Head-Body Partitioned String Matching for Deep Packet Inspection with Scalable and Attack-Resilient Performance*

Yi-Hua E. Yang
Ming Hsieh Dept. of Electrical Eng.
University of Southern California
Email: yeyang@usc.edu

Viktor K. Prasanna
Ming Hsieh Dept. of Electrical Eng.
University of Southern California
Email: prasanna@usc.edu

Chenqian Jiang
Ming Hsieh Dept. of Electrical Eng.
University of Southern California
Email: chenqiaj@usc.edu

***Abstract*—Dictionary-based string matching (DBSM) is a critical component of Deep Packet Inspection (DPI), where thousands of malicious patterns are matched against high-bandwidth network traffic. Deterministic finite automata constructed with the Aho-Corasick algorithm (AC-DFA) have been widely used for solving this problem. However, the state transition table (STT) of a large-scale DBSM AC-DFA can span hundreds of megabytes of system memory, whose limited bandwidth and long latency could become the performance bottleneck We propose a novel partitioning algorithm which converts an AC-DFA into a "head" and a "body" parts. The head part behaves as a traditional AC-DFA that matches the pattern prefixes up to a predefined length; the body part extends any head match to the full pattern length in parallel body-tree traversals. Taking advantage of the SIMD instructions in modern x86-64 multi-core processors, we design compact and efficient data structures packing multi-path and multi-stride pattern segments in the body-tree. Compared with an optimized AC-DFA solution, our head-body matching (HBM) implementation achieves 1.2x to 3x throughput performance when the input match (attack) ratio varies from 2% to 32%, respectively. Our HBM data structure is over 20x smaller than a fully-populated AC-DFA for both Snort and ClamAV dictionaries. The aggregated throughput of our HBM approach scales almost 7x with 8 threads to over 10 Gbps in a dual-socket quad-core Opteron (Shanghai) server.**

***Index Terms*—String matching; SIMD; multi-core processor; DFA; NFA; tree topology; multi-stride tree; intrusion detection; virus scanning**

I. Introduction

Deep packet inspection (DPI) is a critical component of network security systems where the contents of the network traffic are continuously examined. Examples include network intrusion detection [1], virus scanning [2] and content filtering [3]. Dictionary-based string matching (DBSM) is the most widely-used pattern matching mechanism used by DPI to match an input stream against a large number of strings. Due to the explosive growth of network bandwidth and number of malicious attacks, DBSM has become a major performance bottleneck in DPI systems [4].

From an architecture point of view, DBSM solutions can be categorized into two main groups: (1) hardware designs on ASIC or FPGAs [5], [6], [7], [8], [9], [4], [10], [11]; and (2) software designs on multi-core systems [12], [13], [14]. While implementing DBSM in software may not produce the highest performance, it has several critical advantages:

Modularity A DBSM solution is usually part of a more complex DPI system. Implementing DBSM as a software module for multi-core processors makes it easier to integrate DBSM with the rest of the DPI system.

Extensibility A processor-based system is more flexible and extensible than a piece of hardware. For example, the memory size and network bandwidth can be easily upgraded in many cases with a server reboot.

Portability The same (multi-threaded) software executable can run on processors with various number of cores or cache sizes. Its performance can usually be improved substantially by simply upgrading the processor and without changing the source code.

A similar but more powerful pattern matching mechanism is the regular expression matching (REM). While REM can be regarded as a superset problem to DBSM, in this study we focus only on DBSM on multi-core systems for three reasons. First, DBSM is used more widely than REM. A larger number of DPI rules utilize only DBSM together with other higher-level directives. For those rules that do require REM, DBSM is usually also used in the pre-filtering process.

Second, the performance of existing DBSM solutions on multi-core systems still leave much to be desired. In [13], the GPU-accelerated solution achieved 2.3 Gbps against 4000 random strings. In [15], the dual Cell B.E. system achieved 4.5 Gbps with (or 3.5 Gbps without) *a-posteriori* knowledge of the input. In [14], a throughput of 7.5 Gbps was achieved using 32 processors in a Cray XMT supercomputer. There is yet a cost-efficient DBSM solution capable of matching 10 Gbps traffic against several thousand strings on a multi-core platform.

Third, although REM performance on multi-core systems is generally very poor (between 30 to 300 Mbps as in [16]), high-

* Supported by U.S. National Science Foundation under grant CCR-0702784

performance REM solutions achieving over 5 Gbps matching throughput against thousands of regular expressions already exist with FPGA acceleration [17], [18]. Given that the state-of-the-art performance between REM and DBSM differs less than 2x in hardware but almost 20x in software (with DBSM being the higher of both), we are not sure the software/multi-core platform is a good choice for REM.

The Aho-Corasick algorithm, which compiles a dictionary into a deterministic finite automaton (AC-DFA), is a widely-used technique for solving DBSM. An AC-DFA processes the characters in the input stream and reports a pattern match when a "matching state" is reached. On a multi-core system, DBSM using AC-DFA usually gives good throughput when there is none or few pattern match in the input. However, due to increased cache and memory subsystem pressures, the throughput drops quickly when the input stream consists of a higher ratio of matching patterns. This can become a denial-of-service (DoS) opportunity in the network security applications that utilize the DBSM.

Our goal in this study is to analyze and improve the performance of AC-DFA on multi-core systems under all scenarios. Specifically, our contributions include:

1) We demonstrate the dependency between throughput and input match ratio of AC-DFA on multi-core systems. We show that as small as a 5% match ratio in the input stream can cut the throughput by half.
2) We propose the head-body partitioning algorithm which converts an AC-DFA into a small "head" DFA and a compact "body" NFA effectively processing in parallel. The head DFA is made small so that it enjoys better cache performance, whereas the body NFA is made compact and simple to allow fast matching.
3) We design a compact data structure for the body NFA, taking advantage the processor's cache structure and SIMD instructions. The design is suitable for x86-64 compatible processors from both AMD and Intel.
4) We implement, experiment and analyze our head-body matching (HBM) modules with various configurations on x86-64 compatible processors.

Compared with an optimized plain AC-DFA solution, our head-body matching (HBM) approach achieves 1.2x to 3x throughput performance under 2% to 32% input match ratios, respectively, while using 20x less memory. The throughput of our HBM scales almost linearly to 6.8x running on a dual-socket quad-core Opteron (Shanghai) system.

Section II gives the background of the Aho-Corasick algorithm and the related work. In Section III we give the motivation, formal description and theoretical performance analysis of the head-body DBSM. Section IV describes the design of the head DFA (H-DFA) and body NFA (B-NFA) modules. In Section V we evaluate the performance of HBM under various configurations and scenarios. Section VI discusses possible directions for future work, while Section VII concludes the paper.

II. Background

A. The Aho-Corasick Algorithm

Matching high throughput input data against a large dictionary can be both compute and memory intensive. The Aho-Corasick algorithm[19] has been used widely to minimize the computation complexity of dictionary-based string matching (DBSM) by constructing a deterministic finite automaton (AC-DFA) that matches the input against a specific dictionary. Algorithmically, the AC-DFA has computation complexity equal to $O(L)$, where L is the total length of the data input and independent of the size of the dictionary.

The Aho-Corasick algorithm consists of 3 steps. It first constructs a dictionary tree using the entire set of dictionary strings. The set of states Q in the dictionary tree will be the set of states in the final DFA, with the tree root q_0 being the start state. Each character $c \in \Sigma$ in the dictionary, Σ being the dictionary's alphabet, corresponds to a (labeled) tree edge between a parent state q_p and a child state q_c. Such a tree edge is described by the *goto function*, $g(q_p, c) = q_c$. Each string of length l in the dictionary corresponds to the path from q_0 to a *match state* at depth l.

