Carnegie Mellon University
Research Showcase @ CMU

Computer Science Department | School of Computer Science

9-2009

Fast Cache for Your Text: Accelerating Exact Pattern Matching with Feed-Forward Bloom Filters

Iulian Moraru
Carnegie Mellon University

David G. Andersen
Carnegie Mellon University

Follow this and additional works at: http://repository.cmu.edu/compsci

This Technical Report is brought to you for free and open access by the School of Computer Science at Research Showcase @ CMU. It has been accepted for inclusion in Computer Science Department by an authorized administrator of Research Showcase @ CMU. For more information, please contact research-showcase@andrew.cmu.edu.

Fast Cache for Your Text: Accelerating Exact Pattern Matching with Feed-Forward Bloom Filters

Iulian Moraru and David G. Andersen

September 2009
CMU-CS-09-159

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper presents an algorithm for exact pattern matching based on a new type of Bloom filter that we call a feed-forward Bloom filter. Besides filtering the input corpus, a feed-forward Bloom filter is also able to reduce the set of patterns needed for the exact matching phase. 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`, while the filtering speed can be as much as 5× higher than that of a normal Bloom filters.

This research was supported by grants from the National Science Foundation, Google, Network Appliance, Intel Corporation and Carnegie Mellon Cylab.

Keywords: feed-forward Bloom filter, text scanning, cache efficient

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 in one sense 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 cache.

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]. Running Aho-Corasick in these settings would require splitting the patterns into smaller, more manageable chunks, doing multiple passes over the input data, and thus taking a longer time to complete. 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 their memory requirements. They both center around making more efficient use of the cache memory.

Feed-Forward Bloom Filters: For exact-match acceleration, Bloom filters [5] are typically used as a filter before a traditional matching phase, which we refer to as the "grep cleanup" phase. Like a traditional Bloom filter, the feed-forward Bloom filter reduces the size of the corpus before cleanup. Unlike traditional filters, however, it also uses information determined while filtering the corpus to eliminate many of the *patterns* from the second 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 a number of hash values for a query, and using these values as indices when accessing 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 needs to be a lot larger than the cache available on modern

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

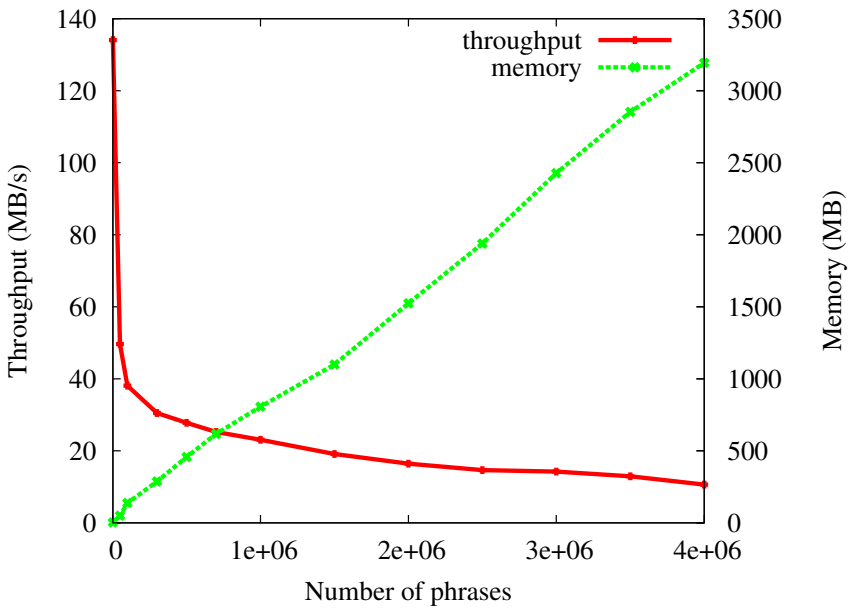

Figure 1: The `grep` processing rate and memory consumption for various numbers of patterns. The average length of the patterns is 29 characters.

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 CPU cache available (typically L2 cache) and is the only one accessed for the wide 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 $5\times$ 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 $2\times$ while consuming $4\times$ less memory, while random ASCII text can be searched $37\times$ faster with $57\times$ 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

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

state that represents the longest prefix of any match that is still possible. The algorithm generates an output every time a state that represents a full match is reached.

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 [7], 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.

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 definitely 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 (we can stop a query 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, $k = ln2 \times m/n$ minimizes the expression of 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.

The performance and effectiveness of the Bloom filter is also highly dependent on the hash functions chosen. A hash function with good uniformity will reduce the false positive rate, making the filter more effective. On the other hand, hash functions that are expensive to compute will impact the speed of the filter.

In this paper we discuss these trade-offs and show how to choose the optimal set of parameters for a given application.

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 3.4. 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 by replacing the bit array with an array of counters—each counter keeps track of how many elements hashed to that location. *Bloomier filters* [8] implement associative arrays that allow a small false positive look-up rate, and are especially effective when the number of keys is small. *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*.

