Robust and Scalable String Pattern Matching for Deep Packet Inspection on Multi-core Processors

Yi-Hua E. Yang and Viktor K. Prasanna, *Fellow, IEEE*

***Abstract*—Conventionally, dictionary-based string pattern matching (SPM) has been implemented as Aho-Corasick deterministic finite automaton (AC-DFA). Due to its large memory footprint, a large-dictionary AC-DFA can experience poor cache performance when matching against inputs with high *match ratio* on multi-core processors. We propose a *head-body finite automaton* (HBFA) which implements SPM in two parts: a *head DFA* (H-DFA) and a *body NFA* (B-NFA). The H-DFA matches the dictionary up to a predefined prefix length in the same way as AC-DFA, but with a much smaller memory footprint. The B-NFA extends the matching to full dictionary lengths in a compact variable-stride *branch* data structure, accelerated by single-instruction multiple-data (SIMD) operations. A *branch grafting* mechanism is proposed to opportunistically advance the state of the H-DFA with the matching progress in the B-NFA. Compared with a fully-populated AC-DFA, our HBFA prototype has < 1/5 construction time, requires < 1/20 run-time memory, and achieves 3x to 8x throughput when matching real-life large dictionaries against inputs with high match ratios. The throughput scales up 27x to over 34 Gbps on a 32-core Intel Manycore Testing Lab machine based on the Intel Xeon X7560 processors.**

***Index Terms*—String matching; DFA; NFA; variable-stride tree; multi-core platform; SIMD; deep packet inspection**

I. Introduction

Deep packet inspection (DPI) is a central component of network security systems where the contents of the network traffic are continuously scanned for malicious attacks. Examples include network intrusion detection [2], virus scanning [1] and content filtering [4]. String pattern matching (SPM) is the most widely-used pattern matching mechanism used by DPI to match a dictionary of strings against a stream of characters. Due to the explosive growth of network bandwidth and types of attacks, SPM has become a major performance bottleneck in DPI systems [9].

Architecturally, SPM solutions can be categorized into two main groups: (1) software programs on multi-core platforms; (2) hardware implementations on ASIC or FPGA. While software-based SPM may not be the most cost or energy efficient, it has the following critical advantages:

- *Modular*: Integrate more easily with other functions.
- *Extensible*: Performance can be improved substantially by upgrading the processor or memory.
- *Portable*: Can be compiled for various number of cores and even different processor families.

Manuscript received September 20, 2011; revised April 9, 2012. This work was supported by the U.S. NSF under grant CCF-1018801.

Y-H. E. Yang is with Futurewei Technologies, Inc.

V. K. Prasanna is with the University of Southern California.

Most existing work on SPM focus on optimizing the *Aho-Corasick deterministic finite automaton* (AC-DFA) [3], whose performance tends to rely heavily on the slow-improving main memory speed. As a result, an AC-DFA usually under-utilizes the computation capability of modern multi-core processors and exhibits SPM throughput that are highly dictionary and input-dependent. In particular, when the dictionary is large and the input stream consists of higher ratios of dictionary sub-strings (the *match ratio*), performance of AC-DFA can degrade significantly Due to increased cache miss and memory pressure. Such behaviors expose performance-based denial-of-service (DoS) attack opportunities in applications that utilize the SPM solution [4].

The main contribution in this work is to propose, implement and evaluate an alternative SPM architecture, *head-body finite automaton* (HBFA), for scalable throughput both in number of concurrent threads and against inputs with high *match ratio*. In summary, an HBFA is composed of two parts: a "head" DFA (H-DFA) and a "body" NFA (B-NFA). The H-DFA consists of an AC-DFA constructed by the set of *compatible prefixes* of the dictionary. The B-NFA stores the rest of the dictionary in an efficient variable-stride *branch* data structure. Compared with original AC-DFA, HBFA takes less time to construct, requires a much smaller memory footprint, and offers higher and more attack-resilient throughput when matching large dictionaries. Other contributions include:

- We analyze the fundamental properties of SPM. Our analysis motivates the head-body partitioning.
- We define an algorithm to extract from any string dictionary a parametrized set of *compatible prefixes*, which allows for efficiently constructing the H-DFA.
- We design an algorithm for constructing a B-NFA in a cache-efficient *branch* data structure, accelerated with single-instruction multiple-data (SIMD) operations.
- We devise a *branch grafting* mechanism for efficient cooperation between the H-DFA and the B-NFA.
- The H-DFA, being itself a (smaller) AC-DFA, can be improved by other AC-DFA optimization techniques.

Section II gives the background of AC-DFA and establishes the theoretical foundation of HBFA. Section IV and Section IV describes the architecture and run-time algorithm of HBFA. Section V evaluates and compares the performance of HBFA. Section VI concludes the paper.

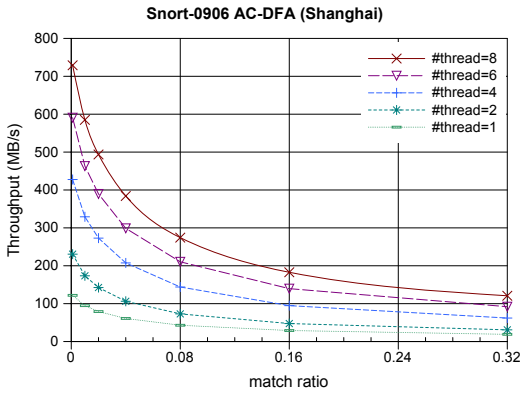


Figure 1: Throughput vs. input match ratio of AC-DFA for Snort on two 2.6 GHz AMD Opteron processors ("Shanghai").

II. Background

A. Problems with AC-DFA based SPM

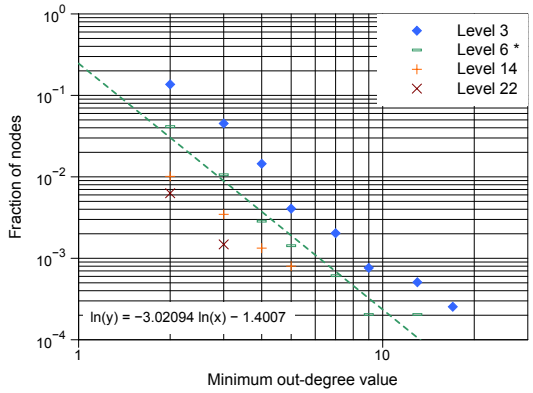
Matching high bandwidth byte streams against a large dictionary can be both compute and memory intensive. In theory, the Aho-Corasick algorithm (AC-alg) [3] can be used to construct a deterministic finite automaton (AC-DFA) which solves string pattern matching (SPM) with minimum asymptotic space and time complexities [3]. (See Appendix I-A for a more detailed discussion on AC-DFA.) In practice, the state transition table (STT) of an AC-DFA can become excessively large and result in poor cache efficiency.

Assume an 8-bit alphabet is used and 32 bits are used to encode a state number. Each state will have $2^8 = 256$ outgoing transitions and occupy $256 \times 4 = 1$ KB memory. Large-scale SPM can require an STT with hundreds of thousands of states, spanning hundreds of megabytes of memory space. In contrast, state-of-the-art multi-core processors usually have $\leq$ 10 MB on-chip cache memory. As a result, an input stream consisting of a minor portion of the dictionary can induce severe cache thrashing and dramatically reduce the SPM throughput. We use *match ratio* to quantify the amount of dictionary substrings in the input.

