Exact Pattern Matching with Feed-Forward Bloom Filters

Iulian Moraru* David G. Andersen*

Abstract

This paper presents a new, memory efficient and cache-optimized algorithm for simultaneously searching for a large number of patterns in a very large corpus. This algorithm builds upon the Rabin-Karp string search algorithm and incorporates a new type of Bloom filter that we call a *feed-forward Bloom filter*. While it retains the asymptotic time complexity of previous multiple pattern matching algorithms, we show that this technique, along with a CPU architecture aware design of the Bloom filter, can provide speedups between 2× and 30×, and memory consumption reductions as large as 50× when compared with `grep`.

1 Introduction

Matching a large corpus of data against a database of thousands or millions of patterns is an important component of virus scanning [18], data mining and machine learning [1], and bioinformatics [19], to name a few problem domains. Today, it is not uncommon to match terabyte or petabyte-sized corpuses or gigabit-rate streams against tens to hundreds of megabytes of patterns.

Conventional solutions to this problem build an exact-match trie-like structure using an algorithm such as Aho-Corasick [3]. These algorithms are asymptotically optimal: matching n elements against m patterns requires only $O(m + n)$ time. In another important sense, however, they are far from optimal: the per-byte processing overhead can be high, and the DFAs constructed by these algorithms can occupy gigabytes of memory, leading to extremely poor cache use that cripples throughput on a modern CPU. Figure 1 shows a particularly graphic example of this: When matching against only a few thousand patterns, GNU `grep` can process over 130 MB/sec (using an algorithm that improves on Aho-Corasick [11]). But as the number of patterns increases, the throughput drops drastically, to under 15MB/sec. The cause is shown by the line in the graph: the size of the DFA grows to rapidly exceed the size of CPU caches.

Un-cached memory accesses on modern CPUs are dauntingly expensive[1]. The Intel Core 2 Quad Q6600 CPU used in the above example with `grep`, for instance, is capable of sequentially streaming over 5GB/sec from memory and (optimistically) executing several billion instructions per second. The achieved 15MB/sec is therefore a disappointing fraction of the machine's capability.

Furthermore, there are situations when running a full-scale Aho-Corasick implementation is very expensive because memory is limited—e.g., multiple pattern matching on netbooks, mobile devices, embedded systems, or some low-power computing clusters [4]. Other applications, such as virus scanning, benefit from efficient memory use in order to reduce the impact on foreground tasks.

This paper makes two contributions that together can significantly boost the speed of this type of processing, while at the same time reducing its memory requirements. They both center around making more efficient use of the cache memory.

Feed-Forward Bloom Filters: Bloom filters [5] have previously been used to accelerate pattern matching by reducing the size of the input corpus before an exact matching phase (in this paper we refer to this exact matching phase as the "grep cleanup" phase). Feed-forward bloom filters reduce the size of the input corpus as well, but during processing, also record information about which patterns could have been matched. They then introduce a second pass filtering step where this information is used to reduce the number of patterns that must be handled during the "cleanup" phase. As a result, it reduces drastically the memory used for cleanup.

Cache-partitioned Bloom filters: A lookup in a typical Bloom filter involves computing k hash values for a query, and using these values as indices into a bit vector. Because the hash values must be randomly distributed for the filter to be effective, and since, for millions of patterns the bit vector must be substantially larger than the cache available on modern CPUs, Bloom filter implementations have poor cache performance. Our solution to this problem is to split the Bloom filter into two parts. The first part is smaller than the largest

*Carnegie Mellon University, Computer Science Department, Pittsburgh, PA 15213. E-mail: imoraru@cs.cmu.edu, dga@cs.cmu.edu

[1]On a 65 nm Intel Core 2 CPU, for example, a cache miss requires 165 cycles.

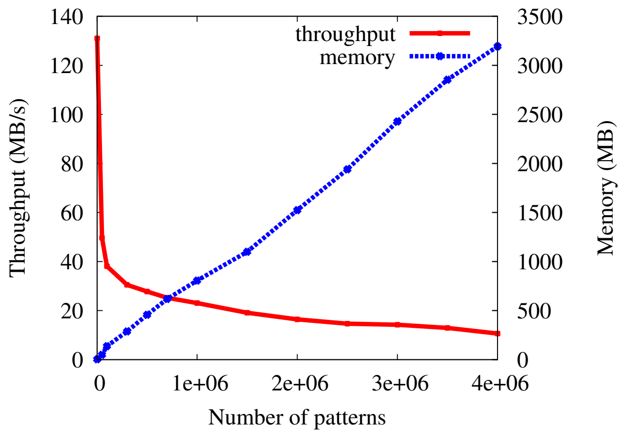


Figure 1: The `grep` processing rate and memory consumption for various numbers of patterns. The average length of the patterns is 29 characters.

CPU cache available (typically L2 cache) and is the only one accessed for the large majority of the lookups[2]. In consequence, it will remain entirely cache-resident. The second part of the filter is larger, but is accessed infrequently (e.g., for true or false positive queries). Its role is to keep the false positive rate small. The result is that the cache-partitioned Bloom filter is as effective as the classic Bloom filter, but has much better cache performance, and is as much as 5× faster, as a result.

We describe these techniques in section 3 and evaluate them in section 4. We show that pattern matching for highly redundant English text can be accelerated by 2× while consuming 4× less memory, while random ASCII text can be searched 37× faster with 57× less memory, when compared with `grep`.

2 Background and Related Work

2.1 Multiple Pattern Search. The classic multiple pattern search algorithm is Aho-Corasick [3]. It is a generalization of the Knuth-Morris-Pratt linear-time matching algorithm that uses a trie structure in which each node represents a state of a finite-state machine—for each input character, the automaton goes to the state that represents the longest prefix of any match that is still possible.

The popular GNU `fgrep` utility uses the Commentz-Walter algorithm [11] for multiple string search. It combines Aho-Corasick with the Boyer-Moore single pattern matching algorithm [6], which achieves sub-linear running time by skipping characters in the input text according to the "bad character" and "good suffix" heuristics. As illustrated in figure 1, the size of the DFA used by Aho-Corasick-like algorithms grows quickly with the number of patterns. This increases setup time (building the trie) and reduces search speed because of poor cache performance.

