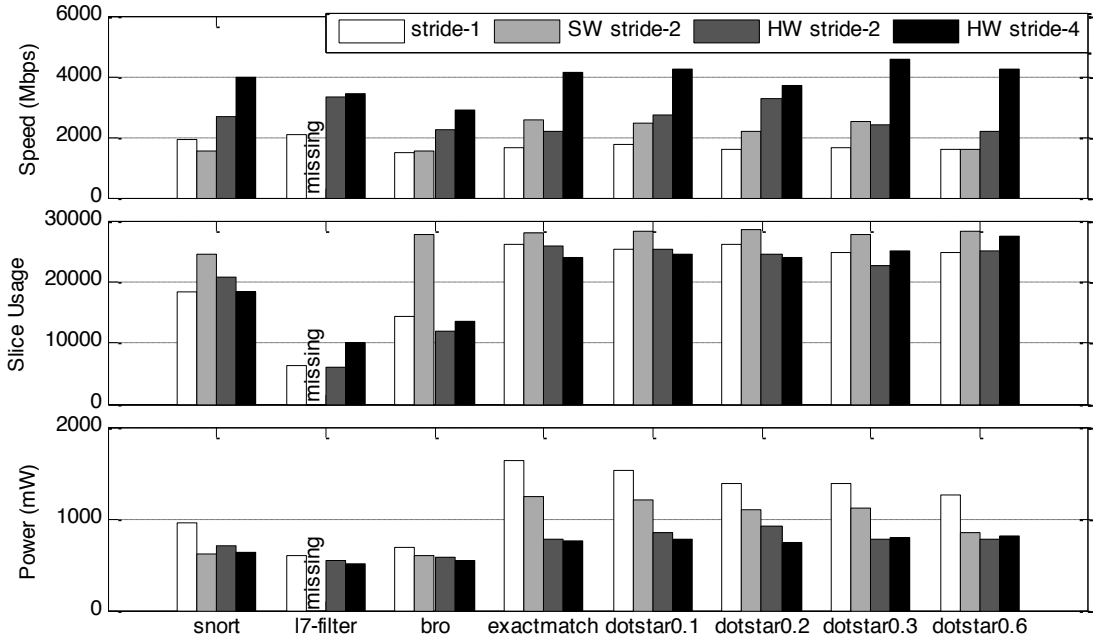
Results from real hardware – Memory based solutions



- TI OMAP 4460 ARM processor
- Rulesets 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.
  - m_{NFA}: the average number of active states in NFA
  - m_{DFA}: the number of DFAs

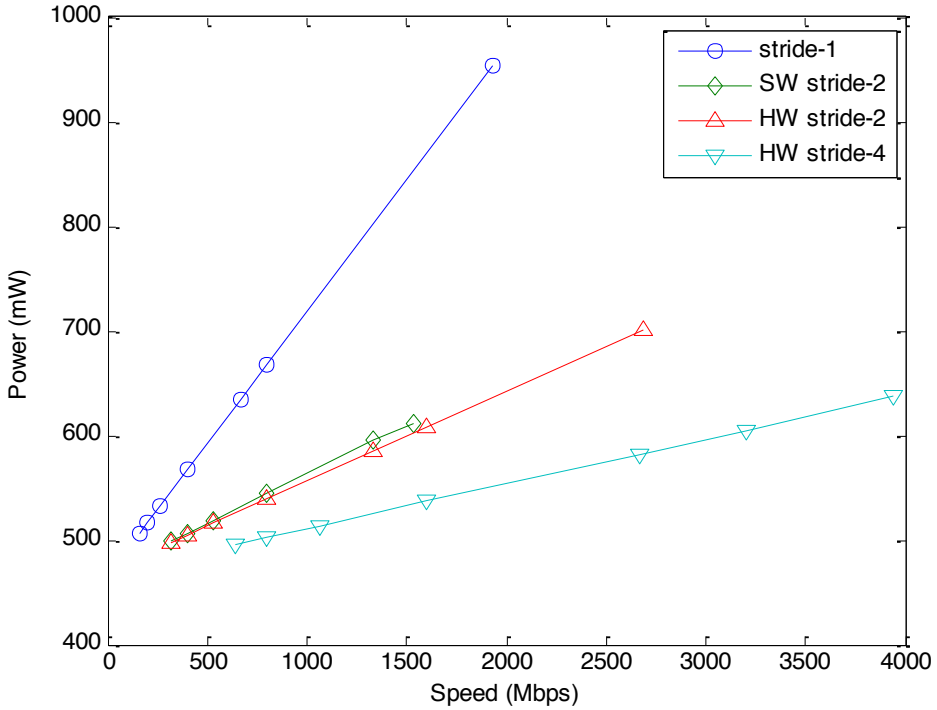
Ruleset	#reg-ex	Length			m_{DFA}	m_{NFA}
		min	max	avg		
snort	462	10	202	44.1	12	2.76
l7-filter	111	6	438	63.2	7	6.02
bro	782	5	211	34.8	8	20.34
exact-match	500	10	256	49.2	2	1.76
dotstar 0.1	500	10	243	49.6	11	8.42
dotstar 0.2	500	11	212	49.0	24	15.64
dotstar 0.3	500	11	251	47.1	33	12.76
dotstar 0.6	500	11	274	50.3	49	26.76

