GPP-Grep: High-Speed Regular Expression Processing Engine on General Purpose Processors

Victor C. Valgenti[1], Jatin Chhugani[2], Yan Sun[1], Nadathur Satish[2],
Min Sik Kim[1], Changkyu Kim[2], and Pradeep Dubey[2]

[1]School of EECS
Washington State University
{vvalgent,ysun,msk}@eecs.wsu.edu
[2]Parallel Computing Lab
Intel Corporation
jatin.chhugani@intel.com

Abstract. Deep Packet Inspection (DPI) serves as a major tool for Network Intrusion Detection Systems (NIDS) for matching datagram payloads to a set of known patterns that indicate suspicious or malicious behavior. Regular expressions offer rich context for describing these patterns. Unfortunately, large rule sets containing thousands of patterns coupled with high link-speeds leave most regular expression matching methods incapable of matching at real-time without specialized hardware.

We present GPP-grep, an NFA-based regular expression processing engine designed for maximum performance on General Purpose Processors. The primary contribution of GPP-grep is the utilization of the data-level parallelism available in modern CPUs to reduce the overhead incurred when tracking multiple states in NFA. In essence, we build and store the NFA in an architecture-friendly manner that exploits locality and then traverse the NFA maximizing the parallelism available and minimizing cache-misses and long-latency memory lookups. GPP-grep demonstrates 24–57× improvement in throughput over standard finite automata techniques on a set of up to 1200 regular-expressions culled from the NIDS Snort, and is within 1.3× of FPGA hardware-based techniques. GPP-grep achieves 2 Gbps throughput on a dual-socket commodity CPU system allowing for line-speed evaluation on *commodity* hardware.

1 Introduction

Pattern matching is a primary component of Network Intrusion Detection Systems (NIDS) that employ Deep Packet Inspection (DPI). DPI necessitates the comparison of every datagram payload against a set of known patterns. Fixed string patterns offer limited ability for expressing the complexities of modern network attacks, especially in the face of evasive techniques employed by attackers [15]. Regular expressions provide much richer context with which to design signatures enabling not only greater precision, but also greater resilience to evasive techniques. However, efficiently matching regular expressions can prove intractable, especially when faced with large sets of regular expressions combined with a high volume of traffic, and can result in near-incapacitation of NIDS when deployed in high-speed environments [12].

D. Balzarotti, S. Stolfo, and M. Cova (Eds.): RAID 2012, LNCS 7462, pp. 334–353, 2012.

To promote efficient regular expression matching the set of regular expressions are reduced to Non-deterministic Finite Automata (NFA) or Deterministic Finite Automata (DFA). While NFA provide for very compact memory utilization, they suffer in throughput as multiple active states must be maintained as all possible paths through the NFA are traversed. DFA exhibit faster throughput, as only one active state is ever needed, but the number of states in the automata can grow exponentially and require excessive amounts of system memory. Current automata solutions such as eXtended Finite Automata (XFA) [24], Hybrid Automata [3], Delayed input DFA (D^2FA) [16], and Ordered Binary Decision Diagrams [30] improve the memory and time efficiency of regular expression matching. Similarly, hardware techniques employing Field Programmable Gate Arrays (FPGA) [19], Graphics Processor Units (GPU) [26,5], or Cell processors [21] utilize specialized hardware to improve matching. However, specialized hardware can prove expensive and unmanageable while purely automata-based techniques do not necessarily exploit the parallelism already embedded in current multi-core and many-core processors.

In this paper, we present **GPP-grep**, a high-speed, NFA-based, regular expression engine for General Purpose Processors (GPP). GPP-grep exploits thread-level and data-level parallelism to achieve gigabit processing rates. Further, GPP-grep utilizes specialized NFA construction and storage to both reduce the total number of active states and exploit locality during NFA traversals. The primary contribution of GPP-grep stems from its ability to merge efficient NFA construction, storage, and traversal techniques with a more complete use of GPP processing power to arrive at a performance of up to $57\times$ faster than traditional NFA engines and within $1.3\times$ of a hard-wired FPGA-based NFA engine (on a 12-core CPU system). With 1,200 real NIDS regular expressions GPP-grep achieves real-time processing rates of 2 Gbps on *commodity* hardware.

2 Related Work

Regular expressions provide signature creators a wide context within which to describe dynamic patterns such as those occurring in polymorphic worms or customized attacks [15]. Unfortunately, matching thousands of regular expressions against the payloads of many thousands of packets-per-second can result in total failure of a NIDS in multi-gigabit environments [12]. DFA provide a fast software implementation for matching but suffer from state explosion when ambiguous characters, such as *wild-card* characters, are used in expressions. Such *wild-card* characters result in an exponential increase in the number of states required for the DFA which, in turn, requires much more memory to store. Since NIDS may employ thousands of regular expressions, nearly 1,600 distinct regular expressions for the default-enabled rules of the Sourcefire Vulnerability Research Team (VRT) Snort rule-set for August 11, 2011 [27], and since these regular expression are complex expressions with an average of six *wild-card* characters, DFA can grow too large to reside in main memory. Conversely, NFA have very compact representations in terms of memory, but require wider bandwidth to traverse as all possible paths through the NFA must be explored simultaneously. Traditionally, NFA are not considered a viable solution for NIDS, nor for regular expression matching, as the single state transitions of DFA appear to offer the best chance at throughput.

However, the growth in number of regular expressions employed by NIDS, as well as the propensity for these regular expressions to use *wild-card* characters, has begun to make DFA matching infeasible as the resultant DFA are simply too large. Compression [1], rule rewriting [31], and add-on data [20] can create smaller DFA, but can also result in added overhead and processing of the DFA.

Much research has sought to improve the efficiency of automata. Smith et al. present XFA [24] which augment Finite State Automata (FSA) with added variables to track state during processing. The added state information serves to provide the FSA with enough hints on processing data that it can both perform faster and in less space, on average, than vanilla FSA. Becchi et al. present Hybrid Finite Automata [3,4] which employ a small, head, DFA for the most common states (closer to the root). However, for matching that extends deep into the automata, tail NFA are employed to succinctly represent these deeper and less traveled regions. This hybrid finite automata demonstrated smaller size than a comparable DFA, but with a much better cache hit ratio and faster processing. Kumar et al. [16,17] introduced Delayed input Deterministic Finite Automata (D^2FA) and Content addressed Delayed input Deterministic Finite Automata (CD^2FA). D^2FA essentially combines identical transitions from multiple states to reduce the total number of states and, ultimately, the size of the Finite Automata. CD^2FA use content labels rather than state identifiers that allow skipping default transitions in certain cases. The end results were much smaller than normal DFA that achieved roughly the same memory bandwidth. Yang et al. [30] adopted an approach more closely aligned with GPP-grep in that they sought to improve the throughput of NFA. They employed Ordered Binary Decision Diagrams to maintain the space-efficiency of typical NFA representations while greatly improving the NFA traversal throughput. Also similar to GPP-grep, Shenoy et al [23] attempt to make the storage of state in Finite State Machines more efficient.