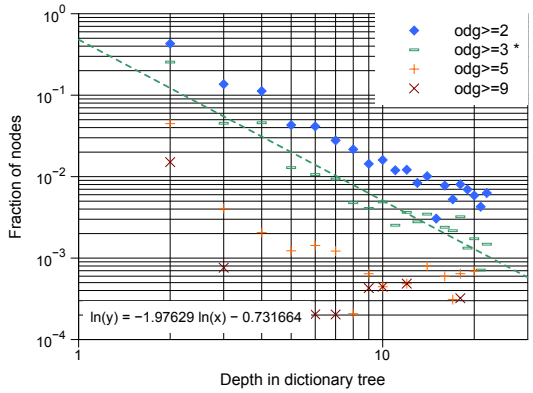
Definition 1: The *match ratio* of a stream of input characters with respect to a given dictionary is the percentage of the input characters matching various *significant prefix substrings* of the dictionary. A prefix substring is *significant* if it covers a significant part (*e.g.*, >80%) of the full string. The prefixes are selected randomly to maximize the coverage on the dictionary within a fixed input size.

Note that an input with high match ratio may not contain any match to the full strings in the dictionary. By embedding the input packets with large number of random prefix substrings from the dictionary, it is possible to launch a performance-based attack on the intrusion detection system (IDS) without actively triggering its defense mechanism.

Figure 1 shows the throughput achieved by running a fully-populated AC-DFA of the Snort [2] (June, 2009) dictionary against inputs with various match ratios. Similar trend of decreasing throughput versus match ratio is observed in all large-scale dictionaries that we tested.



(a) Fraction of nodes with various minimum out-degrees at levels 3, 6, 14 and 22 of the Snort dictionary tree.



(b) Fraction of nodes at various levels of the Snort dictionary tree with minimum 2, 3, 5 and 9 out-degrees.

Figure 2: "Double power-law" distribution of the Snort dictionary tree. In each plot, the series marked with * is fit by the power regression shown in the lower-left corner.

B. Tree Structure of SPM

An important feature of SPM is its dictionary (lexical) tree structure. A core metric of a tree is the minimum out-degrees of the tree nodes. A node is said to have a minimum out-degree d if it has at least d children. Therefore, a node with minimum out-degree d_1 can require larger storage space and result in higher traversal complexity than a node with minimum out-degree d_2 if $d_1 > d_2$.

To better understand the tree structure of SPM, we analyzed the minimum out-degrees of several dictionary trees built from real-world DPI dictionaries (see Table II in Section V). We found a "double power-law" node distribution in all the dictionary trees that we analyzed. Figure 2 shows the node distribution using the Snort dictionary as an example. Specifically, the fraction of nodes at a fixed tree level drops roughly cubically in increasing minimum out-degree (Figure 2a); the fraction of nodes with a fixed minimum out-degree drops roughly quadratically in increasing tree depth (Figure 2b).

The double power-law distribution suggests that, although most nodes in the dictionary tree above certain depth (*e.g.* Level $= 6$) can fit into a fixed node size (*e.g.* odg $= 4$), the number of nodes that do *not* fit can still be significant. There is no "clean" threshold (in either node size or tree depth) above which the fraction of exception nodes (which do not fit into the fixed size) can be statistically ignored.

With AC-DFA based SPM, the dictionary tree structure is often discarded in favor of a general DFA structure. Due to the loss of each state's depth information, a single (uniform) data structure is often used to store the entire AC-DFA. If a large memory size per state is used, then much memory will be wasted for the many small states; if a small memory size per state is used, then much design complexity and computation overhead will be needed to handle the large number of large states. Both result in inefficient SPM implementations.

C. Towards Efficient Finite Automaton for SPM

Based on the analysis above, we propose an efficient finite automaton for SPM with the following characteristics:

1) The lowest few dictionary tree levels are visited most frequently; they are designed with simple access.
2) A *variable* data structure, optimized for widely variable state sizes, is used to store the high-level states.
3) The higher tree levels are compactly stored, together with the AC-DFA failure function, in a tree structure.

Item #1 is based on the fact that the lowest dictionary tree levels consist of states with the highest out-degrees and the largest number of incoming (cross) transitions [7]. Item #2 is a direct consequence of Section II-B. Item #3 aims to strike a balance between memory and compute efficiency: the dictionary tree constitutes the minimum amount of information required for the SPM; whereas the AC-alg failure function allows efficient matching of input against the entire dictionary.

The three characteristics above lead us to the *head-body partitioning* and the design of *head-body finite automaton* (HBFA) for SPM, as detailed in the next section.

III. Construction of HBFA

A. Overall Procedure

The basic idea behind head-body finite automaton (HBFA) is to partition the dictionary into two parts: a "head" and a "body". The "head" part (H-DFA) is built as an AC-DFA from a set of dictionary prefixes. The "body" part (B-NFA) stores the rest of the dictionary in an efficient NFA structure.

The HBFA construction can be described as follows:

1) A set of *compatible prefixes* of the dictionary is identified and used to build the H-DFA; each such prefix produces a *body roots* in the H-DFA to interface with the B-NFA (Section III-B).
2) The B-NFA, constructed from the remaining dictionary suffixes, is organized as a forest of variable-stride body trees, each headed by a body root (Section III-C).
3) The body trees are accessed and traversed in a compact *branch* data structure (Section III-D).
4) To reduce the SPM overhead induced by the body-to-head backtracking, each branch in the B-NFA is associated with a *branch grafting function*, which transitions to an H-DFA state with minimum backtracking when forward traversal in the B-NFA fails (Section III-E).

B. Construction of Head DFA

Let S be the dictionary (set of strings) to match. A head DFA (H-DFA) for S is an AC-DFA constructed from a set of *compatible prefixes* of S. We say P is a set of compatible prefixes of S if (1) every $s \in S$ has a prefix in P, and (2) any $p_1 \in P$ which is a proper infix[1] of some $p_2 \in P$ must also be a full string in S.

Definition 2: Given dictionary S, the following two properties define a set of *compatible prefixes* P of S:

1) $\forall s \in S: \ \exists p \in P, l \leq \text{length}(s) \ \ni \ p = \text{prefix}(s, l)$
2) $\forall p_1, p_2 \in P: p_1 = \text{infix}(p_2, l_1) \rightarrow p_1 \in S$

where $\text{prefix}(t, i)$ and $\text{infix}(t, i)$ denote the length-i prefix and infix of string t, respectively.

Lemma 1: A simple way to obtain a set of compatible prefixes P of dictionary S is as follows. Define some length l_H. $\forall s \in S$ with $\text{length}(s) > l_H$, add $\text{prefix}(s, l_H)$ to P; $\forall s \in S$ with $\text{length}(s) \leq l_H$, add s itself to P. The resulting P is a set of compatible prefixes of S.

Proof: Every string $s \in S$ has either its (proper) prefix p of length l_H or s itself added to P, satisfying Definition 2 item #1. Suppose $p_1, p_2 \in P$ and $\text{length}(p_1) < \text{length}(p_2)$. Since $\text{length}(p_2) \leq l_H$, $\text{length}(p_1) < l_H$, p_1 must be a string in S itself, satisfying Definition 2 item #2. ■

Given a compatible set of prefixes P of dictionary S, an H-DFA can be constructed by first applying AC-alg to P to build an AC-DFA, then for every $p \in P$ which is a proper prefix of some $s \in S$, marking the highest-level state constructed from p as a *body root*. As we shall see in Lemma 2 below, the set of compatible prefixes ensures that, given any input, at most one body root can become "active" in the H-DFA at any input character. This minimizes the overhead in initiating B-NFA instances from H-DFA; it also allows efficient cooperation between H-DFA and B-NFA.