Given a dictionary tree as described above, a *failure function* maps every tree state $q \in Q$ to its *failure state* $q' \in Q$ such that the path from q_0 to q' is the longest proper suffix possible of the path from q_0 to q. Let $\lambda(q)$ be the depth of q in the dictionary tree, it follows that $\lambda(q') < \lambda(q)$ if q' is the failure state of q.

Lastly, a *transition function* maps each pair of $(q \in Q, c \in \Sigma)$ to a target state $q" \in Q$. If there is a valid q_c where $g(q, c) = q_c$, then $q"$ is set to q_c; otherwise, $q"$ is set to the transition function of q' (failure state of q) through label c.

Definition 1: An AC-DFA is a deterministic finite automaton (DFA) constructed from a dictionary of string patterns using the Aho-Corasick algorithm [19]. It consists of an 8-tuple, $(Q, \Sigma, q_0, \delta, M, \lambda, g, f)$, described as follows:

1) A finite set of states Q
2) A finite set of labels, the alphabet Σ
3) A start state $q_0 \in Q$
4) A transition function $\delta : Q \times \Sigma \to Q$
5) A set of match states $M \subseteq \{Q \backslash q_0\}$
6) A depth function $\lambda : \{Q \backslash q_0\} \to \mathbb{N}$ and $\lambda(q_0) = 0$
7) A goto function $g : Q \times \Sigma \to Q$
8) A failure function $f : Q \to Q$

The first 5 elements in the list above are the same as those in an ordinary DFA; the last 3 elements (λ, g and f) are obtained from the Aho-Corasick algorithm during the construction of the AC-DFA.

The following definition extends the concept of failure function of AC-DFA to higher degrees.

Definition 2: The *n-th degree failure function*, $f^n(\cdot), n \in \mathbb{Z}$, of an AC-DFA is defined as the n-th power of the failure function $f(\cdot)$. That is, $f^0(q) = q$ and $f^i(q) = f\left(f^{i-1}(q)\right)$.

B. Related Work

As discussed in Section III-A, DBSM with AC-DFA can suffer from either large state transition table (STT) size or high STT access cost. Much research has been done in recent years to address these two problems. Most of the research falls in the hardware category, where specialized AC-DFAs are designed and targeted at ASIC or FPGA platforms [6], [7], [8], [4], [20], [10]. Software-based DBSM solutions using AC-DFA have been implemented in both ClamAV [2] and Snort [1] for many years. However, due to the computation and memory intensive nature, DBSM can account for 40%–70% of the processing time of network intrusion detection [21]. The performance bottleneck is further aggravated by the increasing size of the dictionary, whose AC-DFA STT can no longer fit into the processor cache.

In [12], [15], the authors improve the *aggregated throughput* of AC-DFA over hundreds of threads, which are interleaved at the instruction (assembly) level to hide the memory access latency. In [14] the technique is brought up further to the Cray XMT supercomputer, taking advantage of the massive hardware multi-threading and memory bandwidth available. In [13], a similar concept is used by implementing hundreds of small AC-DFA on a massively parallel graphics multi-processor (GPU). While these massively multi-threaded implementations achieve a relatively high aggregated throughput, their *per-stream throughput* (number of bytes matched per second per input stream) remains very low. Furthermore, each of these implementations is optimized for a particular processor (CPU or GPU) microarchitecture, making the solution less applicable for the generic multi-core system.

An alternative AC-DFA partitioning approach to this paper was proposed in [6] and extended in [8], targeted for FPGA/ASIC implementations. In both solutions, the root and/or level-1 states are removed from the AC-DFA and processed as an NFA; the remaining AC-DFA states are packed and accessed in a hash table. The goal is to reduce the number of (backward) transitions in the AC-DFA and to improve the memory efficiency of the STT. On processor-based platforms, however, the use of hash functions for every state transition can be computationally expensive. The hashed state accesses may also induce poor cache performance and introduce hash conflicts.

III. Head-Body Partitioning

A. Motivation

Algorithmically, dictionary-based string matching (DBSM) can be solved by the Aho-Corasick deterministic finite automaton [19] (AC-DFA) with minimum asymptotic space and time complexities.[1] Practically, the state transition table (STT) of an AC-DFA is either fully populated by all state transitions (current state number + input character) and directly indexed by the state number, or accessed through a hash function to a (hash) table storing only the valid state transitions.

[1]Both the space and time complexities of an AC-DFA are linearly proportional to the number of states in the AC-DFA.

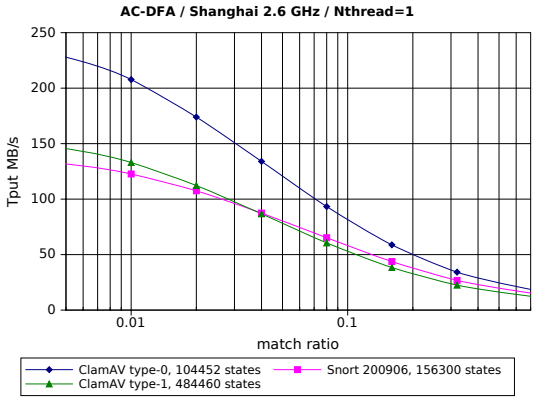

Figure 1. Single-threaded throughput of fully populated AC-DFA vs. match ratio on 2.6 GHz AMD Opteron processor (Shanghai).

Both fully-populated and hash-table STTs, however, result in poor memory and throughput performance for large-scale DBSM on multi-core platforms:

1) When fully populated by all state transitions, the STT of an AC-DFA can become excessively large and result in poor cache efficiency. Assume an 8-bit (1-byte) alphabet is used. Each state will have 2^8 = 256 out-going state transitions, occupying 256×4 = 1k bytes if 4 bytes are used to encode each target state. Thus, a state-of-the-art 10 MB on-chip cache can hold less than 1000 states. On the other hand, a large-scale DBSM with thousands of patterns can easily require an AC-DFA with hundreds of thousands of states.
2) When implemented as a hash table for better memory efficiency, every access to the STT requires potentially time-consuming hashing. Suppose it takes 30 CPU cycles to compute the hashed address,[2] plus 3 cycles to detect hash conflicts and another 12 cycles to compare the transition label and to set the next state variables. On a 2.6 GHz processor core, the best-case performance will be upper-limited by $2600/(30+15) \approx 58$ MBytes/sec, even if *zero* hash conflict or cache miss occurs. (In contrast, a non-hashed STT can sustain $2600/12 \approx 217$ MBytes/sec best-case performance.)

For AC-DFA with a fully populated STT, a few percent of match ratio (# pattern bytes / # total bytes) in the data input can dramatically reduce the DBSM throughput to a fraction of its best case values, as shown in Figure 1. Note that while the absolute throughput depends heavily on the type and size of the dictionary, the decreasing trend of throughput versus match ratio is similar in all three cases.

On the other hand, for AC-DFA with a hash table STT, the DBSM throughput is upper-limited by a relatively low value (<60 MB/s), and can be significantly lower when the input stream continually cause hash conflicts. Both approaches result

[2]A fast hash table lookup function can be found at http://burtleburtle.net/bob/hash/doobs.html, where 30+ clock cycles is needed per hash table access in the best case.

in poor overall performance and create opportunities for the denial-of-service (DoS) type of attacks.