Another Boyer-Moore style algorithm for multiple pattern search is the Wu-Manber algorithm [22], employed by the `agrep` tool. It uses the "bad character" heuristic to skip over characters in the input text. The difference is that it does so not by comparing individual characters, but by comparing the hash values of groups of consecutive characters. This algorithm is most effective for relatively small numbers of patterns—hundreds to tens of thousands of patterns. For larger numbers of patterns, it becomes more memory-hungry and thus less cache-efficient. Lin et al. show that the Wu-Manber algorithm has worse cache performance and worse overall performance than Aho-Corasick as the number of patterns increases [18].

Complementary approaches to multiple pattern matching investigated the idea of encoding the text and the patterns using a compact scheme, such that a word comparison is equivalent to multiple symbol comparisons [15].

The inspiration for the work described in this paper is the algorithm that Rabin and Karp presented in [14]. The patterns—which must be all of the same length—are hashed and the hash values are inserted into a set data structure that allows for fast search (e.g. a Bloom filter, a hashtable or both a bit vector and a hashtable [20]). The actual search consists of a window—of size equal to the size of the patterns—slid over the input text, and a hash value being computed for the text in the window, at each position. This value is then searched in the set of hashes computed from the patterns. If found, it denotes a possible match, which needs to be checked by comparing the string in the current window with every pattern that has the same hash value as it. The average case running time for this algorithm is linear if the hash computations required when sliding the window are done in $O(1)$. This can be achieved by using a rolling hash function—i.e. the hash value for the current window is computed from the hash value of the previous window, the last character in the current window, and the first character in the previous window.

In this paper, we present several improvements to the basic Rabin-Karp technique. They enable fast and memory-inexpensive search for millions of patterns at once.

2.2 Bloom Filters. A Bloom filter [5] is a data structure used for testing set membership for very large sets. It allows a small percentage of false positives in exchange for space and speed.

[2]Assuming that the percentage of true positive queries is small.

Concretely, for a given set S, a Bloom filter uses a bit array of size m, and k hash functions to be applied to objects of the same type as the elements in S. Each hash application produces an integer value between 1 and m, used as an index into the bit array. In the filter setup phase, the k hash functions are applied to each element in S, and the bit indexed by each resulting value is set to 1 in the array (thus, for each element in S, there will be a maximum of k bits set in the bit array—fewer if two hash functions yield the same value, or if some bits had already been set for other elements). When testing membership, the k hash functions are also applied to the tested element, and the bits indexed by the resulting values are checked. If they are all 1, the element is potentially a member of the set S. Otherwise, if at least one bit is 0, the element is not part of the set (false negatives are not possible).

The number of hash functions used and the size of the bit array determine the false positive rate of the Bloom filter. For a set with n elements, the asymptotic false positive probability of a test is $\left(1 - e^{-km/n}\right)^k$ (see section 3.2).

The larger m is, the smaller the false positive rate. Furthermore, since hits in the Bloom filter (false or true positives) are more expensive than misses (a query can stop as soon as one hash function misses), a larger m may also improve the performance (search speed) of the filter. On the other hand, random accesses in a large bit array have poor cache performance on today's machines.

For a fixed m, $k = \ln 2 \times m/n$ minimizes the false positive rate. In practice however, k is often chosen smaller than optimum for speed considerations: a smaller k means computing fewer hash functions.

Improving the performance of Bloom filters has also been the subject of much research. Kirsch and Mitzenmacher [17] show that computing all the hash functions as linear combinations of just two independent hash functions does not affect the false positive rate of a Bloom filter. We use this result, as explained in section 3.5. Putze et al. propose *blocked Bloom filters* in [21], which achieve better cache performance than regular Bloom filters by putting all the hashes of an element in the same cache line of the bit vector. This scheme is most effective for applications with a high true positive search rate, while the cache-friendly technique that we propose in this paper is better suited for applications with a low true positive rate. Hao et al. [13] use partitioned hashing (the elements are divided into groups and each group is hashed with a different set of functions) to reduce the Bloom filter fill factor, and therefore its false positive rate. This optimization is orthogonal to ours.

There exist various extensions to the Bloom filter functionality as well: *counting Bloom filters* [12] allow for deletions; *bloomier filters* [8] implement associative arrays that allow a small false positive look-up rate; *distance-sensitive Bloom filters* [16] are designed to answer queries of the type "is x close to any element in the set S" for a certain, suitable metric; *spectral Bloom filters* [10] allow for queries on the multiplicity of items in a multiset. In section 3.2 we present our own extension to Bloom filters, which we call *feed-forward Bloom filters.*

2.3 Comparison with Regular Expression Matching. The algorithm that we present in this paper is applicable to fixed pattern matching. DFA based algorithms are more powerful in that they can also be used for general regular expression matching. In other work [7] we have succesfully applied feed-forward Bloom filters to regular expressions that contain fixed substrings. Expanding regular expressions to multiple fixed patterns and then using feed-forward Bloom filters could also be a solution for certain workloads, and we leave exploring it for future work.

3 Design and Implementation

3.1 Overview. The multiple pattern matching algorithm that we present in this paper was designed to perform well for situations where a very large numbers of patterns generate a relatively small number of matches. It takes into account the memory hierarchy of modern computers.

The diagram in figure 2 presents a high-level view of our approach:

1. Step 1: Pattern pre-processing. First, a feed-forward Bloom filter (FFBF) is built from the set of patterns.

2. Step 2: Scan items using patterns. The Bloom filter is used to scan the corpus and discard every item (e.g., line of text, if the patterns cannot span multiple lines, or input fragment) that does not generate hits in the filter and therefore cannot contain any matches. If an item matches, it is inserted into the feed-forward structure.

3. Step 3: Scan patterns using matched items. The set of patterns is then scanned using feed-forward information obtained during the corpus scan. Only those patterns for which there is a chance of a match in the filtered corpus are kept for the next phase.

4. Step 4: Cleanup. At this point, all that is left to do is search for a small fraction of the initial number

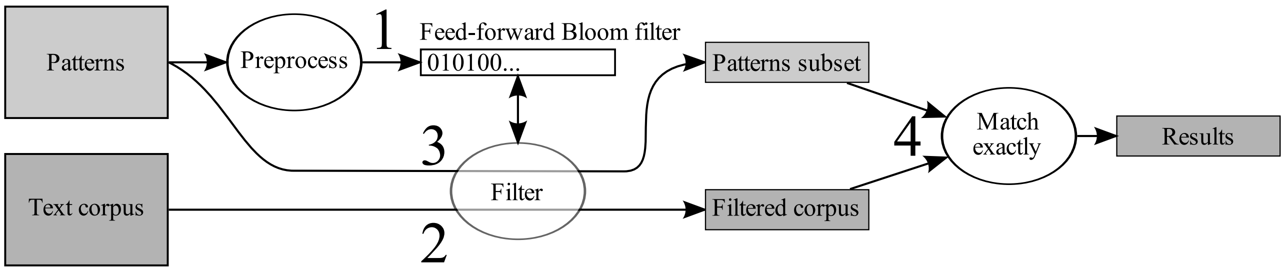


Figure 2: Diagram of the pattern matching algorithm using feed-forward Bloom filters.

of patterns in a small fragment of the corpus. This exact matching step can be performed quickly and with minimal memory requirements using a traditional multiple pattern matching algorithm (e.g. Aho-Corasick). Notice that the large set of patterns does not have to be memory-resident at any point during the execution of our algorithm—we only need to stream it sequentially from external media.