Definition 3: Let S be a dictionary and P a set of compatible prefixes of S. The H-DFA $\mathfrak{H} = (Q_H, q_0, \Sigma, \delta_H, M_H, Q_R)$ is an AC-DFA constructed by running AC-alg on P:

1) Q_H is the set of AC-DFA states.
2) $q_0 \in Q_H$ is the initial state.
3) Σ is the alphabet used to compose S.
4) $\delta_H : Q_H \times \Sigma \rightarrow Q_H$ is the state transition function.
5) $M_H \subset Q_H$ is the set of match states.
   $\forall p \in P \wedge p \in S$, $\exists m_p \in M_H$ such that the transition to m_p signals a match of p found in the input.
6) Q_R is the set of body roots
   $Q_R \triangleq \{q \in Q_H |\ g(q, *) = \varnothing \wedge q \notin M_H\}$
   where $g(q, *)$ is the set of all possible target states of the AC-DFA goto function at q.

[1] We use *infix* to mean any continuous substring within a longer string.

The first 5 items above simply define an AC-DFA constructed from P. Item #6 defines the body roots as the highest-level states constructed from all $p \in P$ which are proper prefixes of some $s \in S$.

Lemma 2: The set of body roots Q_R as defined in Definition 3 satisfy the following property:

$$\forall q \in Q_R, q' \in Q_H : \ f(q') \neq q$$

where $f(\cdot)$ is the AC-DFA failure function.

Proof: As discussed above, Definition 3 item #6 specifies any body root q is the highest-level states constructed from some $p \in P$, which is a proper prefix of some $s \in S$. Since P is a compatible set of prefixes of S, any $p \in P$ that is a proper prefix of some $s \in S$ cannot be an infix of another $p' \in P$ (the reverse of Definition 2 item #2). This implies that q cannot be the failure state of any q' constructed from p'. ■

C. Construction of Body NFA

Definition 4: Let S be a dictionary, Q, g, and M be the set of states, the goto function, and the set of match states of the AC-DFA of S, and $\mathfrak{H} = (Q_H, q_0, \Sigma, \delta_H, M_H, Q_R)$ be an H-DFA of S. The corresponding B-NFA $\mathfrak{B} = (Q_B, Q_R, \Sigma, g_B, M_B, \gamma_B)$ can be found by extending every Q_R to the higher-level nodes in Q:

1) $Q_B = \{Q \backslash Q_H\} \cup Q_R$ is the set of body states.
2) Q_R is the same set of body roots as in $\mathfrak{H}$.
3) Σ is the alphabet used to compose S.
4) $g_B : Q_B \times \Sigma \to \{Q_B \backslash Q_R\}$ is the body goto function. $\forall q \in Q_B, c \in \Sigma : g_B(q, c) = g(q, c)$.
5) $M_B = \{M \backslash M_H\}$ is the set of body match states.
6) $\gamma_B : Q_B \to Q \cup \{\varnothing\}$ is the grafting function.

Topologically, a B-NFA is a *forest* consisting of multiple *body trees*, each headed by a body root in Q_R. Every B-NFA state corresponds to a node in one body tree; the state can be reached from the body root of through a series of calls to the body goto function g_B.

As its name implies, B-NFA is non-deterministic. This is a consequence of the grafting function γ_B, which will be fully defined in Section III-E.

D. Mapping B-NFA to Branches

We design the *branch* data structure to store multiple B-NFA states compactly together in memory. Every B-NFA branch $\beta_\nu^{(h)}$ is defined by a stemming state ν, a branch height h, and the body goto function g_B.

Definition 5: Let $\mathfrak{B} = (Q_B, Q_R, \Sigma, g_B, M_B, \gamma_B)$ be a B-NFA from Definition 4:

1) A *branch* $\beta_\nu^{(h)}$ of $\mathfrak{B}$ with *stemming state* $\nu \in Q_B$ and *height* h is defined as the set of B-NFA states

$$\beta_\nu^{(h)} \triangleq \bigcup_{i=1}^{h} g_B^{(i)}(\nu, *)$$

where $g_B^{(1)}(q, *) = \{q' \mid \exists c \in \Sigma : \ q' = g_B(q, c)\}$, and

$$g_B^{(i+1)}(q, *) = \bigcup_{q' \in g_B^{(i)}(q,*)} g_B^{(1)}(q', *)$$

We say ν *stems* $\beta_\nu^{(h)}$, or $\nu \prec \beta_\nu^{(h)}$.

2) The *fanouts* $\phi_\nu^{(h)}$ of $\beta_\nu^{(h)}$ is defined as

$$\phi_\nu^{(h)} \triangleq \{q \in \beta_\nu^{(h)} \mid q \in M_B \vee g_B^{(1)}(q, *) \cap \beta_\nu^{(h)} = \varnothing\}$$

3) The *width* of $\beta_\nu^{(h)}$ is defined as $|\phi_\nu^{(h)}|$.

For simple notation, we sometimes omit the superscript or subscript of $\beta_\nu^{(h)}$ when the omission causes no ambiguity. Informally, a B-NFA branch stemming from ν with height h consists of states on the paths of all possible stride-h goto transitions starting from ν.

Claim 1: The following statements on branches and stemming states are true for any B-NFA:

- The body roots are the lowest-level stemming state.
- The stemming state of any branch β is either a body root, or a fanout of the parent branch of β.
- A branch fanout is either a match state in the B-NFA, or the stemming state of a child branch, or both.
- Every branch can be uniquely identified by its stemming state, although technically $\nu \notin \beta_\nu$.

Due to the underlying body tree structure, the width w of a branch increases monotonically with respect to h; the rate of increase depends on the number of fanouts of the B-NFA states. Given a fixed storage size k per branch, a body tree can be packed into various branches by finding the maximum h and w from each stemming state such that $h \times w \leq k$, starting with the body roots as the initial set of stemming states. Such mapping transforms the body tree from a tree of variable-size nodes into a tree of fixed-size but variable-stride branches. As discussion in Section II-B, such transformation would allow the B-NFA to achieve excellent memory efficiency for a wide range of dictionaries used by deep packet inspection.

E. B-NFA Grafting Function

We can now formerly define the B-NFA grafting function first introduced in Definition 4.

Definition 6: Let $\mathfrak{B} = (Q_B, Q_R, \Sigma, g_B, M_B, \gamma_B)$ be a B-NFA, Q_H be the corresponding set of H-DFA states, $Q = Q_H \cup Q_R$, and f be the corresponding AC-DFA failure function. The *grafting function* $\gamma_B : Q_B \to Q \cup \{\varnothing\}$ of $\mathfrak{B}$ is defined as follows:

1) Map $\mathfrak{B}$ into branches as described in Definition 5.
2) $\forall q \in Q_B, q' = f(q)$:

$$\gamma_B(q) = \begin{cases} q' & \exists \beta, \beta' : q \prec \beta \wedge (q' \in Q_H \vee q' \prec \beta') \\ \varnothing & \text{otherwise} \end{cases} \tag{1}$$

We say state q *grafts to* either state q' or $\varnothing$, respectively. According to Equation 1, the B-NFA grafting function is essentially the AC-DFA failure function with the exception of mapping some states to $\varnothing$ (nothing). Specifically, $\gamma_B(q) = f(q)$ *only if* both q and $f(q)$ are stemming states in the B-NFA, or q is a stemming state and $f(q)$ belongs to the H-DFA. A non-stemming state in B-NFA is never a valid source nor target state of the grafting function.

