Application-specific Disk I/O Optimisation for a Search Engine

Xiangfei Jia, Andrew Trotman, Richard O'Keefe and Zhiyi Huang
Department of Computer Science
University of Otago
PO Box 56, Dunedin, New Zealand
fei, andrew, ok and hzy@cs.otago.ac.nz

Abstract

Operating systems only provide general-purpose I/O optimisation since they have to service various types of applications. However, application level I/O optimisation can achieve better performance since an application has a better knowledge of how to optimise disk I/O for the application. In this paper we provide a solution for application-specific I/O for optimising a search engine. It shows a 28% improvement when compared to the general-purpose I/O optimisation of Linux. Our result also shows a 11% improvement when the Linux I/O optimisation is bypassed.

1 Introduction

Information Retrieval (IR) is the process of finding relevant information, based on user queries, in a large collection of documents. To increase query throughput, an index of all documents is pre-generated. User queries are satisfied by searching the index. Traditionally hard disks are used to store the index and document collection. However, hard disks suffer from mechanical limitations, including the seek time and rotational latency.

Historically three techniques have been used for disk I/O optimisation. First, *buffer caching* keeps already-referenced disk data in main memory. Further disk I/O requests are reduced if the requests can be satisfied by simply referencing the memory buffer. Since system main memory is limited, not all disk data can be buffered. Various buffer replacement policies have been developed to keep the most-referenced or most-valuable disk data in memory. Second, *prefetching* (also called *readahead*) reads disk data into the memory buffer in advance. When the data is needed by subsequent disk I/O requests, these requests can be served by referencing the memory buffer. Prefetching is normally used for sequential disk access only and can be detrimental to random disk access. Third, *scheduling* re-orders the requests issued by applications, causing the disk head to move in one direction from the centre of the disk to the edge before moving from the edge back to the centre, in order to reduce the total seek distance necessary to service a set of requests. Scheduling minimises overall disk seek times and thus increases disk throughput.

General-purpose operating systems usually provide buffer caching, prefetching and scheduling optimisation algorithms. However, the I/O algorithms are general-purpose, as these operating systems have to serve various kinds of applications. For *special-purpose* applications, it is better for applications to deploy their own I/O optimisation algorithms, rather than using the general ones provided by the underlying operating system.

In the paper, we provide a solution for an application-specific I/O for a search engine, using special-purpose buffer caching, prefetching and scheduling algorithms. We also discuss how to bypass the OS's I/O optimisation (using Linux as an example). Our application-specific I/O optimisation shows a 28% improvement when compared with the general-purpose optimisation algorithms in Linux. Our result also shows a 11% performance increase when the Linux I/O optimisation is bypassed.

2 Overview

2.1 Application Level

Inverted files are a well-known index structure in information retrieval. The index has two parts: a dictionary of unique terms extracted from a document collection and a list of postings (a pair of <document number, term frequency>) for each of the dictionary terms [28]. A key objective in the development of the index is to reduce the size of the inverted files, since large inverted files require more I/O to load the postings. Here we discuss techniques to reduce the size of the list of postings since the size of the postings dominates the total size of the inverted files.

Compression is often used to reduce the size of the postings [26]. By comparing Variable Byte, Elias gamma, Elias

delta, Golomb and Binary Interpolative compressions, Trotman [25] concludes that Variable Byte coding should be used unless disk space is at a premium. Anh & Moffat subsequently [4, 5] construct word-aligned binary codes, which are effective at compression and fast at decompression.

Skipping and impact ordering can also improve runtime performance even though they do not reduce the amount of I/O required. Skipping avoids processing parts of the postings list by skipping over postings which are unlikely to be relevant [17]. For impact ordering, the postings list is sorted using some impact factors (influence on relevance score), and the processing of the list is stopped after reaching a certain threshold [3]. Anh & Moffat [6] develop dynamic impact ordering, which has a better computation performance than searching to completion.

Strohman & Croft [24] provide an in-memory retrieval system, combining impact ordering with inverted list skipping. Their system can evaluate queries 7 times faster than the algorithm presented by Anh & Moffat [6]. However in such systems, only partial postings are stored (due to main memory constrains), so they are unable to search to completion, something essential for medical and legal search.