The starting point for our work is the combination of the Rabin-Karp algorithm and Bloom filters. This multiple pattern matching approach was augmented with two techniques that improve its speed and memory efficiency: *feed-forward Bloom filters* and *cache-partitioned Bloom filters.*

We present the algorithm as a whole in this section, and then describe and evaluate the two techniques in detail. Even though they were designed to improve the performance of our algorithm, we believe that they are independently useful.

We begin by describing the traditional way of using Bloom filters in the Rabin-Karp algorithm. The patterns P represent the set that is used to build the Bloom filter. During the scan, a window of the same length as the patterns (we assume for now that all patterns are of the same length) is slid through the text and tested against the filter. A hit denotes either a true or a false positive. Distinguishing between the two can be done in two ways: (1) during the Bloom filter scan, by looking-up the string in the current window in a hash table that contains all the patterns, or (2) after the Bloom filter scan, which involves saving the regions of text (usually lines of text) that generated hits in the Bloom filter, and running an exact multiple pattern matching algorithm on only this smaller input. In both cases, the disadvantage is that all the patterns are used to build the data structures (either the hash tables or the tries), and this can be memory-inefficient.

Feed-forward Bloom filters help with the second phase exact-matching scan by providing a subset containing all the patterns that will generate matches—and possibly a small number of patterns that will not. In other words, feed-forward Bloom filters not only filter the corpus like regular Bloom filters, but also filter the set of patterns. Usually, the resulting subset contains only a small fraction of the initial number of patterns, so the memory efficiency—and therefore the speed—of the second phase exact-matching scan are drastically improved.

In practice, the patterns may not all be the same length. The solution is to take l consecutive characters of every pattern (e.g. the first l characters) and use only these substrings to build the Bloom filter. If there are patterns shorter than l characters, we will look for them in a concurrent exact matching scan. Choosing l is a trade-off between the effectiveness of the filtering phase (a large l makes the filtering more selective) and the performance of the separate exact matching scan for short patterns (a small l makes for fewer short patterns).

Another common case when filtering effectiveness may be reduced is that when a small number of patterns generate many matches. In this situation, the filtering would only discard a small percentage of the corpus text. A good way of dealing with this case is to test the frequency of the patterns in a sample of the corpus. The most frequent patterns could then be excluded from the filtering phase, and join the short patterns in a separate exact matching scan.

A pseudocode description of the algorithm is presented in figure 3.

3.2 Feed-forward Bloom Filters. Bloom filters are used to test set membership: given a set S, a Bloom filter is able to answer questions of the form "is x in S?" with a certain false positive probability (in our pattern matching application, x is a sequence of characters in the corpus, and S is the set of patterns).

Feed-forward Bloom filters extend this functionality. After answering a number of queries, a feed-forward Bloom filter provides a subset $S' \subset S$, such that:

1. If the query "is z in S?" has been answered and $z \in S$, then $z \in S'$.
2. If $y \in S'$, then there is a high probability that the

```
{$P$ is the set of all fixed-string patterns}
{$T$ is the set of input string elements}
Phase 1 - Preprocessing
1. find $F \subset P$, the subset of the most frequent patterns
2. choose $l$, the minimum size for the patterns to be included in the Bloom filter
3. compute $S \subset P$, the subset of all patterns shorter than $l$
4. build feed-forward Bloom filter $FFBF$ from $P \setminus (F \cup S)$
Phase 2 - Filtering
1. $(T', P') \leftarrow FFBF(T)$
with
$T' \subset T$ and $P' \subset (P \setminus (F \cup S))$
Phase 3 - Exact matching
1. $T_1 \leftarrow$ exact_match$[F \cup S](T)$
2. $T_2 \leftarrow$ exact_match$[P'](T')$
3. output $T_1 \cup T_2$
```

Figure 3: Pseudocode for the multiple pattern matching with feed-forward Bloom filters

query "is y in S?" has been answered.

To implement this functionality, feed-forward Bloom filters use two bit arrays instead of one. The first array is populated using the elements of S (e.g., the patterns to be matched, as in Rabin-Karp), just like a regular Bloom filter. The second array starts with all bits 0, and is modified during the querying process: for every positive test "is x in S?", x is inserted into a Bloom filter that uses the second array as its bit array. The second Bloom filter uses a different set of hash functions to determine which bits to set for an item, but these hash functions are efficiently computable based upon the hashing that was done to test for membership in the first Bloom filter (section 3.5). After a number of queries, S' is obtained by testing every item in S against the second Bloom filter, and putting all the items that generate positive results in S'.

In our implementation, we size the second array the same as the first, but their sizes can be adjusted independently.[3] Because this algorithm is designed for workloads in which positive matches are rare, the total time required for inserting entries into the second Bloom filter is very small.

To understand why this approach is correct, consider that the query "is x in S?" has been answered for a certain $x \in S$ (so the answer must have been positive). Then, according to the procedure described above, x was inserted into the second Bloom filter. In the next phase, when all the elements in S are queried against the second Bloom filter, x will generate a hit and will be included in S'.

The *feed-forward false positives* are those items in S' that were not queried against the first Bloom filter. In pattern matching these false positive items are patterns that we must match against in the cleanup phase, even though no corpus fragment could successfully match them. Given an item $y \in S$ which was not queried against the Bloom filter, what is the probability that y will be inserted in S'?

This false positive rate depends on the standard Bloom filter parameters: the number of items inserted into the filter, the number of hash functions, and the array size. Furthermore, recall that an item is inserted into the second Bloom filter if and only if it matches the first Bloom filter. The feed-forward false positive rate therefore also depends on the number of items tested against the first Bloom filter multiplied by the match rate of the first filter.

