JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 19, 681-695 (2003)

Short Paper

An Inverted File Cache for Fast Information Retrieval*

WANN-YUN SHIEH, JEAN JYH-JIUN SHANN AND CHUNG-PING CHUNG
Department of Computer Science and Information Engineering
National Chiao Tung University
Hsinchu, 300 Taiwan
E-mail: {wyshieh; jjshann; cpchung}@csie.nctu.edu.tw

The inverted file is the most popular indexing mechanism used for document search in an information retrieval system (IRS). However, the disk I/O for accessing the inverted file becomes a bottleneck in an IRS. To avoid using the disk I/O, we propose a caching mechanism for accessing the inverted file, called the inverted file cache (IF cache). In this cache, a proposed hashing scheme using a linked list structure to handle collisions in the hash table speeds up entry indexing. Furthermore, the replacement and storage mechanisms of this cache are designed specifically for the inverted file structure. We experimentally verify our design, based on documents collected from the TREC (Text REtrieval Conference) and search requests generated by the *Zipf*-like distribution. Simulation results show that the IF cache can improve the performance of a test IRS by about 60% in terms of the average searching response time.

Keywords: information retrieval system, inverted file, cache, hashing, memory management

1. INTRODUCTION

To help users find useful information from large-scale information cyberspace, an information retrieval system (IRS) requires a specialized indexing mechanism [1]. Rillof and Hollaar [2] pointed out that most IRSs use inverted files as indexing mechanisms. Zobel et al. [3] further showed that in terms of the querying time, used space, and functionality, inverted files perform better than other mechanisms. However, inverted files themselves require much storage space. This causes a disk I/O problem in an IRS. We believe that this I/O requirement can be alleviated to some extent, and this paper presents our study on this topic.

1.1 Motivation

In an inverted file-based IRS, a user sends a query containing some terms to the system. The system searches these query terms in the inverted file to see which document

Received November 13, 2001; revised July 4, 2002; accepted October 28, 2002.
Communicated by Arbee L. P. Chen.
* This work was supported by National Science Council of R.O.C., NSC-89-2213-E-009-062.

satisfy the request and returns these documents' identifiers to the user. For each distinct term *t*, there is a corresponding list (called the posting list) of the form

$$< t; f_t; D_1, D_2, D_3, ..., D_{f_t} >$$

in the inverted file, where the identifier D_i (also called posting) indicates a document that contains *t*, and the frequency f_t indicates the total number of documents in which *t* appears.

When the number of documents in an IRS increases, the number of terms and the lengths of the posting lists in the inverted file increase accordingly. The increase in the term number slows down the searching speed, while the increase in the posting list length increases the retrieval latency. Moreover, such a large inverted file cannot fit into the memory and usually has to be stored in disks [4]. Thus, the disk I/O for accessing the inverted file becomes a performance bottleneck in a modern IRS. The most common practice for reducing the disk I/O is to apply a caching mechanism before disk accesses are performed.

Most conventional caching mechanisms index the cached entries by means of simple hashing scheme and allocate a fixed-length block to store each cached context. The hashing scheme involves using a hashing function to calculate entry addresses and applying collision resolution to avoid conflict between two or more records at the same address. The collision resolution approach proposed in many textbooks includes the open addressing scheme and the chain hashing scheme [5-8]. The open addressing scheme searches the record in sequence from the conflicting address, while the chain hashing scheme searches the record through the lined-list structure. Both of these searching processes, also called probing processes, have been described and discussed in [6, 7].

However, the schemes described above are not exactly suitable for caching the inverted file for the following reasons:

First, the open addressing scheme may consume an intolerable amount of probing time if the number of cached records increases [5, 6, 8]. Such a situation may happen because in a modern IRS like a search engine, we have to cache as many records as possible to capture the locality behind user queries. In fact, the open addressing scheme performs well only when the hashing records are sparse.

Second, although the chain hashing scheme links the collision records with pointers to shorten the probing length, the uncompleted linked-list structure may cause some cached records to be lost during the insertion of a new record [6]. This problem not only affects the efficiency of the cache, but also causes space to be wasted. Many studies have examined various ways to improve the chain hashing scheme (e.g., two-pass loading [6] and chaining with an overflow area [6]). Those approaches, however, required dynamic memory allocation, which increases the complexity of collision handling and the cache response time [8].

1.2 Research Goal

In this paper, we propose a caching mechanism for the inverted file, called the inverted file cache (IF cache), to reduce the number of disk I/O operations in an IRS. In this cache, we use the linked-list structure to shorten the probing length for the hashing

scheme. To remedy the drawbacks of the linked-list structure in the chain hashing scheme, we add three indicators, *head*, *collision_count*, and *link*, to each cached entry. *Head* indicates that the entry is at the beginning of a list, *collision_count* indicates the length of the list starting from the entry, and *link* indicates the pointer to the next linked entry. When a new record is inserted into the cache, with the help of these indicators, the irrelevant linked lists will neither be broken nor extended to increase the probing length. The probing process needs to be performed only on the entries belonging to the correct linked list. This greatly reduces the probing time needed for searching a record. In addition, along with variable-size posting lists, we suggest two space management schemes, the *compact scheme* and the *chunk-based scheme*, for efficiently using the cache memory. Compared with the traditional IRS without a customized inverted file cache, simulation results show that the performance of a test IRS with the IF cache offers improvement of about 60% in terms of the average response time.

