Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 507173, 16 pages
doi:10.1155/2012/507173

Research Article

Throughput Analysis for a High-Performance FPGA-Accelerated Real-Time Search Application

Wim Vanderbauwhede,[1] S. R. Chalamalasetti,[2] and M. Margala[2]

[1] *School of Computing Science, University of Glasgow, Glasgow, G12 8QQ, UK*
[2] *Department of Electrical and Computer Engineering, University of Massachusetts Lowell, Lowell, MA 01854, USA*

Correspondence should be addressed to Wim Vanderbauwhede, wim@dcs.gla.ac.uk

Received 13 October 2011; Accepted 20 December 2011

Academic Editor: Miaoqing Huang

We propose an FPGA design for the relevancy computation part of a high-throughput real-time search application. The application matches terms in a stream of documents against a static profile, held in off-chip memory. We present a mathematical analysis of the throughput of the application and apply it to the problem of scaling the Bloom filter used to discard nonmatches.

1. Introduction

The focus on real-time search is growing with the increasing adoption and spread and of social networking applications. Real-time search is equally important in other areas such as analysing emails for spam or search web traffic for particular patterns.

FPGAs have great potential for speeding up many types of applications and algorithms. By performing a task in a fraction of the time of a conventional processor, large energy savings can be achieved. Therefore, there is a growing interest in the use of FPGA platforms for data centres. Because of the dramatic reduction in the required energy per query, data centres with FPGA search solutions could operate at a fraction of the power of current data centres, eliminating the need for cooling infrastructure altogether. As the cost of cooling is actually the dominant cost in today's data centres [1], the savings would be considerable. In [2, 3] we presented our initial work on applying FPGAs for acceleration or search algorithms. In this paper, we present a novel design for the scoring part of an FPGA-based high-throughput real-time search application. We present a mathematical analysis of the throughput of the system. This novel analysis is applicable to a much wider class of applications than the one discussed in the paper; any algorithm that performs nondeterministic concurrent accesses to a shared resource can be analysed using the model we present. In particular, the technology presented in this paper can also be used for "traditional," that is, inverted index based, web search.

2. Design of the Real-Time Search Application

Real-time search, in information retrieval parlance called "document filtering," consists of matching a stream of documents against a fixed set of terms, called the "profile." Typically, the profile is large and must therefore be stored in external memory.

The algorithm implemented on the FPGA can be expressed as follows.

(i) A *document* is modelled as a "bag of words," that is, a set D of pairs (t, f), where $f \triangleq n(t, d)$ is the *term frequency*, that is, number of occurrences of the term t in the document d; $t \in \mathbb{N}$ is the *term identifier*.

(ii) The profile M is a set of pairs $p = (t, w)$ where the *term weight* w is determined using the "Relevance Based Language Model" proposed by Lavrenko and Croft [4].

In this work we are concerned with the computation of the document score, which indicates how well a document matches the profile. The document has been converted to the bag-of-words representation in a separate stage. We perform this stage on the host processor using the Open Source

information retrieval toolkit Lemur [5]. We note that this stage could also be very effectively performed on FPGAs.

Simplifying slightly, to determine if a document matches a given profile, we compute the sum of the products of term frequency and term weight

$$\sum_{i \in D} f_i w_i. \tag{1}$$

The weight is typically a high-precision word (64 bits) stored in a lookup table in the external memory. If the score is above a given threshold, we return the document identifier and the score by writing it into the external memory.

2.1. Target Platform. The target platform for this work is the Novo-G FPGA supercomputer [6] hosted by the NSF Center for high-performance reconfigurable computing (CHREC) (http://www.chrec.org/). This machine consists of 24 compute servers which each host a GiDEL PROCStar-III board. The board contains 4 FPGAs with 2 banks of DDR SDRAM per FPGA used for the document collection and one for the profile. The data width is 64 bits, which means that the FPGA can read 128 bits per memory per clock cycle [7]. For more details on the platform, see Section 4.

2.2. Term-Scoring Algorithm. To simplify the discussion, we first consider the case where terms are scored sequentially, and that, as in our original work, we use a Bloom filter to limit the number of external memory accesses.

For every term in the document, the application needs to look up the corresponding profile term to obtain the term weight. As the profile is stored in the external SDRAM, this is an expensive operation (typically 20 cycles per access). The purpose of document filtering is to identify a small amount of relevant documents from a very large document set. As most documents are not relevant, most of the lookups will fail (i.e., most terms in most documents will not occur in the profile). Therefore, it is important to discard the negatives first. For that purpose we use a "trivial" Bloom filter implemented using the FPGA's on-chip memory.

2.2.1. "Trivial" Bloom Filter. A Bloom filter [8] is a datastructure used to determine membership of a set. False positives are possible, but false negatives are not. With this definition, the design we use to reject negatives is a Bloom filter. However, in most cases, a Bloom filter uses a number (k) of hash functions to compute several keys for each element in the set and adds the element to the table (assigns a "1") if element is in the set. As a result, hash collisions can lead to false positives.

Our Bloom filter is a "trivial" edge case of this more general implementation; our hashing function is the identity function $key = elt$, and we only use a single hash function ($k = 1$) so every element in the set corresponds to exactly one entry in the Bloom filter table. As a result, the size of the Bloom filter is the same as the size of the set, and there are no false positives. Furthermore, no elements are added to the set at run time.

2.2.2. Bloom Filter Dimensioning. The internal block RAMs of the Altera Stratix-III FPGA that support efficient single-bit access are limited to 4 Mb; on a Stratix-III SE260, there are 864 M9K blocks that can be configured as $8\,\mathrm{K} \times 1$ [9]. On the other hand, the vocabulary size of our document collection is 16 M terms (based on English documents using unigrams, diagrams, and trigrams). We therefore used a very simple "hashing function," $key = elt \gg 2$. Thus we obtain one entry for every four elements, which leads to three false positives out of four on average. This obviously results in a four times higher access rate to the external memory than if the Bloom filter would be 16 Mb. As the number of positives in our application is very low, the effect on performance is limited.

2.2.3. Document Stream Format. The document stream is a list of *(document identifier, document term pair set)* pairs. Physically, the FPGA accepts a fixed number n of streams of words with fixed width w. The document stream must be encoded onto these word streams. As both elements in the document term pair $d_i = (t_i, f_i)$ are unsigned integers, m pairs can be encoded onto a word if w is larger than or equal to m times the sum of the magnitudes of the maximum values for t and f

$$w \geq m\left(\left\lceil \log_2 t_{\max} \right\rceil + \left\lceil \log_2 f_{\max} \right\rceil\right). \tag{2}$$

To mark the start a document we insert a header word (identified by $f = 0$) followed by the document ID.

2.2.4. Profile Lookup Table Implementation. In the current implementation, the lookup table that stores the profile is implemented in the most straightforward way; as the vocabulary size is 2^{24} and the weight for each term in the profile can be stored in 64 bits, a profile consisting of the entire vocabulary could be stored in the 256 MB SDRAM, which is less than the size of the fixed SDRAM on the PROCStar-III board. Consequently, there is no need for hashing, the memory contains zero weights for all terms not present in the profile.

2.2.5. Sequential Implementation. The diagram for the sequential implementation of the design is shown in Figure 1.

Using the lookup table architecture and document stream format as described above, the actual lookup and scoring system is quite straightforward, the input stream is scanned for header and footer words. The header word action is to set the document score to 0; the footer word action is to collect and output the document score. For every term in the document, first the Bloom filter is used to discard negatives, and then the profile term weight is read from the SDRAM. The score is computed and accumulated for all terms in the document, and finally the score stream is filtered against a threshold before being output to the host memory. The threshold is chosen so that only a few tens or hundreds of documents in a million are returned.

If we would simply look up every term in the external memory, the maximum achievable throughput would be

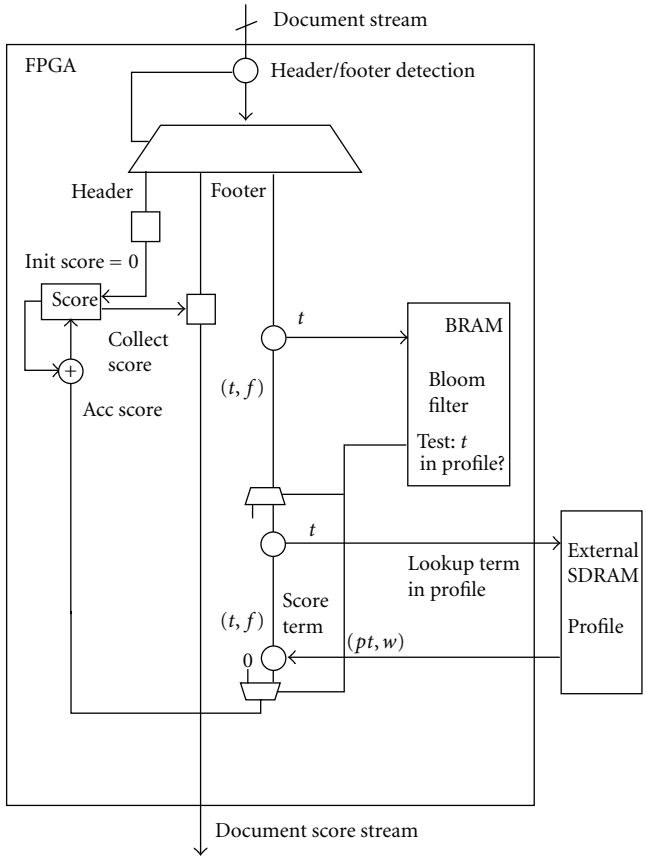

Figure 1: Sequential document term scoring.