Another common tactic is to take advantage of *specialized* high-compute platforms to speed up regular expression matching. Mitra et al. [19] compile PCRE op-codes to Very High speed integrated circuit Description Language (VHDL) so that the matching can be executed on a Field Programmable Gate Array (FPGA). This allows the expression matching to occur in parallel across multiple NFA. The end result is a significant increase in throughput. Other approaches include Smith et al. who map DFA/XFA to Graphics Processing Units (GPU) [26], iNFAnt which also maps NFA to GPUs and provides efficient traversal algorithms [5], the use of the Cell processor as illustrated by Scarpazza et al. [21], and Meiners et al. [18] who employ TCAM to compactly encode multiple DFA and achieve high throughput.

GPP-grep differs from these approaches by creating and implementing an efficient automaton in a manner that fully utilizes the parallelism available in modern multi-core and many-core processors. This presents several advantages for GPP-grep. First, it mitigates the need for *specialized* high-compute platforms by maximizing the full potential of general purpose processors. Thus, GPP-grep can achieve performance benchmarks comparable to hardware implementations on a low-cost, ubiquitous piece of hardware. Secondly, the architecture-friendly layout for automata used in GPP-grep can extend to other approaches, such as those mentioned earlier, to arrive at improved performance. Finally, GPP-grep itself could benefit from the other approaches mentioned earlier thus

potentially allowing for complimentary improvement through the combination of different approaches. Ultimately, GPP-grep attempts to merge both finite automata considerations with general purpose processor considerations to arrive at a regular expression matching engine that demonstrates improvements far beyond what either tactic might manage alone.

3 Modern Architectures

Exposing instruction-level parallelism is critical to utilize the multiple functional units within each processor core. This requires unrolling/software pipelining as well as proper instruction scheduling to expose independent instructions for simultaneous execution. In order to utilize all integrated cores, the application must expose thread-level parallelism. For regular expression matching, parallelization can be done across multiple input packets, with each processor executing matches on different packets. Modern processors also have wide vector Single Instruction, Multiple Data (SIMD) units that can execute instructions on many data items in parallel. For regular expression matching using NFA we utilize SIMD parallelism to perform the next state transitions of multiple active states in parallel. These techniques are further described in Section 5.3.

Instructions with high latency can lead to low utilization of functional units since they block the execution of dependent instructions. This is typically due to long latency memory accesses which can be reduced if the working set of the application (the data size for which the number of cache hits is 90%) fits in the last-level cache of the processor architecture. For regular expression matching the core traversal algorithm involves accesses to distinct memory locations which will ordinarily not be present in cache. In Section 5.2, we rearrange the nodes of the automaton to minimize the impact of cache misses. In addition to cache misses, misses to an auxiliary structure called the Translation Lookaside Buffer (TLB), which is used to perform the conversion from virtual to physical memory addresses prior to each memory access, can also result in significant performance degradation. This is also reduced using our hierarchical blocking scheme described in Section 5.2.

Finally, misses in cache also result in increased use of memory bandwidth. While the compute resources in modern architectures have increased rapidly, memory bandwidth is on a slower curve. To minimize bandwidth utilization, we attempt to ensure that every piece of data brought into cache in the form of cache lines is fully utilized before being evicted. This is ensured by the cache line blocking technique that we describe in Section 5.2.

4 Efficient NFA Construction

The first step in matching regular expressions is to build a finite automaton from a set of given regular expressions. Once construction of the finite automaton is complete, strings of symbols are applied to the automaton as an input until a final (accept) state is reached (a match) or the input is exhausted (no match).

An NFA may require as many state transitions per symbol as the total number of states in the NFA plus the possible addition of an epsilon transition. An epsilon transition allows the advancement to a new state without consuming an input symbol. This

creates a time complexity for each symbol of $O(m)$ where m defines the total number of states in the NFA. For single-byte symbols, this translates to as many as 256 transitions per state (excluding epsilon transitions). The benefit, however, is that the NFA is very efficient in terms of space as only a single state exists for each possible state from the generating alphabet. On the other hand, a DFA requires only one state transition per symbol, but needs a much larger amount of memory to adequately map all possible traversals through the finite automaton. This burden on memory only grows as the number of complex regular expression components, such as Kleene stars, that directly lead to state explosion continue to increase both in frequency (number of rules containing complex components) and density (number of components per expression) with an average of six such components per regular expression in the Snort VRT [27] rule set.

4.1 Challenges in NFA Construction

In order to design efficient NFA previous approaches emphasized the generic reduction of states and transitions. However, reducing the number of "active" states is more important than simply reducing total states since the CPU has to deal with all the active states for each symbol in the input string. When converting regular expressions into an NFA, Thompson's algorithm [28] is often adopted. In Thompson's algorithm, ε is used to represent an empty string and is termed the epsilon transition. Unfortunately, NFA generated by Thompson's algorithm have two major drawbacks: first, there are many redundant states and transitions in the NFA, and second, there are many redundant epsilon transitions in the NFA that can significantly increase the number of active states for a traversal. In order to reduce active states in NFA we can merge some states. Minimizing NFA is a hard problem [13] thus it is necessary to develop heuristic methods to control minimization in order to make the problem tractable.

4.2 Reducing the Number of Active States

An NFA is defined by a quintuple $A = (S, \Sigma, \delta, s_0, F)$, where S is a finite set of states, Σ is the alphabet, δ is the transition function, s_0 is the initial state and F is the set of final states.

Given an NFA, we reduce the number of active states by combining "mergeable" states into one. Formally, given an NFA $A = (S, \Sigma, \delta, s_0, F)$, let p and q be two different states in S. Then, by combining p and q, we obtain another NFA, $A' = (S', \Sigma, \delta', s_0, F')$, that satisfies the following conditions:

$$S' = S - \{q\}, \tag{1}$$

$$\delta'(s, \omega) = \begin{cases} \delta(p, \omega) \cup \delta(q, \omega) & \texttt{if } s = p \\ \delta(s, \omega) & \texttt{otherwise,} \end{cases} \tag{2}$$

and

$$F' = \begin{cases} (F - \{q\}) \cup \{p\} & \texttt{if } q \in F \\ F & \texttt{otherwise.} \end{cases} \tag{3}$$

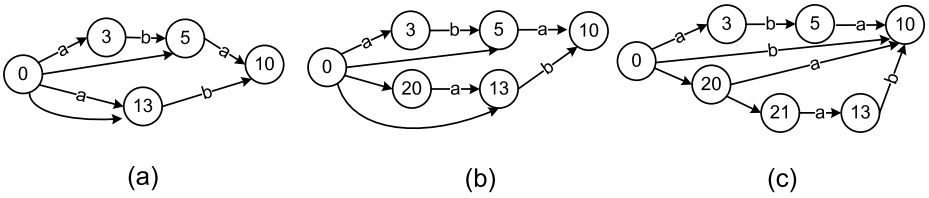


Fig. 1. Various NFA types for the RE ((ab|ε)a|(a|ε)b); (a) MM (b) SM (c) SS

Let $L(A)$ be the language accepted by A. Then states p and q are *mergeable* if and only if $L(A) = L(A')$. In fact, there exists a weaker condition for two states to be mergeable [7].