Markatos [16] shows the existence of temporal locality in queries and suggests that dynamic caching should be used when a large cache memory is available, and static caching is better for small cache memory. Baeza-Yates et al [7] suggest that caching posting lists results in better hit ratios than caching query answers, and that static caching can be more effective than dynamic caching.

Fagni et al [12] propose an adaptive prefetching strategy, which anticipates future requests, to improve the performance of web search engines. Lempel & Moran [15] develop a dynamic caching algorithm (Probabilistic Drive Caching), that includes a prefetching capability.

2.2 Kernel Level

An old and yet still widely used buffer replacement algorithm is the Least Recently Used (LRU) algorithm, due to its simple and effective exploitation of temporal locality: a block that is accessed recently is likely to be accessed again in the near future. There are also a large number of other algorithms such as LRU-K, 2Q, MQ, LRFU, ARC, LIRS [11]. All these algorithms focus only on temporal locality. Song et al [13] propose a scheme called Dual Locality (DULO), which takes consideration of both temporal and spatial locality to improve I/O performance.

Butt et al [9] show that prefetching has a significant impact on the performance of various replacement policies, and prefetching can significantly improve the performance of sequential access applications, *but not random access applications*. We believe access to inverted files is essentially random and so low level prefetching will be ineffective.

Pai et al [19] and Wu et al [27] discuss the default readahead[1] algorithm in the Linux 2.6 kernel. Pai et al impove random workloads with small changes to the readahead algorithm, while Wu et al develop a new algorithm called on-demand readahead, which reduces the complexity of the default algorithm with slight performance increases.

Pratt & Heger [20] conduct an comparison of four I/O schedulers implemented in the Linux 2.6 kernel under various workload scenarios, including Noop, Deadline, Anticipatory and Completely Fair Queueing (CFQ). They conclude that there is no best I/O scheduler and *the choice of an I/O scheduler depends on the workload pattern*, the hardware setup and the file system used. Seelam et al [23] implement a new I/O scheduler called Cooperative Anticipatory Scheduler (CAS) base on the Anticipatory Scheduler (AS). CAS addresses the starvation encountered in AS.

2.3 The Linux I/O Subsystem

An operating system must provide protection against unauthorised access to system resources. Modern processors provide a hardware solution by having at least two different protection levels in the CPU itself. The Linux kernel utilises such a hardware solution, allowing kernel code to execute at the highest level (supervisor mode) and user applications to execute at the lowest level (user mode). Figure 1 shows a global picture of various components that form the block I/O subsystem in the Linux 2.6 kernel.

At the top level of the subsystem, the Virtual File System (VFS) provides a common file model for various different file systems. The generic block layer provides a common interface for various disks. The disk cache provides a data buffering mechanism and the readahead reads data in advance. The kernel uses a variant buffer replacement policy similar to the 2Q replacement policy [14]. The kernel maintains two LRU lists called the active list and the inactive list. The active list is used for holding the actively accessed pages of all processes, while the inactive list contains less frequently accessed pages. Pages on the inactive list are replaced when the system has high memory usage. There are two types of readahead; static and dynamic. For static readahead, the kernel always performs I/O requests in blocks, where a block is normally two physical disk sectors. Dynamic readahead only applies to sequential access, here a current window holds the disk data for the current request and a readahead window holds the data that satisfies expected future requests.

The disk drivers, at the lowest level, perform the actual I/O requests to disk. The order of I/O requests issued by applications can differ from the ones performed by the drivers as disk requests are scheduled by the I/O scheduler. In case of a read, the drivers read data from disk to the buffer cache

[1] In the Linux kernel, prefetching is often referred as readahead.

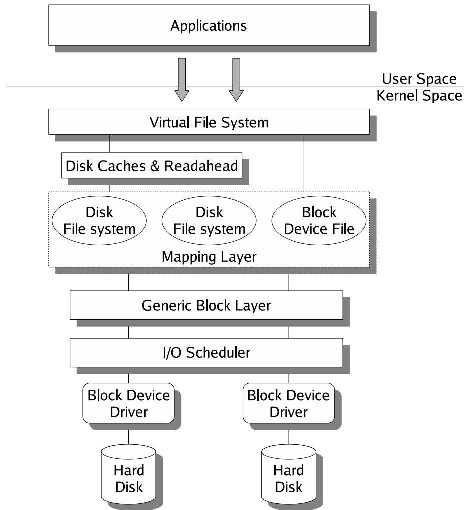

Figure 1. The Linux I/O Subsystem (from [8])

and then the kernel copies the requested buffer to the application. The extra copy operation is required as the kernel and applications operate in different CPU modes.