Assume for now that true postives (items that actually match a pattern that we care about) are rare. Let m be the number of bits in the first array, let k be the number of hashes used for every item insertion/search in it, and let m' and k' be the corresponding parameters for the second Bloom filter. If $n = |S|$ is the number of items inserted, then the false positive probability of the first filter is $P_{FP}^{filter1} = function(n, m, k)$. The feed-forward false positive probability will then be $P_{FP}^{filter2} \approx function(P_{FP}^{filter1} \times \text{number of queries}, m', k')$ (we explain the approximation below).

Assuming perfectly uniform hash functions, after inserting n items into the first Bloom filter, the probability that any particular bit is still 0 in the first array is: $P_0 = \left(1 - \frac{1}{m}\right)^{kn}$. The probability of a false positive when searching in the Bloom filter is then:

$$P_{FP} \equiv P_{FP}^{filter1} = (1 - P_0)^k = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k$$

We begin by ignoring true positives (in most applications the number of true positives is negligible when compared to the number of false positives), but we factor them in in the next section. For now, we consider $P_{hit} = P_{FP} + P_{TP} \approx P_{FP}$. We approximate the probability that a bit is 0 in the second array, after w queries, by using the expectation for the number of false positive queries in the first filter ($w \cdot P_{FP}$):

$$P'_0 \approx \left(1 - \frac{1}{m'}\right)^{k' w P_{FP}}$$

[3]In an earlier version of this algorithm, we used the same hash functions for the two filters to avoid additional computation. Doing so, however, substantially increases the false positive rate of the pattern-filtering phase.

Thus, the probability that an item in S will be selected to be part of S' (as a feed-forward false positive) is:

$$P_{feed-fwdFP} \equiv P_{FP}^{filter2} = (1 - P'_0)^{k'} \approx$$

$$\left(1 - \left(1 - \frac{1}{m'}\right)^{k'w\left(1-e^{-kn/m}\right)^k}\right)^{k'} \approx$$

$$\left(1 - e^{-k'\frac{w}{m}\left(1-e^{-kn/m}\right)^k}\right)^{k'}$$

This expression is represented in figure 4 as a function of w/m, for different values of k and m/n, in the case where $k' = k$ and $m' = m$. The feed-forward false positive probability is small, and it can be drastically reduced by small adjustments to the feed-forward Bloom filter parameters: For example, consider a target feed-forward false-positive probability of 1% when searching for 3 million patterns (i.e., we want around 30,000 patterns for exact matching). With 10 MB bit vectors and five hash functions for each bit vector ($k = k' = 5$) we can scan 67 GB of text and not surpass the target feed-forward false positive probability. With 10 MB bit vectors and $k = k' = 6$, we can scan 156 GB of text. With 20 MB bit vectors and still six hash functions we can scan 14 terabytes of text and still select only under 1% of the patterns for exact matching.

3.3 Factoring in True Positives. Intuitively, it is not the number of true positive tests (against the first Bloom filter) that affects the feed-forward false positive rate, but the percentage of items that generate them. For example, if only one item generates all the true positives, then at most k' bits will be set in the second bit array.

Assume that there are n' items from S that will generate true positives (we usually expect $\frac{n'}{n}$ to be small). Then the probability that a bit is 1 in the second bit array because of the true positive tests is:

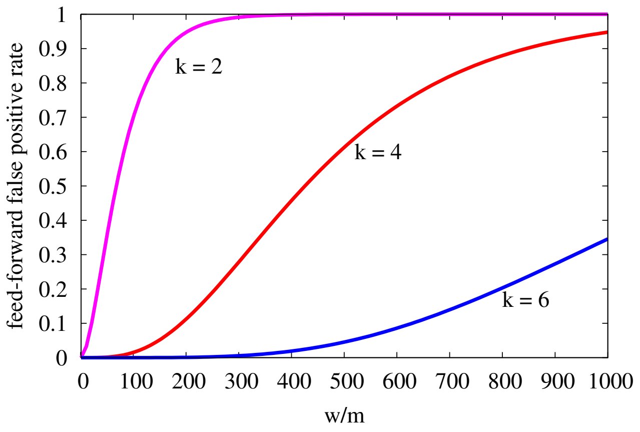
$$P'_{1,TP} = 1 - \left(1 - \frac{1}{m'}\right)^{k'n'}$$

In conclusion, the probability that a bit is set in the second array, after w tests that are not true positives and any number of tests that are true positives, is:

$$P'_1 = P'_{1,TP} + \left(1 - P'_{1,TP}\right)(1 - P'_0)$$

where P'_0 has the expression presented in the previous section.

The probability of a feed-forward false positive becomes:



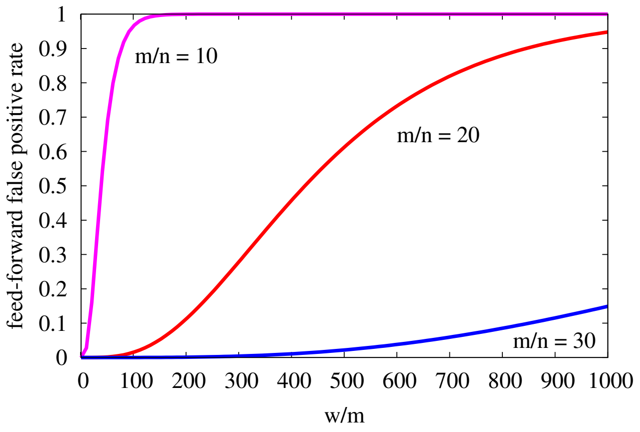


Figure 4: The feed-forward false positive rate as a function of w/m for $m/n = 20$ and varying k (up), and $k = 4$ and varying m/n (down). For both cases we consider $k' = k$ and $m' = m$.

$$P_{feed-fwdFP} = (P'_1)^{k'}$$

Figure 5 presents the same cases as the first graph in figure 4, and shows how the feed-forward false positive rate is affected if $\frac{n'}{n} = 0.5$. We conclude that the effect of the true positives is small even if a relatively large percent (50%) of patterns are present in the corpus.

3.4 Cache-partitioned Bloom Filters. Consider a machine with a simple memory hierarchy: a small cache memory[4] that can be accessed rapidly and a large main memory that is accessed more slowly. In the cases we examine, hits in the Bloom filter are rare. A Bloom filter miss requires one or more lookups in the bit array, where the number of lookups is inversely proportional

[4]For a multi-level cache hierarchy this will usually be the largest cache.