Definition 1. *The* left language *of a state s in an automaton A is* $\overleftarrow{L}^A(s) = \{\omega \in \Sigma^* \mid s \in \delta(s_0, \omega)\}$.

Definition 2. *The* right language *of a state s in an automaton A is* $\overrightarrow{L}^A(s) = \{\omega \in \Sigma^* \mid \delta(s, \omega) \cap F \neq \varnothing\}$.

Note the generalized usage of δ. It represents a set of NFA states reachable by the string ω and any number of ε transitions preceding ω, following ω, and between any successive symbols in ω. Additionally, [7] prove the following proposition.

Proposition 1. *Two states, p and q, in an automaton A are mergeable if and only if* $\overleftarrow{L}^A(p) = \overleftarrow{L}^A(q)$ *or* $\overrightarrow{L}^A(p) = \overrightarrow{L}^A(q)$.

Unfortunately, testing the condition in Proposition 1 is NP-hard, requiring global knowledge [6]. However, because we build an NFA from a regular expression, which is a linear representation of the NFA, we do not need to deal with an arbitrary NFA. Based on this observation, we propose a sufficient mergeability condition, and design a novel heuristic algorithm to identify most mergeable states during a single-pass conversion from a regular expression to an NFA.

Proposition 2. *Two states, p and q, in an automaton $A = (S, \Sigma, \delta, s_0, F)$ are mergeable if* $\delta(p, \varepsilon) = \{q\}$ *and either (i)* $\bigcup_{c \in \Sigma} \delta(p, c) = \varnothing$ *or (ii)* $\{(s, c) \mid c \in \Sigma \cup \{\varepsilon\} \wedge q \in \delta(s, c)\} = \{(p, \varepsilon)\}$.

Proof. Suppose p and q satisfy the condition in the proposition. We prove (i) and (ii) separately.

(i) Because the epsilon transition is the only outgoing transition of p, we have $\delta(p, \omega) = \bigcup_{t \in \delta(p, \varepsilon)} \delta(t, \omega)$. Since $\delta(p, \varepsilon) = \{q\}$, the equation becomes $\delta(p, \omega) = \delta(q, \omega)$. Therefore, $\overrightarrow{L}^A(p) = \overrightarrow{L}^A(q)$.

(ii) Because the epsilon transition from p to q is the only incoming transition of q, for any ω such that $q \in \delta(s_0, \omega)$, we get $p \in \delta(s_0, \omega)$ by removing the last epsilon transition. Similarly, for any ω such that $p \in \delta(s_0, \omega)$, we get $q \in \delta(s_0, \omega)$ by adding another epsilon transition at the end. Therefore, $\overleftarrow{L}^A(p) = \overleftarrow{L}^A(q)$.

Thus, by Proposition 1, p and q are mergeable. □

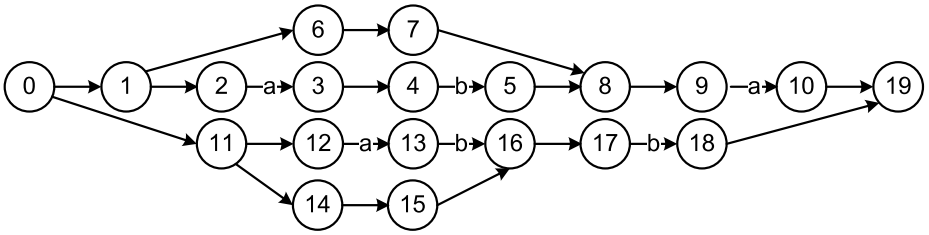


Fig. 2. NFA produced by the Thompson Algorithm for the RE ((ab|ε)a|(a|ε)b)

Note that to test conditions in Proposition 2, we only need the local knowledge of outgoing transitions for p and incoming transitions for q. To demonstrate the effectiveness of our state merging, we illustrate Thompson's NFA for $(ab|)a|(a|)b$ in Figure 2, and the result of state merging in Figure 1(a), which has significantly fewer active states than Thompson's NFA.

4.3 Building an Efficient NFA

After merging states, we will get an NFA with each state having multiple transitions, even for the same symbol, and multiple epsilon transitions (a multi-multi NFA, or MM). Such an NFA requires a dynamic data structure, such as a linked list, to maintain all outgoing transitions for each input symbol. A simpler way is to force each state to have at most one transition per symbol and at most one epsilon transition (a single-single NFA, or SS), and implement it as an array. Essentially, we wish to reduce an MM to an SS. Then the whole transition table of a state can be stored as a single entity with 257 elements (256 input symbols and one epsilon transition). To achieve at most one transition per symbol, we have to introduce new epsilon transitions to distribute multiple transitions for the same symbol over multiple states. In order to have room to add such new epsilon transitions, we first remove epsilon transitions from MM using Algorithm 1. For each epsilon transition from p to q followed by another transition from q to r, Algorithm 1 adds a shortcut from p to r, and eventually removes the epsilon transition.

After removing epsilon transitions, we serialize all outgoing transitions for the same symbol using new epsilon transitions. The pseudocode is shown in Algorithm 2. Note that every state on the chain of epsilon transitions created by this algorithm has exactly one incoming epsilon transition and one outgoing transition. In this way, we can guarantee that each state has at most one transition per input symbol and at most one epsilon transition. Given an MM NFA as shown in Figure 1(a), its SS counterpart is shown in Figure 1(c). We can use a similar algorithm to build an SM (a single-multi NFA), as shown in Figure 1(b). The only difference is to connect a new state n in Algorithm 2 to s directly using an epsilon transition instead of inserting it into the epsilon chain.

Overall, the algorithm to optimize an NFA created from a given set of regular expressions requires three steps: first, to create a compact NFA by merging states; second, to remove all epsilons from the NFA; and third, to force a single-transition per input symbol by creating an epsilon chain. This simplifies the memory representation of an NFA and its traversal algorithm, as we explain in Section 5.

Algorithm 1. Remove-Epsilons(State *s*)

```
if s has been marked then
    return
end if
mark s visited
for all e such that e is an epsilon transition out of s do
    d ← destination of e
    if d has not yet been marked AND d ≠ s then
        add transitions in d AND NOT in s to the transitions of s
        if d is accepting then
            set s to accept
        end if
    end if
    remove e
end for
for all t such that t is a transition out of s do
    d ← destination of t
    Remove-Epsilons(d)
end for
```

5 Efficient Layout and Traversal

In this section, we describe our architecture-friendly NFA layout and the efficient traversal algorithm. These optimizations are aimed at reducing the traversal cost and the resultant working set (the most frequently accessed data), thereby resulting in increased NFA processing throughput.