Results from real hardware – Logic based solutions



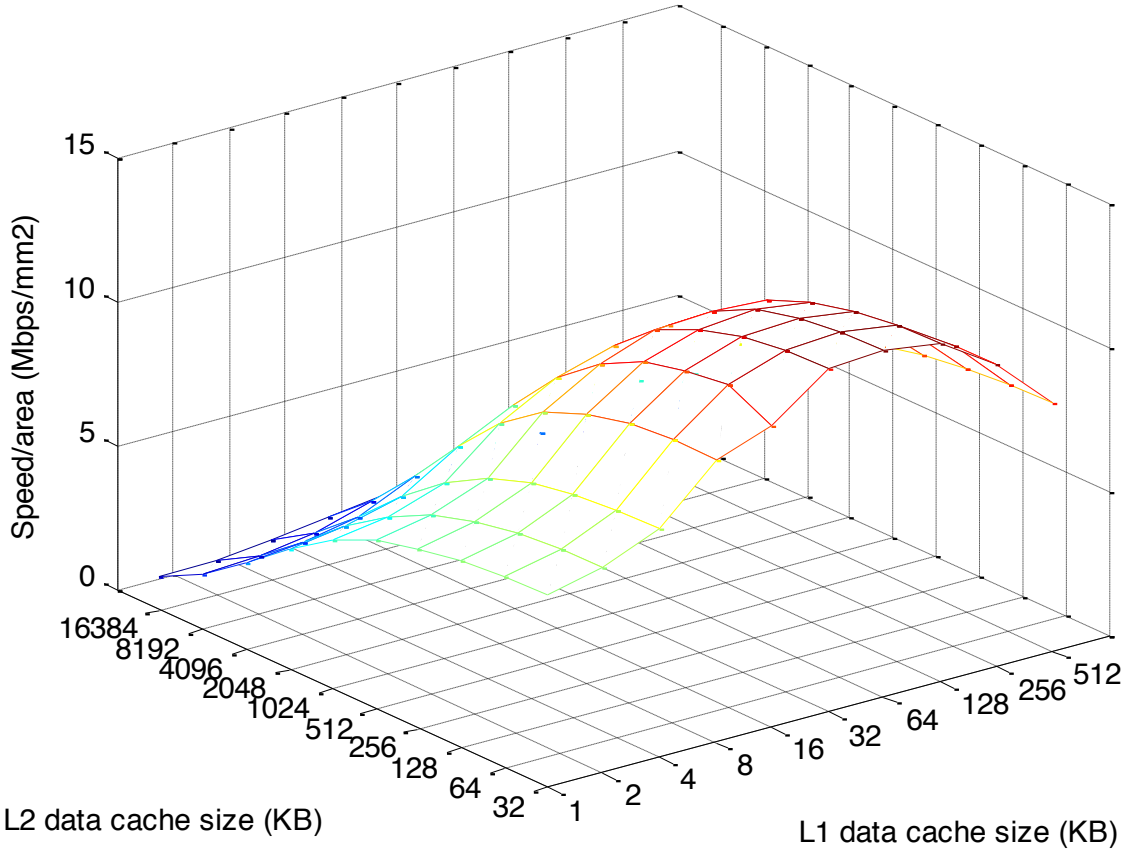
- Xilinx Virtex 5 (XC5VLX50)
- $speed = clk_{FPGA} \times stride \times 8\ bits$
- Smaller circuit can operate at higher frequency
- Hardware-based stride 4 implementation leads to the best results

Results from real hardware – Logic based solutions



- Different frequency: power vs. speed trade-off;
- $P = P_{static} + P_{dynamic} = P_{static} + \alpha C V^2 clk_{FPGA}$
- Should choose highest achievable clk_{FPGA} to get highest speed/power ratio.