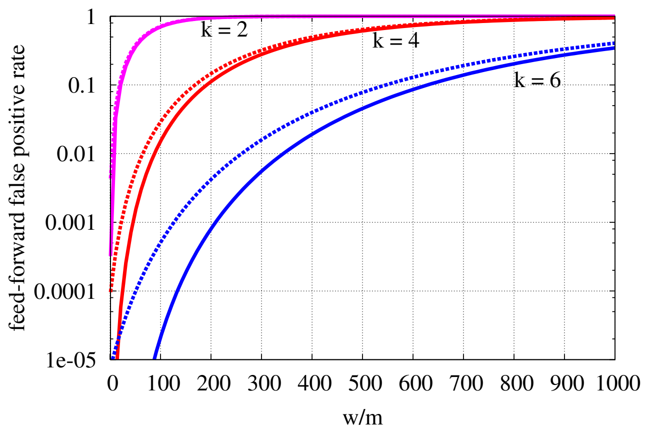


Figure 5: The feed-forward false positive rate as a function of w/m for $m/n = 20$ and varying k ($k' = k$ and $m' = m$). The dashed lines show the effect of 50% of the items generating true positives. Note that the y axis uses a logarithmic scale.

to the fraction of bits that are set to 1—the filter returns "NO" when it finds the first 0 bit. These lookups, therefore, have a computational cost to hash the data and compute the Bloom filter bit index, and a memory lookup cost that depends upon whether the lookup hits in L2 cache and whether it incurs a TLB miss. Because of the large cost penalty for cache misses, reducing the number of cache misses for negative Bloom filter lookups can substantially reduce the total running time. We therefore propose an improvement to Bloom filters that we call *cache-partitioned Bloom filters.*

The bit array for a cache-partitioned Bloom filter is split into two components: a small bit array that fits completely in cache and a large bit array that resides only in main memory. The first s hash functions hash into the small cache-resident array, while the other $q = k - s$ functions hash only into the non-cache-resident part. Unlike for the regular Bloom filter, in cache-partitioned filters most accesses are made to the part that resides in cache: the first bits checked are always in cache, and most of the time one of them will be 0, which will allow the lookup to abort.

Cache behavior. We assume that the cache uses an approximation of least-recently-used with some degree of set associativity (≥ 2). As a result, pages for the cache-resident part of the filter are likely to remain in cache. We ensure this further by doing non-temporal reads[5] when accessing the non-cache resident part of the bit array.

[5]We accomplish this using non-temporal prefetches with the `prefetchNTA` instruction available for Intel CPUs.

TLB behavior. We use the large pages support available in most modern processors to ensure that the number of pages required by the bit array is smaller than the number of TLB entries. Avoiding TLB miss penalties improves speed by 15%. This optimization also simplifies our analysis because it lets us ignore TLB effects.

After inserting n patterns in the filter, the probability that any particular bit is 1 in the cache resident part is: $P_{1c} = 1 - \left(1 - \frac{1}{c}\right)^{sn}$. For the non-resident part, the corresponding probability is: $P_{1m} = 1 - \left(1 - \frac{1}{m}\right)^{qn}$. Therefore, the expected time spent per Bloom filter lookup is:

$$E[t_{lookup}] = t_c + t_c P_{1c} + t_c P_{1c}^2 + \ldots + t_c P_{1c}^{s-1} +$$

$$t_m P_{1c}^s + t_m P_{1c}^s P_{1m} + \ldots + t_m P_{1c}^s P_{1m}^{q-1} =$$

$$t_c \frac{1 - P_{1c}^s}{1 - P_{1c}} + t_m P_{1c}^s \frac{1 - P_{1m}^q}{1 - P_{1m}}$$

To refine this model further, note that for CPUs that perform branch prediction, the branch predictor will be wrong every time a bit vector access hits a set bit, thus incurring a branch misprediction penalty t_p. The expected lookup time becomes:

$$E[t_{lookup}] = t_c \frac{1 - P_{1c}^s}{1 - P_{1c}} + t_m P_{1c}^s \frac{1 - P_{1m}^q}{1 - P_{1m}} +$$

$$t_p \left(\frac{1 - P_{1c}^s}{1 - P_{1c}} - 1 + P_{1c}^s \frac{1 - P_{1m}^q}{1 - P_{1m}} \right)$$

This expression can be used to predict the best values for the parameters of cache-partitioned Bloom filters, as shown in seciton 4.4.

3.5 Fast Rolling Hash Functions.

Besides the cache behavior, another possible bottleneck in a Bloom filter implementation is the computation of the hash functions.

When using Bloom filters for scanning text, most implementations employ rolling hash functions to easily update the hash values based on the characters sliding out of, and into the current window. The classic rolling hash function used in the Rabin-Karp algorithm computes the hash value of a string as the value of the corresponding ASCII sequence in a large base. This computation, however, requires multiplications and the expensive modulo operation, and can thus have a high overhead.

An inexpensive and effective rolling hash method is hashing by cyclic polynomials [9]. It uses a substitution box to assign random 32-bit values to characters, and combines these values with bit-wise rotations and the exclusive-OR operation, avoiding expensive multiplications and modulo operations.

In our implementation, we use cyclic polynomial hashing to obtain two distinct hash values for each window. We then use the idea of Kirsch and Mitzenmacher [17] and quickly compute all the other hashes as linear combinations of these two values. Our hash computations are therefore very efficient.

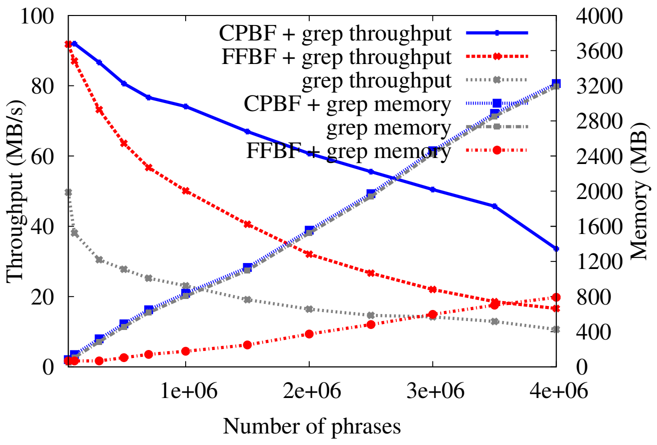
4 Evaluation

We run our tests on a 2.4 GHz Intel Core 2 Quad Q6600 CPU with split 8 MB L2 cache (each core has access to only 4 MB), and 4 GB of RAM memory. The optimal cache-partitioned configuration for this setup is: 2 MB and two hash functions for the resident part, 32 MB and three hash functions for the non-resident part (see section 4.4 for details about choosing these parameters). For the feed-forward Bloom filter, both bit vectors have the same size. All tests are performed with a warm file system buffer cache. Every time we compare with `grep`, we discount the `grep` initialization time. The results were averaged over four runs and the standard deviation was always smaller than 5% of the mean.