5.1 Motivation

NFA are typically stored using an adjacency list representation, with each state storing its outgoing *symbol* and ε-transitions. The NFA traversal algorithm maintains a list of *active states*, labeled *AS*, initialized to the `Start` state(s). For each input symbol α, the traversal consists of the following two steps:

Step 1: Computing the list of neighboring states (*NS*) for the symbol α for all states in *AS*.

Step 2: Computing the ε-closure[1] of all the states in *NS*, and assigning the resultant states to *AS*.

In the case where *AS* consists of the `End` state(s), a *match* is found, and the NFA execution may continue until *AS* contains no `End` state(s) (to find the longest sub-string match). On the other hand, in the case where all the input symbols are processed without ever reaching `End` state(s), the input stream does not match any regular expression in the set of regular expressions.

[1] By definition, ε-closure of any state *A* consists of the state *A* and all the states reachable using ε-transitions from ε-neighbors of *A*.

Algorithm 2. Force-Single-Transition-Per-Symbol (State s)

```
if s has been marked then
   return
end if
mark s visited
for all c such that c ∈ Σ do
   if there are multiple transitions for c then
      for all t such that t is a transition s -c-> d except the first do
         remove t
         insert a new state n at the head of epsilon chain
         add a transition n -c-> d
      end for
   end if
end for
for all t such that t is a transition out of s do
   d ← destination of transition
   Force-Single-Transition-Per-Symbol(d)
end for
```

The runtime of the traversal algorithm is dependent on the following 3 factors: (1) For each *active state* in `AS`, the time taken to lookup the neighboring states in Step 1. (2) The time taken to lookup the ε-neighbors (and the subsequent ε-closure) for each state in `NS` in Step 2. (3) Maintaining a unique list of states in `AS` and `NS` to avoid redundant computation.

Typical implementations use either linear or log-time complexity algorithms [2] to lookup neighbors and maintain unique lists. One primary reason has been the focus on reducing the total memory footprint and using compact representation at the expense of increased traversal cost. In contrast, our layout aims to exploit the existence of hardware caches and reduce the actual working set to make it fit in the cache, thereby minimizing the access cost and thus reducing the instructions required for the traversal. This layout makes our traversal algorithm compute-bound (rather than latency or bandwidth-bound), and also provides an opportunity to exploit the Single Instruction, Multiple Data (SIMD) execution units to further improve run-times.

5.2 Architecture-Friendly Layout

For the remainder of this section, let $\mathcal{S}$ denote the set of symbols, and $|\mathcal{S}|$ the cardinality of $\mathcal{S}$ (e.g. 256 for 8-bit input symbols—the most common case). Furthermore, let $\mathcal{M}$ denote the set of states (also referred to as nodes). We define the depth of any node as the shortest distance (in terms of number of edges) traversed from the `Start` state to that node. By definition, the `Start` state has a depth of *zero*.

On modern architectures, the data transfers (from main memory to caches, and within caches) are performed at the granularity of cache lines (typically 64 bytes or longer). In order to reduce memory accesses during traversal, it is important to reduce the number of accessed cache-lines, which implies storing temporally coherent data in proximity. Note that the NFA traversal involves neighborhood queries for all the active states for a

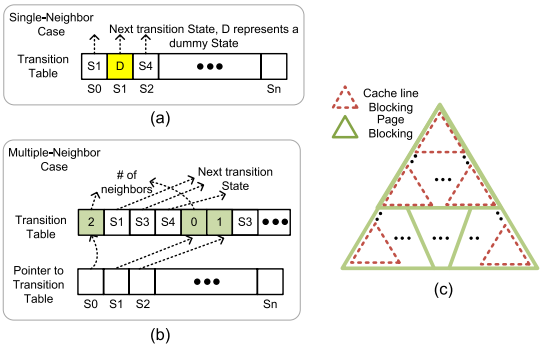


Fig. 3. (a) Layout for single-neighbor case. (b) Layout for multiple-neighbor case. (c) 2-Level hierarchical blocking of NFA nodes.

specific input symbol under consideration. Hence we need to store the transitions from various states for any given symbol close to each other. We therefore cluster all the transitions into ($|\mathcal{S}|$+1) groups ($|\mathcal{S}|$ for symbols and one for ε transitions). We now describe our layout for these transitions for any particular symbol. For ease of explanation, the remainder of the section uses the term symbol to include both input symbols and the ε transition.

We adopt different layout schemes dependent on whether we wish to construct a Single-neighbor (`S`) NFA where all the states have ≤ 1 neighbor or a Multiple-neighbor (`M`) NFA where states may have many neighbors. These NFA represent different design choices and are constructed as explained in Section 4.3. Consider the `S` case, and the symbol α. It is indeed possible for some states to have no transition for α. However, in the Single-neighbor NFA we must provide exactly one neighbor for every state in our layout in order to provide for a predictable data structure as illustrated in Figure 3(a). Thus, our layout employs a `Dummy` state for all non-existent transitions. While this increases the total number of states for the NFA by one, it offers an efficient method for determining no transitions.

Now consider the Multiple-neighbor (`M`) case. Different states may have varying number of neighbors, and hence we need to store a *count* for the number of neighbors. Furthermore, in order to reduce the number of accessed cache-lines, we need to store the *count* close to the state IDs. We therefore store the *count* followed subsequently by the neighboring IDs. This representation requires storing a pointer to the address of the memory location storing *count* (Figure 3(b)).

Our NFA layout described so far stores the states in increasing order, starting from state ID 0. However, the traversal pattern follows a specific order—for any given state, it accesses its neighboring (outgoing) state IDs. To improve cache locality, we need to store all the neighbors of a node close to each other. Doing so for all depths results in a breadth-first storage. However, this increases the storage distance between any node and its neighbors at larger depths. For input streams matching the given NFA, the depth of active states increases as we traverse through the inputs, thereby resulting in memory accesses that are separated by increasing distances. As described in Section 3, memory is laid out as pages and, for memory accesses offset by more than the page size, each

access would result in a TLB miss which would increase memory access latency. In the worst case, we may incur a TLB miss for each symbol of the input stream. We propose a hierarchical blocking scheme that reorders the state IDs and reduces both the number of cache and TLB misses simultaneously.

Hierarchical 2-Level Blocking. The aim of hierarchical blocking is to *partition* the nodes into groups that fit entirely in a memory page (typical size of 4KB). Furthermore, we need to rearrange nodes within *every page* so that a node and its neighbors are stored close to each other—thereby fully exploiting a cache-line. We refer to this as our hierarchical blocking scheme. Assuming a 32-bit representation for each state, we can fit 16 states in a cache-line and 1K nodes in a 4KB memory page. We perform a global reordering of the nodes, and apply the same permutation to the states for all the symbols.

We start with the root node (`Start` state), and perform a depth-first traversal of the graph and assign depth values to all the nodes. Furthermore, we maintain a list of nodes that have not been assigned to any cluster—initialized to all the states in the NFA. We begin clustering by including the `Start` state and all its neighbors (irrespective of the symbol). We make progress by picking one of the unassigned nodes at the lowest depth and including it and its unassigned neighbors in the cluster. We continue the process until the threshold for the number of nodes in a page is reached. We then start clustering for the next page by either continuing with the neighbors of the node just being considered or starting with a new unassigned node. The process is terminated when all the NFA states have been assigned to any of the clusters.