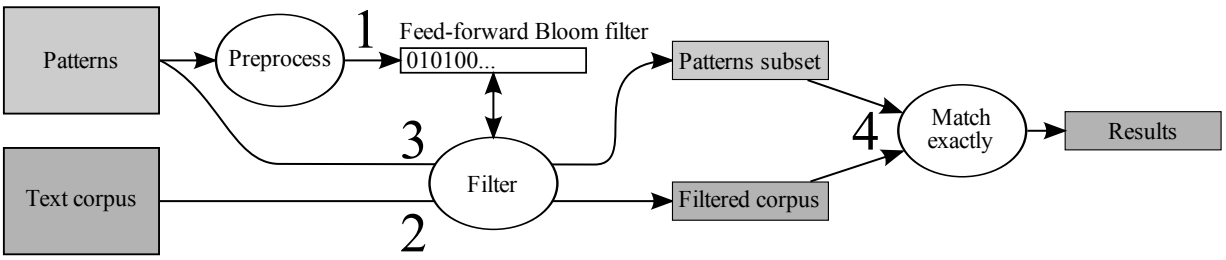

Figure 2: Diagram of the pattern matching algorithm using feed-forward Bloom filters.

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. First, a feed-forward Bloom filter (FFBF) is built from the large set of patterns.

2. It 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.

3. 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. At this point, all that is left to do is search for a small fraction of the initial number of patterns in a small fragment of the text corpus. Therefore, this exact matching step can be performed quickly and with minimal memory requirements using any 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-friendly 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, all of which must be of the same length, represent the set that is used to build the Bloom filter. During the scan, a window of the same length as the patterns is slid through the text

and tested against the filter. A hit denotes either a true match, or a false positive. To distinguish between the two, the string in the current window needs to be compared with each pattern. This step can be performed during or after the Bloom filter scan. Performing the comparison during the scan is efficient only if additional structures (e.g. hash tables) are used to reduce the number of patterns that need to be tested. This means that the entire set of patterns needs to be memory resident and therefore contradicts our design goals. The alternative is to do the exact matching step after the Bloom filter scan. This 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 only on this smaller input. The disadvantage in this case is that all the patterns need to be used for this second phase run, so it will require large amounts of memory and will exhibit poor cache performance.

Feed-forward Bloom filters help with the second phase 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 speed and memory efficiency of the second phase exact matching scan are drastically improved.

In practice, it often happens that the patterns are not all of the same length. One solution is to take the size of the shortest pattern (l), and consider for the first phase only l consecutive characters of every pattern (e.g. the first l characters). If, however, l is too small, then the filtering will not be very effective, since the chance of any combination of only a few characters is likely to be very common in the text. The solution in this case is to remove the shortest patterns from the first phase, and look for them separately in an exact match scan. This scan is faster for fewer small patterns, so choosing l is a trade-off between the effectiveness of the filtering phase—and as a result, the performance of the second phase scan—and the performance of the separate exact match scan for the 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.

$\{P$ 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. $T_1 \leftarrow \text{exact_match}[F \cup S](T)$
2. $T_2 \leftarrow \text{exact_match}[P'](T')$
3. output $T_1 \cup T_2$

Figure 3: Pseudocode for the multiple pattern matching algorithm based on feed-forward Bloom filters

A critical aspect of the performance of Bloom filters is the way they use the CPU caches. Section 3.3 presents a technique for making Bloom filters take advantage of the architectural characteristics of modern CPUs.

3.2 Feed-forward Bloom Filters

$n = \|S\|$	the number of items in the set
k	the number of hashes used for the Bloom filter
m	the number of bits in the Bloom filter
u	the number of bits set in the first bit vector
w	the number of queries

Table 1: The notation used in section 3.2.

Bloom filters are used to test set membership: given a set S, a Bloom filter is able to answer questions of the form "does x belong to S?" (we will write $x \in^? S$) with a certain false positive probability.

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 $z \in^? S$ was answered and $z \in S$, then $z \in S'$.

2. If $y \in S'$, then there is a high probability that $y \in^? S$ was answered.

To implement this functionality, feed-forward Bloom filters use two bit arrays instead of one. The first array is used like a regular Bloom filter bit array. The second one, of the same size as the first, starts with all bits 0, and is modified during the querying process: for every positive test, the bits indexed by the hash values of the tested item—which are set in the first array, since the test is positive—are also set in the second array. After a number of queries, S' is obtained by testing every item in S against the Bloom filter that uses the second array as its bit array, and putting all the items that generate positive results in S'.

To understand why this implementation is correct, consider that the query $x \in^? S$ has been answered for a certain $x \in S$. Then, according to the procedure described above, the bits indexed by the hash values of x (the values obtained by applying the Bloom filter hash functions to x) have also been set in the second array. In the next phase, when all the items in S are queried using the second array, x will again be hashed using the same hash functions, yielding the same hash values. The bits associated to these values have all been set in the second array, so the test will be positive and x will be included in S'.

Next, given an item $y \in S$ which was not queried against the Bloom filter, we compute the probability that $y \in S'$—in other words, the *feed-forward false positive probability*. Intuitively, this depends on the number of bits set in the second array, which in turn is determined by (1) the same factors that affect the false positive rate of the Bloom filter that uses the first bit array, since fewer hits in the first array mean fewer bits set in the second, and, for the same reason, (2) the number of queries that are run against the first array.

Table 1 contains the notations used in this section.