4.1 Overall Performance. We compare our algorithm with `grep` version 2.5.4, run as `fgrep`, which is optimized for fixed-string patterns. We use cache-optimized feed-forward Bloom filters for the first phase, and `grep` for the second phase. We report aggregate throughput and memory consumption. In this comparison we use the following three workloads:

Read the Web: The Read the Web project [1] aims at building a probabilistic knowledge base using the content of the Web. The workload that we use in our evaluation consists in determining semantic classes for English words by putting those words in phrases with similar structure and finding the relative frequencies of these phrases in Web documents. In total, there are approximately 4.5 million phrases that we search in 244 MB of Web documents. Note that, because of the way they were built, the patterns are very similar, which means that this workload is almost the best case for `grep` and the worst case for the feed-forward Bloom filter. Around 85% of these patterns are over 19 characters, so we choose the first 19 characters of each phrase (that is long enough) to put in the Bloom filter. The results presented in figure 6 (top graph) are for phrase sets that do not contain any short patterns. Since the distribution of pattern lengths is highly application-specific, we present results for experiments with short patterns separately, in section 4.2.

Random ASCII text: We search for random 19-character strings consisting of printable ASCII characters in a random corpus. Each line of the corpus has 118 characters (resulting in 100 Bloom filter lookups per line) and there are one million lines in the corpus. Since there is no redundancy in the patterns, and the probability that a pattern will be found in the corpus is very small, this workload represents the best case for Bloom filters, but the worst case for `grep`. The results are presented in figure 6 (bottom graph).



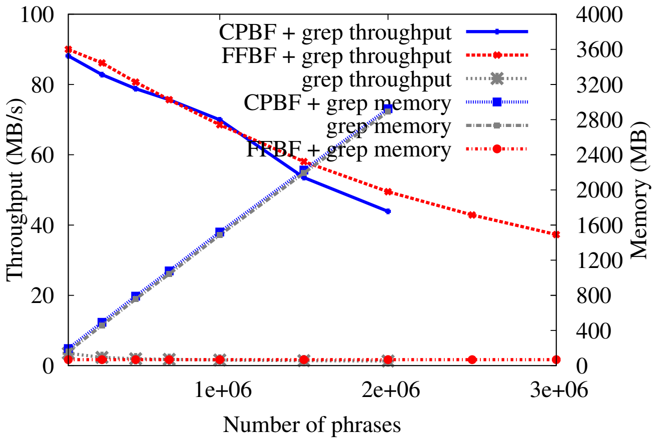


Figure 6: Comparison between scanning text with cache-partitioned Bloom filters, feed-forward Bloom filters and `grep` for the Read the Web project (top graph) and random ASCII text (bottom graph). The CPBF and FFBF throughputs include the first (filter) and second (grep cleanup) phase. The FFBF also contains the cache-partitioned optimization (i.e., FFBF is a cache-partitioned feed-forward Bloom filter, whereas CPBF is just a cache-partitioned Bloom filter). At 2.5 million random ASCII phrases, `grep` runs out of memory on our system.

DNA: This consists in looking for 200,000 random DNA sequences of various lengths in the genomes of three strains of Streptococcus Suis [2] [6]. Our goal is to assess the limitations of our approach for a potentially important application which has the particularity that the alphabet is very small (four base pairs). Our method (feed-forward, cache-partitioned Bloom filter in the first phase and grep in the second) performs increasingly better for DNA sequences larger than 10 base pairs—it generates a 17× speedup for 15 base pairs.

Comparisons between the memory requirements of the two approaches (FFBF + `grep` versus simple `grep`) for the Read the Web and random text workloads are also presented in figure 6 (the cache-partitioned Bloom filter without the feed-forward technique (CPBF) + `grep` requires only 34 MB of memory more than `grep`—i.e., the size of the bit vector.

As expected, feed-forward Bloom filters are much better than `grep` for the random text workload. `Grep` builds a very large DFA because the alphabet is large and all symbols occur in the pattern set with almost equal frequency, while the feed-forward Bloom filter only needs to run the first pass, since there are no patterns false positives (even if there are false positive matches in the corpus).

The Read the Web scenario is more favorable to `grep` because there are many similar patterns (i.e. the first 19 characters that we use to build the feed-forward Bloom filter are the same for many patterns), so the number of patterns that must be checked in the second phase is large. Even so, feed-forward Bloom filters perform substantially better.

`Grep` works well for DNA lookups because the alphabet is very small (four symbols)—and usually the patterns are short, so the DFA that `grep` builds is small. Furthermore, with patterns containing only four distinct characters, the hash values computed in the FFBF algorithm will be less uniform. However, as the size of the sequences increases, the relative performance of FFBFs improves, making them a viable solution even in this setting.

Initialization time, while discounted from the results presented in figure 6, is always substantially smaller for feed-forward Bloom filters—e.g., 18 s for our algorithm versus 70 s for `grep`, for the random text workload, at the 2 million patterns point.

We have also applied feed-forward Bloom filters to virus scanning. The results, reported in [7], show an overall speed improvement of 2× to 4×, while at the same time using less than half the memory, overall.

[6]We do not claim this to be representative of workloads in bioinformatics, even though popular sequence alignment algorithms, such as BLAST, start with a multiple-patterns matching phase.

4.2 The Impact of Short Patterns. We repeat the comparison with `grep` for the Read the Web workload at the 4 million phrases point, but this time 15% of the phrases are shorter than 19 characters. Simple `grep` achieves a throughput of 6.4 MB/s. Using FFBFs and searching for the short patterns in a separate `grep` scan, achieves an aggregate throughput of 6.7 MB/s. A better strategy is to apply the feed-forward technique recursively. For example, using three FFBFs—one for patterns at least 19 characters, another for patterns at least 14 characters and at most 18, and another for patterns between 10 and 13 characters long—and a separate scan for the shortest patterns (shorter than 10 characters in length), we can achieve a throughput of 8.3 MB/s.