We now describe our scheme for performing cache-line blocking within each cluster. We maintain a list of nodes that have not been assigned an index within the cluster. We start with the node at the lowest depth, and consider its neighboring states in the cluster. In the case where the number of unassigned neighbors is greater than 16 (the maximum number of neighbors within a cache-line), we assign the neighbors contiguously. We continue with the remaining neighbors, and assign them in a similar fashion. If the number of unassigned neighbors is less than 16, we fill up the cache-line partially and then continue with the process by selecting the unassigned node at the lowest depth. The process is carried out until all the nodes have been assigned an index (Figure 3(c)).

For any distribution of the symbols in the input stream, our hierarchical blocking scheme aims to reduce the average number of cache- and TLB-misses. In practice, the NFA traversal spends most of its time in the first few levels of the NFA, which are clustered together by our algorithm. Hence we have very few cache- and TLB-misses as shown in Section 6. Using our blocking scheme, we obtain a large performance improvement (of around 2.7×) for large NFA. Note that we reorder the nodes for each of the MM, SM and SS types of NFA as a pre-process step.

5.3 Traversal Algorithm

We first explain our technique to maintain unique lists in $O(1)$ time for our NFA layout, followed by the complete traversal algorithm for different NFA types. We maintain a time stamp (referred to as τ, and initialized to *zero*) for the simulation that gets updated for each step of the NFA traversal. Furthermore, we maintain an array (referred to as `TimeStamp`), that stores one time stamp value per state—representing the time stamp

for the most recent access. This requires an additional two accesses per state (to check and potentially update its time stamp), but helps maintain unique states without the significant overhead due to sorting or linear searches. For each input symbol, τ gets incremented by one for the symbol-transitions and again incremented for the ε transitions. Depending on the granularity of the time stamp, we need to reset the `TimeStamp` array for all the states once the timer τ truncates to zero. Assuming a 32-bit timer, this reset process needs to be performed after traversing through 2 MB of input packets, rendering the amortized cost to be negligible.

We first describe the traversal algorithm for the Single-neighbor (`S`) case. Let α be the input symbol, and τ represent the current time step. Before starting the execution, `TimeStamp[Dummy]` $\leftarrow \tau$. Consider the Step 1 of the traversal (Section 5.1). The traversal consists of:

(a) Loading the next active state (S_i) in `AS`, followed by
(b) Looking up its α neighbor (say $S^{\alpha}{}_i$),
(c) Looking up the corresponding time stamp value (τ') of the neighbor (`TimeStamp`[$S^{\alpha}{}_i$]).
(d) Comparing τ' and τ, and
(e) In case $\tau' \neq \tau$, `TimeStamp`[$S^{\alpha}{}_i$] $\leftarrow \tau$ and append $S^{\alpha}{}_i$ to `NS` (since $\tau' = \tau$ implies that $S^{\alpha}{}_i \in$ `NS`).

The same process is carried out for Step 2 (for `S` case).

In the case of Multiple-neighbors (`M`), we replace the above steps by the following computations. We first (i) compute the address of the memory location containing the counter of the number of neighbors, and (ii) lookup the counter value. The rest of the process involves iterating over all the neighbors, and performing steps (a)–(e).

The `M` case clearly involves the added *overhead* of address computation and accessing the counter value prior to starting the efficient process of neighbor lookup and appending entries the list of unique active states. Further, it carries the additional *overhead* of loop computation and checking for termination. For a small number of neighbors this overhead contributes substantially to execution time, but gets amortized for large numbers of neighbors.

Improving ILP (Instruction-Level Parallelism). During the NFA traversal, a node may access cache-lines that are not resident in the cache. This scenario arises as we traverse deeper into the NFA graph. Such accesses may incur long latencies and stall the execution pipeline. In order to reduce the impact of such latencies on the execution time, we issue *software prefetches* in advance, before actually accessing such cache-lines. This reduces (and in most cases eliminates) the memory latency stalls. We modify our traversal algorithm as follows. During the execution of Step 1, we look at the subsequent input symbol (say β). As we identify and add states to the *NS*, we also issue software prefetches for all the memory locations storing the neighbors of these states (for symbol β and ε neighbors). This process is carried out during the execution of Step 2. Note that we can issue prefetches for all memory accesses, except for the ones that arise from the ε-transitions of the ε-neighbors of states in *NS*.

In addition to software prefetches, we also perform *loop unrolling* while iterating over the active states. Since steps (a), (b), and (c) above are completely independent for

the various active states, these instructions increase the amount of parallel instructions available for the processor scheduler to improve the IPC (Instructions Per Cycle), and hence reduce the effective amount of execution cycles.

Exploiting Thread- and Data-Level Parallelism. We exploit the available multiple cores on current CPUs by dividing the input packets among the cores. This requires keeping a separate copy of the `TimeStamp`, `AS` and `NS` arrays. Since the cores perform *independent* traversals, we obtain near-linear scaling with number of cores.

In order to further reduce the executed instructions, we take advantage of the SIMD execution units available on modern computing units. We exploit SIMD by *operating on multiple active states* simultaneously. Modern CPUs have a SIMD width of 128-bits, and hence we operate on *four* active states simultaneously. Although there exist schemes that exploit SIMD by performing traversal on multiple input streams simultaneously, they do not achieve any speedup on CPUs [8].

For the Single-neighbor `S` case, performing step (a) in SIMD involves doing an *aligned vector load* from the memory into a vector register, while step (b) involves *gathering* the α neighbors for four different states (hence four memory addresses) into a register. Similarly step (c) is another *gather* operation from the `TimeStamp` array, while step(d) is a vector compare (equality between two registers). Finally step (e) needs to be performed only for the SIMD lanes that correspond to states which failed the equality test. Two operations are performed for the same—a masked *scatter* operation to update the `TimeStamp` array, and a *packed vector store* [22] to the `NS` array. Similarly for the `M` case, steps (i) and (ii) translate to vector gather instructions that gather the addresses of the memory locations storing the counter value, followed by another gather to obtain the count values themselves. The remaining operations can utilize SIMD in a similar fashion to `S`.

The speedup due to SIMD is governed by the following 3 factors: (i) efficient hardware support for load, compare, gather, scatter and packed-store vector instructions, (ii) the number of active states that are available for SIMDfication, and (iii) the results of comparison in step (d) that compute the number of elements that need to be scattered and pack-stored.

The current generation of CPUs do not have efficient support for gather, scatter and packed-store instructions. We therefore emulated them using scalar instructions. Hardware support for these instructions would have a greater impact on reduction in runtime. We provide performance numbers for SIMD speedup in Figure 4(a) in Section 6.1. Finally, we note that a kernel implementation reduces the overhead of context switches and user-space copies and serves to further improve performance. As such, we have implemented GPP-grep in the Linux 2.6 kernel.