Consider a Bloom filter of size m (the filter's bit array has m bits) and let k be the number of hashes used for every item insertion/search. Assuming perfect hash functions, after inserting $n = |S|$ items into the filter, the probability that any particular bit is still 0 is:

$$P_0 = \left(1 - \frac{1}{m}\right)^{kn}$$

Then, the number of bits that are set is:

$$u = m * \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)$$

The probability of a false positive when searching in the Bloom filter is then[3]:

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

We begin by ignoring true positives (in most applications the number of true positives is negligible when compared with the number of false positives), but we factor them in the next subsection. For now, we can write:

[3]As shown in [6], this formula is not the exact expression for the false positive rate of a Bloom filter, but it is a good approximation if m is very large and k is small, which is the case for most Bloom filter applications.

$$P_{hit} = P_{FP} + P_{TP} \approx P_{FP}$$

After doing w queries against the Bloom filter, the probability that a particular 1 bit did not correspond to any hashes of the queries that did hit—so the probability of a bit set in the first array not being set in the second array—will be:

$$P_{1,0} = \left(1 - \frac{1}{u}\right)^{kwP_{hit}}$$

Thus, the fraction of items that are covered by the hits (i.e. their hash values are amongst those of the positive queries), and will be selected to be part of S' is:

$$\frac{|S'|}{|S|} = \left(1 - \left(1 - \frac{1}{u}\right)^{kwP_{hit}}\right)^k \approx$$

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

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

This expression is represented in figures 4 and 5 as a function of w/m, for different values of k and m/n.

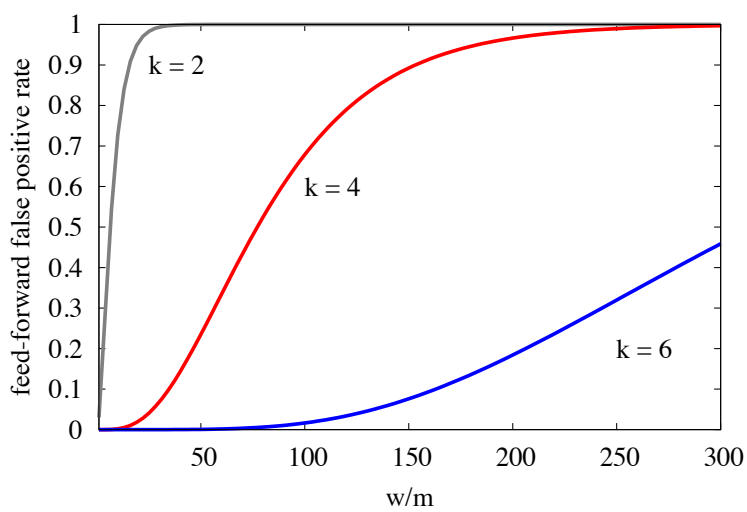

Figure 4: The feed-forward false positive rate as a function of w/m when $m/n = 20$ and $k = 2, 4, 6$.

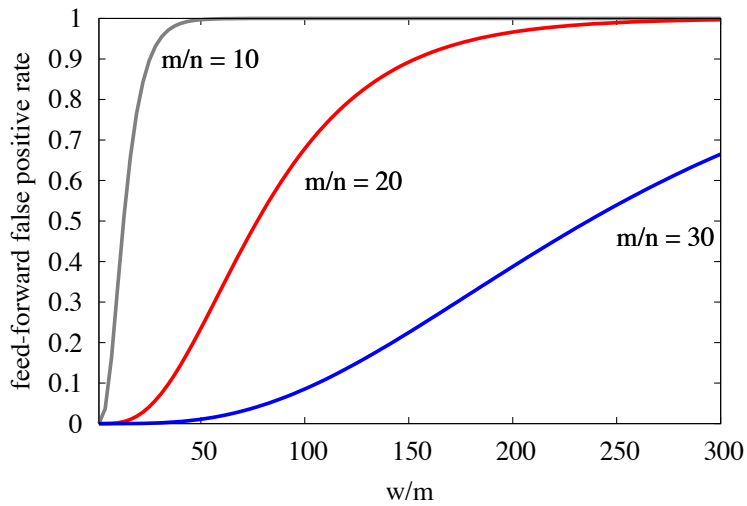

Figure 5: The feed-forward false positive rate as a function of w/m when $k = 4$ and $m/n = 10, 20, 30$.

3.2.1 Factoring in True Positives

Intuitively, it is not the number of true positives that affects the feed-forward false positive rate, but the percentage of items that generate them. For example, if only one item generates a very large number of true positives, then only 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). The number of bits that are 1 in the first bit array due to these n' items is:

$$u' = m\left(1 - \left(1 - \frac{1}{m}\right)^{kn'}\right)$$

Then, the probability that a bit set in the first array is also set in the second array, after w tests that are not true positives and any number of test that are true positives, is:

$$P_{1,1} = (1 - P_{1,0})\left(1 - \frac{u'}{u}\right) + \frac{u'}{u}$$

The probability of a feed-forward false positive becomes:

$$P_{feed-fwdFP} = P_{1,1}^{k}$$

Figure 6 presents the same cases as figure 4, and shows how the feed-forward false positive rate is affected if $\frac{n'}{n} = 0.1$. The effects of $\frac{n'}{n} = 0.5$ are presented in figure 7. We conclude that the effect of the true positives is negligible if we expect only a small percent ($\leq 10\%$) of items to be present in the corpus.