To counter these performance bottlenecks, we propose partitioning the AC-DFA into two parts: a small (in terms of number of states) "head" which constantly filters the input stream for potential pattern matches, and a much larger but more compact "body" which quickly finds out any pattern match from the "head" output. The idea is to make the head part small so it may be fully-populated, and to make the body part simple so it can be packed and accessed efficiently without the use of expensive hashing.

B. Construction of H-DFA and B-NFA

The head-body partitioning consists of the construction of the head DFA (H-DFA) and the body NFA (B-NFA) from an Aho-Corasick DFA (AC-DFA). We will first describe the H-DFA construction. Let S be the dictionary (a set of strings) used to construct the AC-DFA. We say P is a set of *compatible prefixes* of S if P consists of a set of prefix strings, one of every string in S, where no string in P that is a proper prefix in S, is also a proper suffix of another string in P.

Definition 3: A set of *compatible prefixes* P of the dictionary S is defined as follows:

$$\forall p \in P:\ \exists s \in S, i \in \mathbb{N} \ni p = \mathrm{Prefix}\,(s,i)$$
$$\forall s \in S:\ \exists p \in P, i \in \mathbb{N} \ni p = \mathrm{Prefix}\,(s,i)$$
$$\forall p_1, p_2 \in P, s \in S:\ p_1 \in \mathrm{Prefix}\,(s,*) \Rightarrow p_1 \notin \mathrm{Suffix}\,(p_2,*)$$

where $\mathrm{Prefix}\,(s,i)$ is the length-i prefix of s; $\mathrm{Prefix}\,(s,*)$ and $\mathrm{Suffix}\,(s,*)$ are the set of proper prefixes and suffixes of s, respectively.

Proposition 1: A simple way to obtain a set of compatible prefixes P from an arbitrary dictionary S is as follows:

1) Start with an empty P; decide a head length l_H.
2) For each string $s \in S$,
   a) If $\mathrm{length}\,(s) > l_H$, then add $\mathrm{Prefix}\,(s, l_H)$ into P.
   b) If $\mathrm{length}\,(s) \leq l_H$, then add s itself into P .
3) The resulting P is a set of compatible prefixes of S.

Proof: All strings in P have the same length l_H except those that are *not* proper prefixes of some string in S. Thus all requirements of Definition 3 are satisfied and P is a set of compatible prefixes of S. ∎

After finding P, the H-DFA can be constructed by running the Aho-Corasick algorithm on P.

Definition 4: Given an AC-DFA $(Q, \Sigma, q_0, \delta, M, \lambda, g, f)$, a head DFA (H-DFA) of the AC-DFA is a 6-tuple, $(Q_H, \Sigma, q_0, \delta_H, M_H, Q_R)$, consisting of the following:

1) The same alphabet Σ as that of the AC-DFA
2) The same start state q_0 as that of the AC-DFA
3) A finite set of head states $Q_H \subset Q$, such that
   $\forall q_a \in Q_H:\ q_b \in \mathrm{ShortestPath}\,\{q_0, q_a\} \Rightarrow q_b \in Q_H$
4) A head transition function $\delta_H:\ Q_H \times \Sigma \rightarrow Q_H$
5) A finite set of head matches $M_H = \{Q_H \cap M\}$
6) A finite set of body roots $Q_R \subset \{Q_H \backslash q_0\}$, such that
   $\forall q \in Q_R:\ g\,(q,*) \neq \phi \ and\ g\,(q,*) \nsubseteq Q_H$
   $\forall \{q_a, q_b\} \in Q_R, i \in \mathbb{N}:\ q_a \neq f^i\,(q_b)\ and\ q_b \neq f^i\,(q_a)$

where $\mathrm{ShortestPath}\,\{q_0, q\}$ is the set of states on the shortest path from q_0 to q in the AC-DFA, and $g\,(q,*)$ is the set of possible target states of the goto function at state q.
Note that the first 5 items above specify a DFA, while the last item requires the DFA to be constructed from a compatible set of prefixes. This requirement guarantees that no two body roots can be reached at the same time by any input stream; *i.e.*, it guarantees the uniqueness of any B-NFA invocation by the H-DFA.

Next we describe the construction of the B-NFA. Let P be a set of compatible prefixes of S. We say T is the *set of compatible suffixes for P with respect to S* if concatenating every string in P by a corresponding string in T produces the dictionary S.

Definition 5: Given a dictionary S and a set of compatible prefixes P of S, a set of strings T is called *the set of compatible suffixes for P with respect to S* if and only if

$$\forall s \in S:\ \exists p \in P, t \in T \ni s = p \cdot t$$

where $p \cdot t$ means the concatenation of p and t.
Given an AC-DFA constructed from a dictionary S and an H-DFA constructed from some compatible prefixes P of S, the corresponding B-NFA can be constructed by extending the set of body roots of the H-DFA by the strings in T, where T is the set of compatible suffixes for P with respect to S. This results in a multi-tree data structure for the B-NFA, as described below.

Definition 6: Given an AC-DFA $(Q, \Sigma, q_0, \delta, M, \lambda, g, f)$ and an H-DFA $(Q_H, \Sigma, q_0, \delta_H, M_H, Q_R)$, the corresponding body NFA (B-NFA) is a 5-tuple, $(Q_B, Q_R, \Sigma, g_B, M_B)$, consisting of the following:

1) The same alphabet Σ as that of the AC-DFA
2) The same set of body roots Q_R as that of the H-DFA
3) A finite set of body states $Q_B = \{Q \backslash Q_H\} \cup Q_R$
4) A body goto function $g_B:\ Q_B \times \Sigma \rightarrow \{Q_B \backslash Q_R\}$ and
   $\forall q \in Q_B, c \in \Sigma:\ g_B\,(q,c) = q' \Rightarrow g\,(q,c) = q'$
5) A finite set of body matches $M_B = \{M \backslash M_H\}$

Topologically, the B-NFA described above consists of a number of *body-trees*, each headed by a body root in Q_R. Every body state represents a node in exactly one of the body-trees and is reachable only from the root of that body-tree through a series of goto functions (*i.e.*, forward AC-DFA transitions).

The operations of H-DFA and B-NFA can be outlined as follows. The H-DFA receives a stream of data input, starting at state q_0, and makes state transitions just like an ordinary AC-DFA. Whenever a body root is reached in the H-DFA, however, the body root and the input position are recorded and sent to the B-NFA to invoke a body matching. Starting from the body root, the B-NFA matches the subsequent data input against the labels of a series of goto functions in the body-tree traversal. The body matching terminates when the goto function does not produce a valid next state for the next input character. Concurrent to the body matching, the H-DFA continues to process the subsequent data input and invokes a next body matching if another next body root is reached, even

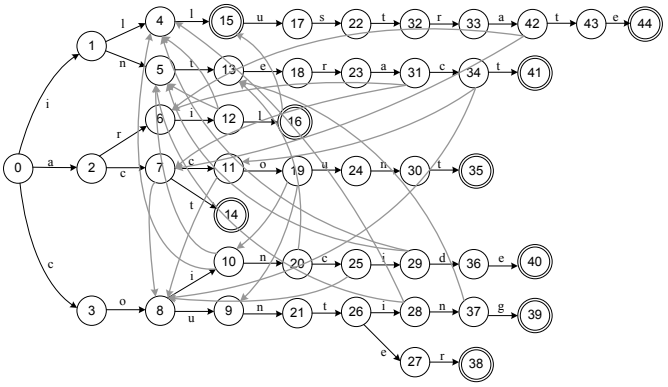

Figure 2. An example AC-DFA.