5.4 Analytical Model

To predict performance we create an analytical model by computing the total number of ops[2] executed during the traversal (for an active state and the input symbol). The corresponding number of cycles depends on memory access patterns. Our layout is also

[2] 1 op implies 1 operation or 1 executed instruction.

optimized for efficient memory access and achieves close to maximal IPC (detailed results in Section 6.2). Thus with appropriate NFA layout optimizations, our analytical model may also project the number of executed cycles.

First consider the Single-neighbor (S) case. For each active state the total cost of the Steps one and two, as outline in Section 5.1, is the sum of the sub-steps (a)–(e) and is termed $\text{cost}_{\text{step1/2}}$. However, step (e) is only executed with certain probability, dependent on the input stream and NFA characteristics. Let ρ denote the corresponding probability. Furthermore, we also issue *software prefetch* instructions to improve the IPC. This adds to the instruction overhead too. Let the cost of prefetches be denoted by $\text{cost}_{\text{pref}}$. Let the number of symbol neighbors be $\mathcal{N}_{\text{symbol}}$ and the ε-closure consist of $\mathcal{N}_{\varepsilon}$ elements. Therefore $\text{cost}_{\text{step1/2}} = \text{cost}_{\text{pref}} + \text{cost}_{(a)} + \text{cost}_{(b)} + \text{cost}_{(c)} + \text{cost}_{(d)} + \rho\text{cost}_{(e)}$, and hence $\text{cost}_{\text{S}} = (\mathcal{N}_{\text{symbol}} + \mathcal{N}_{\varepsilon})\ \text{cost}_{\text{step1/2}}$. For the Multiple-neighbor (M) case, we also need to include the overhead of steps (i) and (ii) (Section 5.3). Hence $\text{cost}_{\text{M}} = (\mathcal{N}_{\text{symbol}} + \mathcal{N}_{\varepsilon})\ \text{cost}_{\text{step1/2}} + \text{cost}_{\text{step(i)}} + \text{cost}_{\text{step(ii)}}$.

For any given architecture, the costs for each of the above terms is known, and can be plugged in to get an estimate of the number of executed ops. We provide data and the projected results by our model in Figure 4(b) in Section 6. We believe the model serves as a metric to compare the performances of different architectures for the NFA traversal with different NFA representations.

6 Experimental Evaluation

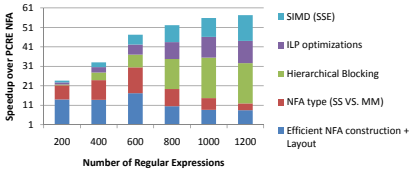
We now evaluate the performance of GPP-grep on the Intel Xeon DP Westmere-EP X5680 CPU. Our CPU platform has 2 sockets with a total of 12 cores running at 3.33 GHz. Our system has 12 GB RAM and runs SUSE Enterprise Edition Linux 11. The peak CPU compute power per socket is 150 Gops (300 Gops on 2 sockets) and achievable memory bandwidth of 44 GBps on 2 sockets.

We collected regular expressions from the Snort rule-sets as provided by the VRT [27]. We choose random subsets consisting of 200, 400, 600, 800, 1000, and 1200 regular expressions from the backdoor, exploit, spyware, web-activex, and web-client rule-sets as the basis for the NFA and our analysis. We employ a packet trace from the 1999 DARPA Intrusion Detection Evaluation Data Sets distributed by the MIT Lincoln Laboratory [9], and simulate a million packets as input data.

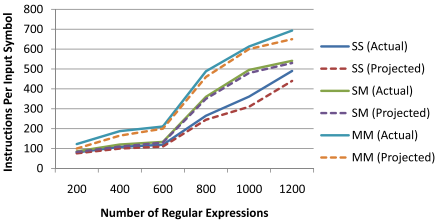
We first describe the impact of various algorithmic and architectural optimizations that we performed in GPP-grep. We then discuss the key static and runtime characteristics of NFA traversal and show that they correlate well to the performance model. Finally, we compare the performance of our regular expression matching against other state-of-the-art systems.

6.1 Impact of Optimizations

Figure 4(a) shows the speedup obtained over the baseline PCRE (v8) performance due to the various optimizations with the baseline PCRE performance normalized to 1. We parallelized PCRE for a fair comparison. The speedups obtained from each optimization are *multiplicative* on top of previous optimizations. The lowermost bar gives the



(a) Speedup over PCRE with various optimizations.



(b) Comparison of real instructions executed per input symbol to performance model predictions across MM, SM, and SS NFA.

Fig. 4. Speedup and fit to Analytical Model

impact of our efficient NFA construction that helps reduce the number of active states and efficient layout that enables a constant time neighbor lookup/addition to the active state list as described in Section 5.2. This provides a large benefit of 9–17× over PCRE. The effect of this optimization is primarily to reduce the number of instructions executed. In the absence of architectural optimizations, some of the impact of such instruction reduction weakens for large NFA due to increasing cache and TLB misses and consequently lower IPC. Thus we see a smaller benefit for large regular expression sets. This motivates the need for our architectural optimizations.

The next bar is the impact of picking NFA with multiple transitions per symbol and multiple epsilon transitions (MM) versus a single transition per symbol and epsilon transition (SS). This impact is consistently 1.5–1.8× on top of the previous optimizations. As before, the impact is primarily instruction reduction due to efficient address computation possible in the SS code (Section 5.3).

The next benefits come from an improvement in IPC. The impact of hierarchical NFA blocking increases with NFA size (larger number of regular expressions) up to a maximum of an additional multiplicative impact of 2.7×. As described in Section 5.2, this is due to a decrease in the number of cache and TLB misses. The next IPC optimization is the impact of ILP optimizations such as unrolling and software prefetching described in Section 5.3. This has an impact of 1.1–1.4×, and has more impact for larger NFA (similar to hierarchical blocking). This also includes the effect of SMT (Simultaneous Multi-Threading), which helps hide memory latency when software prefetching techniques do not succeed. This occurs, for instance, when fetching nodes that are in the ε-closure of an active state but not in the neighbor list of the state—we do not prefetch these nodes and rely on SMT.

Finally, we also obtain up to a 1.3× speedup due to the impact of using SIMD instructions. Since we use SIMD to process multiple active states at once, the impact of SIMD increases when we have a relatively large number of active states. This happens for larger NFA that correspond to more regular expressions. Hence our SIMD speedup increases with increasing numbers of regular expressions.