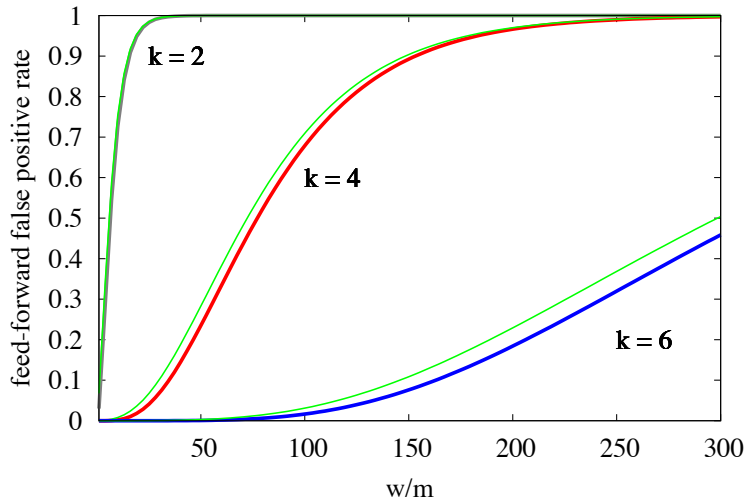

Figure 6: The feed-forward false positive rate as a function of w/m when $m/n = 20$ and $k = 2, 4, 6$. The green lines show the effect of 10% of the items generating true positives.

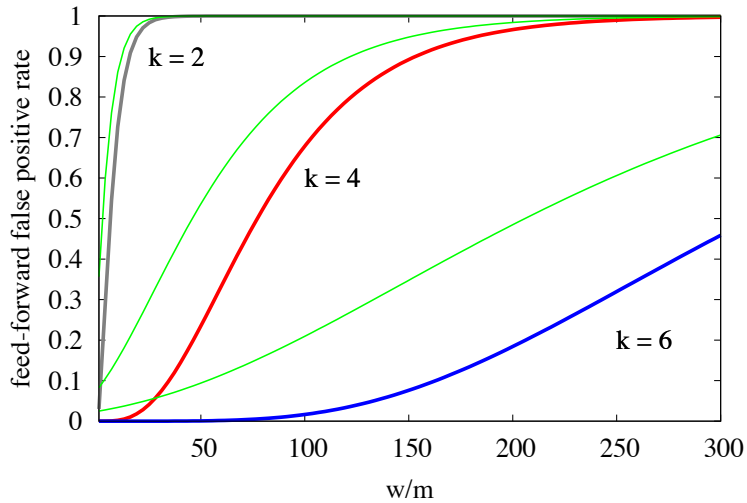

Figure 7: The feed-forward false positive rate as a function of w/m when $m/n = 20$ and $k = 2, 4, 6$. The green lines show the effect of 50% of the items generating true positives.

3.3 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 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 heavily 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. Figure 8 gives the intuition behind cache-partitioned Bloom filters. Unlike for the regular and k-partitioned Bloom filters[5], 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 unset, which will determine the lookup to be aborted.

Table 2 contains the notation used in this section.

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[6] when accessing the non-cache resident part of the bit array.

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 phrases 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:

$$P_{1m} = 1 - \left(1 - \frac{1}{m}\right)^{qn}$$

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

[5]A k-partitioned Bloom filter uses a bit array that is split into as many parts as there are hash functions. Each hash function is used to set and test bits in only one part of the array—in other words, a k-partitioned Bloom filter is the composition of k smaller Bloom filters, each using only one hash function

[6]In fact non-temporal prefetches with the `prefetchNTA` instruction available for Intel CPUs.

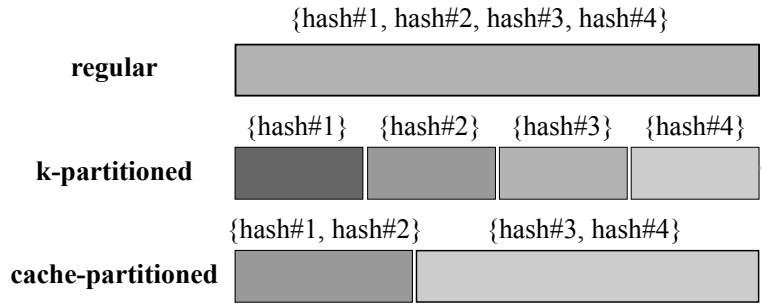

Figure 8: Comparison between the patterns of access in a regular Bloom filter, a k-partitioned Bloom filter and a cache-partitioned Bloom filter, with $k = 4$ and $s = 2$. Darker areas have a higher access density (average number of accesses per bit).

n	the number of patterns
k	total number of hash functions used
s	the number of hashes used for the cache-resident part of the filter
q	the number of hashes used for the non-resident part of the filter
c	the number of bits in the cache-resident part of the filter
m	the number of bits in the non-resident part of the filter
t_c	cache access time
t_m	memory access time
t_p	branch misprediction penalty

Table 2: Notation used in section 3.3.

Assuming that the cache-resident part will only rarely evicted from cache, the average time spent per Bloom filter lookup will be:

$$\begin{aligned}\bar{t}_{lookup} &= t_c + t_c P_{1c} + t_c P_{1c}^2 + ... + t_c P_{1c}^{s-1} + \\ &\quad t_m P_{1c}^s + t_m P_{1c}^s P_{1m} + ... + t_m P_{1c}^s P_{1m}^{q-1} \\ &= t_c \tfrac{1-P_{1c}^s}{1-P_{1c}} + t_m P_{1c}^s \tfrac{1-P_{1m}^q}{1-P_{1m}}\end{aligned}$$

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 average lookup time becomes:

$$\bar{t}_{lookup} = t_c \frac{1-P_{1c}^s}{1-P_{1c}} + t_m P_{1c}^s \frac{1-P_{1m}^q}{1-P_{1m}} + t_p \left(\frac{1-P_{1c}^s}{1-P_{1c}} - 1 + P_{1c}^s \frac{1-P_{1m}^q}{1-P_{1m}} \right)$$

3.4 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, respectively. 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 bitwise rotations and the exclusive-OR operation, avoiding 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 compute all the hash functions needed in the Bloom filter algorithm as linear combinations of these two values.