The design of the Linux I/O subsystem has drawbacks. First, an extra copy operation is required for read and write requests. However, the O_DIRECT in the Linux 2.6 kernel bypasses the I/O subsystem and thus allows direct disk read and write. Second, buffer replacement policies manage buffer caches usually in terms of sectors. Individual applications often use their own data structures. It is more efficient to buffer the application's data structures. However, such data structures are not visible to the kernel. The same is true for readahead and I/O scheduling.

In summary, what an operating system can provide is *general-purpose* I/O algorithms since it has to serve various types of applications. Applications define their own *special-purpose* data structures, while the OS kernel operates in sectors. This extra mapping of application's data structures into sectors is time consuming.

3 Application-specific I/O Optimisation

Caching can be achieved using either query results or posting lists. Caching query results not only optimises disk I/O, but also avoids reprocessing of queries evaluation. However, queries tend to have low frequency of repetition and thus result in a high cache miss rate [7].

Caching posting lists, on the other hand, can reach a high hit ratio [7]. One way of implementing a cache is to deploy a buffer replacement policy, like LRU or Least Frequently Used (LFU), which defines how efficiently a finite amount of cache memory is used for large amounts of data on disk. This is so called dynamic, due to frequent update of the cache memory. One of the challenges for dynamic caching is that caching variable sized posting lists is difficult in terms of cache memory management.

Instead of frequent update of cache memory, another way to cache posting lists is to define what posting lists are the most important and then let them stay in cache memory without eviction. This is called static caching because the most important posting lists are already cached, there is no need to further update the cache memory. The open question is how to define the importance of postings lists. One solution is to choose query terms which have highest query-term frequencies $f_q(t)$. Another solution can be based on document-term frequencies $f_d(t)$. Terms with high $f_d(t)$ have long posting lists, thus consume more cache memory. However, caching long posting lists reduces disk I/O further as it takes more time to read long posting lists from disk. Baeze-Yates et al [7] argue that the importance should be defined as $\frac{f_q(t)}{f_d(t)}$ (Posting lists with high query-term frequencies and short length in size are preferred). Static caching simplifies cache management by eliminating the buffer replacement policy problem. However, there is a price to pay. Because the importance of posting lists is based on the analysis of the early query log, the importance needs to be redefined if incoming queries are outside the coverage of the early query log. The operation of dealing such problem is to re-fill the cache, resulting in very low hit ratio and performance decrease.

Both dynamic and static caching have pros and cons. Dynamic caching is good at keeping up with frequent change in queries, while static caching simplifies cache management and results in high hit ratio under normal circumstances. An obvious question to address is the possibility of combining both dynamic and static caching. The potential problems are: (1) which posting lists to cache dynamically and which to cache statically, (2) how to distribute cache memory among them, (3) is there a performance gain for such a combination.

Prefetching and scheduling are straightforward. The access pattern of a search engine can be predicted by the query terms. Posting lists for the next term can be prefetched while the current term is being processed.

If we consider that posting lists are sorted in alphabetic order of the dictionary terms, we can define a new scheduler which sorts disk I/O requests in the order of the dictionary terms. The sorting can be either local or global, where local means sorting terms in a single query and global means sorting terms in several queries executed concurrently. Local sorting has quick individual response time while global sorting has better overall performance. For the scheduler, once the sorting is performed in ascending order, then next time the sorting should be done in descending order, and so on. This allows the disk head to move from the centre of the disk to the edge and then back to the centre.

4 Test Data

We used the .GOV collection [10]. For evaluation of the search engine, we concentrated on efficiency, rather than effectiveness. However, we deployed Okapi BM25 [21] for ranking:

$$RSV_d = \sum_{t \in q} log\left(\frac{N}{df_t}\right) \cdot \frac{(k_1+1)\,tf_{td}}{k_1\left((1-b)+b\times\left(\frac{L_d}{L_{avg}}\right)\right)+tf_{td}}$$

Here, N is the total number of documents, and df_t and tf_{td} are the number of documents containing the term t and the frequency of the term in document d, and L_d and L_{avg} are the length of document d and the average length of all documents. The empirical parameters k_1 and b have been shown to be effective when set to 1.2 and 0.75 respectively [21]. There are other models of Okapi BM25. However, this model guarantees positive RSV_d values as $log\frac{N}{df_t}$ is always positive. Positive numbers are required when Variable Byte coding is used to compress/decompress quantised RSV_d values.