$1/\Delta t_S$, with Δt_S the number of cycles required to look up the term weight in the external memory and compute the term score. The use of a Bloom filter greatly improves the throughput as the Bloom filter access will typically be much faster than the external memory access and subsequent score computation. If the probability for a term to occur in the profile is P_P and the access time to the Bloom filter is Δt_B, the average access time will become $\Delta t_B + P_P \cdot \Delta t_S$. In practice P_P will be very low as most document terms will not occur in the profile (because otherwise the profile would match all documents). The more selective the profile, the fewer the number of document terms that match it.

2.3. Parallelising Lookups. The scoring process as described above is sequential. However, as in the bag-of-words representation all terms are independent, there is scope for parallelisation. In principle, all terms of a document could be scored in parallel, as they are independent and ordering is of no importance.

2.3.1. Parallel Document Streams. In practice, even without the bottleneck of the external memory access, the amount of parallelism is limited by the I/O width of the FPGA, in our case 64 bits per memory bank. A document term can be encoded in 32 bits (a 24-bit term identifier and an 8-bit term frequency). As it takes at least one clock cycle of the FPGA clock to read in two new 64-bit words (one per bank), the best case for throughput would be if 4 terms per document would be scored in parallel in a single cycle. However, in practice scoring requires more than one cycle; to account for this, the process can be further parallelised by demultiplexing the document stream into a number of parallel streams. If, for example, scoring would take 4 cycles, then by scoring 4 parallel document streams the application could reach the maximal throughput.

2.4. Parallel Bloom Filter Design. Obviously, the above solution would be of no use if there would be only a single, single-access Bloom filter. The key to parallelisation of the lookup is that because the Bloom filter is stored in on-chip memory, accesses to it can be parallelised by partitioning the Bloom filter into a large number of small banks. The combined concepts of using parallel streams and a partitioned Bloom filter are illustrated in Figure 2. To keep the diagram uncluttered, only the paths of the terms (Bloom filter addresses) have been shown.

Every stream is multiplexed to all m Bloom filter banks; every bank is accessed through an n-port arbiter. It is intuitively clear that, for large numbers of banks, the probability of contention approaches zero, and hence the throughput will approach the I/O limit—or would if none of the lookups would result in an external memory access and score computation.

3. Throughput Analysis

In this section, we present the mathematical throughput analysis of the Bloom filter-based document scoring system. The analysis consists of four parts.

(i) In Section 3.1 we derive an expression to enumerate all possible access patterns for n concurrent accesses to a Bloom filter built of m banks and use it to compute the probability for each pattern.

(ii) In Section 3.2 we compute the average access time for each pattern, given that n_H accesses out of n will result in a lookup in the external memory. We consider in particular the cases of $n_H = 0$ and $n_H = 1$ and propose an approximation for higher values of n_H.

(iii) In Section 3.3 we compute the probability that n_H accesses out of n will result in a lookup in the external memory.

(iv) In Section 3.4, combining the results from Section 3.2 and Section 3.3, we compute the average access time over all n_H for a given access pattern; finally, we combine this with the results from 3.1 to compute the average access time over all access patterns.

3.1. Bloom Filter Access Patterns. We need to calculate the probability of contention between c accesses out of n, for a Bloom filter with m banks. Each bank has an arbiter which sequentialises the contending accesses, so c contending accesses to a given bank will take a time $c \cdot \Delta t_B$, with Δt_B

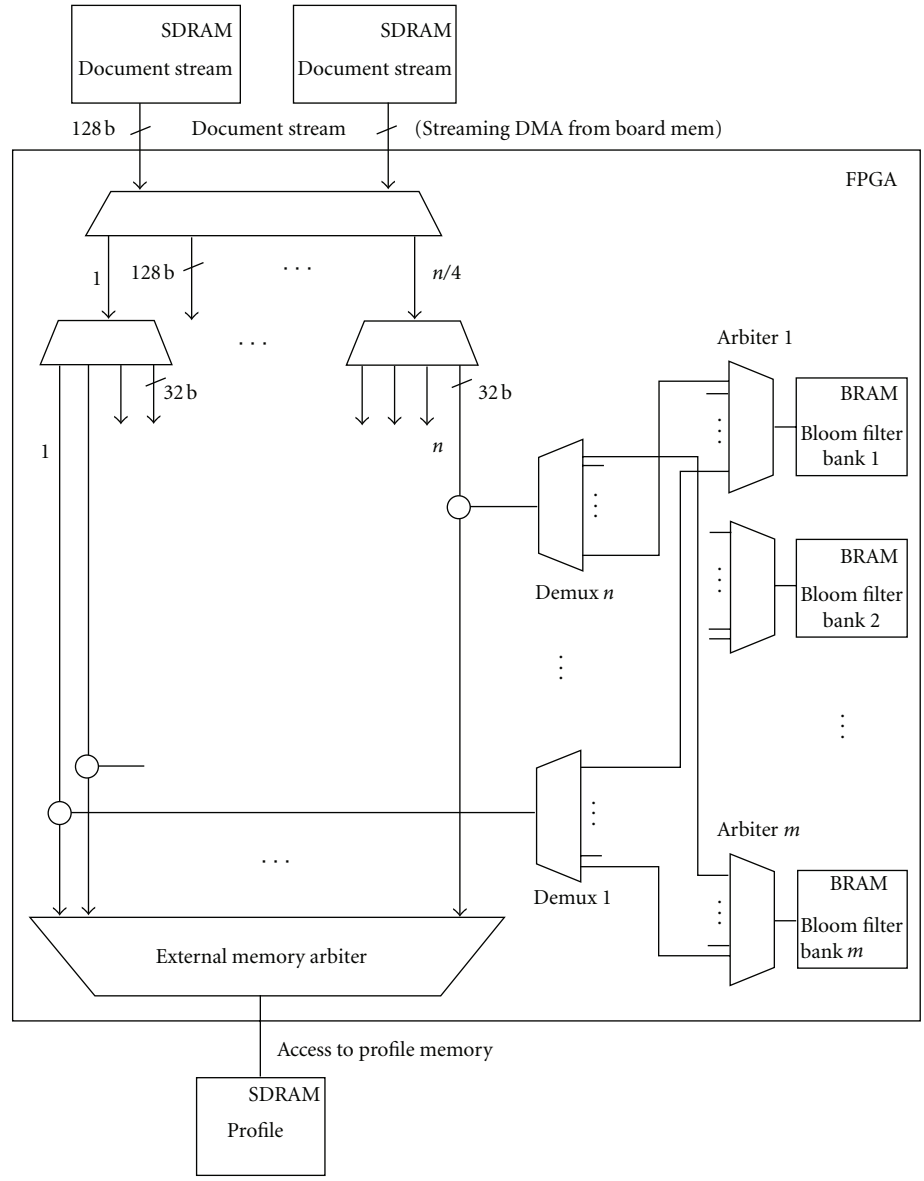

Figure 2: Parallelizing lookups using parallel streams and a multibank Bloom filter.

the time required for a single lookup in the Bloom filter. We also account for a fixed cost of contention Δt_C. We use a combinatorial approach; we count all possible arrangements of n accesses to m banks. Then we count the arrangements that result in c concurrent accesses to a bank.

To do so, we need first to compute the *integer partitions* of n [10] as they constitute all possible arrangements of n accesses. For the remainder of the paper, we will refer to "all possible arrangements that result in x" as the *weight* of x. Each partition of n will results in a particular average access time over all accesses. If we know the probability that each partition will occur and its resulting average access time, we can compute the total average access time.

3.1.1. Integer Partitions. A *partition* $p(n,k)$ of a positive integer n is a nonincreasing sequence of k positive integers $p_1, p_2, \ldots, p_k$ with n as their sum. Each integer p_i is called a *part*. Thus, with n in our case being the number of access ports to our Bloom filter, each partition is a possible access pattern for the Bloom filter. For example, if $n = 16$ and $k = 8$, the partition (5 3 2 2 1 1 1 1) means that the first bank in the Bloom filter gets 5 concurrent accesses, the next 3, and so on. For $n \leq m$, $k \in [1,n]$; if $n > m$, we must restrict k to $k \in [1,m]$ because we cannot have more than m parts in the partition as m is the number of banks. In other words, $k \in [1, \min(n,m)]$. We denote this as $p(n,k)$.

3.1.2. Probability of Each Partition. For each partition, we can compute the probability of it occurring as follows: if there are n concurrent accesses to the Bloom filter's m banks, $n \leq m$, then each access pattern can be written as a sequence of numbers. We are not interested in the actual numbers, but in

the patterns, for example, with $n = 8$ and $m = 16$, we could have a sequence $(a\ a\ a\ b\ b\ c\ c\ d)$, $a, b, c, d \in 0 \ldots m-1; a \neq b \neq c \neq d$ which results in a partition (3 2 2 1). Consequently, given a partition we need to compute the probability for the sequence which it represents. The probability for each number occurring is the same, $1/m$. We can compute this probability as a product of three terms. First, we consider the probabilities for sequences of length n of events with probability α_i where each event occurs x_i times. These are given by the multinomial distribution

$$n! \prod_{i=1}^{k} \frac{\alpha_i^{x_i}}{x_i!}, \tag{3}$$

where $0 < x_i \leq n$ and $n = \sum_{i=1}^{k} x_i$.

In our case, each event has the same probability $1/m$, and the number of times each event occurs is the size of each part p_i in the partition, so

$$\frac{n!}{m^n} \prod_{i=1}^{k} \frac{1}{p_i!}. \tag{4}$$

This gives the probability for a sequence of k groups of p_i events, n events in total.