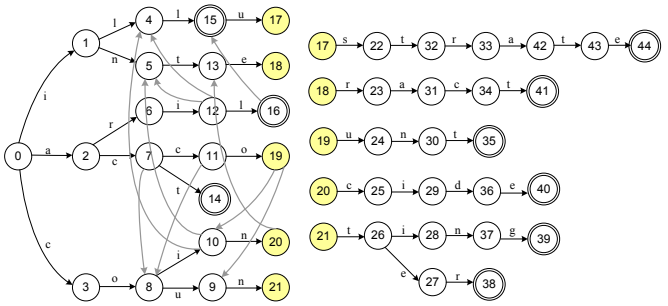

Figure 3. The head (left) and body (right) parts for the example AC-DFA.

before any (previous) body matching terminates. Any head or body matches visited during the operations above are recorded and reported as match output.

C. Illustrative Example

Given a dictionary S = {`aril`, `act`, `account`, `interact`, `illustrate`, `ill`, `counting`, `counter`, `coincide`} with the Roman alphabet, the Aho-Corasick DFA (AC-DFA) can be constructed as shown in Figure 2. For clarity, we do not show transitions back to states at depth 0 or depth 1. We also omit the labels on the cross transitions (gray lines) since they are the same as the forward transitions that reach the same target states. The match states (upon which a string match is found) are represented by double-circles.

The AC-DFA can be partitioned into a head part and a body part, as shown in Figure 3. The head part consists of an Aho-Corasick DFA built upon a set of prefixes P of S, P = {`aril`, `act`, `acco`, `inte`, `illu`, `ill`, `coun`, `coun`, `coin`}. In this example, P is obtained simply by taking all length-4 prefixes of S. For those strings in S shorter than 4 characters, the string itself is put in P. The resulting head DFA (H-DFA) is shown on the left side of Figure 3. Note that a few states in the H-DFA are shaded: they are the body roots, which represents states after which a remaining suffix should follow.

The body part is constructed by extending the body roots by the remaining suffixes in multiple body-tree structures, as shown on the right side of Figure 3. Note that there is a one-to-one mapping from the body roots in the head part to the body-trees in the body part.

The string matching begins at the head part as a sequence of ordinary DFA transitions. Assume the input stream consist of "`accountillustrate`". The state transitions in the head part will be $0 \xrightarrow{\texttt{a}} 2 \xrightarrow{\texttt{c}} 7 \xrightarrow{\texttt{c}} 11 \xrightarrow{\texttt{o}} 19 \xrightarrow{\texttt{u}} 9 \xrightarrow{\texttt{n}} 21 \xrightarrow{\texttt{t}} 0 \xrightarrow{\texttt{i}} 1 \xrightarrow{\texttt{l}} 4 \xrightarrow{\texttt{l}} 15 \xrightarrow{\texttt{u}} 17 \xrightarrow{\texttt{s}} 0 \ldots \xrightarrow{\texttt{a}} 2 \xrightarrow{\texttt{t}} 0 \xrightarrow{\texttt{e}} 0$. At each of the states 19, 21, and 17, the corresponding body-tree in the body part will be invoked to match the following input. Note that both "`account`" and "`illustrate`" will be matched in the body part, while "`ill`" will be matched in the head part.

Although a transition is made from H-DFA to B-NFA whenever a body root is reached, *no* transition is possible from B-NFA back to H-DFA. In fact, due to its non-deterministic nature, the B-NFA does not have any backward or cross-tree transition at all. As discussed in Section IV-C, this "forward-only" property of B-NFA allows the body part to be stored compactly and accessed efficiently in a cache-friendly data structure.

As can be seen in the example, the main point of the head-body partitioning is to reduce the AC-DFA into a small head DFA and a set of simple body-trees. The head DFA can be small enough to achieve good processor cache locality, while the simple structures of the body-trees can be exploited for both processing and storage efficiency.

D. Head-Body Performance Analysis

For convenience and clarity, whenever not explicitly stated in this section, we always assume the following definitions:

- Aho-Corasick DFA, AC-DFA: $(Q, \Sigma, q_0, \delta, M, \lambda, g, f)$
- Head DFA, H-DFA: $(Q_H, \Sigma, q_0, \delta_H, M_H, Q_R)$
- Body NFA, B-NFA: $(Q_B, Q_R, \Sigma, g_B, M_B)$

1) Length of Failure Chains: In B-NFA, multiple matching instances can be active (each invoked by an active body root in H-DFA) concurrently while the H-DFA is running. Since the time complexity of the head part (H-DFA) is the same as any DFA, the additional processing of the body part makes DBSM with the head-body approach algorithmically more complex than the AC-DFA. In this subsection we prove that the complexity of the B-NFA is bounded by the maximum length of the failure chains that completely reside within the B-NFA.

Definition 7: The *failure chain of length* n *from state* $q \in Q$ of an AC-DFA, $F^n(q)$, consists of the set of states $F^n(q) = \{q, f(q), \ldots, f^n(q)\}$. $F^n(q)$ is said to *reside in the B-NFA* if and only if $F^n(q) \subset Q_B$.

Lemma 1: Two states in a B-NFA are active concurrently if and only if they are in the same failure chain.

Proof: According to the original Aho-Corasick algorithm, two states in an AC-DFA are connected by a failure function if and only if they can be reached using the same input stream starting at different offsets. Let the two states be $q_x \in Q_B$ and $q_y \in Q_B$, $q_x \neq q_y$. Since the same input stream, starting at two different offsets, can reach both q_x and q_y concurrently, it follows that there must be two active body matching (invoked by the H-DFA separately), one for each of the reached states q_a and q_b. ■

Direct applications of Lemma 1 to all adjacent pairs of states in a failure chain gives the following result.

Corollary 1: The processing complexity of a B-NFA for any input character is upper bounded by the maximum length of the failure chains that reside in the B-NFA.

Note that although algorithmically the complexity is bounded by the *maximum* length of the failure chains, this worst-case is rarely relevant due to the following reasons:

1) In order to continuously invoke the longest failure chain, a small number of states must be repeatedly visited, resulting in good state locality and fast processing speed in both the head and the body parts.
2) Further analysis of the Snort dictionary shows that nearly all the long (> 8) failure chains are caused by a small number (< 100) of strings with shortly repeating patterns, such as "aaaaaa..." or $\{00\ 00\ 00\ \ldots\}_h$. These patterns can be extracted from the dictionary and matched specially.

As shown in Figure 4, the number of failure chains decreases sharply with the increasing length of the failure chain as well as the depth of the body roots.

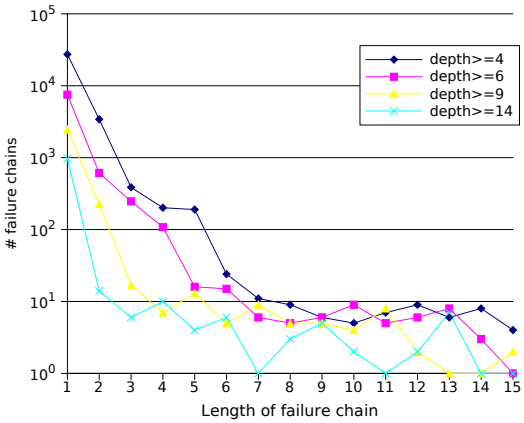

Figure 4. Number of failure chains versus length above various depth in Snort dictionary's AC-DFA.

