Hardware-accelerated regular expression matching with overlap handling on IBM PowerEN™ processor

Kubilay Atasu, Florian Doerfler, Jan van Lunteren, and Christoph Hagleitner
Outline

- Regex decompositions
- The IBM PowerEN™ PME
- Subexpression overlaps
- Compilation for the PME
- Experiments and results
Finite State Machine Processing & Regular Expressions

- One of the 13 dwarfs of computation: “inherently sequential”

- A fundamental part of network intrusion detection systems
  - attack signatures $\rightarrow$ regular expressions $\rightarrow$ state machines

- Alternative hardware architectures for regex matching:
  - nondeterministic finite state automata (NFA) $\rightarrow$ reconfigurable
  - deterministic finite state automata (DFA) $\rightarrow$ programmable
  - decomposed automata $\rightarrow$ combination of NFA and DFA

- This work focuses on compiling regexs to decomposed automata
  - implemented in IBM’s PowerEN™ Processor

Composition of State Machines

- Parallel composition of state machines
- Cascade composition of state machines
- Combining parallel composition and cascade composition
Decomposition of regular expressions

- Parallel decompositions: $(abc|def|ghi) \rightarrow \{abc, def, ghi\}$
  – Yu et al. ANCS 06, Rohrer et al. CODES+ISSS 09

- Cascade decompositions: $abc.*def.*ghi \rightarrow \{abc, def, ghi\}$
  – Kumar et al. ANCS 07, Smith et al. SIGCOMM 08
Outline

- Regex decompositions
- The IBM PowerEN™ PME
- Subexpression overlaps
- Compilation for the PME
- Experiments and results
The Pattern Matching Engine of IBM PowerEN™ Processor

- Millions of concurrent network sessions
- Small session state (fast context switches)
- Memory-based data structures (dynamic updates)
- Input-independent processing rates (DoS attacks)
- Wire-speed processing rates for large regex sets

The Pattern Matching Engine of IBM PowerEN™ Processor

- The throughput rate is a function of memory efficiency
  - memory accesses are irregular, cache misses are costly

- The throughput rate is a function of resource efficiency
  - complex regex sets do not fit into a single lane
  - need to stream the data through multiple lanes

- 4 lanes → 8 concurrent data streams
- 4 DFAs (B-FSMs) per scan lane
- 32 KB cache per DFA (512 KB total)
- 8 instructions per cycle per scan lane
- 9.2 Gb/s single stream throughput rate
- 73.6 Gb/s aggregate throughput rate
Outline

- Regex decompositions
- The IBM PowerEN™ PME
- Subexpression overlaps
- Compilation for the PME
- Experiments and results
Subexpression Overlaps

- Example: abc.*cde → {abc, cde}
  - the string abcde creates a false positive
  - We call this a “partial overlap”
  - abcabcde should still result in a match
Computing Partial Overlaps

- Need to define the language of subexpression overlaps
- Assuming that $R_1.*R_2$ is decomposed into $R_1$ and $R_2$
  - e.g., abc.*cde is decomposed into abc and cde
- The language of partial overlaps is
  \[ PO = (R_1.* \cap .* R_2) \backslash (R_1.* R_2) \]
  - e.g., abcde belongs to PO for the above example
- We can construct an automaton that recognizes PO
  - and check for false positives on the accepting states
Computing Full Overlaps

- Assume that abc.*xabcy is decomposed into abc, xabcy – abc is contained by xabcy: we call this a full overlap

- The language of full overlaps is
  \[ FO = (.*R_1.* \cap R_2) \setminus (R_1.* R_2) \]

- The language of all overlaps is
  \[ AO = .* (PO \cup FO) = (.*R_1.* \cap .* R_2) \setminus (.*R_1.* R_2) \]

- We can construct an automaton that recognizes AO – and check for false positives on the accepting states
Preventing False Positives

- The language AO is split into non-overlapping languages
  - EO(1), which contains exactly one instance of $R_1$
  - EO(2), which contains exactly two instances of $R_1$