Because each stemming state uniquely identifies a branch in the B-NFA (see Claim 1 in Section III-D), the function γ_B can be regarded as a *branch* grafting function. Suppose $q' = \gamma_B(q)$; if $q \prec \beta$ and $q' \prec \beta'$, then γ_B essentially grafts β to β'; if $q \prec \beta$ but $q' \in Q_H$, then γ_B grafts β back to the H-DFA. In both cases, γ_B describes an ϵ-transition from every B-NFA branch to a (lower-level) branch or an H-DFA state where an alternative point in the finite automaton (either H-DFA or B-NFA) can be reached by the input.

IV. Run-Time Processing of HBFA

In this section, we explain the architecture and run-time processing of the proposed head-body finite automaton (HBFA). An example showing the matching progress of a small HBFA is given in Appendix II. For convenience of presentation, the following definitions from Section III:

- Original AC-DFA: $\mathfrak{D} = (Q, q_0, \Sigma, \delta, M, \lambda, g, f)$
- H-DFA: $\mathfrak{H} = (Q_H, q_0, \Sigma, \delta_H, M_H, Q_R)$
- B-NFA: $\mathfrak{B} = (Q_B, Q_R, \Sigma, g_B, M_B, \gamma_B)$

The run-time processing of HBFA can be outlined in the three steps below, each explained in a subsection:

1) The input characters are matched in the H-DFA until a body root is reached, where a B-NFA instance is initiated to match the following input. (Section IV-A)
2) The input characters are compared with the path labels at each activated branch in the B-NFA. If the path comparison leads to a fanout state, then the child branch stemming from the fanout state becomes the next active branch. (Section IV-B)
3) When no fanout state in the active branch can be reached, the B-NFA either grafts to another B-NFA branch, or to an H-DFA state. (Section IV-C)

Any match state visited (in either H-DFA or B-NFA) during the process above is reported as a match output.

A. Architecture of H-DFA

As discussed in Section III-B, given a dictionary S, an H-DFA is essentially the AC-DFA constructed from a set of compatible prefixes of S. In our implementation we use the simple method described in Lemma 1 (Section III-B) to find the set of compatible prefixes, whose size can be configured by a single "prefix length" parameter.

To access the δ_H transitions, we assume a byte-oriented input alphabet $|\Sigma| = 2^8 = 256$ and a fully-populated state transition table (STT). There are $|Q_H|$ rows and $|\Sigma|$ columns in the STT, one row for each $q \in Q_H$ and one column for each $c \in \Sigma$. Every entry in the STT stores the next state number given the current state number (row index) and the next input character value (column offset). To encode up to $2^{16} = 65536$ states in a moderately large H-DFA, two bytes (16 bits) are needed for each entry. The size of the STT is therefore $|Q_H| \times 512$ bytes.

To easily identify the type of the H-DFA states, we organize the state numbers into 3 ranges:

1) Internal state numbers: 0 — $(n_1 - 1)$
2) Match state numbers: n_1 — n'_1
3) Body root numbers: n_2 — n'_2

State numbers in range 1 encode those H-DFA states that are neither match states nor body roots ($Q_H \setminus (M_H \cup Q_R)$). State numbers in range 2 and range 3 encode the match states (M_H) and body roots (Q_R), respectively, in the H-DFA. In general, $n_1 < n_2 \leq n'_1 < n'_2$; numbers between n_2 and n'_1 represent states that are both match states and body roots. There are no more than n'_2 states in the H-DFA.

Run-time algorithm of H-DFA is similar to that of AC-DFA. Specifically, at any time index t, the current state $q \in Q_H$ and the next input character c_{t+1} are used to access the STT for the next state $q' = \delta_H(q, c_{t+1})$. If $q \in M_H$ is reached at time t, then the *state position* (q, t) is reported as a match output from the H-DFA. It is important to note that the "time index" t here is defined as the current byte (character) offset in the input: a state position represents a "snapshot" of the H-DFA matching up to the input byte offset. In particular, t does *not* map to either the wall time or the processor cycle time.

Definition 7: A *state position* (q, t) of a finite automaton with respect to input stream $[c_0, c_1, \ldots]$ is reached when the state q of the finite automaton is reached by consuming the input characters $[c_0, \ldots, c_t]$.

Different from AC-DFA, if some $q \in Q_R$ is reached in H-DFA at time t, then the state position (q, t) is used to initiate a B-NFA instance to continue the input matching process; meanwhile, the H-DFA makes an ϵ-transition to $(f(q), t)$ and become temporarily "frozen." At some later time, the B-NFA instance can transfer the string matching process back to the H-DFA, either via the grafting function to some "future" state position (q', t'), $t' \geq t$, or simply to $(f(q), t)$ by default. The branch grafting process is explained in Section IV-C below.

Due to the additional "body root" state type and the overlap between the match states and body roots, it takes up to three compare instructions to identify a state in H-DFA, compared to just one compare to find the match status in ordinary AC-DFA. This overhead makes H-DFA slightly slower than the best-case AC-DFA , although the few cycles can be easily offset by the better cache performance of H-DFA on a modern processor.

B. Architecture of B-NFA

As discussed in Section III-C & III-D, a B-NFA branch consists of multiple body trees nodes packed in a variable-stride data structure. We design a compact and SIMD-friendly architecture to access every branch in 64 bytes of memory, matching the cache line size of most modern processors.

Table I describes the data structure of B-NFA branch in detail. In general, a branch of size k can have any height h and width w as long as $h \times w \leq k$ (see Section III-D). This allows the branch to store up to w different goto transition paths from the branch's stemming state of the branch, each path with a stride up to h characters. For simplicity and efficient processing, we assign to any B-NFA branch a "type" with pre-defined $h \times w$, as shown in the description below Table I. For example, a branch of type "4x8" can store up to 8 goto transition paths, each with a stride up to 4 characters. The

Table I: Data structure of B-NFA branch.

0	1	2	3	4 5 6 7	8 9 a b	c d e f
b	*h*	*n*		next_state	graft_state	mat_msk
input_msk						
path_labels[1]						
path_labels[2]						

type (*b*) One of the following: {1x256, 1x32, 1x16, 2x16, 2x8, 4x8, 4x4, 8x4, 8x2, 16x2, 16x1, 32x1}
height (*h*) Maximum path stride in the branch: {1, 2, 4, 8, 16, 32}
nfouts (*n*) No. fanout states leading to a valid child branch.
next_state State number of the first fanout to a valid child branch.
graft_state State number of the grafting function target; -1 if $\varnothing$.
mat_msk Bit-mask of the match states. One bit for each byte in the path_labels vector.
input_msk Byte-mask of the valid input characters to be compared with by the path_labels vector.
path_labels Up to 32 bytes of path labels in branch.

branch height is also stored in the height field for easy access. The nfouts field stores number of full-length paths which lead to the fanout states of the branch. All fanout states of any branch are numbered consecutively, with the first (lowest) fanout state number stored in the next_state field.