4 Evaluation

Unless specified otherwise, we run our tests on a 2.4 GHz Intel Core 2 Quad Q6600 CPU with split 8 MB L2 cache (each core only has access to 4 MB), and 4 GB of RAM memory. All tests are performed with a warm file system buffer cache. Every time we compare with `grep`, we discount the `grep` initialization time.

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, initialization time, and memory consumption.

In this comparison we use three workloads, described below:

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 90% 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

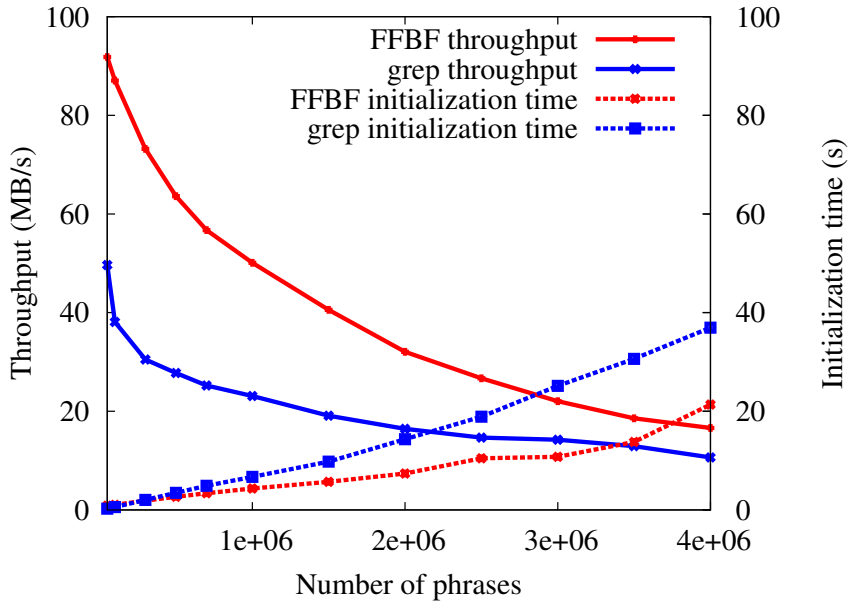

Figure 9: Comparison between scanning text from the Read the Web project with feed-forward Bloom filters and using `grep`. The FFBF throughput is the overall throughput including the first (filter) and second (grep cleaning) phase.

presented in figure 9 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 lines (there are 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 patterns 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 10. Note that at 2.5 million patterns, `grep` requires more memory than available, making it unusable.

DNA: This consists in looking for 200,000 random DNA sequences of various lengths (9, 10, 15 and 20 base pairs) in the genomes of three strains of Streptococcus Suis [2] [7]. 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). The results are presented in figure 11.

[7] 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 (the patterns are usually 11 base pairs for a human genome).

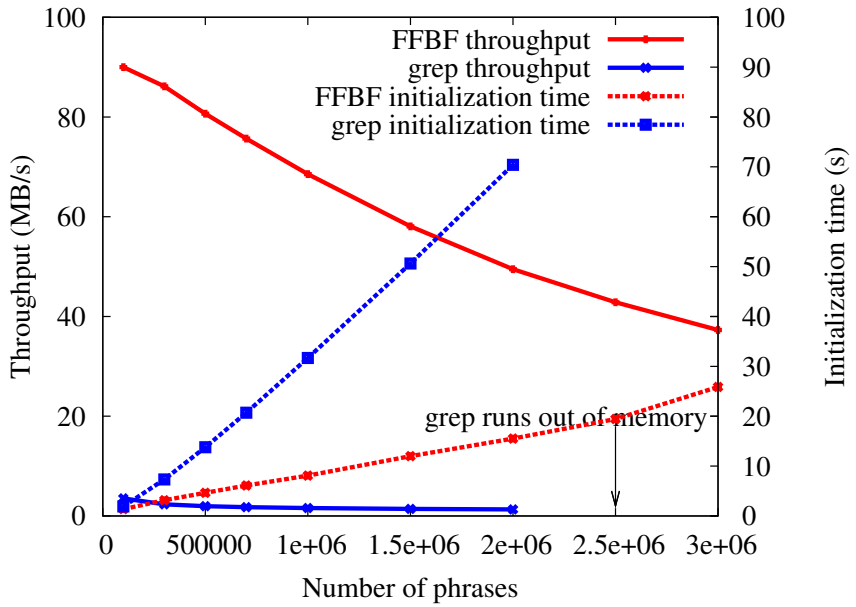

Figure 10: Comparison between scanning random text (printable ASCII characters) with feed-forward Bloom filters and using `grep`. At 2.5 million phrases `grep` requires more than the available 4 GB of RAM, which practically makes it unusable. The FFBF throughput is the overall throughput including both the first (filter) and second (grep cleaning) phase.