The actual sequence will consist of numbers $1 \ldots m$, so we must consider the total number of different sequences of numbers that result in a given partition. This is simply the number of possible combinations of k numbers out of m, C_m^k.

Finally, we must consider the permutations as wells for example, for (2 1 1) we must also consider (1 2 1) and (1 1 2). This is a combinatorial problem in which the bins are distinguishable by the number of elements they contain; however, the actual number of elements is irrelevant, only the fact that the bins are distinguishable. The derivation is slightly more complicated. We proceed as follows: we transform the partition into a tuple with as many elements as the number of different integers in the partition, and the value for each element is the number of times this integer occurs in the partition, for example, (5 5 3 3 2 1 1) $\rightarrow$ (2 2 1 2) and (4 3 2 2 1 1 1) $\rightarrow$ (1 1 2 3). We call the new set the frequencies of the partition p, $F(p(n,k))$. As partitions are nonincreasing sequences, the transformation is quite straightforward.

First we create an ordered set $\mathcal{S} = \{S_1, \ldots, S_i, \ldots\}$ with $P = \bigcup S_i$; that is, $\mathcal{S}$ is a set partition of P. The elements of $\mathcal{S}$ are defined recursively as

$$S_1 = \left\{ \forall p_j \in P \mid p_j = p_1 \right\}, \tag{5}$$

$$S_i = \left\{ \forall p_j \in P \setminus \bigcup_{k=1 \ldots i-1} S_k \mid p_j = p_1 \right\}. \tag{6}$$

That is, S_1 contains all parts of P identical to the first part of P; for S_2, we remove all elements of S_1 from P and repeat the process, and we continue recursively until the remaining set is empty. Finally, we create the (ordered) set of the cardinal numbers of all elements of $\mathcal{S}$

$$F = \left\{ f_i \triangleq S_i,\ \forall S_i \in \mathcal{S} \right\}. \tag{7}$$

We are looking for the permutations with repetition of $F(p(n,k))$, which is given by

$$n'! \prod_{\forall f_i \in F\left(p(n,k)\right)} \frac{1}{f_i!}, \tag{8}$$

where $n' = \sum f_i$.

Thus the final probability for each partition of n and a given m becomes

$$\mathcal{P}\left(p(n,k), m\right) = \frac{C_m^k}{m^n} \cdot n! \prod_{\forall p_i \in p(n,k)} \frac{1}{p_i!} \cdot n'! \prod_{\forall f_i \in F\left(p(n,k)\right)} \frac{1}{f_i!}. \tag{9}$$

We observe that

$$\sum_{k=1}^{n} \mathcal{P}\left(p(n,k), m\right) = 1 \tag{10}$$

regardless of the value of m.

In the next section we derive an expression for the access time for a given partition, depending on the number of accesses that will result in an external memory lookup.

3.2. Average Access Time per Pattern. The time to perform n lookups in the Bloom filter is of course determined by the number of contending accesses. For c contending accesses, it will take a time $c\Delta t_B$. However, not all Bloom filter lookups will result in a subsequent access to the external memory—in fact most of them will not, this is exactly the reason for having the Bloom filter. We will call a Bloom filter lookup that results in an access to the external memory a *hit*.

3.2.1. Case of No Hits. First, we will consider the case of 0 hits, that is, the most common case. In this case, the average access time for a given partition $p(n,k)$ is the average of all the parts in the partition

$$\overline{\Delta t_{H,p,0}} = \frac{k_{>1}}{k} \cdot \Delta t_C + \frac{n}{k} \Delta t_B, \tag{11}$$

where $k_{>1}$ is the number of parts $p_i > 1$. For the case of $k = n$ (no contention), $k_{>1} = 0$ so there is no fixed cost of contention Δt_C. Note again that $k \leq \min(n, m)$.

In practice, a small number of Bloom filter lookups will result in a hit, and consequently there is a chance of having one or more hits for concurrent accesses.

3.2.2. Case of a Single Hit. Consider the case of a single hit (out of n lookups). The question we need to answer is, how long on average will it take to encounter a hit? Because as soon as we encounter a hit, we can proceed to perform the external memory access, without having to wait for subsequent hits. This time depends on the particular integer partition. To visualise the partition, we use a so-called Ferrers diagram [11], in which every part is arranged vertically as a list of dots. For example, consider the Ferrers diagram for the partition (8 4 1 1 1 1), that is, $n = 16$, $k = 6$. (Figure 3).

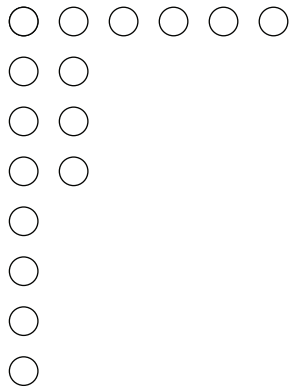

Figure 3: Ferrers diagram for the partition (8 4 1 1 1 1).

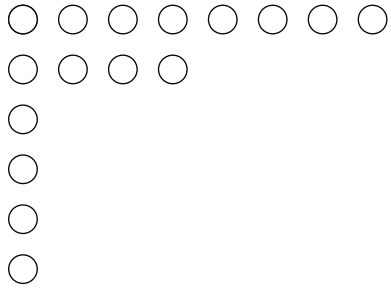

Figure 4: Ferrers diagram for the conjugate partition (6 2 2 2 1 1 1 1).

Each row can be interpreted as the number of concurrent accesses to different banks; each column represents the number of contending accesses to a particular bank.

From this graph it is clear that the probability for finding the hit on the first cycle is 6/16; on the second to fourth cycle 2/16, on the fifth to eighth cycle 1/16. Consequently, the average time to encounter a hit will in this case be

$$1 \cdot \frac{6}{16} + (2+3+4) \cdot \frac{2}{16} + (5+6+7+8) \cdot \frac{1}{16}. \tag{12}$$

To generalise this derivation, we observe first that the transposition of the Ferrers diagram of an integer partition p yields a new integer partition $p'(n,k')$ for the same integer called the *conjugate* partition. In our example $p' = (6\ 2\ 2\ 2\ 1\ 1\ 1\ 1)$ with $k' = 8$ (Figure 4).

We observe that the time it takes to reach a hit in part p'_i is $\Delta t_B \cdot i$. Using the conjugate partition p', we can write the lower bound for average time it takes to reach a hit in partition p as

$$\overline{\Delta t_{H,p,1}} = \left(1 - \frac{k - k_{>1}}{n}\right) \cdot \Delta t_C + \Delta t_B \cdot \frac{1}{n} \sum_{i=1}^{k'} i \cdot p'_i. \tag{13}$$

The term in Δt_C only occurs when the hit is in a bank with contention, that is, in a part greater than 1. There are $k - k_{>1}$ parts of size 1, so the chance of a hit occurring in one of them (i.e., a hit on a bank without contention) is $k - k_{>1}/n$. Thus, the probability for the term in Δt_C is

$$1 - \frac{k - k_{>1}}{n}. \tag{14}$$

And of course, as the hit results in an external access, the average access time is

$$\overline{\Delta t_{A,p,1}} = \left(1 - \frac{k - k_{>1}}{n}\right) \cdot \Delta t_C + \Delta t_B \cdot \frac{1}{n} \sum_{i=1}^{k'} i \cdot p'_i + \frac{1}{n} \cdot \Delta t_S. \tag{15}$$

For the case of $k = n$, the equation reduces to

$$\overline{\Delta t_{A,p,1}} = \Delta t_B + \frac{1}{n} \cdot \Delta t_S. \tag{16}$$

3.2.3. Case of Two or More Hits. If there are two or more hits, the exact derivation would require enumerating all possible ways of distributing n_H hits over a given partition; furthermore, simply enumerating them is not sufficient; we would have to consider the exact time of occurrence of each hit to be able to determine if a subsequent hit was encountered during or after the time to perform an external lookup and compute the score (Δt_S) from a given hit. It is easy to see that, for large n_H, this problem is so complex as to be intractable in practice. However, we can make a simplifying assumption; in practice, Δt_S will be much larger than the time to perform a Bloom filter lookup.

If that is the case, a good approximation for the total elapsed time is the time until the *first* hit is encountered plus n_H times the time for external access. This approximation is exact as long as the time it takes to sequentially perform all external lookups is longer than the time between the best and worst case Bloom filter access time for n_H hits on a single bank, in other words as long as $\Delta t_S > p_i \Delta t_B$. The worst case is of course $p_1 = n$, but this case has a very low probability; for example, for $n = 16$, the average value of all parts is 2.5; even considering only the parts >1, the average is still <4. For $n = 32$, the numbers are, respectively, 3 and 5. In practice, if $\Delta t_S / \Delta t_B > 10$, the error will be negligible.

Conversely, we could consider the time until the *last* hit is encountered plus n_H times Δt_S. This approximation provides an upper bound for the access time.

Therefore, we are only interested in these two cases, that is, the lowest, respectively, highest part of the partition with at least one hit. We need to compute the probability that the lowest (resp. highest) part will contain a hit, and the next but lowest (resp. highest) one, and so forth. For simplicity, we leave off Δt_C in the following derivation.

Lower Bound. The number of all possible cases is $N_{p'} = C(n, n_H)$, all possible arrangements of n_H elements in n bins. To compute the weight of a hit in the lowest part p'_1, we compute the complement, all possible arrangements without any hits in p'_1. That means that we remove p'_1 from n. Then, using the notation $\neg p_1$ for "not a hit in p'_1," we compute

$$N_{\neg p_1} = C(n - p'_1, n_H). \tag{17}$$

These are all the possible cases for not having a hit in p'_1. Thus, $N_{p_1} = N_p - N_{\neg p_1}$ is the number of possible arrangements with $1 \dots n_H$ hits in p'_1.