Table 1 shows a summary of the document collection. The documents in the collection were parsed with common stopwords included (not removed) and no stemming. The postings were stored in alphabetic order of the words in the dictionary, and were compressed using Variable Byte coding. The postings file has two versions: (1) the standard postings file which stores the pairs of <document number, term frequency>, (2) the pre-BM25 version stores the pairs of <document number, pre-calculated BM25 result>, where the pre-calculated BM25 ranking result (+0.5) were rounded to the nearest integer.

Our pre-BM25 quantisation is a variant of Moffat & Anh's quantisation method [18, 3], in which term frequencies are approximately mapped to the range $(0..2^b)$ and stored in b bits. Moffat & Anh claim that quantisation only slightly affects the retrieval precision when the value of b is greater than 5. In our current research, we are not concerned with retrieval precision instead we concentrate on performance and rely on the result of Moffat and Anh when it comes to the precision loss through quantisation.

We created two raw disk images with a simple structure for the two postings files. Each image had a copy of the corresponding postings file at the beginning, followed by the dictionary file which started at a new sector location. The raw disk images were copied to the disk using the dd command. Two macros were defined to convert a file location to a sector location and to find the offset in the sector. By using raw disk images, we avoided the overhead of the file system and the variance of testing results due to fragmentation.

The TREC 2007 Million Query Track [2] was used for throughput evaluation. The track has a total of one million queries with a total of 41095 terms. The average query length is about 4 terms.

Collection	18GB	Documents	1247753
Unique Words	8849995	Avg Doc Length	975 words
Postings File	819MB	With Pre-BM25	818MB
Dictionary File	394MB		
Raw Image	1.2GB	With Pre-BM25	1.2GB

Table 1. Summary of the .GOV collection

Specification	ST380215A	Capacity	80GB
Speed	7200RPM	Transfer Rate	100MB/sec
Cache Buffer	2MB	Avg Latency	4.16ms

Table 2. Specification for the test disk [22]

5 Experiments

We conducted tests on a PC with an Intel single core Pentium 4 CPU running at 2.4GHz, with 512KB of L2 cache and a speed of 533MHz for the Front Side Bus. The system has 768MB of DDR266 main memory. We used separate disks for installation of kernels and testing. The testing disk is the IDE primary master, while the kernel disk is configured as the primary slave. The kernel disk is not needed for the performance test. Table 2 shows the specification for the testing disk. The chosen Linux distribution was Debian Etch, with the default Linux 2.6.8 kernel.

Throughout our experiments, only the pre-BM25 raw disk image was used because preliminary experiments show that ranking is always faster than the standard BM25 image. The RDTSC instruction was used for timing purpose and the number of cycles returned by RDTSC was converted to seconds. In order to minimise bias, we re-booted the testing machine for each run and the swap partition used for Linux was disabled. Each test was run five times and the average was taken as the final result.

We carried out two categories of testing. First we used O_DIRECT to demonstrate the effectiveness of caching, prefetching and scheduling for the search engine. The reason we used O_DIRECT but not the default read() is because we want to minimise the impact of the Linux I/O subsystem and concentrate on application level optimisation. Then we compared the performance of the application level I/O optimisation with the Linux I/O subsystem by comparing O_DIRECT to normal read().

5.1 Application Level

For caching, we deployed static caching, dynamic caching and the combination of both. Stopwords tend to have long posting list and caching them can reduce disk I/O. We used the list of stopwords from [1] and there were 571 stopwords in the list. The posting list of a stopword was