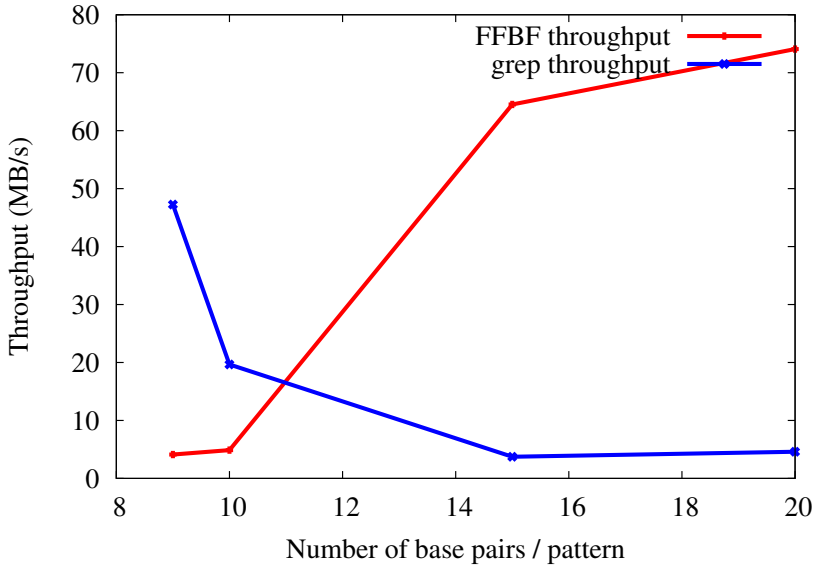

Figure 11: Comparison between scanning a DNA genome (Streptococcus Suis strains P1/7, BM407 and SC84 [2]) for random DNA sequences of different length with feed-forward Bloom filters and `grep`. The FFBF throughput is the overall throughput including the first (filter) and second (grep cleaning) phase.

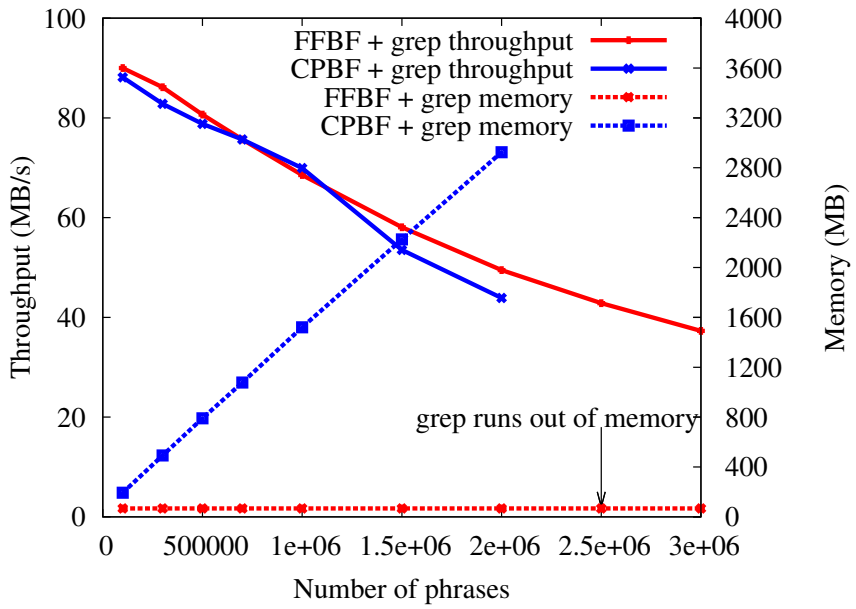

Figure 12: Throughput and memory consumption comparison between FFBF + `grep` and simple cache-partitioned Bloom filter (no feed-forward) + `grep` for scanning random ASCII text. At 2.5 million phrases `grep` requires more than the available 4 GB of RAM.

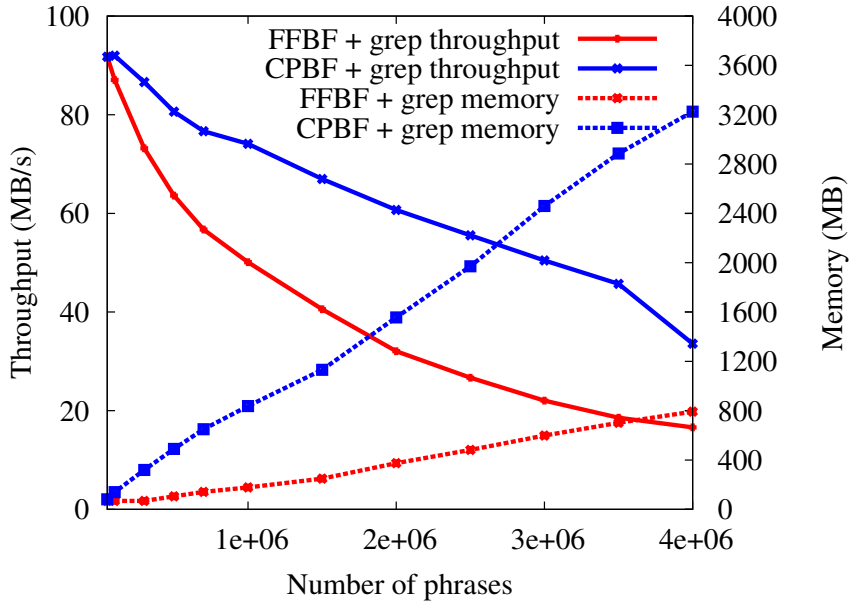

Figure 13: Throughput and memory consumption comparison between FFBF + `grep` and simple cache-partitioned Bloom filter (no feed-forward) + `grep` for scanning text from the Read the Web project with. The simple Bloom filter uses all our performance enhancing techniques.

A comparison between the memory requirements of the two approaches is presented for the Read the Web and random text workloads in figures 12 and 13 (the cache-partitioned Bloom filter (CPBF) + `grep` requires only 34 MB—the size of the bit vector—more than `grep`).