Combined in a single DFA
Generalizing the Method

- We have covered the decomposition of \( R_1 \cdot R_2 \) into \( R_1 \) and \( R_2 \)
- Extension to \( R_1 \cdot R_2 \cdot R_3 \cdot \ldots \cdot R_K \) is very straightforward
  - First between \( R_1 \) and \( R_2 \), then between \( R_2 \) and \( R_3 \), ...
- Extension to \( R_1[^\backslash n] \cdot R_2[^\backslash n] \cdot R_3[^\backslash n] \cdot \ldots \cdot R_K \) is more complex
  - \([\backslash n]\) can appear in \( R_2, R_3, \ldots \)
- Works also for \(((R_1 \cdot R_2) \mid (R_3 \cdot R_4)) \cdot R_5 \cdot R_6\)
  - e.g., combine parallel and cascade decompositions
  - We get \( R_1 \cdot R_2 \cdot R_5 \cdot R_6 \) and \( R_3 \cdot R_4 \cdot R_5 \cdot R_6 \)
- When doesn’t it work?
  - If \( R_2 \) contains \( R_1^* \): need counters of unlimited size
Outline

- Regex decompositions
- The IBM PowerEN™ PME
- Subexpression overlaps
- Compilation for the PME
- Experiments and results
Compilation Flow

1. Decompose regexs
2. Overlap handling
3. \( L = \text{MinLanes} \)
4. \( L < \text{MaxLanes} \)
   - NO: EXIT
   - YES: Distribute regexs to \( L \) lanes
5. \( L = L + 1 \)
6. Allocate registers in each lane
7. Feasible allocation?
   - NO
   - YES: Store solution

Covered by this work

Rohrer, CODES+ISSS, 2009

Covered by this work
Microarchitectural Constraints of The PME

- Instructions per cycle
  - 4 default, 4 regular per lane
  - 1 default, 1 regular per DFA

- Size of the register file
  - 16 x 16-bit registers per lane

- Instruction types
  - set, reset, inc, shift, match

- Instruction formatting
  - condition register, target register
  - e.g., set v2 if v1, set v3 if v1
  - bitmasks to address specific bits
Multi-register Reset Instructions

- The need for multi-register reset instructions
  - consider the regex “abc[^\n]*def[^\n]*ghi[^\n]*jkl”
  - \[n\] resets the variables set by abc, def, ghi
Register Allocation Problem

- Problem Statement
  - Given a set of variables $v_1$ to $v_N$ ($N$ variables)
  - Given a set of registers $r_1$ to $r_M$ ($M$ registers)
  - Find a mapping between variables and registers
  - This simple formulation is a bin packing problem

- Diagram showing the allocation of variables to registers.
Register Allocation Constraints

- The size of the registers, and the number of registers
- The number of instructions that can be executed per cycle
- Only a single multi-reset instruction per register
  - assume “a” resets $v_1$ and $v_2$, “b” resets $v_2$ and $v_3$
  - $v_1, v_2, v_3$ cannot be packed in the same multi-reset area
Outline

- Regex decompositions
- The IBM PowerEN™ PME
- Subexpression overlaps
- Compilation for the PME
- Experiments and results
Experiments – Proprietary Regex Sets

- Parallel comp.: 4 DFAs per lane, no post-processor (Yu, ANCS 06)
- Cascade comp.: 1 DFA + 1 post-processor (Smith, SIGCOMM 08)
- PME: 4 DFAs + 1 post-processor per lane (8 lanes)
  - 4-lane solution (2 replicas) → 18.4 Gb/s throughput rate
  - 2-lane solution (4 replicas) → 36.8 Gb/s throughput rate
Experiments – Public Regex Sets

- L7 filter for Linux™ (left): ~100 regexs
  - 2-lane solution (4 replicas) → 36.8 Gb/s throughput rate

- Emerging Threats (right): ~1100 regexs (many duplicates)
  - 1-lane solution (8 replicas) → 73.6 Gb/s throughput rate
Summary and Conclusions

- Methods for handling sub-expression overlaps
  - correct decompositions $\rightarrow$ no false positives
  - the increase in the memory usage is not high

- Register allocation methods
  - minimize the number of resources used
  - maximize the aggregate throughput rate

- Comparisons with related work
  - more storage efficient than parallel composition
  - more resource efficient than cascade composition
Hardware-accelerated regular expression matching with overlap handling on IBM PowerEN™ processor

Kubilay Atasu, Florian Doerfler, Jan van Lunteren, and Christoph Hagleitner