In practice, for a large dictionary with thousands of patterns, the worst-case performance is induced when a large number of states in many long failure chains are visited in turn, causing not only high processing complexity but also poor state locality. The severity of the worst case scenario can be better estimated by the accumulated length of all failure chains. A detailed model for the complexity of DBSM is out of the scope of this paper. Instead, in Section V we experimentally measure the performance of DBSM under various conditions with both the head-body and the AC-DFA approaches.

2) State Out-degree: A good feature of the B-NFA is its tree structure, versus a more generalized DFA structure of the AC-DFA. One of the most important properties of a tree structure is its out-degrees. Both the memory complexity to store a tree and the time complexity to traverse a tree are positively proportional to the out-degree of the tree nodes. In the case of B-NFA, the out-degree of each state can be found using the following definition.

Definition 8: The state out-degree, $d(q)$, of state $q \in Q$ is the number of possible target states of the goto function at state q. That is, $d(q) = |g(q, *)|$.

As shown in Figure 5, the number of states with high out-degrees decreases exponentially with respect to the state depth. More than 95% of the states at depth 4 or higher have an out-degree of 2 or less. This means that on the average, an AC-DFA state at depth 4 or higher should require no more than 2 bytes for label storage or 2 operations for label comparison.

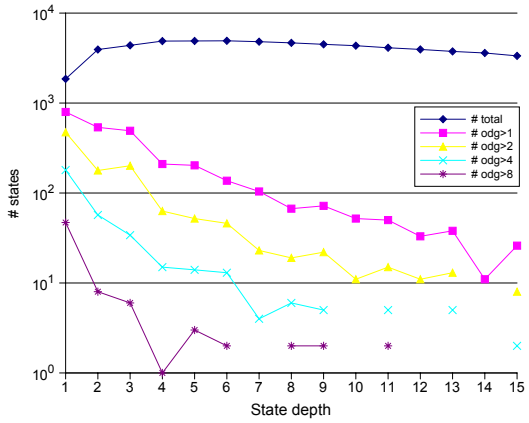

Figure 5. Number of states versus state depth with various out-degrees in Snort dictionary's AC-DFA.

This finding shows that both fully-populated and hash-table STTs are inefficient for storing and accessing the AC-DFA for large-scale real-world DBSM (such as Snort). On the other hand, the head-body partitioning can take advantage of this out-degree statistics by storing the lower-depth states in a fully-populated but relatively small head DFA, and the higher-depth states in a compact body NFA.

3) Memory Efficiency: Due to the lack of "cross transitions" in the tree structure of B-NFA, any state in the body-tree can have only one previous (parent) state. A corollary to this property is that all the next (children) states of a body state can always be placed consecutively.

Thus, instead of performing a complex hash operation to find the next state while risking the chance of hash conflict, in a B-NFA the next (child) state address of any state can be found by simply taking an non-negative offset to its first child. This offset can be found by comparing the input character with an ordered list of transition labels. Based on the discussion in Section III-D2, a B-NFA state should on the average require no more than 2 bytes for storing labels and 4 bytes for storing the first next-state address, giving a memory efficiency 2 orders better than the H-DFA and AC-DFA.

As is shown in Section IV-C, the memory efficiency can be further improved by packing multiple body-tree paths into one body-tree "branch," taking advantage of the cache block size and the single-instruction multiple-data (SIMD) instructions on modern processors.

IV. Design of HBM

A. Overall Architecture

The top-level entity of our head-body implementation is the *head-body matching (HBM) module*, which consists of a *head module* as an H-DFA and a *body module* as a B-NFA.

Given a string dictionary, the main parameter to an HBM module is the set of compatible prefixes to be used for the H-DFA. In our implementation we use the simple method of Proposition 1 to find the set of compatible prefixes, reducing the parameter to a single prefix length value. This prefix length determines the state depth of all the body roots, where the H-DFA and the B-NFA interface with each other.

Although conceptually the H-DFA and the B-NFA run in parallel, for programming simplicity the head and body modules are executed serially (in a single program thread) in our HBM module. Initially, the program processes the input stream using the head module. Whenever a body root is reached, the program records the body root's state number and the position of the next input character. It then branches to the body module to match the subsequent data input beginning at the reached body root. After the body module terminates, the program returns to the head module, restores the previously recorded state number (the body root), and continues the matching from the recorded input position.

Our HBM module records both the state number and the input character position of all the dictionary matches in a queue, which grows dynamically to accommodate large number of matches efficiently. The size of the queue is upper-bounded by a preset value (12 million elements, for example), after which it becomes a ring where the earliest elements are erased to hold new insertions.

B. The Head Module

The head module consists of an H-DFA for the dictionary and a pre-defined *head length* (the l_H in Proposition 1). It is implemented as a DFA with $|Q_H| < 65536 = 2^{16}$ states in a fully-populated STT. Using the notations in Definition 4, there are $|Q_H|$ rows and $|\Sigma|$ columns in the STT, one row for each head state and one column for each input label.

The string matching is performed byte-wise, resulting in the input width of 8 bits and the alphabet size of $2^8 = 256$. The size of the H-DFA's STT is thus $|Q_H| \times 512$ bytes, where every next-state value occupies 2 bytes (16 bits) of storage.

The head length l_H must be 3 (characters) or larger, up to the maximum value where the number of head states remains below 65536. That is,

$$3 \leq l_H \leq \max_l \left(|s \in Q : \ \lambda(s) \leq l| < 65536\right)$$

To make it easier to identify a match state ($s \in M_H$) and a body root ($s \in Q_R$), we organize all head states into 3 ranges:

1) Non-match states
2) Match states
3) Body roots

A non-match state in the H-DFA can be neither a match state nor a body root. On the other hand, a state can be both a match state and a body root at the same time. This is resolved by allowing an overlap between the range 2 and the range 3. It takes two branches to identify what range(s) a state belongs to in the H-DFA, compared to just one branch to find out the state's match status in an ordinary AC-DFA. This overhead makes the H-DFA slightly slower than the AC-DFA in the best case. The difference, however, is quite small on a modern multi-core processor.

C. The Body Module

The body module consists of a number of body-trees each headed by a body root. We design a simple yet compact data structure to store the body-trees in blocks of 64 bytes, which match the cache block size of most modern processors. Each 64-byte block is called a *branch*, which contains one or more body paths of multiple bytes in length. The data structure, shown in Table I, is described as follows:

Table I
THE BODY BRANCH DATA STRUCTURE.

0	1	2	3	4 5 6 7	8 9 a b	c d e f
b	d	n		next_state	mat_mask	misc_info
input_mask						
path_labels[1]						
path_labels[2]						

branch width (b) Number of paths stored in the branch. Valid values include 1, 2, 4, 8, 16, or 32.

stride size (d) Maximum stride of each path. Valid values include 1, 2, 4, 8, 16, or 32.

max offset (n) Total number of paths that lead to a valid next branch.

next_state State number of the first valid next branch.

mat_mask Bit-mask of dictionary pattern matches. One bit for each byte in the path_labels vector.

misc_info Miscellaneous information about the branch.

input_mask Masking the input vector before comparing it to the path_labels vector.

path_labels Up to 32 bytes of path labels containing from a single 32-byte path to 32 one-byte labels.