Results from Processor Simulation – Cache



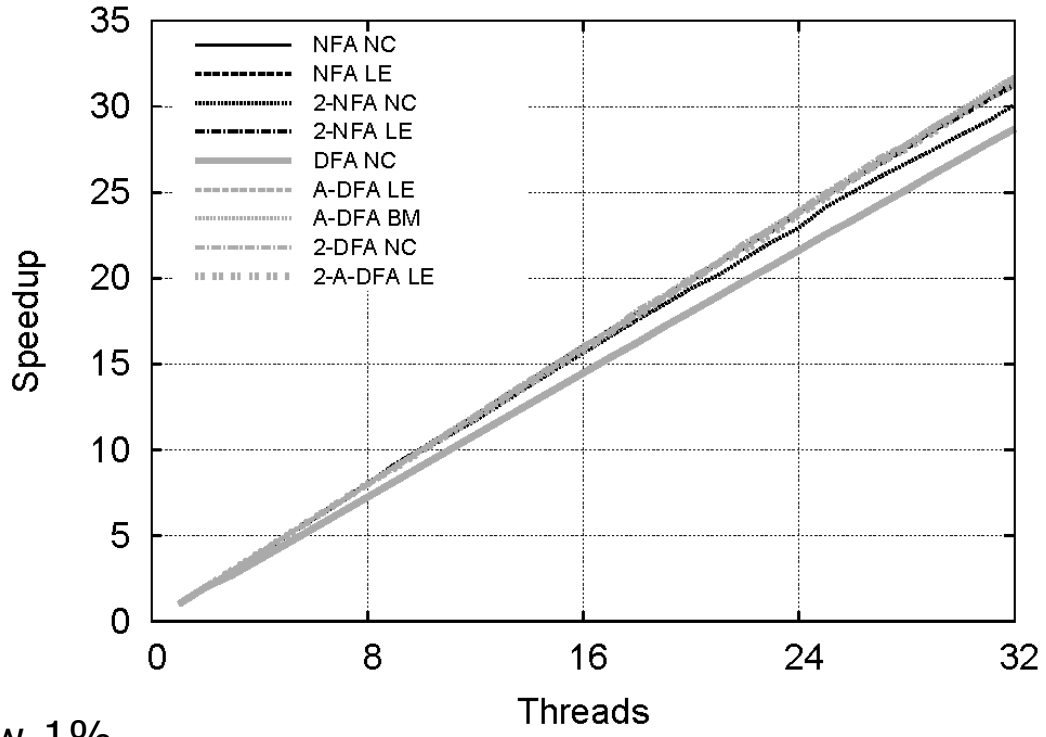
Best cache size for different configurations
Selected by maximum speed/area

	L1 size (KB)	L2 size (KB)
NFA	16	64
NFA linear	16	32
2-NFA	64	1024
2-NFA linear	64	512
DFA	64	128
D^2FA linear	64	64
D^2FA bitmap	32	64
2-DFA	128	4096
2-D^2FA linear	128	4096

- SimpleScalar simulator
- We select the best cache size based on speed/area.

Results from Processor Simulation – Memory bandwidth

	Utilization of bw_{mem} (%)	Max threads supported
NFA	0.25	81
NFA linear	0.17	120
2-NFA	0.38	52
2-NFA linear	0.23	88
DFA	0.17	118
D^{2}FA linear	0.04	454
D^{2}FA bitmap	0.04	480
2-DFA	0.26	76
2-D^{2}FA linear	0.20	101



Demonstration of scalability on Intel x86 CPU.

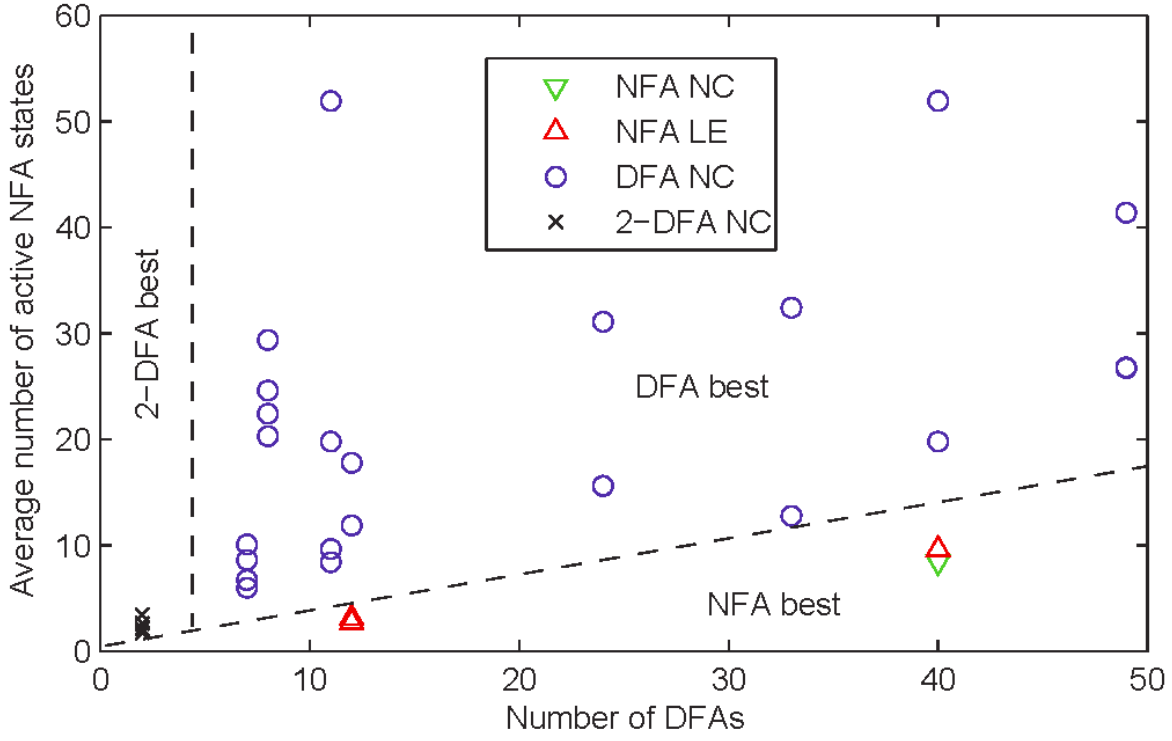
- Most cache miss rates are below 1%
- Low memory bandwidth utilization
- High parallelism is possible