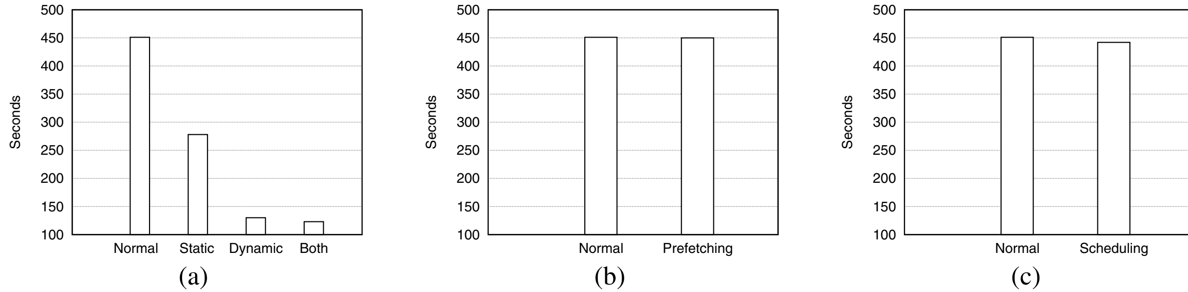

Figure 2. Performance of the caching, prefetching and scheduling algorithms for the search engine

statically cached when it first appeared in a query. We developed a simple LRU cache policy for dynamic caching and the cache policy was defined to cache postings lists of terms which were repeated more than once. We did not want to simply cache all terms dynamically, so as to avoiding frequent cache eviction. The size of the dynamic cache memory was defined as the total available physically memory after that which had been allocated for the static caching. We found that the maximum number of posting lists, the dynamic caching could hold, was 2400 for the given query set.

In this paper, we are not trying to find an optimal solution for application level caching. Instead, we want to use simple but effective caching algorithm to demonstrate that application level caching is better than the generic kernel level I/O optimisation.

Figure 2(a) shows the performance of the static and dynamic caching individually and in combination against normal O_DIRECT in which there is no optimisation at all. Static and dynamic caching performed about 38% and 71% better than normal O_DIRECT (which took about 451 seconds), and the overall performance of static and dynamic caching had about 73% increase. As shown in the figure, the dynamic caching performed a lot better than the static caching. This did not mean that the dynamic caching algorithm was superior to the static since they were allocated with different size of cache memory. Note that the maximum number of entries for dynamic cache was 2400 for both individual and overall testing.

For prefetching, we defined two buffers for storing posting lists: one for serving the current request, while the other serves the next request. Two threads were introduced to interchangeably switch between the two buffers. While one thread waits for I/O, the other can process the posting list already loaded in the other buffer. As shown in Figure 2(b), the performance of prefetching was almost the same as normal O_DIRECT. This is not what we expect and is probably due to the overhead of multi-threading (in our implementation, new threads were allocated for each single query).

For scheduling, we re-ordered the query terms in alphabetic order of dictionary terms. The first query was sorted in ascending order while the second was sorted in descending order, and so on. Since we processed one query at a time, we only sorted query terms locally. The performance of the scheduling was about 9 seconds faster than normal O_DIRECT as shown in Figure 2(c).

5.2 Search Engine Comparison

We carried out four tests: (1) O_DIRECT Normal, (2) O_DIRECT Optimised, (3) read() Normal and (4) read() Optimised. The optimised tests had the application level I/O subsystem enabled, while the normal tests had no application level optimisation. The O_DIRECT tests was essentially the same as the read() tests, and the only difference was the way the disk was accessed.

As Shown in Figure 3, the processing time for ranking and decompression were similar for all tests, where decompression is the process of decompressing posting lists, and the ranking is the process of accumulating ranking results.

The I/O read time is the total time taken reading the postings from the disk. The O_DIRECT Normal test took the longest time for reading I/O (about 451 seconds). The O_DIRECT Optimised test performed about 73% better than O_DIRECT Normal. The read() Normal and Optimised tests took about 167 and 136 seconds, respectively. Interestingly, read() Optimised also benefited from the application level optimisation. O_DIRECT Optimised beat read() Normal by 47 seconds (28% improvement), showing that application-specific optimisation was superior to that is offered by the Linux kernel. O_DIRECT Optimised also beat read() Optimised by 16 seconds (11% improvement), demonstrating the overhead of the Linux I/O subsystem when the application provided its own I/O optimisation.

The total time is comprised of the ranking time, decoding time and the I/O read time. The ranking time represents the part of the search engine which is CPU-intensive, whereas the I/O read time represents the part which is I/O-intensive. By using either application level or kernel level I/O optimisations, we have shifted the search engine to

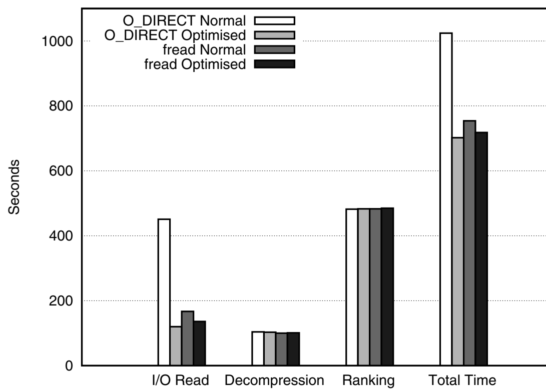

Figure 3. Overall Comparison

CPU-bound. The overall total improvement for O_DIRECT Optimised was about 31%, 7% and 2% when compared with O_DIRECT Normal, read() Normal and Optimised.