We now do the same for p_2, and so forth. That gives us all possible cases for *not* having a hit in p_i

$$N_{\neg p_i} = C\left(n - \sum_{j=1}^{i} p'_j, n_H\right). \tag{18}$$

Obviously, there must be enough space in the remaining parts to accommodate n_H hits, so i is restricted to values where

$$n - \sum p'_i \geq n_H. \tag{19}$$

We call the highest index for which (19) holds, k^*.

To obtain the weight of a hit in p'_i, we must of course subtract the weight of a hit in p'_{i-1}, because $N_p - N_{\neg p_i}$ would give the weight for having a hit in all parts up to p_i. It is easy to show (by substitution of (17)) that

$$N_{p_i} = N_{\neg p_{i-1}} - N_{\neg p_i}. \tag{20}$$

Finally, the average time it takes to reach a part in a given p' with at least one hits out of n_H is

$$\overline{\Delta t_{H,p,n_H}} = \Delta t_B \cdot \frac{1}{N_{p'}} \sum_{i=1}^{k^*} i \cdot N_{p_i}. \tag{21}$$

With the above assumption, the average access time for n_H hits can then be approximated as

$$\overline{\Delta t_{A,p,n_H}} = \left(1 - \frac{k - k_{>1}}{n}\right) \cdot \Delta t_C + \Delta t_B \cdot \frac{1}{N_{p'}} \sum_{i=1}^{k^*} i \cdot N_{p_i} + n_H \Delta t_S. \tag{22}$$

We observe that for $n_H = 1$, (22) indeed reduces to (15) as $N_{p'} = n$ and $N_{p_i} = p'_i$. For $n = k$ the equation reduces to $\Delta t_B + n_H \Delta t_S$.

Upper Bound. The upper bound is given by the probability that the highest part is occupied, and so forth, so the formula is the same as (18) but starting from the highest part p_c, that is,

$$N_{\neg p_{c-i}} = C\left(n - \sum_{j=c-i+1}^{c} p'_j, n_H\right) \tag{23}$$

with the corresponding restriction on i that

$$\sum_{i=1}^{k^*} p'_i \geq n_H. \tag{24}$$

As we will see in Section 3.5, in practice the bounds are usually so close together that the difference is negligible.

3.3. Probability of External Memory Access. The chance that a term will occur in the profile depends on the size of the profile $N_{\mathcal{P}}$ and the size of the vocabulary $N_{\mathcal{V}}$

$$P_{\mathcal{P}} = \frac{N_{\mathcal{P}}}{N_{\mathcal{V}}}. \tag{25}$$

This is actually a simplified view; it assumes that the terms occurring in the profile and the documents are drawn from the vocabulary in a uniform random way. In reality, the probability depends on how discriminating the profile is. As the aim of a search is of course to retrieve only the relevant documents, we can assume that actual profiles will be more discriminating than the random case. In that case (25) provides a worst case estimate of contention.

The probability of n_H hits, that is, contention between n_H accesses to the external memory is then

$$C(n, n_H) \cdot P_{\mathcal{P}}^{n_H} \cdot (1 - P_{\mathcal{P}})^{n - n_H}. \tag{26}$$

That is, there are $C(n, n_H)$ arrangements of n_H accesses out of n, and, for each of them, the probability that it occurs is $P_{\mathcal{P}}^{n_H} \cdot (1 - P_{\mathcal{P}})^{n-n_H}$. Furthermore, n_H contending accesses will take a time $n_H \Delta t_S$. Of course, if no external access is made, the external access time is 0.

3.4. Average Overall Throughput

3.4.1. Average Access Time over All n_H for a Given Pattern. We can now compute the average access time over all n_H for a given access pattern p by combining (22) and(26)

$$\overline{\Delta t_{A,p}} = \sum_{n_H=0}^{n} C(n, n_H) \cdot P_{\mathcal{P}}^{n_H} \cdot (1 - P_{\mathcal{P}})^{n-n_H} \overline{\Delta t_{A,p,n_H}}. \tag{27}$$

3.4.2. Average Access Time over All Patterns for Given n and m. Finally, using (9) and (27), we can compute the average access time over all patterns for given n and m, that is, the average overall throughput of the application with n parallel threads and an m-bank Bloom filter

$$\overline{\Delta t_A(n, m)} = \sum_{\forall p(n)} \mathcal{P}(p(n, k), m) \cdot \overline{\Delta t_{A,p}}. \tag{28}$$

3.5. Analysis. In this section the expression obtained in Section 3.4 is used to investigate the performance of the system and the impact of the values of n, m, Δt_B, Δt_S, and $P_{\mathcal{P}}$ on the throughput.

3.5.1. Accuracy of Approximation. To evaluate the accuracy of the approximations introduced in Section 3.2.3, we compute the relative difference between the "first hit" approximation and the "upper bound" approximation. From Figure 5, it can be seen that the difference is less than 1% of the throughput over all simulated cases. As the upper bound always overestimates the delay, and the "first hit" approximation will in most cases return the correct delay, this demonstrates that both approximations are very accurate. An interesting observation is that for $\Delta t_B = 4$ the error is almost the same as

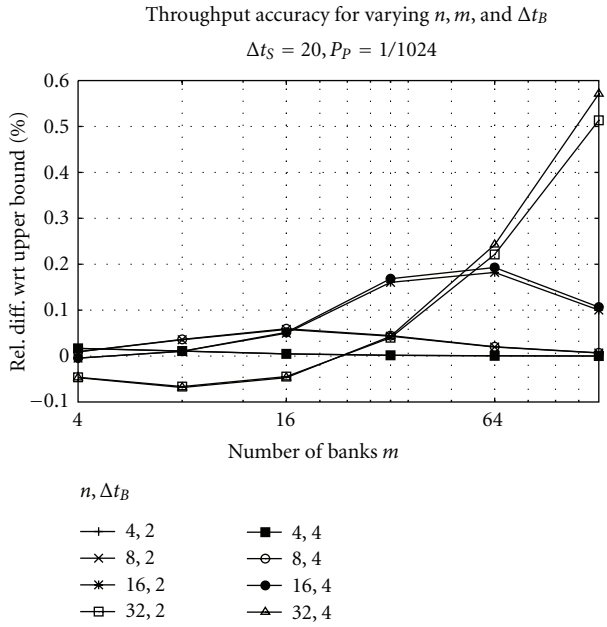

FIGURE 5: Accuracy of the approximation for $n_H \geq 2$.

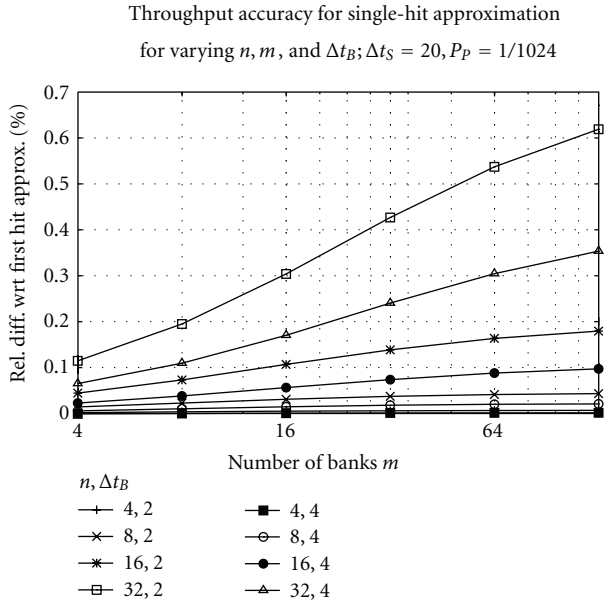

FIGURE 6: Accuracy of the single-hit approximation for $n_H \geq 2$.

for $\Delta t_B = 2$, which illustrates that the condition $\Delta t_S > p_i \Delta t_B$ is sufficient but not necessary.

Next, we consider a more radical approximation; we assume that, for $n_H > 1$, $P_{\mathcal{P}} = 0$, in other words we ignore all cases with more than 1 hit.

From Figure 6 we see that the relative difference between the throughput using this approximation and the "first hit" is very small, to such an extent that in almost all cases it is justified to ignore $n_H > 1$. This is a very useful result as this approximation speeds up the computations considerably.

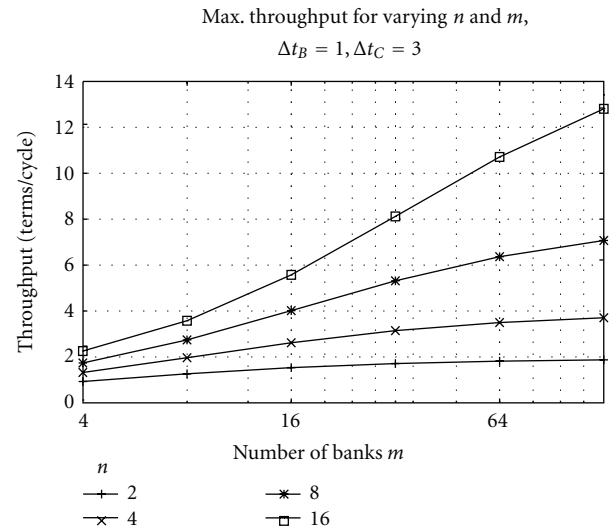

FIGURE 7: Best case (0 hits) average access time for a Bloom filter with m banks and n access ports, $\Delta t_B = 1$, $\Delta t_C = 3$.