In practice, the short patterns scan can be performed in parallel with the FFBF scan. In our test however, we ran them sequentially in order to maintain a fair comparison with the single threaded `grep`.

4.3 The Benefit of Individual Optimizations.

Feed-forward. Figure 6 presents the results of comparing cache-partitioned feed-forward Bloom filters with cache-partitioned Bloom filters (no feed-forward) for random text and Read the Web workloads. The no-feed-forward implementation gains time by not having to process the phrases after filtering the corpus, but needs an expensive grep cleanup phase using all the phrases. Although the FFBF-based implementation achieves higher throughput only for the random text case, it uses much less memory for both workloads. This is important because the amount of available memory limits the size of the pattern set that we can search for. For example, we are not able to search for 2.5 million phrases on a machine with 4 GB of RAM, in the random text case. Even the Read the Web workload is problematic for a low-power system—with 1 GB of RAM we can search for no more than 1 million phrases.

Figure 7 shows the benefits of each of the following three optimizations: cache-partitioning, non-temporal reads, and super pages.

Cache-partitioning. Cache-partitioning is the optimization that provides the biggest speed-up. Note that we used more hash functions for the partitioned filters because, even if this made them slightly slower, we wanted their false positive rate to be at least as small as that of the non-partitioned filter. Table 1 compares the false positive rates of the two Bloom filter variants for 3 million phrases.

Super pages. Using super pages provides an almost constant time reduction, since most of the TLB misses are triggered by one of the first Bloom filter lookups—even the cache-resident part of the filter is too

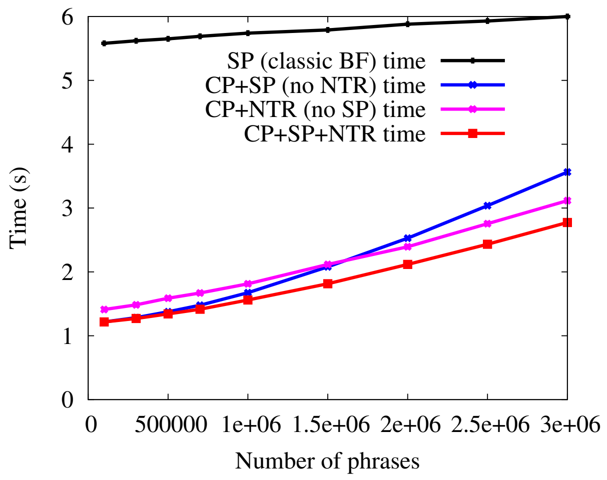


Figure 7: The graph shows the benefit of each optimization: CP (cache-partitioning), SP (super pages), and NTR (non-temporal reads). The filters were run on 114 MB of random ASCII text, for different numbers of 19-characters phrases. The cache-partitioned filters use five hash functions (two of which are for the cache-resident part) while the non-partitioned filter uses four. They are of similar size: 32 MB for the non-partitioned Bloom filter, and 2 + 32 MB for the cache-partitioned ones.

Filter Type	# Hashes	FP Rate	Throughput
Classic	4	0.205%	19 MB/s
Partitioned	4	0.584%	41.2 MB/s
Partitioned	5	0.039%	41.1 MB/s

Table 1: FP rates for cache-partitioned and classic Bloom filters, for random text, 3M phrases.

large for all its 4 KB pages to fit in the TLB.

Non-temporal reads. As the number of phrases increases, the non-temporal reads optimization becomes more important, because there are more accesses to the non-resident part of the filter. When non-temporal reads are not used, these accesses determine fragments of the cache-resident part to be evicted from cache, and this produces cache misses during the critical first lookups.

4.4 Choosing Parameters for Feed-Forward Bloom Filters. In this section we describe the way we choose the feed-forward Bloom filter parameters.

The size of the bit vectors and their partitioning depend on:

- The amount of memory we are willing to allocate for the filter.
- The number of TLB entries for super pages. If the required number of super pages is too large, there will be a TLB miss penalty that will add to the average filter lookup time.
- The size of the largest CPU cache. We determined empirically that for CPUs with large caches, the filter is faster when we don't use the entire cache. This is because there will usually be some cache contention between the Bloom filter and other processes or other parts of the program (e.g. reading the input data). In our case, since our hash functions are faster if the size of their codomain is a power of 2, we used half of the available L2 cache. For CPUs with small caches on the other hand, using less than the entire cache may produce too many false positives in the first part of the filter for cache-partitioning to provide any benefit.

The sizes of the two bit vectors used by the feed-forward Bloom filter may differ. However, if memory is not very limited, making them equal is convenient for the reasons presented above.

The number of hash functions affects not only the false positive rate of the filter, but also its speed—even with the efficient hash function computation scheme that we use, too many hash functions may cause too many memory accesses, while too few hash functions for the cache resident part will determine many tests against non-cache-resident memory. The expected lookup time model that we presented in section 3.4 is useful for determining how many hash functions to use in each section of the feed-forward Bloom filter, if we aim for optimal speed. Figure 8 shows a comparison between the speed of the fastest filter and that of the filter that uses the settings recommended by our model.[7]

After determining the settings that provide the best speed, the desired false positive rate can be achieved by increasing the number of hash functions in the non-resident part—assuming a low true positive rate, lookups in this section have little influence on the speed of the filter. Notice the large decrease of the false positive rate reported in table 1 after adding just one more hash function to the non-resident section of the filter.

Finally, the last parameter we need to determine is how to partition the input corpus, i.e., how many input items (e.g. text lines) to scan before performing the grep cleanup phase. A coarse partitioning implies fewer cleanup runs, while a finer partitioning determines

[7]The parameters that we used for modeling the behavior of the Intel Core 2 Quad Q6600 CPU are: 14 cycles for an L1 miss, 165 cycles for an L2 miss and 6 cycles for a branch misprediction.