Up to 32 bytes of path labels, derived from the goto transitions between states within the branch, are stored in the two path_labels fields; the exact arrangement of the path labels depends on the branch type. The input_msk is used to mask bytes read from the input stream when comparing them with a path of stride less than the branch height. For example, a branch of type "4x8" has the two path_labels fields logically partitioned into 8 chunks of 4 bytes each, while some 4-byte chunk can actually store a path of only 3 (or fewer) bytes long. The mat_msk masks the comparison result for match outputs; a non-zero bit in mat_msk indicates the corresponding byte in path_labels is on a path that comprises a dictionary match.

The input stream can be compared with all paths stored in the path_labels field(s) in parallel using single-instruction multiple-data (SIMD) instructions.[2] To do so, the B-NFA first reads h *input bytes* from the input stream and replicates them w times to form an *input vector* of length $h \times w$ (both h and w are known from the branch type). The resulting input vector should be either 16 or 32 bytes long, matching the length of one or two path_labels fields, respectively. A number of SIMD instructions are then performed on the input vector and the path_labels field(s) in the following steps:

1) Mask the input vector by input_msk and compare it with the path_labels field(s).
2) Find the fanout state which can be reached by matching the input bytes to the path labels.
3) Extract any branch path that comprises a dictionary match and has path labels matching the input bytes.

There can be at most one fanout state found in step 2 above, which causes the B-NFA to traverse to the branch stemming from the fanout state. On the other hand, two paths can have path labels matching the input bytes if one is a prefix of the other. All the branch paths extracted in step 3 above are recorded and reported as match output at the branch. Due to the body tree structure, each B-NFA branch can have only one previous (parent) branch. A corollary to this property is that all the fanout states of any branch can be numbered sequentially, allowing the respective children branches to be placed consecutively in memory. The address of any child branch can be found simply by taking a non-negative offset to the state number of the branch's first fanout state (*i.e.*, the next_state field).

On average, the processing in each branch consists of 25–35 instructions (SIMD or not), taking 20–30 clock cycles on a modern super-scalar processor. While this is not much faster than a hash table lookup, we note that most branches have heights from 4 to 16 bytes, allowing the B-NFA to be traversed in multi-byte strides very often *and* with good memory and cache efficiency.

C. Head-Body Cooperation through Branch Grafting

A B-NFA can work either independently or cooperatively with the H-DFA. Suppose a B-NFA instance is initiated at some state position (r, t), $r \in Q_R$. An *independent* B-NFA starts its matching process at branch β_r (stemming from r) and input byte offset $t + 1$. A forward branch transition is made when the following input bytes match the path labels of a full-length path in the branch. The B-NFA instance continues to make forward branch transitions until at some branch the input bytes do not match the path labels of any full-length path. Then the B-NFA instance terminates and the H-DFA is reactivated at $(f(r), t)$, as discussed in Section IV-A.

A *cooperative* B-NFA makes forward branch transitions in the same way as an independent B-NFA. In addition, at every branch a cooperative B-NFA also updates a local "grafting point" variable, gp, which specifies the best alternative branch (state) position in the B-NFA (H-DFA) where the string matching process can continue when no more forward transition can be made at the current branch:

1) When the B-NFA instance is initiated at state position (r, t), $r \in Q_R$, $gp \triangleq (f(r), t)$.
2) Suppose branch β_ν stemming from $\nu \in Q_B$ is reached after consuming input characters $[c_{t+1}, \ldots, c_{t'}]$; let γ_B be the grafting function:
   a) If $\gamma_B(\nu) = \varnothing$, then gp is unchanged.
   b) If $\gamma_B(\nu) \neq \varnothing$, then $gp \triangleq (\gamma_B(\nu), t')$.

   Note that either $\gamma_B(\nu) \in Q_R$ or $\gamma_B(\nu) \in Q_H \backslash Q_R$.
3) Suppose (ν', t') is the latest gp, and no forward branch transition can be made in the B-NFA:
   a) If $\nu' \in Q_H \backslash Q_R$, then the B-NFA instance terminates and the H-DFA is reactivated at (ν', t').
   b) If $\nu' \in Q_B$, then the B-NFA transitions to branch $\beta_{\nu'}$ to match the input at offset $t' + 1$.

By following the grafting function, the terminating B-NFA instance does not always need to "roll back" to the original state position $(f(r), t)$ where the H-DFA was "frozen." Instead, the grafting point (ν', t') is used to either transition the

[2]Our HBFA prototype was implemented on the x86-64 instruction set with the Streaming SIMD Extensions (SSEx); similar extensions can be found in other instruction sets as well.

B-NFA to $\beta_{\nu'}$ (if $\nu' \in Q_B$) or reactivate the H-DFA at ν' (if $\nu' \in Q_H \backslash Q_R$). Since in both cases $t' > t$, the overall string matching progress is advanced even when no forward path is found in the B-NFA.

Note that the independent B-NFA can be seen as a special case of the cooperative B-NFA where all branches in the B-NFA have grafting targets set to $\varnothing$, thus the grafting point, once initialized at $(f(r), t)$, remains $(f(r), t)$ till the end of the B-NFA instance.

V. Performance Evaluation

A. Environments and Datasets

1) Platform and implementation: We implemented HBFA for x86-64 with 128-bit SSE2/SSE3 extensions. The source is written in C/C++/SIMD assembly, compiled with GCC 4.2 and the "`-O2`" flag, and run on 64-bit Linux with kernel version 2.6. We measured the performance of our program on a dual-socket server with two AMD Opteron 2382 processors (quad-core "Shanghai" at 2.6 GHz) and 16 GB DDR2-667 main memory; we also measured scalability in number of threads on a 32-core Intel Manycore Testing Lab machine based on Intel Xeon X7560 processors (8-core "Nehalem" at 2.26 GHz).[3]

We implemented three versions of SPM finite automaton: a fully-populated AC-DFA, an HBFA without head-body cooperation, and an HBFA with head-body cooperation. The same data structure is used by the two HBFA versions; their only difference is in whether the grafting point is updated and utilized in the B-NFA. For fair evaluation, all dictionary matches found by the finite automaton are stored in an output queue, which grows dynamically to accommodate large number of matches. On the other hand, we did not apply any hashing or compression to the AC-DFA for the following reasons:

1) Many hashing and compression techniques on AC-DFA improve the throughput performance only for specific types of dictionaries or input streams.
2) Any technique that could improve the performance of AC-DFA can also be applied to the H-DFA and improve the overall performance of HBFA.

A fair comparison of various STT hashing and compression techniques requires a detailed study of their performance characteristics, which is out of the scope of this paper.

2) Dictionaries: We use three dictionaries from popular intrusion detection (Snort) and virus scanning (ClamAV) applications. Table II shows the statistics of the dictionaries. Snort-0906 represents a dictionary with many relatively short strings, whereas ClamAV type-1 and type-3 represent dictionaries with long strings. All three dictionaries are "binary," *i.e.*, they utilize the full 8-bit alphabet.

[3]Intel "Nehalem" and later processors feature Hyper-Threading and Turbo Boost technologies, which can speed up both HBFA and AC-DFA. The speedup may be subject to additional variables such as L1D contention and core temperature. A fair assessment of these technologies is out of the scope of this paper.

Table II: Dictionary statistics.