Table 1. Runtime characteristics of NFA traversal

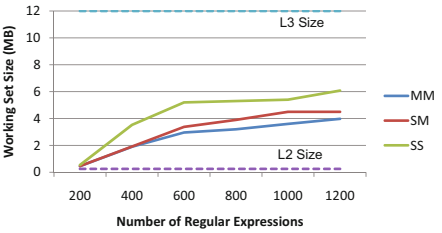
NFA Type	# REs	# states	# transitions	Avg. Active States	Max Active States	Avg. Depth
MM	200	15,883	1,922,997	4.2	24	3.5
	400	43,535	6,729,191	5.4	42	6.1
	600	62,188	9,462,802	7.0	46	8.9
	800	83,965	13,118,674	13.1	81	11.0
	1000	106,815	16,915,665	19.2	82	12.9
	1200	137,927	22,833,917	23.6	94	14.3
SM	200	16,291	1,923,405	4.3	24	3.7
	400	44,336	6,729,992	5.4	42	7.1
	600	63,377	9,463,991	6.9	46	10.2
	800	85,503	13,120,212	13.0	81	13.5
	1000	108,762	16,917,612	18.9	83	15.0
	1200	140,322	22,836,312	23.7	96	18.1
SS	200	16,252	1,957,905	4.3	24	3.7
	400	44,288	6,804,066	5.2	43	8.1
	600	63,313	9,580,679	6.8	47	15.1
	800	85,424	1,327,3108	13.1	82	17.6
	1000	108,731	17,124,534	19.0	84	20.5
	1200	140,491	23,088,684	23.6	98	23.2

6.2 Performance Analysis

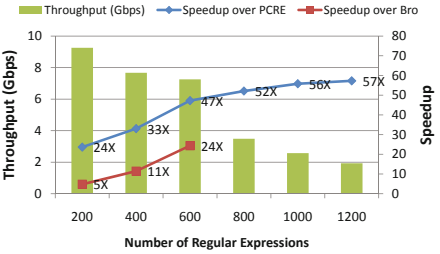
Table 1 shows the salient static and runtime characteristics of the MM, SM, and SS NFA. Col. 5 shows average number of active states during traversal. The average number of active states is a good indicator of the number of instructions required to perform the traversal. Col. 7 shows the average depth of NFA states traversed, which gives us an indication of the true working set of traversal. This has implications on the number of cache misses during traversal. Table 1 also illustrates that the number of active states increases with the number of regular expressions. Since the number of active states directly impacts the number of instructions (and hence final performance), we expect to see a drop in performance for larger sets of regular expressions. Further, all three NFA types—MM, SM, and SS—have similar numbers of active states. However, the number of instructions executed for SS is the least because we can use the fact that there is only one neighbor per symbol and only one ε transition to simplify the address calculations during traversal.

Comparison with Performance Model. To compute the number of ops for various operations listed in Section 5.4, we analyzed the static assembly file and hand-counted the number of instructions. These numbers were then plugged in together with the average number of symbol neighbors and nodes in the ε-closure to compute the projected number of instructions. Our projected number matches closely to actual results, only slightly less (5–10%) than the real performance, and is illustrated in Figure 4(b). This is due to the register spills and fills that are not accounted for by our model. Furthermore, our optimized layout results in a per-core IPC of around 1.8 on 12 cores, and hence the resultant run-times are also within 8–12% of the projected run-times (obtained using the maximum per-core IPC of 2).

Working Set Analysis. We measured the working set (data size for which the number of cache hits is $\geq 90\%$), for our MM, SM, and SS NFA types with our input regular expression sets. The working set increases with increasing number of regular expressions



(a) Working sets for MM, SM, and SS NFA with a per-socket shared L3 cache of 12 MB and per-core private L2 cache of 256 KB.



(b) Throughput of GPP-grep and speedup over PCRE and Bro. For fair comparison, we parallelize PCRE and Bro. Bro exceeds memory capacity for >600 regular expressions.

Fig. 5. Working Set and Throughput Evaluation

and is highest for the SS type. Even for the SS type NFA for 1200 regular expressions (our largest input), the working set entirely fits in the L3 cache (12 MB per socket). However, the working set usually does not fit in the L2 caches of the individual cores. Finally, we do see an improvement in IPC using the software prefetches described in Section 5.3. The results of those improvements are illustrated in Figure 5(a).

6.3 Comparison with State of the Art Systems

Figure 5(b) demonstrates the throughput of GPP-grep using the best performing SS NFA. Also, Figure 5(b) shows the speedup of GPP-grep over PCRE and Bro (v.1.5.1) systems. After our optimizations, we obtain a final throughput between 2 and 9.3 Gbps, with a performance of **2 Gbps for 1200 regular expressions**. The absolute performance of GPP-grep drops with increases in the number of regular expressions. This results from increases in the number of active states as the number of regular expressions grows and is shown in Table 1.

We parallelized both PCRE and Bro. PCRE uses NFA with small working sets; these scale perfectly with the number of cores. GPP-grep is 24–57× faster than parallel PCRE, depending on the number of regular expressions. The speedup improves with larger sets of regular expressions. This is because PCRE provide best performance when handling regular expressions one at a time; hence the runtime is proportional to the number of regular expressions. However, the number of active states in GPP-grep only increases by about 1.5× from 200 to 600 regular expressions though there is a sudden increase in the number of active states from 600 onwards resulting in a stabilization of the speedup over PCRE. GPP-grep is also 5–24× faster than Bro. Since the Bro system demonstrated poor results when run one regular expression at a time, we grouped all the regular expressions together. The Bro system adopts a DFA based approach for matching. For greater than 600 regular expressions, the DFA sizes generated by Bro were greater than our system memory of 12 GB. Hence we do not report the resultant performance numbers. Our experiments indicate that the Bro system rapidly becomes bandwidth bound for 400 regular expressions and beyond, since the working set of the DFA does not fit in the L3 cache. Bandwidth bound applications are unable to take advantage of the full

computational capabilities of multi-core platforms; we only obtained a parallel scalability of about 3.6× on 12 cores for the 600 regular expression DFA. On the other hand, the working set of the NFA produced by GPP-grep fits in the L3 cache, even for large numbers of regular expressions—hence we are compute-bound and scale near-linearly with the 12 cores of a Xeon X5680. The difference in parallel runtime scaling results in an increase in speedup over Bro as the number of regular expressions increases.

We compare our performance with the FPGA based solution proposed by Mitra et al. [19]. We first note that our single-threaded PCRE performance for 200 regular expressions is similar to their PCRE performance (also for 200 Snort regular expressions) on a single 3.0 GHz Xeon; this indicates that the regular expressions used have similar complexity in terms of active states, making our performance comparisons fair. They report a speedup of 335× over single-threaded PCRE for 200 regular expressions when using FPGAs, while we are about 258× faster than single threaded PCRE on a single CPU (24× faster than PCRE running on 12 cores). Thus our CPU performance is about 1.3× off their FPGA performance using commodity processors. Further, our CPU implementation can scale to a much larger number of regular expressions, while FPGA are more resource limited in terms of on-board memory and do not scale as well to larger sets of regular expressions.

7 Discussion

Our fast regular expression matching algorithm is useful in contexts other than NIDS. In particular, XML queries expressed in XPath [29] often have to be matched against incoming documents in publish/subscribe systems [10] where efficient regular expression matching would prove a boon. Containment queries on trees and graphs wherein the task is to match a tree (or graph) against another tree (or graph), as illustrated by GraphDB [14], pose another potential arena for GPP-grep. Improving NFA traversal can be applied to the general graph traversals used in many contexts including graph searches and graph matching similar to the A* graph search which, in several forms, found use in graph database shortest path searches [14], as well as matches of subgraphs in protein databases, image databases, and software repositories [11].