6 Conclusion & Future Work

We have shown the efficiency of an application-specific I/O optimisation over the Linux kernel I/O optimisation for search engines. Among the caching, prefetching and scheduling algorithms, caching is the most effective method for optimising disk I/O. However, caching is the most complicated I/O algorithm to implement. For the disk access of the search engine, the overall improvement of the application-specific I/O subsystem is about 73% and 28% when compared with no disk I/O optimisation and the Linux I/O subsystem respectively. Our results also show an 11% overhead of the Linux I/O subsystem when the application provides its own I/O optimisation.

In the future, we plan to further improve the caching, prefetching and scheduling algorithms and to address the CPU boundness of the application.

References

[1] http://en.wikipedia.org/wiki/SMART_Information_Retrieval_System, July 2008.

[2] J. Allan, B. Carterette, J. Aslam, V. Pavlu, B. Dachev, and E. Kanoulas. Million query track 2007 overview. In *TREC*, 2008.

[3] V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *SIGIR '01*, 2001.

[4] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1):151–166, 2005.

[5] V. N. Anh and A. Moffat. Improved word-aligned binary compression for text indexing. *IEEE T Knowl Data En*, 18(6):857–861, 2006.

[6] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *SIGIR '06*, pages 372–379, 2006.

[7] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *SIGIR '07*, pages 183–190, 2007.

[8] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel, 3rd Edition*. O'Reilly, November 2005.

[9] A. R. Butt, C. Gniady, and Y. C. Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. *IEEE Trans. Comput.*, 56(7):889–908, 2007.

[10] N. Craswell and D. Hawking. Overview of the TREC-2002 web track. In *TREC*, 2002.

[11] X. Ding, S. Jiang, and F. Chen. A buffer cache management scheme exploiting both temporal and spatial localities. *Trans. Storage*, 3(2):5, 2007.

[12] T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans. Inf. Syst.*, 24(1):51–78, 2006.

[13] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. DULO: An effective buffer cache management scheme to exploit both temporal and spatial locality. 2005.

[14] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB '94*, pages 439–450, 1994.

[15] R. Lempel and S. Moran. Predictive caching and prefetching of query results in search engines. In *WWW '03*, 2003.

[16] E. P. Markatos. On caching search engine query results. *Computer Communications*, 24(2):137–143, 2001.

[17] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4):349–379, 1996.

[18] A. Moffat, J. Zobel, and R. Sacks-Davis. Memory efficient ranking. *Inf. Process. Manage.*, 30(6):733–744, 1994.

[19] R. Pai, B. Pulavarty, and M. Cao. Linux 2.6 performance improvement through readahead optimization. In *Proceedings of the Linux Symposium*, 2004.

[20] S. L. Pratt and D. A. Heger. Workload dependent performance evaluation of the linux 2.6 I/O schedulers. In *Proceedings of the Linux Symposium*, 2004.

[21] S. E. Robertson, S. Walker, M. Hancock-Beaulieu, A. Gull, and M. Lau. Okapi at TREC-4. In *TREC-4*, 1992.

[22] Seagate. Barracuda 7200.10 PATA, August 2007.

[23] S. Seelam, R. Romero, and P. Teller. Enhancements to linux I/O scheduling. In *Proceedings of the Linux Symposium*, 2005.

[24] T. Strohman and W. B. Croft. Efficient document retrieval in main memory. In *SIGIR '07*, pages 175–182, 2007.

[25] A. Trotman. Compressing inverted files. *Inf. Retr.*, 6(1):5–19, 2003.

[26] H. E. Williams and J. Zobel. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201, 1999.

[27] F. Wu, H. Xi, and J. Li. Linux readahead: less tricks fo more. In *Proceedings of the Linux Symposium*, 2007.

[28] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.*, 23(4):453–490, 1998.