Depending on the values of b (branch width) and d (stride size) of a body branch, one of the 11 *body functions* is called to match d characters from the input against the (up to) b paths stored in the path_labels array. By design, the value of $b \times d$ is either 16 or 32. If $b \times d$ = 16, then only path_labels[2] is used for storing tree paths; if $b \times d$ = 32, then both path_labels[1] and path_labels[2] are used for storing tree paths. The input_mask is responsible of handling paths that are shorter than the stride size of the branch. These shorter-than-branch-stride paths always represent dictionary matches.

The matching is performed efficiently using the single-instruction multiple-data (SIMD) instructions available on modern processors (*e.g.*, x86-64 compatible CPUs from both AMD and Intel). In order to do so, the d input characters read by the branch are first replicated b times to form an *input vector* of the same length as the path_labels vector (either 16 or 32 bytes, depending on the value of $b \times d$). Then a

number of SIMD instructions are performed on the input and the path_labels vectors to obtain the following results:

1) All paths in the path_labels vectors that generate one or more pattern matches for the dictionary (with the help from the mat_mask bit-mask)
2) The offset, if any, of the path in the path_labels vector that leads to a valid next branch

The first result above causes the body part to record the input position and the matching states. The second result causes the body part to traverse to the next body branch.

On the average, the processing in each body branch has 25–35 instructions (SIMD or not), taking 20–30 clock cycles on a modern superscalar processor. While this is not much faster than a hash table lookup, we note that the out-degree of the body states are usually small, allowing each branch to process multiple (up to 32) characters very often. Furthermore, our B-NFA optimizations are applicable to all modern superscalar processor with SIMD capabilities, including all x86-64 compatible CPUs from AMD and Intel.

D. Parallelism on Multi-Core

To utilize all available cores in the multi-core platforms, we exploit the data parallelism of DBSM to scale up the aggregated throughput to multiple threads. To generate meaningful and realistic results, we use multiple data input streams, one for each thread, in separate memory storage. 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.

This parallel processing approach allows us to scale the number of software threads easily to the number of cores in a shared-memory multi-core system. While it is possible to perform instruction-level interleaving as in [12] to hide the memory access latency for both H-DFA and B-NFA, doing so in an optimal way requires assembly-level manipulation of the source code for each particular platform. Simple interleaving in the C/C++ source level produces lower performance. Unlike [12], our solution is not targeted at any particular platform. We believe that future processor technologies such as hardware multi-threading may be able to hide the memory latency more efficiently (with respect to development effort).

V. Performance Evaluation

A. Environments and Datasets

We measure the performance of our head-body matching (HBM) modules on both AMD Opteron and Intel Xeon platforms. Table II compares the processor specifications. All servers are equipped with 16 GB or more DDR2 667 MHz main memory. Due to the space limit, we only show the results on the two AMD processors (to compare performance between two processor generations with the same system architecture). However, we note that the performance results on Clovertown is very similar to that on Barcelona, with slightly (2%–3%) greater advantage towards HBM due to Clovertown's larger cache size.

Table II
Processors for performance evaluation.

	Barcelona	Shanghai	Clovertown
CPU	Opteron 2350	Opteron 2382	Xeon 5335
# Cores	4 (8/server)	4 (8/server)	4 (8/server)
Cache	4 MB L2+L3	8 MB L2+L3	8 MB L2
Freq.	2.0 GHz	2.6 GHz	2.0 GHz

The source code is compiled using GCC 4.2 and run on 64-bit SuSE Linux (kernel version 2.6.18). We use either "`gcc -O2`" or "`g++ -O2`" to compile the source codes, depending on the language used in the source files.

We use three dictionaries from popular intrusion detection (Snort) and virus scanning (ClamAV) applications for performance evaluation. Table III compares the statistics of the dictionaries and the size of their state transition tables (STT) when implemented as an AC-DFA and various HBM configurations. The latest Snort dictionary as of June 2009 (Snort0906) consists of 8,673 words in 196,967 characters. In the ClamAV "main.ndb" file, the type-0 strings (CAV-0) consist of 1,444 words in 109,686 characters, whereas the type-1 strings (CAV-1) consist of 5,225 words in 497,984 characters. The HBM-{1–4} rows correspond to the four HBM configurations in Figure 6 and in Figure 7, respectively. The values in the HBM rows describe the memory size used by the respective H-DFA and B-NFA modules.

As shown in Table III, the HBM modules effectively compact the dictionary into a much smaller memory than the AC-DFA. The H-DFA is essentially a DFA with less number of states (only the "head" of the AC-DFA), so its memory size is directly proportional to the number of states it contains. The B-NFA, on the other hand, compacts up to 32 single-character state transitions into a 64-byte memory block, resulting in a memory footprint as small as 2 B/state. In practice, the memory footprints of the B-NFA for these real-world dictionaries vary from 2.5 B/state to 4 B/state.

As shown in Figure 1, the match ratio of the data input has a great impact on the performance of the dictionary-based string matching (DBSM) running on processors. In our experiments, we control the match ratio by embedding the dictionary strings randomly into an "innocent" data input template. For the Snort dictionary, we use the plain text of the entire King James Bible as the input stream template; for the ClamAV type-0 and type-1 dictionaries, we use the concatenation of all files under `/usr/bin` in a typical Linux server installation.

A match ratio of 0.01 means the embedded dictionary patterns amount to 1% of the input characters (for example, there is a single 15-byte dictionary pattern in a 1500 B network packet). This would represent an occasional occurrence of intrusion-like pattern in the network traffic. A greater match ratio, *e.g.*, 0.3 or higher, would represent a performance-based denial-of-service attack where 30% or more of the network packets are sent by the attacker to slow down the system.

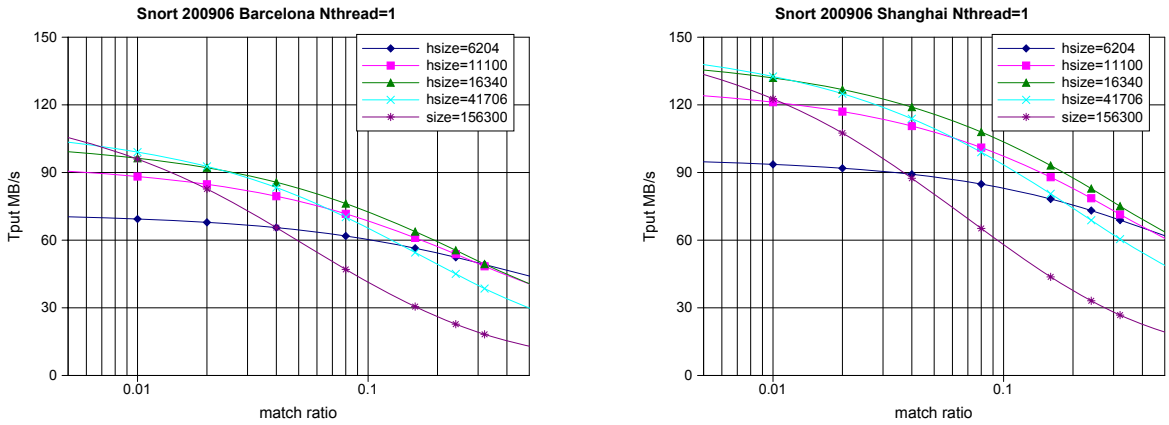

Figure 6. Throughput of AC-DFA and HBM with various head sizes for Snort0906 on Barcelona (left) and Shanghai (right).