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 functions will be less uniform. As the size of the sequences increases, however, the relative performance of feed-forward Bloom filters improves, making them a viable solution even in this settings.

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. When using feed-forward Bloom filters, if we search for the short patterns in a separate scan, we will obtain a 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.

4.3 The Benefit of Individual Optimizations

Feed-forward. Figures 12 and 13 present the results of comparing 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. 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 internal memory, in the random text case. Even the Read the Web workload is problematic for a low-power system (like the one that we used to run the test that corresponds to figure 15)—with 1 GB of RAM we can search for no more than 1 million Read the Web phrases.

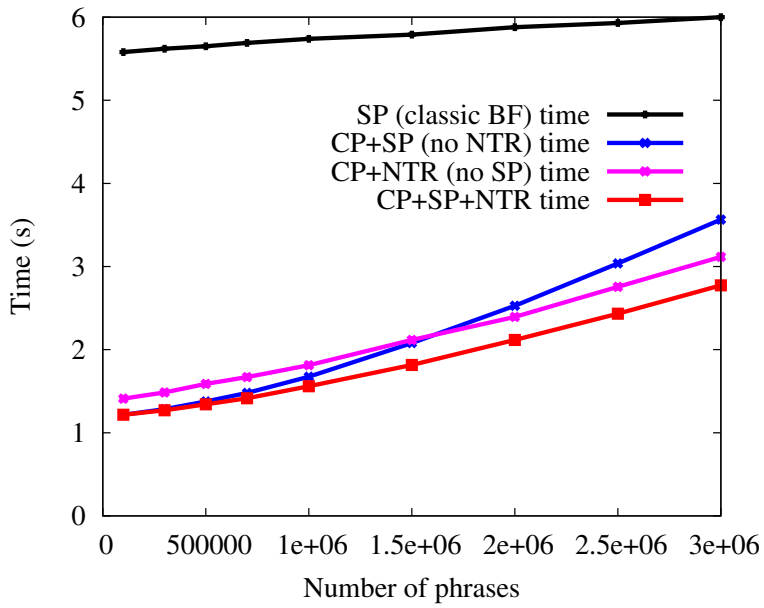

Figure 14: The graph shows the benefit of each optimization: CP (cache-partitioning), SP (super pages), and NTR (non-temporal reads). The filters were run on 114 MB of random ASCII text, for different numbers of 19-characters phrases. The cache-partitioned filters use five hash functions (two of which 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.

Cache-partitioning. Figure 14 shows the benefits of each of the following three optimizations: cache-partitioning, non-temporal reads and super pages. 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 3 compares the false positive rates of the two variations of filters for 3 million phrases and different numbers of hash functions. Figure 15 shows that cache-partitioning is effective in providing speedups even for CPUs that have small caches. In our experiment we used the Intel Atom 330, which has an L2 cache of only 512 KB.

Filter Type	Number of Hashes	FP Rate
Classic	4	0.205%
Partitioned	4	0.584%
Partitioned	5	0.039%

Table 3: The false positive rates for cache-partitioned and non-partitioned filters for the random text workload.

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 large for the number of 4 KB pages it contains 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.

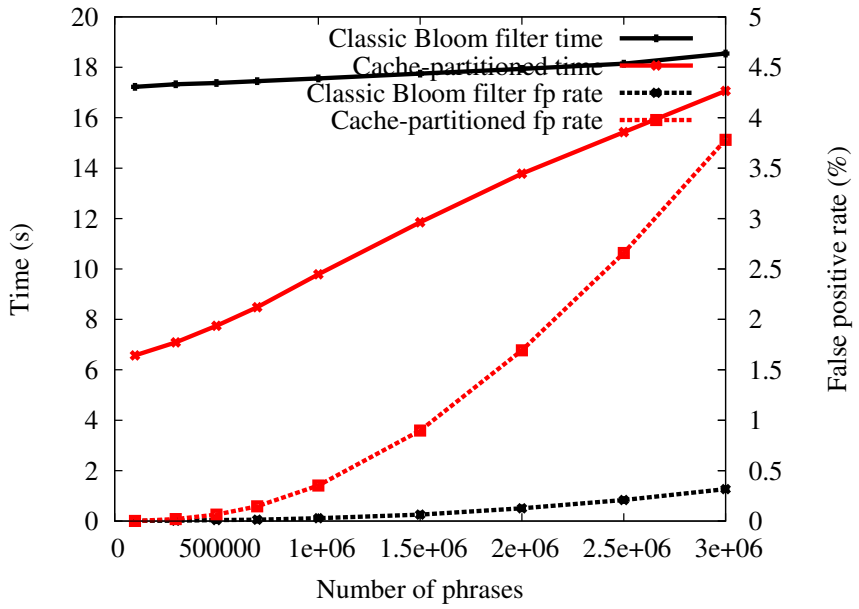

Figure 15: Comparison between the cache-partitioned Bloom filter and the classic Bloom filter on an Atom 330 CPU with 512 KB L2 cache. The filters were run on 114 MB of random ASCII text, for different numbers of phrases. Both filters use four hash functions (two-and-two for the cache-partitioned), and are of similar size: 16 MB for the classic Bloom filter, and 512 K + 16 MB for the cache-partitioned one. The classic Bloom filter was implemented using large virtual memory pages, just like the cache-partitioned one.

4.4 Choosing Parameters

In this section we describe the way we choose the feed-forward Bloom filter parameters.