3.5.2. Maximum Achievable Throughput. The throughput depends on the number of hits in the Bloom filter. Let us consider the case where the Bloom filter contains no hits at all. This is the maximum throughput the system could achieve, and it corresponds to a profile for which no document in the stream has any matches. We can use (11) and (9) to calculate the best-case average access time for a Bloom filter with m banks and n access ports

$$\overline{\Delta t_{\min}(n,m)} = \sum_{k=1}^{n} \sum_{\forall p(n,k)} \left(\left(\frac{k_{>1}}{k} \Delta t_C + \frac{n}{k} \Delta t_B \right) \cdot \mathcal{P}(p(n,k),m) \right). \tag{29}$$

Note that for $m < n$, $\mathcal{P}(p(n,n)) = 0$.

The results are shown in Figure 7. The figure shows that for $\Delta t_B = 1$, $\Delta t_C = 3$ (the values for our current implementation), the I/O-limited throughput (4 terms/cycle for the PROCStar-III board) is achieved with $n = 8$ and $m = 16$. That means that we need to demultiplex both input streams into 4 parallel streams because each 64-bit word contains 2 terms.

3.5.3. Throughput Including External Access. Figure 8 shows the effect of the external memory access and score computation. The important observation is that the performance degradation is quite small for low hit rates, and still only around 25% for a relatively high hit rate of 1/512. This demonstrates that the assumptions underlying our design are justified.

3.5.4. Impact of Bloom Filter Access Time. A further illustration of the impact of Δt_B is given in Figure 9, which plots the throughput as a function of Δt_B on a log/log scale. This figure illustrates clearly how a reduction in throughput as result of slower Bloom filter access can be compensated for by increasing the number of access streams. Still, with $\Delta t_B = 4$,

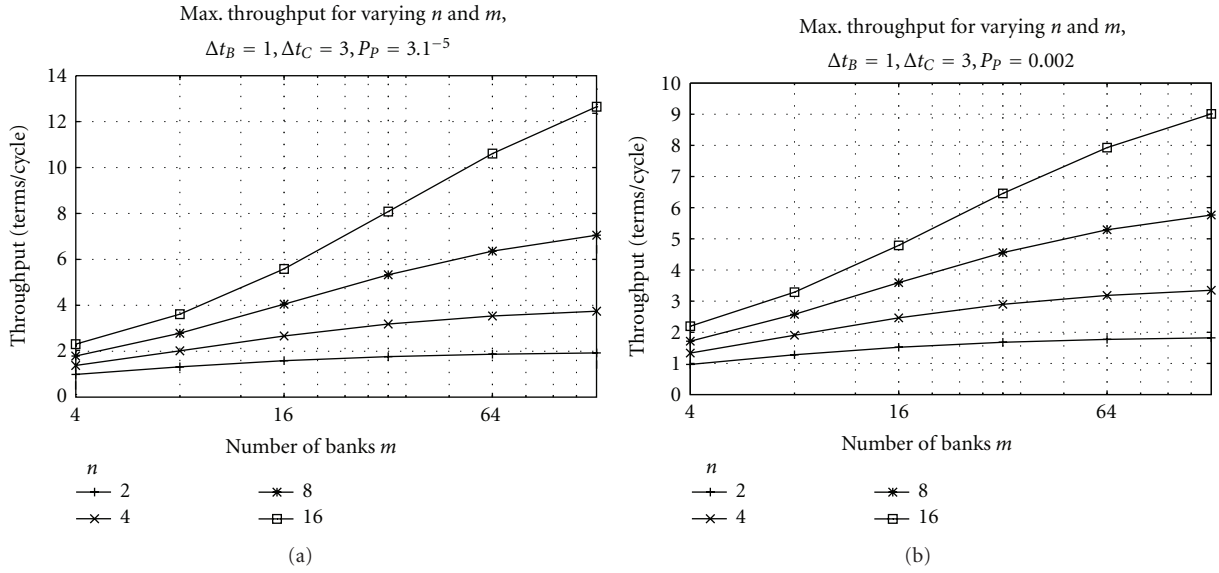

Figure 8: Average access time for a Bloom filter with m banks and n access ports, $\Delta t_S = 20$, (a) $P_{\mathcal{P}} = 3 \cdot 10^{-5}$ (b) $P_{\mathcal{P}} = 0.002$.

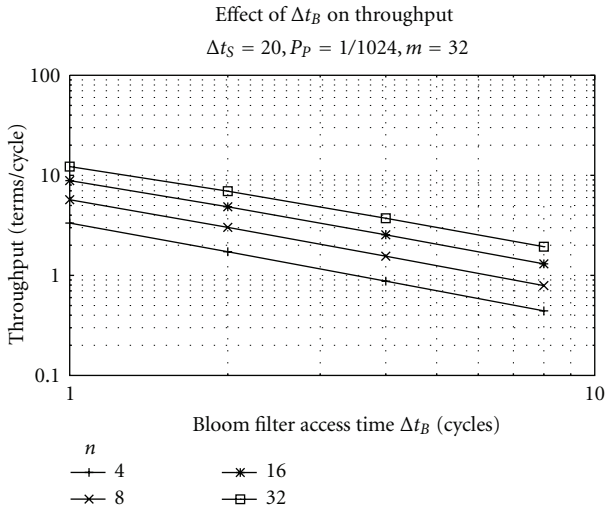

Figure 9: Impact of Bloom filter access time on throughput.

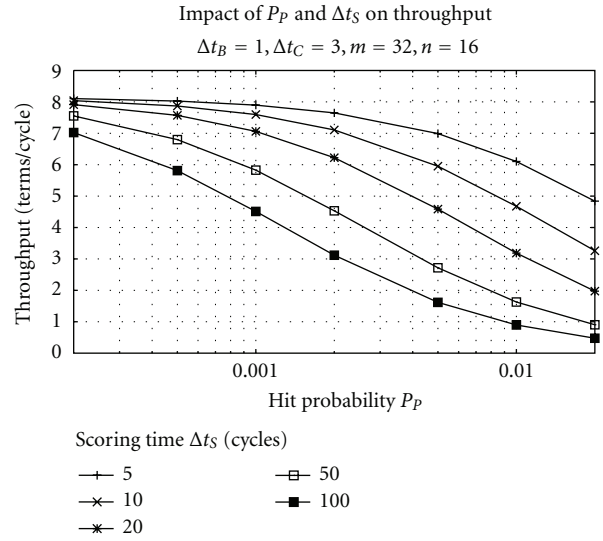

Figure 10: Impact on throughput of hit probability and external memory access time.

we would need 32 parallel streams per input stream, or we would need a very large number ($\gg 128$) Bloom filter banks. On the one hand, the upper limit is 512 (the number of M9K blocks on the Stratix-III 260E FPGA); on the other hand, the size of the demultiplexers and arbiters would become prohibitive as it grows as $m \cdot n$.

3.5.5. Impact of Profile Hit Probability and External Memory Access Time. The final figure (Figure 10) is probably the most interesting one. It shows how, for very selective profiles (i.e., profiles resulting in very low hit rates), the effect of long external memory access times is very small.

4. FPGA Implementation

We implemented our design on the GiDEL PROCStar-III development board (Figure 11). This system provides an extensible high-capacity FPGA platform with the GiDEL PROC-API library-based developer kit for interfacing with the FPGA.

4.1. Hardware. Each board contains four Altera Stratix-III 260 E FPGAs running at 125 MHz. Each FPGA supports a five-level memory structure, with three kinds of memory blocks embedded in the FPGA:

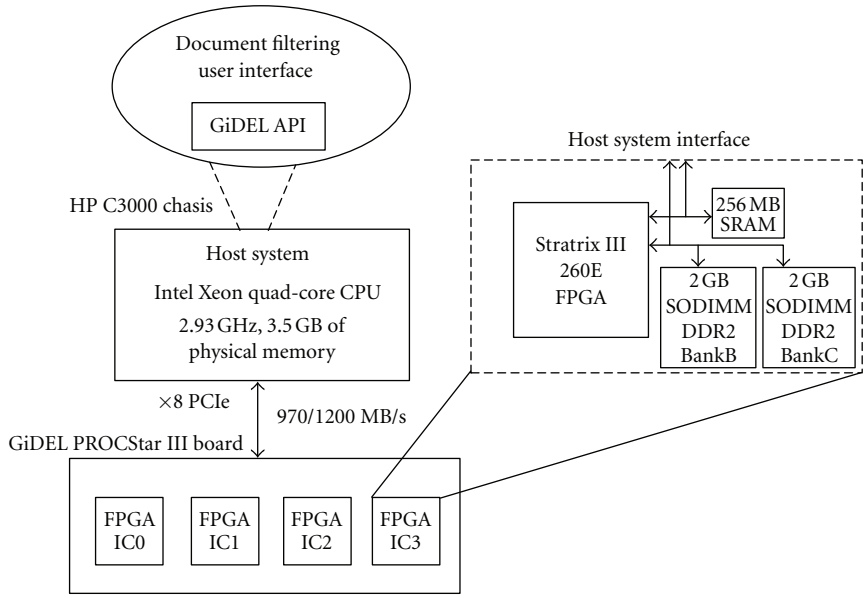

Figure 11: Block diagram of FPGA platform and photograph of experimental hardware.

(i) 5,100 MLAB RAM blocks (320 bit),

(ii) 864 M9K RAM blocks (9 Kbit), and

(iii) 48 M144K blocks (144 Kbit)

and 2 kinds of external DRAM memory:

(i) 256 MB DDR2 SDRAM onboard memory (Bank A) and

(ii) two 2 GB SODIMM DDR2 DRAM memories (Bank B and Bank C).