The performance of our SIMD algorithm for graph traversal would further improve with hardware support for gathers/scatters and packed-store operations. The upcoming Intel MIC (Many Integrated Core) architecture will add such support and should improve SIMD utilization [22]. Coupling this with a kernel implementation is conducive to System-on-Chip (SoC) implementations where packet I/O is combined with matching on a single chip and which makes this approach applicable to current and future trends in regular expression processing. Finally, we note that our algorithm will benefit from any technique that helps further minimize the number of active states during traversal such as: XFA [25] and HFA [2]. Our techniques are complementary and should result in cumulative improvement.

8 Conclusion

We present GPP-grep, a fast regular expression processing engine on commodity general purpose processors. GPP-grep exploits thread-level and data-level parallelism, and

employs an architecture-friendly layout and graph traversal scheme to improve efficiency. On a dual-socket commodity CPU system, GPP-grep attains a maximum throughput of 9.3 Gbps, and is up to 57× faster than traditional PCRE engines. In the future we hope to expand this engine to present a single tool for handling NIDS DPI processing of all criteria for any rule, fixed string or regular expression, in one pass. Ultimately, GPP-grep offers an economical solution, both financially and in terms of system resources, for high-speed regular expression matching.

References

1. Becchi, M., Cadambi, S.: Memory-efficient regular expression search using state merging. In: INFOCOM. IEEE (2007)
2. Becchi, M., Crowley, P.: A hybrid finite automaton for practical deep packet inspection. In: CoNEXT. ACM (2007)
3. Becchi, M., Crowley, P.: Extending finite automata to efficiently match perl-compatible regular expressions. In: CoNEXT. ACM (2008)
4. Becchi, M., Wiseman, C., Crowley, P.: Evaluating regular expression matching engines on network and general purpose processors. In: Architecture for Networking and Communications Systems. ACM (2009)
5. Cascarano, N., Rolando, P., Risso, F., Sisto, R.: iNFAnt: NFA pattern matching on GPGPU devices. SIGCOMM Comput. Commun. Rev. 40, 20–26 (2010)
6. Champarnaud, J.-M., Coulon, F.: NFA reduction algorithms by means of regular inequalities. Theoretical Computer Science 327(3), 241–253 (2004)
7. Champarnaud, J.-M., Coulon, F.: Erratum to NFA reduction algorithms by means of regular inequalities. Theoretical Computer Science 347(1-2), 437–440 (2005)
8. Chong, J., You, K., Yi, Y., Gonina, E., Hughes, C., Sung, W., Keutzer, K.: Scalable HMM-based inference engine in large vocabulary continuous speech recognition. In: International Conference on Multimedia and Expo. IEEE Press (2009)
9. Cunningham, R.K., Lippmann, R.P., Fried, D.J., Garfinkel, S.L., Graf, I., Kendall, K.R., Webster, S.E., Wyschogrod, D., Zissman, M.A.: Evaluating intrusion detection systems without attacking your friends: The 1998 DARPA intrusion detection evaluation. In: Intrusion Detection and Response (1999)
10. Diao, Y., Altinel, M., Franklin, M.J., Zhang, H., Fischer, P.M.: Path sharing and predicate evaluation for high-performance XML filtering. Trans. on Database Systems 28, 467–516 (2003)
11. Djoko, S., Cook, D.J., Holde, L.B.: An empirical study of domain knowledge and its benefits to substructure discovery. Trans. on Knowledge and Data Engineering 9, 575–586 (1997)
12. Dreger, H., Feldmann, A., Paxson, V., Sommer, R.: Operational experiences with high-volume network intrusion detection. In: Computer and Communications Security. ACM (2004)
13. Gramlich, G., Schnitger, G.: Minimizing NFA's and regular expressions. J. Comput. Syst. Sci. 73, 908–923 (2007)
14. Güting, R.H.: GraphDB: Modeling and querying graphs in databases. In: Very Large Data Bases. Morgan Kaufmann Publishers Inc. (1994)
15. Handley, M., Paxson, V., Kreibich, C.: Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In: USENIX Security. USENIX (2001)
16. Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P., Turner, J.: Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In: SIGCOMM. ACM (2006)

17. Kumar, S., Turner, J., Williams, J.: Advanced algorithms for fast and scalable deep packet inspection. In: Architecture for Networking and Communications Systems. ACM (2006)
18. Meiners, C.R., Patel, J., Norige, E., Torng, E., Liu, A.X.: Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. In: USENIX Security. USENIX (2010)
19. Mitra, A., Najjar, W., Bhuyan, L.: Compiling PCRE to FPGA for accelerating Snort IDS. In: Architecture for Networking and Communications Systems. ACM (2007)
20. Pasetto, D., Petrini, F., Agarwal, V.: Tools for very fast regular expression matching. IEEE Computer 43(3), 50–58 (2010)
21. Scarpazza, D.P., Russell, G.F.: High-performance R.E. scanning on the Cell/B.E. processor. In: International Conference on Supercomputing, pp. 14–25. ACM (2009)
22. Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P.: Larrabee: A Many-Core x86 Architecture for Visual Computing. ACM Trans. Graph. 27(3), 18:1–18:15 (2008)
23. Shenoy, G.S., Tubella, J., Gonzalez, A.: A performance and area efficient architecture for intrusion detection systems. In: Parallel & Distributed Processing Symposium. IEEE Computer Society (2011)
24. Smith, R., Estan, C., Jha, S.: XFA: Faster signature matching with extended automata. In: Security and Privacy. IEEE Computer Society (2008)
25. Smith, R., Estan, C., Jha, S., Kong, S.: Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata. In: SIGCOMM. ACM (2008)
26. Smith, R., Goyal, N., Ormont, J., Sankaralingam, K., Estan, C.: Evaluating GPUs for network packet signature matching. In: Performance Analysis of Systems and Software. IEEE (2009)
27. Sourcefire Vulnerability Research Team: Sourcefire Vulnerability Research Team (VRT) Snort Rule-set, 2.9.0 edn. (August 2011), `http://www.snort.org/vrt`
28. Thompson, K.: Programming techniques: Regular expression search algorithm. Commun. ACM 11, 419–422 (1968)
29. XML path language (XPath) 2.0. W3C Recommendation (2007), `http://www.w3.org/TR/xpath20/`
30. Yang, L., Karim, R., Ganapathy, V., Smith, R.: Improving NFA-Based Signature Matching Using Ordered Binary Decision Diagrams. In: Jha, S., Sommer, R., Kreibich, C. (eds.) RAID 2010. LNCS, vol. 6307, pp. 58–78. Springer, Heidelberg (2010)
31. Yu, F., Chen, Z., Diao, Y., Lakshman, T.V., Katz, R.H.: Fast and memory-efficient regular expression matching for deep packet inspection. In: Architecture for Networking and Communications Systems. ACM (2006)