The size of the bit vector and its partitioning depend on:

- The amount of memory we are willing to allocate for the filter.
- The number of TLB entries for super pages. Once 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 number of hash functions affects not only the false positive rate of the filter, but also its speed. The average lookup time model that we presented in section 3.3 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 16 shows a comparison between the speed of the fastest filter and that of the filter that uses the setting recommended by our model.[8]

After determining the settings that provide the best speed, the desired false positive rate can be achieved by increasing the number of hash function 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 3 after adding just one more hash function to the non-resident section of the filter.

Finally, the last parameter that needs to be determined 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 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 (it depends on the false positive rate), and therefore we do not attempt to find a general solution.

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.

[8]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.

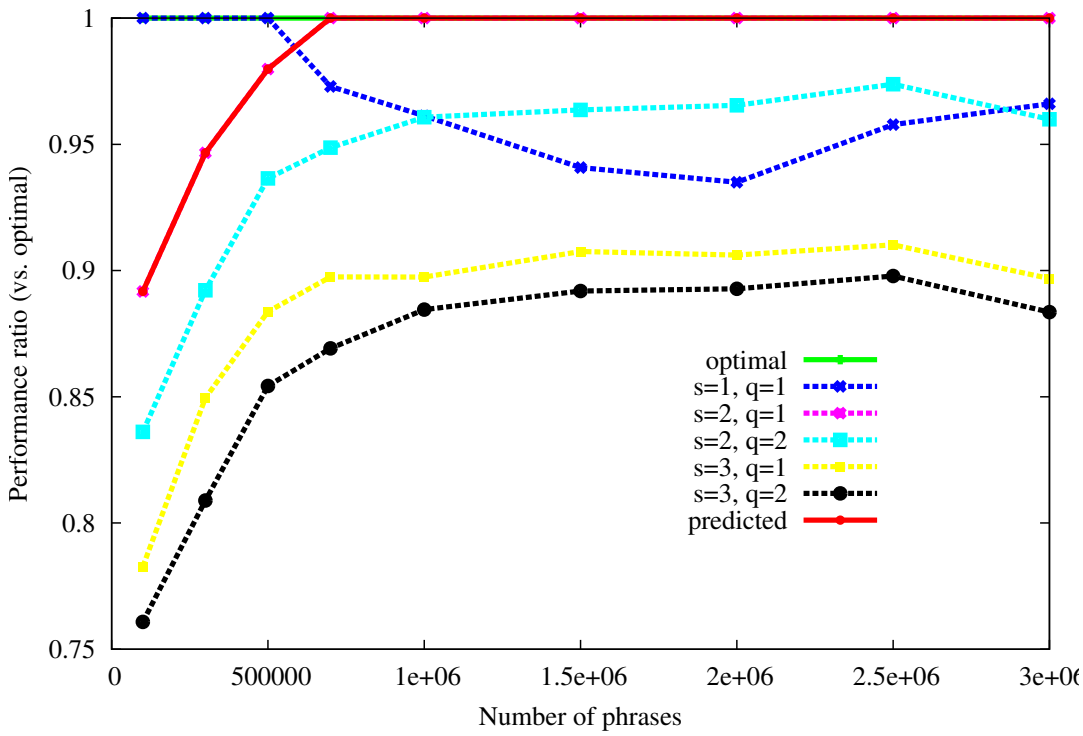

Figure 16: The ratio between the speed of scanning using cache-partitioned Bloom filters with different numbers of hash functions and the speed of the optimal (fastest) setting. The filtered corpus contains 114 MB of random ASCII text. The predicted line shows the speed of the filter using the setting that the mathematical model of the average filter lookup time deems to be the fastest.

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] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.

[4] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.

[5] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel Smid, and Yihui Tang. On the false-positive rate of bloom filters. Technical report, School of Computer Science, Carleton University, 2007.

[7] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[8] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, Ayellet Tal, and Oh 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*, pages 30–39, 2004.

[9] Jonathan D. Cohen. Recursive hashing functions for n-grams. *ACM Transactions on Information Systems*, 15(3):291–320, 1997.

[10] Saar Cohen and Yossi Matias. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 241–252. ACM, 2003.

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

[12] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area Web cache sharing protocol. In *Proc. ACM SIGCOMM*, pages 254–265, Vancouver, British Columbia, Canada, September 1998.

[13] Fang Hao, Murali Kodialam, and T. V. Lakshman. Building high accuracy bloom filters using partitioned hashing. *SIGMETRICS Performance Evaluation Review*, pages 277–288, 2007.

[14] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research Developments*, (2):249–260, March 1987.

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

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

[17] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. *Random Structures & Algorithms*, 33(2):187–218, 2008.

[18] Po-ching Lin, Zhi-xiang Li, Ying-dar Lin, and Yuan-cheng Lai. Profiling and accelerating string matching algorithms in three network content security applications. *IEEE Communications Surveys & Tutorials*, 8, April 2006.

[19] Harry Mangalam. tacg - a grep for dna. *BMC Bioinformatics*, 3(1):8, 2002.

[20] Robert Muth and Udi Manber. Approximate multiple string search. In *Proceedings CPM'96, LNCS 1075*, pages 75–86. Springer-Verlag, 1996.

[21] Felix Putze, Peter Sanders, and Singler Johannes. Cache-, hash- and space-efficient bloom filters. In *Experimental Algorithms*, pages 108–121. Springer Berlin / Heidelberg, 2007.

[22] Sun Wu and Udi Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, Department of Computer Science, University of Arizona, 1994.