The embedded FPGA memories run at a maximum frequency of 300 MHz, Bank A and Bank B at 667 MHz, and Bank C at 360 MHz. The FPGA-board is connected to the host platform via 8-lane PCI Express I/O interface. The host system consists of a quad-core 64-bit Intel Xeon X5570 CPU with a clock frequency of 2.93 GHz and 3.5 GB DDR2 DRAM memory, the operating system is 32-bit Windows XP. The host computer transfers data to the FPGA using 32-bit DMA channels.

4.2. Development Environment. FPGA-accelerated applications for the PROCStar board are implemented in C++ using the GiDEL PROC-API libraries for interacting with the FPGA. This API defines a hardware abstraction layer that provides control over each hardware element in the system; for example, Memory I/O is implemented using the GiDEL MultiFIFO and MultiPort IPs. To achieve optimal performance, we implemented the FPGA algorithm in VHDL (as opposed to Mitrion-C as used in our previous work). We used the Altera Quartus toolchain to create the bitstream for the Stratix-III.

4.3. FPGA Implementation Description. Figure 12 presents the overall workflow of our implementation. The input stream of document term pairs is read from the SDRAM via a FIFO. A Bloom filter is used to discard negatives (terms that do not appear in the profile) for multiple terms in parallel. Profile weights are read corresponding to the positives, and the scores are computed for each term in parallel and accumulated to achieve the final score described in (1). Below, we describe the key modules for the implementation: document streaming, profile negative hit filtering, and profile lookup and scoring.

4.3.1. Document Streaming. Using a bag-of-words representation (see Section 2) for the document, the document stream is a list of *(document id, document term tuple set)* pairs. The FPGA accepts a stream of 64-bit words from the 2 GB DRAM (Bank B). Consequently, the document stream must be encoded onto this word stream. The document term tuple $d_i = (t_i, f_i)$ can be encoded in 32 bits: 24 bits for the term id (supporting a vocabulary of 16 million terms) and 8 bits for the term frequency. Thus, we can combine two tuples into a 64-bit word. To mark the start and end of a document, we insert a marker words (64 bits) followed by the document id (64 bits).

4.3.2. Profile Negative Hit Filtering. As described in Section 2, we implemented a Bloom filter in the FPGA's on-chip MRAM (M9K blocks). The higher internal bandwidth of the MRAMs leads to very fast rejection of negatives. Although the MRAM is fast, concurrent lookups lead to contention. To reduce contention we designed a distributed Bloom filter. Based on the analysis presented in this paper, the Bloom filter memory is distributed over a large number of banks (16 in the current design) and a crossbar switch connects the document terms streams to the banks. As shown in our analysis, in this way contention is significantly reduced. The design was implemented as shown in Figure 2, but due to

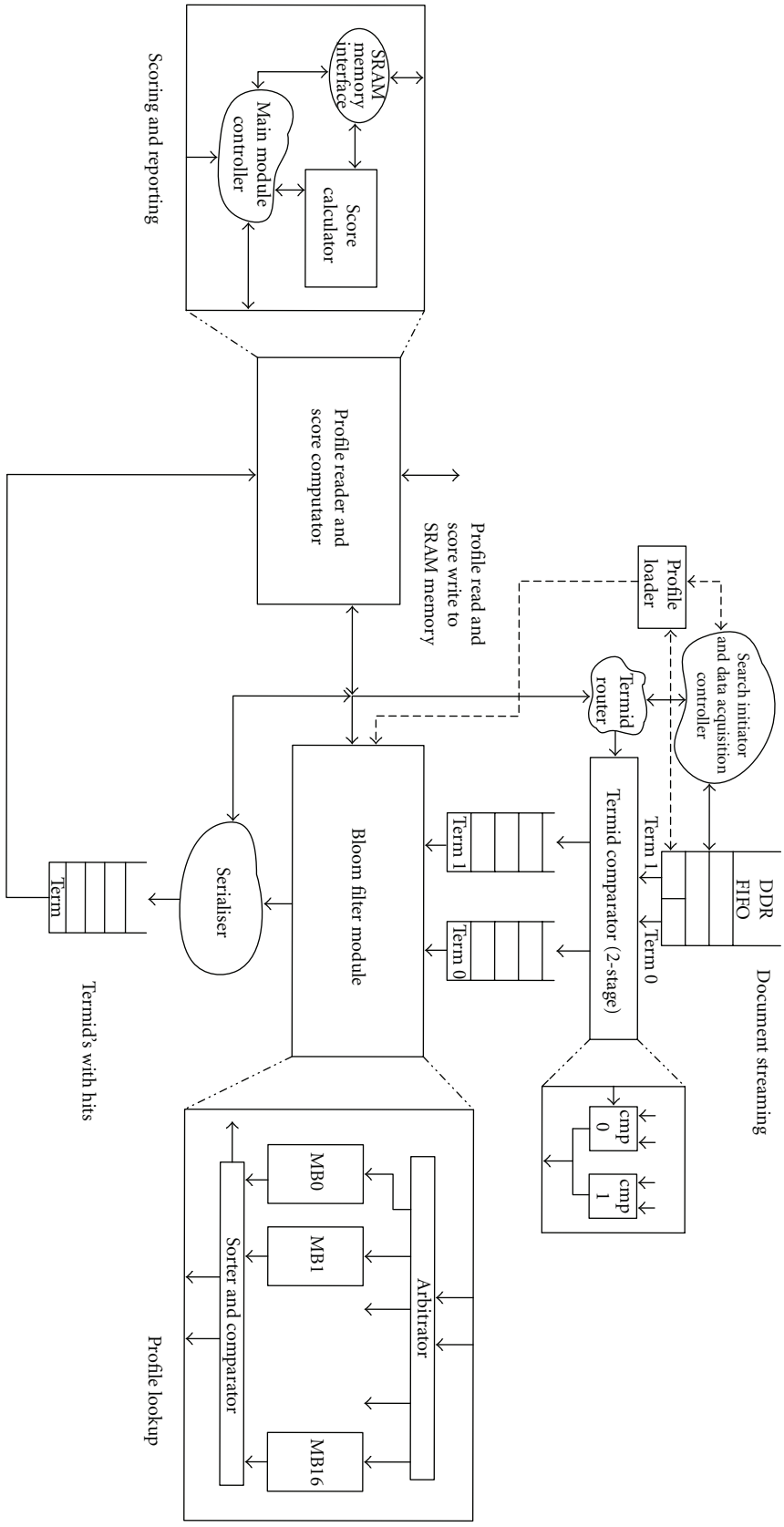

Figure 12: Overall block diagram of FPGA implementation.

an issue with the board we could only use one SDRAM to store the collection. As a result we have only two parallel terms in the current implementation.

4.3.3. Profile Lookup and Scoring. As explained in Section 2.2, the actual lookup and scoring system is quite straightforward; the input stream is scanned for header and footer words. The header word action is to store the subsequent document ID and to set the corresponding document score to 0; the footer word action is to collect and output the *(document ID, document score)* pair if the score exceeds the threshold. For every two terms in the document, first the Bloom filter is used to discard negatives, and then the weights corresponding to positives are read from the SDRAM. The score is computed for each of the terms in parallel and added. The score is accumulated for all terms in the document, and finally the score stream is filtered against a limit before being output to the host. Figure 13 summarises the implementation of the profile lookup and scoring.

4.3.4. Discussion. The implementation above leverages the advantages of an FPGA-based design, in particular the memory architecture of the FPGA; on a general-purpose CPU-based system, it is not possible to create a very fast, very low-contention Bloom filter to discard negatives. Also, a general-purpose CPU-based system only has a single, shared memory. Consequently, reading the document stream will contend for memory access with reading the profile terms, and as there is no Bloom filter, we have to look up each profile term. We could of course implement a Bloom filter, but, as it will be stored in main memory as well, there is no benefit; looking up a bit in the Bloom filter is as costly as looking up the term directly. Furthermore, the FPGA design allows for lookup and scoring of several terms in parallel.

4.4. FPGA Utilisation Details. Our implementation used only 11,033 of the 203,520 logic elements (LEs) or a 5% utilisation of the logic in the FPGA, and 4,579,824 out of 15,040,512 for a 30% utilisation of the RAM. Of the 11,033 LEs utilised by whole design on the FPGA, the actual document filtering algorithm only occupied 1,655 LEs, which is less than 1% of utilisation, and rest was used by the GiDEL Memory IPs. The memory utilised for the whole design (4,579,824 bits) was mainly for the Bloom filter that is mapped on embedded memory blocks (MRAMs). The Quartus PowerPlay Analyzer tool estimates the power consumption of the design to be 6 W. The largest contribution to the power consumption is from the memory I/O.

5. Evaluation

In this section we discuss our evaluation results. We present our experimental methodology and the data summarising the performance of our FPGA evaluation and comparison with non-FPGA-accelerated baselines, and we conclude with the learnings from our experiments.

Table 1: Summary statistics from representative real-world collections that we used as templates for our synthetic data sets.

Collection	No. docs.	Avg. Doc. Len.	Avg. Uniq. Terms
Aquaint	1,033,461	437	169
USPTO	1,406,200	1718	353
EPO	989,507	3863	705

5.1. Creating Synthetic Data Sets. To accurately assess the performance of our FPGA implementation, we need to exercise the system on real-world input data; however, it is hard to get access to such real-world data; large collections such as patents are not freely available and governed by licenses that restrict their use. For example, although the researchers at Glasgow University have access to the TREC Aquaint collection and a large patent corpus, they are not allowed to share these with a third party. In this paper, therefore, we use synthetic document collections statistically matched to real-world collections. Our approach is to leverage summary information about representative datasets to create corresponding language models for the distribution of terms and the lengths of documents; we then use these language models to create synthetic datasets that are statistically identical to the original data sets. In addition to addressing IP issues, synthetic document collections have the advantages of being fast to generate and easy to experiment with, and not taking up large amounts of disk space.