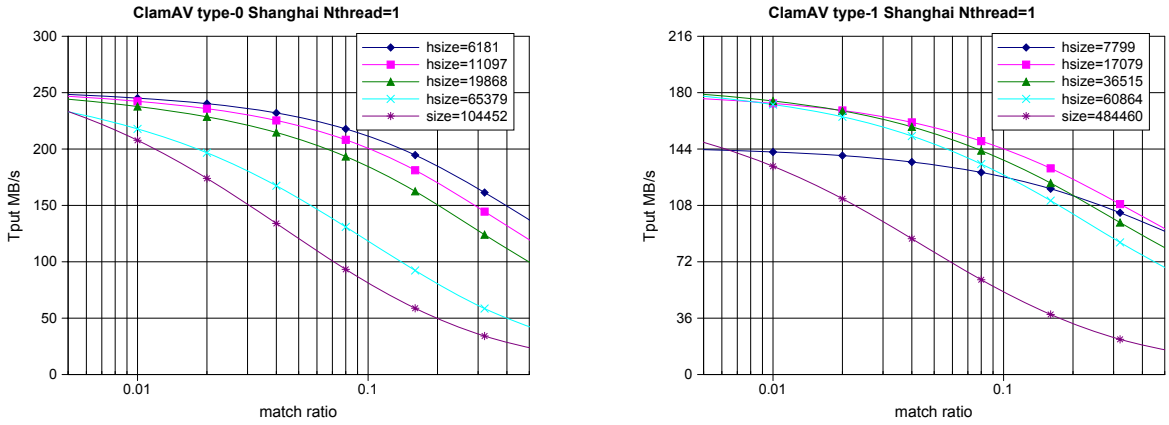

Figure 7. Throughput of AC-DFA and HBM with various head sizes for CAV-0 (left) and CAV-1 (right) on Shanghai.

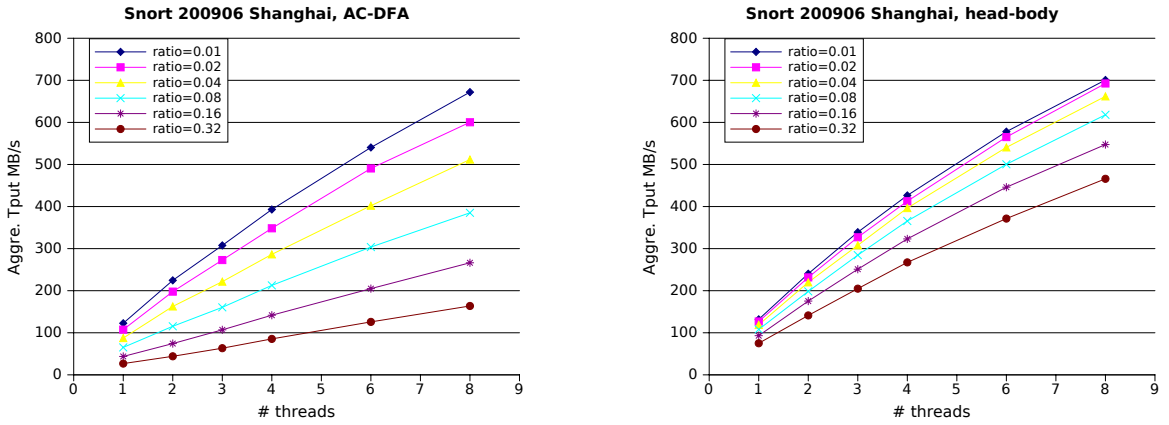

Figure 8. Aggregated throughput of AC-DFA (left) and HBM with 16,340 head states (right) for Snort0906 on Shanghai.

Table III
STATISTICS OF THE SNORT AND CLAMAV DICTIONARIES AND THE RESPECTIVE SIZES OF THE HBM AND AC-DFA STTS.

	Snort0906		*CAV-0*		*CAV-1*	
# strings	**8,673**		**1,444**		**5,225**	
# chars	**196,967**		**109,686**		**497,984**	
	H-DFA size	*B-NFA size*	*H-DFA size*	*B-NFA size*	*H-DFA size*	*B-NFA size*
HBM-1	**3.0** MB	**0.65** MB	**3.0** MB	**0.23** MB	**3.8** MB	**1.09** MB
HBM-2	**5.5** MB	**0.58** MB	**5.4** MB	**0.22** MB	**8.3** MB	**1.08** MB
HBM-3	**8.0** MB	**0.57** MB	**9.7** MB	**0.21** MB	**18** MB	**1.04** MB
HBM-4	**20** MB	**0.43** MB	**32** MB	**0.11** MB	**30** MB	**1.01** MB
AC-DFA	**153** MB		**102** MB		**473** MB	

B. Performance on Various Processors

Figure 6 shows the matching throughput versus match ratio for the Snort dictionary. Four different H-DFA sizes (with 6,204 / 11,100 / 16,340 / 41,706 states) are used for four HBM configuration, contrasted by an optimized AC-DFA with total 156,300 states. We repeat the same set of experiments on both the Barcelona-based and the Shanghai-based servers to see how processor speed affects the throughput of DBSM with AC-DFA and various HBM configurations.

All the HBM configurations (except "hsize = 6204", or HBM-1 in Table III) achieve higher matching throughput than the AC-DFA for match ratios above 0.01. When the H-DFA is too small, however, body matching is invoked excessively even without any matched pattern in the input stream, causing relatively lower HBM performance at low match ratios. Increasing the H-DFA size greatly improves the best-case throughput of HBM while only marginally decreases the throughput when the system is under attack. Using a "moderate" head size between 10 K to 20 K states, an HBM module can achieve 1.2x to 3x the throughput of AC-DFA for match ratios varying from 0.02 to 0.32.

Also note that HBM performs increasingly better than AC-DFA on the faster Shanghai processor than on the older Barcelona processor. Compared with AC-DFA, HBM's throughput performance depends more on the CPU processing power than on the system memory latency. Since the speed of processor increases at a much faster pace than that of memory, we expect the DBSM throughput to scale better with HBM than with AC-DFA to future generations of multi-core processors.

While the on-chip cache of the Shanghai processor is twice as large as the Barcelona processor, on both servers the HBM-3 configuration in Table III (the "hsize = 163,400" label in Figure 6) produces the best performance. This shows that although head-body partitioning helps improve processor cache performance, the throughput of HBM is not very sensitive to the on-chip cache size.

C. Performance with Various Dictionaries

Figure 7 shows the matching throughput versus match ratio of two dictionaries, ClamAV type-0 and ClamAV type-1. These dictionaries have dramatically different sizes in terms of both number of strings and number of characters.

While the absolute matching throughput in the two plots in Figure 7 are very different (notice their different y-axis scales), they both follow the same trend with increasing input match ratio. All HBM configurations perform consistently better than AC-DFA across the entire range (0.01–0.3) of input match ratios. Furthermore, HBM enjoys greater performance advantage over AC-DFA for the larger dictionary (ClamAV type-1). This shows that the proposed head-body partitioning effectively alleviates the cache pressure incurred by the large STT access of the AC-DFA.

In general, to optimize throughput over all input match ratios, we should use a relatively small H-DFA for a small dictionary, and gradually increase the size of the H-DFA when the dictionary becomes larger. Note that the "size" of a dictionary depends on not only the total number of characters but also the number of strings in the dictionary. For example, although the number of characters in the Snort dictionary is closer to that in the ClamAV type-0 dictionary, the relative performance of various HBM configurations for Snort is more similar to that for ClamAV type-1. Compared with the Snort dictionary, ClamAV type-1 has almost 3 times the number of characters (497,984 vs. 196,967), but a similar number of strings (5,225 vs. 8,678).