Outline

- Introduction to regular expressions
- Design space exploration
- Results
- **Optimal Regular Expression Matching Configuration**
- Conclusion

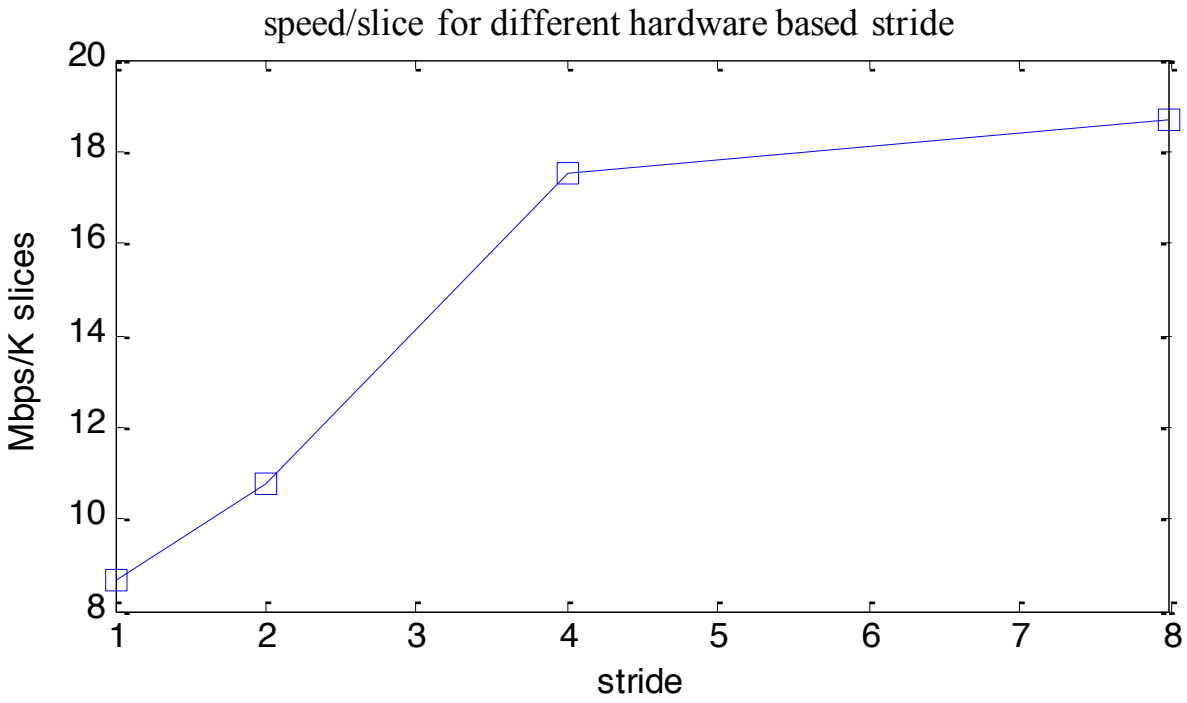
Optimal Memory-Based Configurations

- Select the optimal configuration by speed/area
- Parallel processing is allowed
- When $m_{NFA}/m_{DFA}<0.35$, an NFA-based implementation is preferable;
- Otherwise DFA-based implementations are preferable.
- For some simple rulesets, 2-DFA is faster than DFA.



Optimal Logic-Based Configurations

- Hardware-based multi-stride is the best.
- There seems to be a peak speed/slice value at higher stride, but this is beyond the chip's resource to validate.



Conclusion

- 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.
- In this work, we:
  - define the regular expression matching design space
  - propose a benchmark of configurations that evaluate the design space both on simulator and on real hardware.
  - present the analysis of ruleset to obtain optimal configuration.

Thank you!

Calibration