5.1.1. Real-World Document Collections. We analysed the characteristics of several document collections—a newspaper collection (TREC Aquaint) and two collections of patents from the US Patent Office (USPTO) and the European Patent Office (EPO). These collections provide good coverage on the impact of different document lengths and sizes of documents on filtering time. We used the Lemur (http://www.lemurproject.org/) Information Retrieval toolkit to determine the rank frequency distribution for all the terms in the collection. Table 1 shows the summary data from the collections we studied as templates.

5.1.2. Term Distribution. It is well known (see, e.g., [12]) that the rank-frequency distribution for natural language documents is approximately Zipfian

$$f(k;s;N) = \frac{1/k^s}{\sum_{n=1}^{N} 1/n^s}, \tag{30}$$

where f is frequency of term with rank k in randomly chosen text of natural language, N is number of terms in the collection, and s is an empirical constant. If $s > 1$, the series becomes a value of a Riemann ζ-function and will therefore converge. This type of distribution approximates a straight line on a log-log scale. Consequently, it is easy to match this distribution to real-world data with linear regression.

Special purpose texts (scientific articles, technical instructions, etc.) follow variants of this distribution. Montemurro [13] has proposed an extension to Zipf's law which better captures the linguistic properties of such collections.

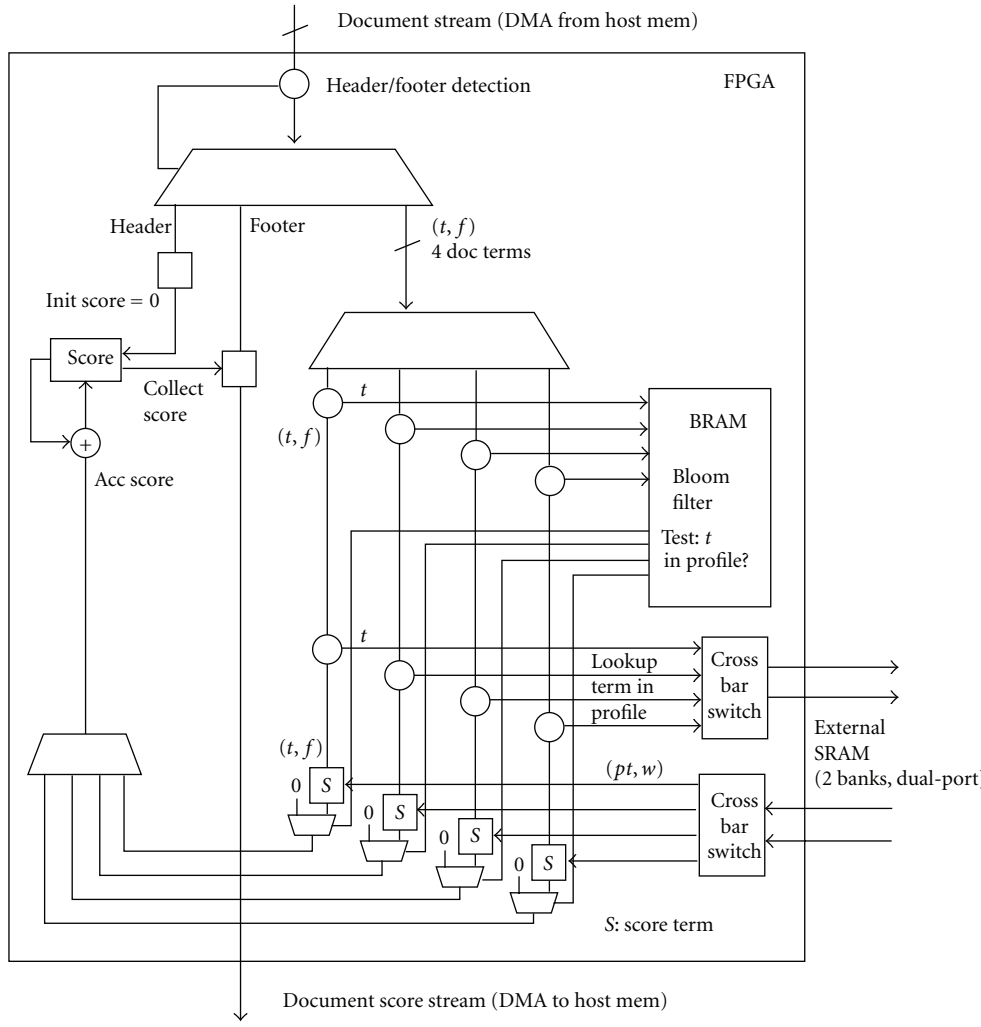

Figure 13: Implementing profile lookup and scoring.

His proposal is based on observation that, in general after some pivot point p, the probability of finding a word of rank r in the text starts to decay much faster than in the beginning. In other words, in log-log scale, the low-frequency part of the distribution has a steeper slope than the high-frequency part. Consequently, the distribution can be divided into two regions each obeying the power law, but with different slopes

$$F(r) = \begin{cases} a_1 r + b_1 & r < p, \\ a_2 r + b_2 & \text{otherwise}, \end{cases} \tag{31}$$

We determine the coefficients a_1, a_2, b_1, and b_2 from curve-fitting on the summary statistics from the real-world data collections. Specifically, we use the sum of absolute errors as the merit function combined with a binary search to obtain the pivot. We then use a least-squares linear regression, with χ^2 statistics as a measure of quality (taken from [14]). A final normalisation step is added to ensure that the piecewise linear approximation is a proper probability density function.

5.1.3. Document Length. Document lengths are sampled from a truncated Gaussian. The hypothesis that the document lengths in our template collections have a normal distribution was verified using a χ^2 test with 95% confidence. The sampled values are truncated at the observed minimum and maximum lengths in the template collection.

Once the models for the distribution of terms and document lengths are determined, we use these models to create synthetic documents of varying lengths. Within each document, we create terms that follow the fitted rank-frequency distribution. Finally, we convert the documents into the standard bag-of-words representation, that is, a set of unordered *(term, frequency)* pairs.

5.2. Experimental Parameters. Statistically, the synthetic collection will have the same rank-frequency distribution for the terms as the original data sets. Consequently, the probability that a term in the collection matches a term in the profile will be the same in the synthetic collection and the original

collection. The performance of the algorithm on the system now depends on

(i) the size of the collection,

(ii) the size of the profile,

(iii) the "hit probability," that is, the probability that the profile corresponding to a term has a nonzero weight.

To evaluate these effects, we studied a number of different configurations—with different document sizes, different profile lengths, and different profile constructions. Specifically, we studied profile sizes of 4 K, 16 K, and 64 K terms, the first two are of the same order of magnitude as the profile sizes for TREC Aquaint and EPO as used in our previous work [2], and the third, larger profile was added to investigate the impact of the profile size. We studied two different document collections: 128 K documents of 2048 terms, which is representative for the patent collections, and 512 K documents of 512 terms, similar to the Aquaint collection. Note that the total size of the collection is not important for the performance evaluation; for both the CPU and FPGA implementation, the time taken to filter a collection is proportional to its size.

We evaluated four ways of creating profiles. The first way ("Random") is by selecting a number of random documents from the collection until the desired profile size is reached. These documents were then used to construct a relevance model. The relevance model defined the profiles which each document in the collection was matched against (as if it were being streamed from the network). The second type of profiles ("Selected") was obtained by selecting terms that occur in very few documents (less than ten in a million). For our performance evaluation purpose, the main difference between these profiles is the hit probability, which was 10^{-5} for the "Random" profiles and $5 \cdot 10^{-4}$ for the "Selected" profiles. For reference, we also compared the performance against an "Empty" profile (one that results in no hits).

5.3. FPGA Performance Results

5.3.1. Access Time Measurements. The performance of the FPGA was measured using a cycle counter. The latency between starting the FPGA and the first term score is 22 cycles. For the subsequent terms, the delay depends on a number of factors. We considered three different cases:

(i) "Best Case": no contention on the Bloom filter access and no external memory access

(ii) "Bloom Filter Contention": contention on the Bloom filter access for every term but no external memory access

(iii) "External Access": no contention on the Bloom filter access, external memory access for every term

These cases were obtained by creating documents with contending/not contending term pairs and by setting all Bloom filter bits to 0 (no external access, which corresponds to an empty profile) or 1 (which correspond to a profile that would contain all terms in the vocabulary).

Table 2: FPGA Cycle counts for different cases.

Case	No. cycles/2 terms	Probability
Best case	1	.9375
Bloom filter contention	5	.0625
External access	37	<0.00001

The results are shown in Table 2. As we read two terms in parallel, the Best Case (i.e., the case of no contention and no hits) demonstrates that the FPGA implementation does indeed work at I/O rates, that is, $\Delta t_B = 1$. The table also shows the probability for each case when filtering an actual document collection.

The most interesting result in Table 2 is the "Bloom Filter Contention," which shows that in our design $\Delta t_C = 3$. The case of "External Access," which means no contention on the Bloom filter, and lookup of both terms in the external memory shows that $\Delta t_S = 18$.