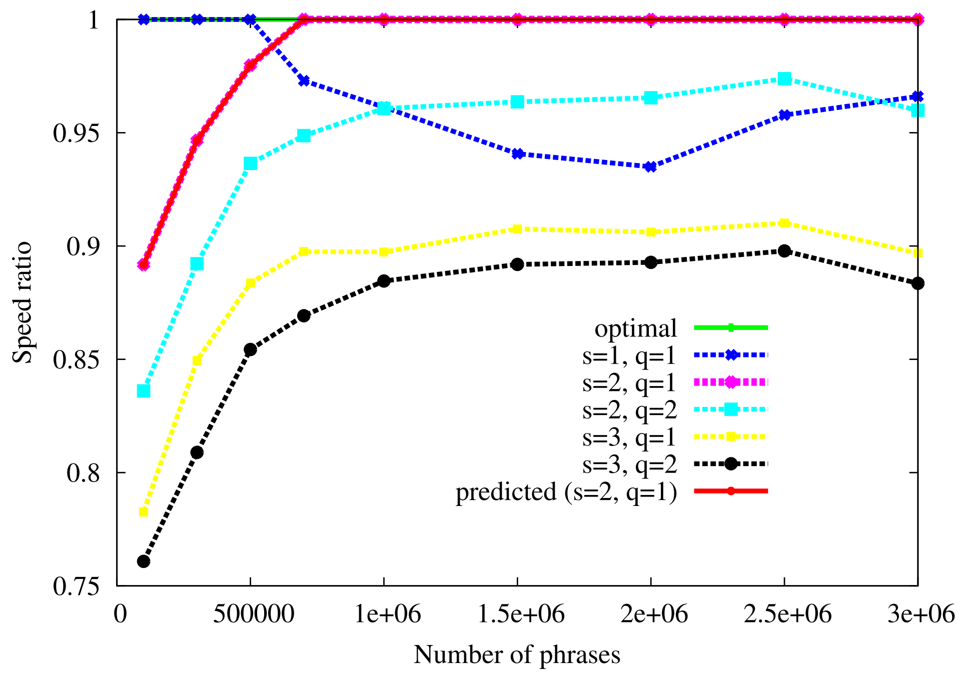


Figure 8: The ratio between the speed of scanning using cache-partitioned Bloom filters with different numbers of hash functions and the speed of the optimal (fastest) setting. The filtered corpus contains 114 MB of random ASCII text. The predicted line shows the speed of the filter using the setting that the mathematical model of the average filter lookup time deems to be the fastest.

these runs to be shorter, because the feed-forward false positive rate will be smaller, as explained in section 3.2. As seen in section 4.1, this is highly application specific, and therefore we do not attempt to find a general solution. We mention that for workloads approaching the ideal case (i.e., patterns that are very different from each other), the partitioning can be vey coarse (hundreds of gigabytes or even terabytes, as seen in section 3.2).

5 Conclusion

We have presented a new algorithm for exact pattern matching based on two Bloom filter enhancements: (1) feed-forward and (2) CPU architecture aware design and implementation. This algorithm substantially reduces scan time and memory requirements when compared with traditional DFA-based multiple pattern matching algorithms, especially for large numbers of patterns that generate relatively few matches.

6 Acknowledgments

We thank Christos Faloutsos, Kanat Tangwongsan, Srinivasan Seshan and the anonymous reviewers for their helpful suggestions. This work was supported by the National Science Foundation under grant CCF-0964474 and by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office.

References

[1] *Read the Web Project Webpage.* `http://rtw.ml.cmu.edu/readtheweb.html`.

[2] *Streptococcus Suis Sequencing Webpage.* `http://www.sanger.ac.uk/Projects/S_suis`.

[3] A. V. Aho and M. J. Corasick, *Efficient string matching: An aid to bibliographic search*, Communications of the ACM, 18 (1975), pp. 333–340.

[4] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, *FAWN: A fast array of wimpy nodes*, in Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP), Big Sky, MT, Oct. 2009.

[5] B. H. Bloom, *Space/time trade-offs in hash coding with allowable errors*, Communications of the ACM, 13 (1970), pp. 422–426.

[6] R. S. Boyer and J. S. Moore, *A fast string searching algorithm*, Communications of the ACM, 20 (1977), pp. 762–772.

[7] S. K. Cha, I. Moraru, J. Jang, J. Truelove, D. Brumley, and D. G. Andersen, *SplitScreen: Enabling efficient, distributed malware detection*, in Proc. 7th USENIX NSDI, San Jose, CA, Apr. 2010.

[8] B. Chazelle, J. Kilian, R. Rubinfeld, A. Tal, and O. Boy, *The bloomier filter: An efficient data structure for static support lookup tables*, in In Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2004, pp. 30–39.

[9] J. D. Cohen, *Recursive hashing functions for n-grams*, ACM Transactions on Information Systems, 15 (1997), pp. 291–320.

[10] S. Cohen and Y. Matias, *Spectral bloom filters*, in Proceedings of the 2003 ACM SIGMOD international conference on Management of data, ACM, 2003, pp. 241–252.

[11] B. Commentz-Walter, *A string matching algorithm fast on the average*, in Proceedings of the 6th Colloquium, on Automata, Languages and Programming, London, UK, 1979, Springer-Verlag, pp. 118–132.

[12] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, *Summary cache: A scalable wide-area Web cache sharing protocol*, in Proc. ACM SIGCOMM, Vancouver, British Columbia, Canada, Sept. 1998, pp. 254–265.

[13] F. Hao, M. Kodialam, and T. V. Lakshman, *Building high accuracy bloom filters using partitioned hashing*, SIGMETRICS Performance Evaluation Review, (2007), pp. 277–288.

[14] R. M. Karp and M. O. Rabin, *Efficient randomized pattern-matching algorithms*, IBM Journal of Research Developments, (1987), pp. 249–260.

[15] S. Kim and Y. Kim, *A Fast Multiple String-Pattern Matching Algorithm*, in Proceedings of the 17th AoM/IAoM Conference on Computer Science, 1999.

[16] A. Kirsch and M. Mitzenmacher, *Distance-sensitive bloom filters*, in In Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX), 2006.

[17] ——, *Less hashing, same performance: Building a better bloom filter*, Random Structures & Algorithms, 33 (2008), pp. 187–218.

[18] P.-c. Lin, Z.-x. Li, Y.-d. Lin, and Y.-c. Lai, *Profiling and accelerating string matching algorithms in three network content security applications*, IEEE Communications Surveys & Tutorials, 8 (2006).

[19] H. Mangalam, *tacg - a grep for dna*, BMC Bioinformatics, 3 (2002), p. 8.

[20] R. Muth and U. Manber, *Approximate multiple string search*, in Proceedings CPM'96, LNCS 1075, Springer-Verlag, 1996, pp. 75–86.

[21] F. Putze, P. Sanders, and S. Johannes, *Cache-, hash- and space-efficient bloom filters*, in Experimental Algorithms, Springer Berlin / Heidelberg, 2007, pp. 108–121.

[22] S. Wu and U. Manber, *A fast algorithm for multi-pattern searching*, Tech. Rep. TR-94-17, Department of Computer Science, University of Arizona, 1994.