Dictionary	Snort-0906	CAV-type1	CAV-type3
# strings	8,673	5,225	3,062
# chars	196,967	497,984	262,438
# levels	232	362	382

3) Input match ratio: The input match ratio is controlled by embedding *proper prefixes* of the dictionary strings randomly into an "innocent" data stream. For the Snort dictionary, we use the plain text from an HTML-formatted King James Bible as the innocent data stream; for the ClamAV dictionaries, we use the concatenation of all files under `/usr/sbin` in a typical Linux server installation as the innocent data stream.

Our experiments show that the throughput of AC-DFA and HBFA against the innocent data streams is about the same as that against a "clean" plain-text network trace collected on a typical university workstation. This is expected since the (clean) transfers on such a workstation consist mostly of files of the same nature as our innocent data streams. In contrast, matching a data stream of random 8-bit characters results in slightly (~5%) higher throughput with AC-DFA than with HBFA. When a larger or more complex dictionary is used, the matching throughput against the innocent data stream can drop significantly (see comparison between Figure 4 and Figure 5), but the matching throughput against the random data stream remains high. We use the innocent data stream for performance evaluation because it better captures the realistic behaviors of deep packet inspection.

To construct data streams with different levels of attack activities, we embed various amount of *significant prefixes* from each dictionary into the innocent data stream. The level of attack activity is described by the input *match ratio*, as defined Definition 1 in Section II-A. A 1% (0.01) match ratio could represent an occasional occurrence of intrusion-like pattern in the network traffic. A 10% or higher match ratio could represent a scenario where 10% or more network packets are sent by the attacker to slow down the system. Since only proper prefixes of the dictionary strings are embedded to the data streams, no match output is generated and the DPI system detects *no* abnormality other than the lowered matching throughput.

4) Parallelism on multi-core: To utilize all available cores in the multi-core platforms, our program runs multiple threads (up to the total number of processor cores in the system), each with one instance of HBFA or AC-DFA, matching the same dictionary. To generate meaningful and realistic results, each thread accesses its own input stream in a separate memory location. All threads share the same dictionary data structure, although each thread may (and usually will) access a different state due to the different input streams they process. The aggregated throughput over all parallel threads is used to measure the overall system performance.

Future multi-core processors are expected to have increasing number of cores or hardware threads. This parallel processing approach allows us to scale the number of software threads

Table III: Memory efficiency and construction time.

FA type	Head size # states (levels)	H-DFA MB	B-NFA MB	B-NFA B/st.	Time sec
Snort-0906					
HBFA	6,204 (3)	3.03	0.75	5.2	1.02
	21,564 (6)	10.5	0.65	5.0	2.75
	41,706 (10)	20.4	0.50	4.6	4.82
AC-DFA	156,300 (232)	153	n/a	n/a	10.0
CAV-type1					
HBFA	7,799 (3)	3.81	1.13	2.4	2.47
	21,905 (6)	10.7	1.11	2.4	4.41
	41,402 (10)	20.2	1.07	2.4	6.67
AC-DFA	484,460 (362)	473	n/a	n/a	25.6
CAV-type3					
HBFA	6,347 (6)	3.10	0.60	2.7	1.21
	21,564 (10)	11.2	0.55	2.6	2.42
	41,276 (22)	20.2	0.51	2.7	4.03
AC-DFA	232,412 (382)	153	n/a	n/a	11.2

easily to the number of cores in a shared-memory multi-core system. In some scenarios it may be preferable to match different dictionaries on various cores in the system. We note that doing so would reduce the effectiveness of the on-chip shared cache and give the memory-efficient HBFA even greater performance advantage over AC-DFA.

B. Memory Efficiency and Construction Time

For each dictionary, we constructed the AC-DFA as well as three HBFAs with various head sizes for performance evaluation. As shown in Table III, HBFA effectively compacts every dictionary into a much smaller memory than AC-DFA.

1) Memory efficiency: State-of-the-art multi-core processors can execute instructions much faster than they can access high-level caches and off-chip memory. It is advantageous to store the most-frequently accessed part of the dictionary into a private and lowest-level cache to the processing core. Thus memory efficiency is critical to the throughput scalability (in both number of cores and dictionary size) of the SPM solution.

Memory footprint of an H-DFA is directly proportional to the number of states in the H-DFA. Because by design an H-DFA can have at most 65536 states, the state numbers of H-DFA occupies only 2 bytes each; this makes H-DFA twice as memory efficient as an AC-DFA whose state numbers occupies 4 bytes each. Note that any AC-DFA compression technique is by definition applicable to H-DFA; in all cases, keeping the total number of DFA states below 64K always improves memory efficiency.

The B-NFA compacts up to 32 body tree nodes into a 64-byte branch, resulting in a memory footprint as small as 2 B/state (bytes per state). In practice, the memory footprints of B-NFA vary from 2.4 B/state to 5.2 B/state for the dictionaries evaluated in this study.

2) Construction time: The construction time is another important performance metric of SPM, especially for DPI. Most dictionaries of computer patterns (protocols, viruses, field values, etc.) are dynamic. An optimized SPM solution usually requires reconstruction for every dictionary update. A more efficient construction allows the SPM to sustain higher update rate, reduce update down-time, and require less resource for the update engine.

As shown in Table III, HBFA takes 1/3 to 1/10 time to construct than AC-DFA. To fully populate an AC-DFA, all state transitions must be traversed at least once, a process which can be slow due to the irregular memory accesses over a relatively large memory space. The HBFA construction, in contrast, compacts the dictionary tree in a depth-first manner while constructing the B-NFA. Although potentially more operations are needed, the operations are performed with much higher spatial and temporal locality over a much smaller memory space than populating the AC-DFA.

C. Throughput

Figures 3, 4 and 5 show the matching throughput of HBFA for the Snort, ClamAV type-1 and type-3 dictionaries, respectively, on the dual quad-core Opteron 2382 platform. For each dictionary, three head sizes (as in Table III) are used for the HBFA construction to show the effect of head size on input resilience and matching throughput. With each head size, the "HBFA_coop" line plots the throughput of HBFA with branch grafting; the "HBFA_ind" line plots the throughput without branch grafting. The "AC-DFA" line in every sub-figure also plots the (same) AC-DFA throughput.

Compared with AC-DFA, the matching throughput of HBFA is much more resilient to high match ratio in the input stream. The head size of HBFA offers a tradeoff between input resilience and best-case throughput: a smaller H-DFA makes the HBFA more resilient to high match ratio, whereas a larger H-DFA slightly increases the best-case throughput at low match ratio. In practice, the head size that achieves the "optimal" tradeoff will be selected for implementation. For all three dictionaries examined, a good tradeoff is obtained when the H-DFA consists of approximately the lowest 6 levels of the AC-DFA. This corresponds to 21564 H-DFA states for Snort (Figure 3 middle), 21905 for ClamAV type-1 (Figure 4 middle) and 6347 for ClamAV type-3 (Figure 5 left) dictionaries.

HBFAs with and without branch grafting have similar match ratio resilience. Branch grafting allows the B-NFA to work cooperatively with the H-DFA in advancing the matching progress. As a result, an HBFA with branch grafting always out-performs one without branch grafting matching the same dictionary. The difference is slightly greater for the ClamAV dictionaries where (on average) longer string suffixes are matched in the B-NFA.