As explained in Section 3, the Bloom filter contention depends on the number of Bloom filter banks (2 parallel terms, 16 banks), and, in the current design, the contention probability is 1/16 (from (9)). The probability for external access depends on the actual document collection and profile, but as the purpose of a document filter is to retrieve a small set of highly relevant documents, this probability is typically very small, as demonstrated by the experiments discussed in the next section. Consequently, the typical performance is determined by the cycle counts for Best Case and Bloom Filter Contention. Using (29), we get $(1/2)\Delta t_B \cdot (15/16) + ((1/2)\Delta t_C + \Delta t_B) \cdot (1/16) = 0.625$ cycles per term. At a clock speed of 125 MHz, this results in a throughput of 200 million terms per second (200 MT/s) per FPGA.

5.3.2. Comparison with CPU Reference Systems. Table 3 presents performance results for our FPGA implementation for various workload types. Focusing on a "Random" profile of 16 K terms for 128 K documents, our measured performance is close to 800 million terms/second for the design—close to the estimated performance; the earlier calculations showed 200 million terms/second *per FPGA*: across the four FPGAs in the GiDEL board, that translates to 800 million terms/second for the design. Table 3 also shows the sensitivity to various other parameters. The performance of the FPGA design is comparable for different profile sizes and document sizes. However, as expected, the performance varies based on different hit probabilities for different profiles.

To compare the FPGA performance against a conventional CPU, we ran the experiments discussed in Section 5.1 on an optimised reference implementation (compared to the Lemur-based implementation used in our previous work), written in C++, compiled with g++ with optimisation -O3, and run on two different platforms: *System1* (an iMac) has an Intel Core 2 Duo Mobile E8435 CPU with clock frequency 3.06 GHz and 8 GB RAM, bus speed 1067 MHz. *System2* (a Linux server) has an Intel Core i7-2600 CPU running at 3.4 GHz, with 16 GB RAM, bus speed 1333 MHz. The higher memory configurations are required to enable sufficient

Table 3: Throughput of document filtering application (M terms/s) for (a) 128 K documents of 2048 terms and (b) 512 K documents of 512 terms.

(a)

Profile	System1	System2	FPGA board
Empty, 4 K	31	48	800
Empty, 16 K	31	48	800
Empty, 64 K	31	48	800
Random, 4 K	25	42	800
Random, 16 K	24	41	800
Random, 64 K	24	41	800
Selected, 4 K	21	37	792
Selected, 16 K	18	35	792
Selected, 64 K	18	25	792

(b)

Profile	System1	System2	FPGA board
Empty, 4 K	30	53	800
Empty, 16 K	32	53	800
Empty, 64 K	32	53	800
Random, 4 K	26	47	800
Random, 16 K	26	46	800
Random, 64 K	25	46	800
Selected, 4 K	20	40	796
Selected, 16 K	19	38	792
Selected, 64 K	17	27	796

memory for the algorithm; it is not possible to run the reference implementation on the 32-bit Windows XP server which hosts the FPGA board as the 3.5 GB of memory is not sufficient. We could of course run the algorithm several times on smaller data sets but then in that case the time required to read the data from disk would dominate the performance. We keep the entire data set in memory because the memory I/O is much higher than the disk I/O. While this approach might not be practical on a CPU-based system, on the FPGA-based system this is entirely practical as the PROCStar-III board has a memory capacity of 32 GB. This means that, for example, the Novo-G FPGA supercomputer, which hosts 48 PROCStar-III boards, can support a collection of 1.5 TB. Note also that the format in which the documents are stored on the disk is a very efficient bag-of-words representation, which is much smaller than the actual textual representation of the document.

The results are summarised in Table 3. For example, focusing on one example case, for the random profile with 16 K terms and 128 K documents, compared to the 800 million terms/second performance per FPGA achieved by our design, the *System2* system achieves 41 million terms/second, and the *System1* system achieves 24 million terms/sec. This translates to a 36-fold speedup for the FPGA-based design relative to the *System1* system and a 20-fold speedup relative to the *System2* system. Additionally, examining the results for various workload configurations, the FPGA's performance is relatively constant across different workload inputs. This bears out the rationale for our design because in general hits are rare, the FPGA works at the speed determined by I/O and Bloom Filter performance. Unlike the FPGA-based design, the CPU-based system sees more variation in performance with profile size (degraded performance with increased profile size) and document size (degraded performance with larger documents) and a bigger dropoff in performance between various profile types compared to the FPGA-based design.

6. Discussion

In the above sections we have used a preliminary implementation of our proposed design to validate the analytical model. The design does indeed behave in line with the model, for the case of two parallel terms and a 16-bank Bloom filter. The performance is 200 M terms/s. This design is not optimal for several reasons. On the one hand, the original aim was to support four parallel terms, but an issue with the access to one of the memories prevented this. On the other hand, as is clear from the model, a 16-bank implementation does not result in operation close to I/O rates. For four parallel terms, this would require 64 banks; even for two parallel terms, the performance is 80% of the I/O rate. Our aim was not so much to achieve optimal performance as to implement and evaluate our novel design and compare it to the analytical model. We therefore decided to limit the number of banks to 16 to reduce the complexity of the design, as the implementation was undertaken as a summer project.

This means that there is a lot of scope for improving the current implementation.

(i) We will deploy our design on a PROCStar-IV board which does not have this issue, and thus we will be able to score 4 terms in parallel rather than 2 terms.

(ii) Even with a single SDRAM, we can be more efficient; the SDRAM I/O rate is 4 GB/s (according to the PROCStar-III databook); our current rate is only 1 GB/s. By demultiplexing the scoring circuit, it should be possible to increase this rate to 4 GB/s.

(iii) Combining both improvements, an improved design could score 16 terms in parallel. This will of course require a Bloom filter with more banks to reduce contention, but considering the current resource utilisation that is not a limitation. Consequently, the improved design should be able to operate up to 8× faster than the current design.

In terms of the analytical model itself, there is some scope for further refinement, in particular for the external access; we currently use a single access time for one and more hits. Just like for the Bloom filter, we can include a fixed cost for concurrent accesses on the external memory. We also want to refine the model to include the effect of grouping terms: that is, the n parallel terms are usually grouped per two or four depending on the I/O width. This affects the waiting time on contention, as all terms in a single group need to wait before a new group can be fetched. Currently, the model

assumes all terms are independent. For the case of two terms, this assumption is corrects for more terms there is a slight underestimation of the access time in the case of contention. The counting problem for this case is complicated as it requires enumerating all the possible groupings and working out the effect if one or more accesses per group are in contention.

7. Conclusion

In this paper we have presented a novel design for a high-performance real-time information filtering application using a low-latency "trivial" Bloom filter. The main contribution of the paper is the derivation of an analytical model for the throughput of the application. This combinatorial model takes into account the access times to the Bloom filter and the external memory, the access probability, and the probability and cost of contention on the Bloom filter. The approach followed and the intermediate expressions are applicable to a large class of resource-sharing problems.

We have implemented our design on the GiDEL PROCStar-III board. The analysis of the system performance clearly demonstrates the potential of the design for delivering high-performance real-time search; we have shown that the system can in principle achieve the I/O-limited throughput of the design. Our current, suboptimal implementation works at 80% of its I/O rate, and this already results in speedups of up to a factor of 20 at 125 MHz compared to a CPU reference implementation on a 3.4 GHz Intel Core i7 processor. Our analysis indicates how the system should be dimensioned to achieve I/O-limited operation for different I/O widths and memory access times.

Our future work will focus on achieving higher I/O bandwidth by using both memory banks on the board and time-multiplexing the memory access. Our aim is to achieve an additional 8× speedup.

Acknowledgments

The authors acknowledge the support from HP, who hosted the FPGA board and provided funding for a summer internship. In particular, we'd like to thank Mitch Wright for technical support and Partha Ranganathan for managing the project.

We'd like to acknowledge Anton Frolov who implemented the synthetic document model.

Wim Vanderbauwhede wants to thank Dr. Catherine Brys for fruitful discussions on probability theory and counting problems.

References

[1] C. L. Belady, "In the data center, power and cooling costs more than the IT equipment it supports," *Electronics Cooling*, vol. 13, no. 1, 2007.

[2] W. Vanderbauwhede, L. Azzopardi, and M. Moadeli, "FPGA-accelerated information retrieval: High-efficiency document filtering," in *the 19th International Conference on Field Programmable Logic and Applications (FPL '09)*, pp. 417–422, September 2009.

[3] L. Azzopardi, W. Vanderbauwhede, and M. Moadeli, "Developing energy efficient filtering systems," in *the 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '09)*, pp. 664–665, July 2009.

[4] V. Lavrenko and W. Bruce Croft, "Relevance-based language models," in *the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 120–127, September 2001.

[5] Lemur, "The Lemur toolkit for language modeling and information retrieval," 2005, http://www.lemurproject.org/.

[6] V. Kindratenko, R. Wilhelmson, R. Brunner, T. J. Martíez, and W. M. Hwu, "High-performance computing with accelerators," *Computing in Science and Engineering*, vol. 12, no. 4, Article ID 5492949, pp. 12–16, 2010.

[7] GiDEL Ltd, "PROCStar III, Data Book," September 2009.

[8] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[9] Altera Corp, "Stratix III, Device Handbook," July 2010.

[10] G. Andrews and K. Eriksson, *Integer Partitions*, Cambridge University Press, 2004.

[11] C. Chen and K. Koh, *Principles and Techniques in Combinatorics*, World Scientific, 1992.

[12] R. M. Losee, "Term dependence: a basis for Luhn and Zipf models," *Journal of the American Society for Information Science and Technology*, vol. 52, no. 12, pp. 1019–1025, 2001.

[13] M. A. Montemurro, "Beyond the Zipf-Mandelbrot law in quantitative linguistics," *Physica A*, vol. 300, no. 3-4, pp. 567–578, 2001.

[14] W. Press, *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, 2007.