D. Performance with Multiple Cores

Figure 8 shows the aggregated throughput scaling from 1 to 8 threads for the Snort dictionary. The AC-DFA is compared with an HBM module configured with 16,340 head states. Both AC-DFA and HBM scale well for all input match ratios between 0.01 and 0.32. This implies that the throughput reduction at high input match ratios is not due to insufficient memory bandwidth but rather the long memory access latency caused by higher cache miss rates.

We observed similarly good aggregated throughput scaling for all three dictionaries with the HBM-{1,2,3} configurations in Table III. Figure 9 shows the throughput scaling for matching ClamAV type-0 with HBM-3 and ClamAV type-1 with HBM-2, respectively. Due to space limitation, we could not show all 9 possible cases. In all cases, the matching throughput is penalized by less than 50% while the match ratio increases over 30x (from 0.01 to 0.32).

The good throughput scaling of all HBM configurations also shows that the performance of HBM is not sensitive to the (effective) size of the on-chip cache, since four threads on the same quad-core processor compete for the on-chip (level-3) cache.

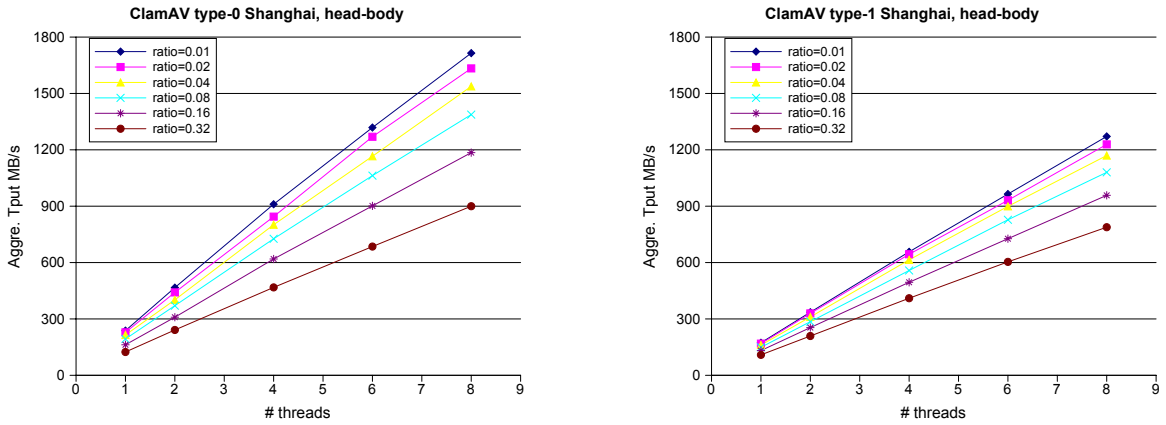

Figure 9. Aggregated throughput of HBM for CAV-0 with 19,868 head states (left) and for CAV-1 with 17,079 head states (right).

VI. Future Work

Currently, the head-body partitioning is performed statically, where the head part is bounded by a pre-defined depth value. In general, the ending states (body roots) in the head part do not need to have a constant or fixed depth value. One future direction is to design an algorithm that efficiently constructs a head part ending at "uneven" depths, to include a longer prefix of the "hot" dictionary patterns.

Because both H-DFA and B-NFA can be made small to a size comparable to the on-chip cache size, simultaneous multi-threading (SMT) may effectively scale up the aggregated throughput of the head-body approach. We are looking forward to test our HBM implementations on systems with hardware multi-threading support, such as Sun's Niagara 2 and Intel's Nehalem processors.

VII. Conclusion

We proposed and formally defined the head-body partitioned DBSM, which converts an AC-DFA into a small H-DFA and a memory-efficient tree-structured B-NFA. The H-DFA gives good cache performance on multi-core processors, while the B-NFA allows cache-friendly multi-stride transitions utilizing SIMD instructions. Unlike AC-DFA, large-scale DBSM with our head-body partitioned approach does not suffer from significant performance degradation when the match ratio increases in the input stream. The head-body approach also scales well to larger dictionaries and faster processors.

References

[1] "SNORT," http://www.snort.org/.

[2] "Clam AntiVirus," http://www.clamav.net/.

[3] 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.

[4] L. Tan and T. Sherwood, "Architectures for Bit-Split String Scanning in Intrusion Detection," in *IEEE MICRO*, vol. 26, no. 1, 2006, pp. 110–118.

[5] Z. K. Baker and V. K. Prasanna, "Time and area efficient pattern matching on FPGAs," in *Proc. of the 12th International Symposium on Field Programmable Gate Arrays (FPGA '04)*. ACM, 2004, pp. 223–232.

[6] J. van Lunteren, "High-performance pattern matching for intrusion detection," in *INFOCOM*, 2006.

[7] D. Pao, W. Lin, and B. Liu, "Pipelined Architecture for Multi-String Matching," in *IEEE Computer Architecture Letters*, vol. 7, 2008.

[8] T. Song, W. Zhang, D. Wang, and Y. Xue, "A Memory Efficient Multiple Pattern Matching Architecture for Network Security," in *IEEE INFOCOM*, 2008.

[9] I. Sourdis and D. Pnevmatikatos, "Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System," *Lector Notes in Computer Science*, vol. 2778, pp. 880–889, 2003.

[10] Y.-H. E. Yang and V. K. Prasanna, "Memory-Efficient Pipelined Architecture for Large-Scale String Matching," in *Proc. of 17th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2009.

[11] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit rate packet-matching using TCAM," in *IEEE Intl. Conf. on Network Protocols (ICNP '04)*, 2004.

[12] D. P. Scarpazza, O. Villa, and F. Petrini, "Exact Multi-pattern String Matching on the Cell/B.E. Processor," in *Proc. of 2008 Conf. on Computing Frontiers*, 2008, pp. 33–42.

[13] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High Performance Network Intrusion Detection Using Graphics Processors," in *Recent Advances in Intrusion Detection*, vol. 5230, 2008, pp. 116–134.

[14] O. Villa, D. Chavarria, and K. Maschhoff, "Input-independent, Scalable and Fast String Matching on the Cray XMT," in *Proc. of IEEE Int. Parallel & Distributed Proc. Sym.*, 2009.

[15] D. P. Scarpazza, O. Villa, and F. Petrini, "High-Speed String Searching against Large Dictionaries on the Cell/B.E. Processor," in *Proc. of IEEE Int. Parallel & Distributed Proc. Sym.*, 2008.

[16] R. Smith, C. Estan, and S. J. S. Kong, "Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata," in *ACM SIGCOMM*, August 2008.

[17] N. Yamagaki, R. Sidhu, and S. Kamiya, "High-Speed Regular Expression Matching Engine Using Multi-Character NFA," in *Proc. of International Conference on Field Programmable Logic and Applications (FPL)*, Aug. 2008, pp. 697–701.

[18] Y.-H. E. Yang, W. Jiang, and V. K. Prasanna, "Compact Architecture for High-Throughput Regular Expression Matching on FPGA," in *Proc. of 2008 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, November 2008.

[19] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[20] Q. Wang and V. K. Prasanna, "Pipelined Multi-Core Architecture on FPGA for Large Dictionary String Matching," in *Proc. of 17th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2009.

[21] S. Antonatos, K. G. Anagnostakis, and E. P. Markatos, "Generating Realistic Workloads for Network Intrusion Detection Systems," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 1, pp. 207–215, 2004.