Figure 6 shows the throughput speedup of HBFA with branch grafting over AC-DFA for all three dictionaries. With a properly chosen head size, HBFA with branch grafting achieves 1.2x ~ 1.6x the throughput of AC-DFA when matching inputs with 1% match ratio, 1.6x ~ 3x with 4% match ratio, and 3x ~ 8x with 16% match ratio. At zero match ratio where AC-DFA runs optimally with few cache misses, the HBFA achieves over 99% the throughput of AC-DFA for the Snort and ClamAV type-1 dictionaries, and 85% for the ClamAV type-3 dictionary. We notice that the dictionary tree constructed with the ClamAV type-3 dictionary has much

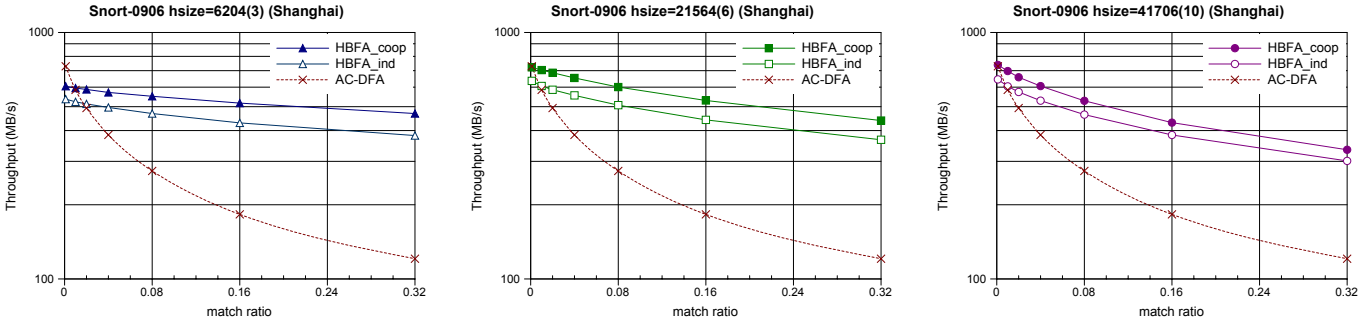


Figure 3: Throughput of HBFA for Snort with 6204, 21564 and 41706 H-DFA states (3, 6 and 10 levels).

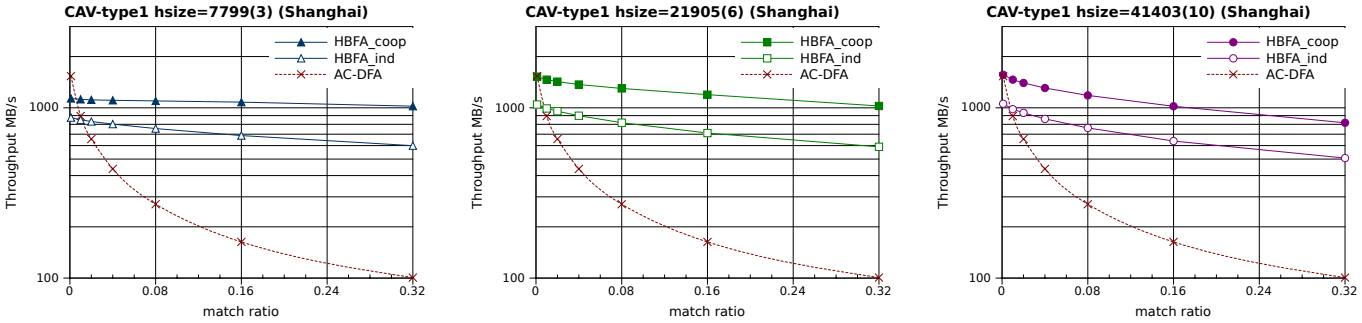


Figure 4: Throughput of HBFA for ClamAV type-1, with 7799, 21905 and 41403 H-DFA states (3, 6 and 10 levels).

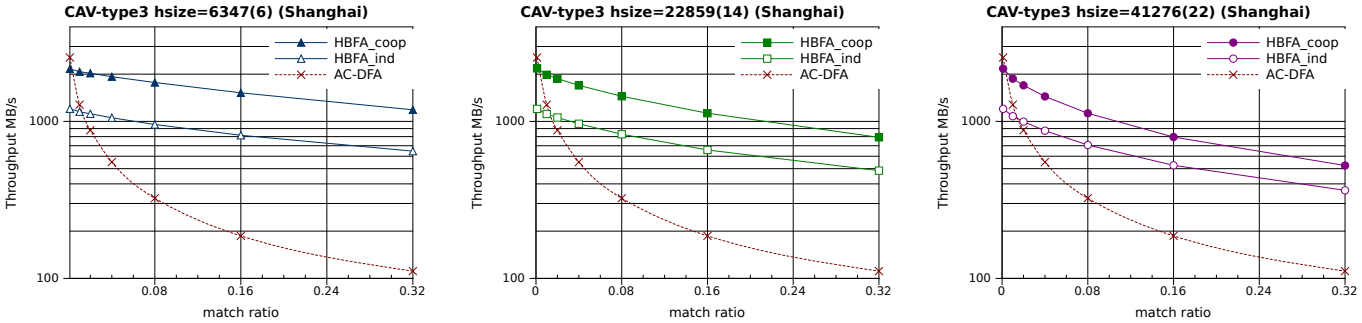


Figure 5: Throughput of HBFA for ClamAV type-3, with 6347, 22859 and 41276 H-DFA states (6, 14 and 22 levels).

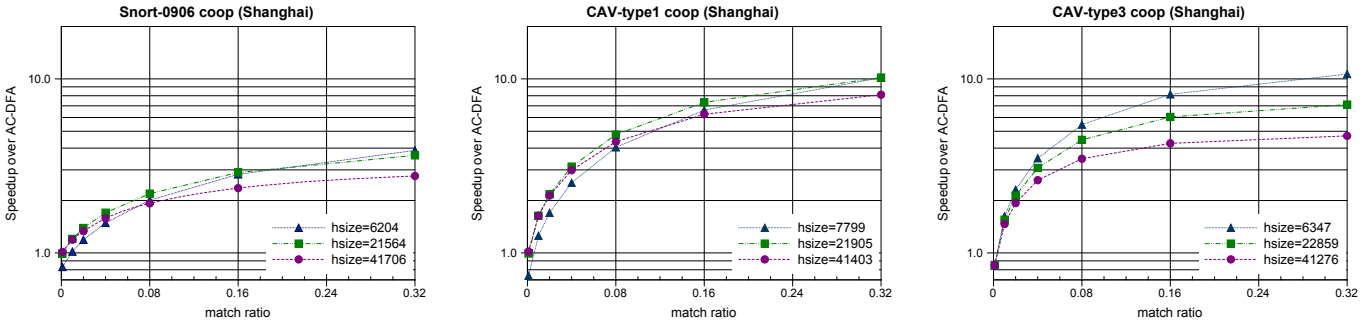


Figure 6: Speedup of HBFA with branch grafting over AC-DFA for the three dictionaries.

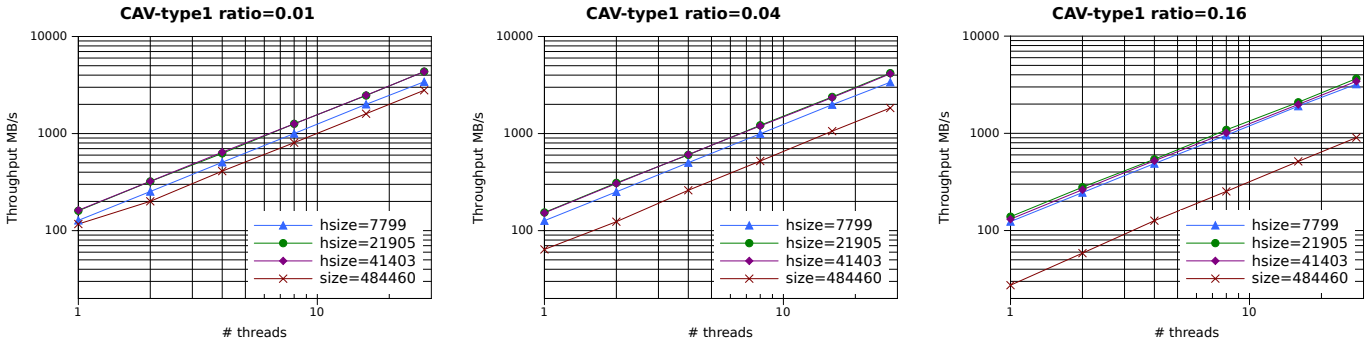


Figure 7: Throughput scaling of HBFA (lines with various hsize) and AC-DFA (lines with size = 484460) in number of threads on 4-socket Intel Xeon X7560 (up to 28 threads in a total of 32 cores).

fewer nodes in the lowest tree levels. Therefore, when the input has zero match ratio (hence containing no significant match to any dictionary string), the AC-DFA would stay mostly within a small set of states near the root (initial) state, resulting in good cache performance. However, the throughput of AC-DFA drops quickly below HBFA with only a 1% match ratio, where the AC-DFA transitions occasionally to a state farther away from the root state.

Figure 7 shows the aggregated throughput on the quad octo-core Xeon X7560 system, scaling from 1 to 28 threads[4] while matching the largest dictionary (ClamAV type-1). Both HBFA and AC-DFA scale well for all input match ratios from 0.01 up to 0.16. This implies that the throughput degradation of AC-DFA at high input match ratios is not due to insufficient memory bandwidth but rather long memory access latency. The good throughput scaling of all HBFA configurations also shows that the performance of HBFA is not sensitive to the (effective) on-chip cache size, since in the experiment system, all 28 threads compete for the 24 MB on-chip (level-3) cache.

VI. Conclusion and Future Work

We proposed head-body finite automaton (HBFA), a robust and scalable string pattern matching (SPM) solution. Unlike AC-DFA, HBFA does not suffer from significant performance degradation when processing input with high match ratios. For large dictionaries and a moderate (16%) match ratio, HBFA achieves 3x$\sim$8x matching throughput, requires $< 1/20$ run-time memory and $< 1/5$ construction time of an AC-DFA.

Currently, head-body partitioning is performed statically where the head part is bounded by a pre-defined depth value. In general, all body roots in the H-DFA do not need to have the same depth. One future direction is to design an algorithm that efficiently constructs an H-DFA with body roots at varying depths, for example, to include longer prefixes for "hot" dictionary strings. The B-NFA processing may be further accelerated by the upcoming AVX and XOP x86-64 extensions or the OpenCL/CUDA capable GPGPUs.

[4]The system is accessed remotely and shared by a large community. We instantiated only 28 threads in the 32-core system to mitigate the effect of an occasional shared access that we do not control.

References

[1] Clam AntiVirus. http://www.clamav.net/.

[2] SNORT. http://www.snort.org/.

[3] Alfred V. Aho and Margaret J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, 1975.

[4] S. Antonatos, K. G. Anagnostakis, E. P. Markatos, and M. Polychronakis. Performance Analysis of Content Matching Intrusion Detection Systems. In *Proc. of Int. Sym. on Applications and the Internet*, Jan. 2004.

[5] Spyros Antonatos, Kostas G. Anagnostakis, and Evangelos P. Markatos. Generating Realistic Workloads for Network Intrusion Detection Systems. *SIGSOFT Softw. Eng. Notes*, 29(1):207–215, 2004.

[6] Michela Becchi, Charlie Wiseman, and Patrick Crowley. Evaluating Regular Expression Matching Engines on Network and General Purpose Processors. In *Proc. of ACM/IEEE Sym. on Architecture for Networking and Communications Systems (ANCS'09)*, 2009.

[7] Weirong Jiang, Yi-Hua E. Yang, and Viktor K. Prasanna. Scalable Multi-Pipeline Architecture for High Performance Multi-Pattern String Matching. In *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, August 2010.

[8] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. High-Speed String Searching against Large Dictionaries on the Cell/B.E. Processor. In *IEEE Intl. Parallel & Distributed Proc. Sym.*, 2008.

[9] Lin Tan and Timothy Sherwood. Architectures for Bit-Split String Scanning in Intrusion Detection. In *IEEE MICRO*, volume 26, pages 110–118, 2006.

[10] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. *INFOCOM*, 4:2628–2639, March 2004.

[11] J. van Lunteren. High-performance Pattern-matching for Intrusion Detection. In *IEEE INFOCOM*, 2006.

[12] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Recent Advances in Intrusion Detection*, volume 5230, pages 116–134, 2008.

[13] Giorgos Vasiliadis and Sotiris Ioannidis. GrAVity: A Massive Parallel Antivirus Engine. In *Recent Advances in Intrusion Detection*, 2010.

[14] Oreste Villa, Daniel Chavarria, and Kristyn Maschhoff. Input-independent, Scalable and Fast String Matching on the Cray XMT. In *Proc. of IEEE Int. Parallel & Distributed Proc. Sym.*, 2009.

Yi-Hua E. Yang is a Staff Algorithm Engineer at FutureWei Technologies (Huawei USA in Santa Clara, CA). He received his B.Sc. in Electrical Engineering from the National Taiwan University, M.Sc. in Electrical & Computer Engineering from the University of Maryland, College Park, and Ph.D. in Electrical Engineering from the University of Southern California. He worked as a research assistant at the Ming Hsieh Department of Electrical Engineering, University of Southern California from 2008 to 2011, and at the Information Sciences Institute, University of Southern California From 2005 to 2007. His research interests include algorithmic optimization on parallel architectures, high performance architecture designs for network processing, security and cryptography.

Viktor K. Prasanna (http://ceng.usc.edu/~prasanna) is Charles Lee Powell Chair in Engineering in the Ming Hsieh Department of Electrical Engineering and Professor of Computer Science at the University of Southern California. His research interests include High Performance Computing, Parallel and Distributed Systems, Reconfigurable Computing, and Embedded Systems. He received his BS in Electronics Engineering from the Bangalore University, MS from the School of Automation, Indian Institute of Science and Ph.D in Computer Science from the Pennsylvania State University. He is the Executive Director of the USC-Infosys Center for Advanced Software Technologies (CAST) and is an Associate Director of the USC-Chevron Center of Excellence for Research and Academic Training on Interactive Smart Oilfield Technologies (Cisoft). He also serves as the Director of the Center for Energy Informatics at USC. He served as the Editor-in-Chief of the IEEE Transactions on Computers during 2003-06. Currently, he is the Editor-in-Chief of the Journal of Parallel and Distributed Computing. He was the founding Chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is the Steering Co-Chair of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) and is the Steering Chair of the IEEE International Conference on High Performance Computing (HiPC). Prasanna is a Fellow of the IEEE, the ACM and the American Association for Advancement of Science (AAAS). He is a recipient of the 2009 Outstanding Engineering Alumnus Award from the Pennsylvania State University.