This paper is organized as follows. In section 2, we present the detailed design of the IF cache with the link-based hashing scheme and the variable-length space management scheme. In section 3, we present simulation results and evaluation. Finally, section 4 is our conclusion.

2. INVERTED FILE CACHE (IF CACHE)

The IF cache consists of four components as shown in Fig. 1: the cache manager, stop word filter, index block, and posting block.

1. The cache manager is a finite-state controller that handles the search status of the cache. If a query term is found in the cache, the manager returns the cached posting list to the user. Otherwise, it issues a disk access command to retrieve results in the disk and then writes the results to the cache.
2. The stop word filter filters out unnecessary terms, such as “the,” “a,” “of” etc. to prevent those words from being issued in the search. This is because these terms typically carry no meaning.
3. The index block contains a hash function and a hash table for cached terms. The hash function maps each query term to a unique entry in the hash table. Each entry in the hash table includes a term field, the three indicators described in section 1.2, and a posting list pointer (PLP). The term field stores the term that has been queried, and the three indicators maintain the link structure among the entries. The PLP simply stores the pointer to the location of the corresponding posting list in the posting block.
4. The posting block stores the posting lists of various lengths.

When a query term is issued to the IF cache, the cache manager searches it in the hash table using the hashing scheme. If this term hits an entry in the table, its posting list in the posting block is returned. Otherwise, its posting list is retrieved from the disk and replaces some elements in the cache. We discuss each component in the following.

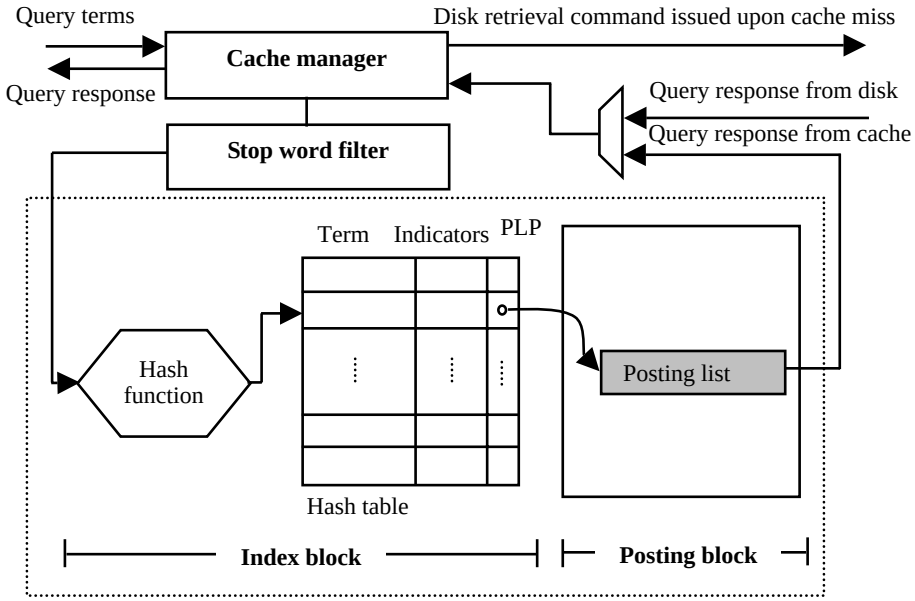

Fig. 1. Essential components of the IF cache.

2.1 Hashing Scheme

To avoid intolerable probing time and complex cache management, we select a linked-list structure as the basis of the hashing scheme. In this scheme, a hash table is implemented as a linear array of entries as in the open addressing scheme, but those entries whose terms have collided at the same hashing address are linked together as in the chained hashing scheme, as shown in Fig. 2. When a collision occurs, we select an empty entry or a nonempty entry in the hash table and insert into it or replace it with, respectively, a new term, and then link this new entry to the related list. The probing process needs to be performed only on those entries belonging to the same linked list, not all entries in the hash table.

The linked list of a set of collision terms in the hash table is maintained by means of the three indicators in each entry: *head*, *collision_count*, and *link*. The *head* of an entry is set to 1 if this entry is identified as the direct consequence of hashing. The *collision_count* of an entry records the number of other terms colliding at this entry with the *head*. And the *link* of an entry is the pointer to the next linked entry. Fig. 2 shows an example in which entries 2, 6, 12, and 16 form a linked list. In this case, terms t_2, t_6, t_{12}, and t_{16} share the same hashing address and collide at the same entry. Hence, the *head* and *collision_count* of entry 2 are set to 1 and 3, respectively.

If we apply this hashing scheme to the IF cache, the search process stops at one of three states: *hit*, *false hit*, or *miss*. The three search states are explained below.

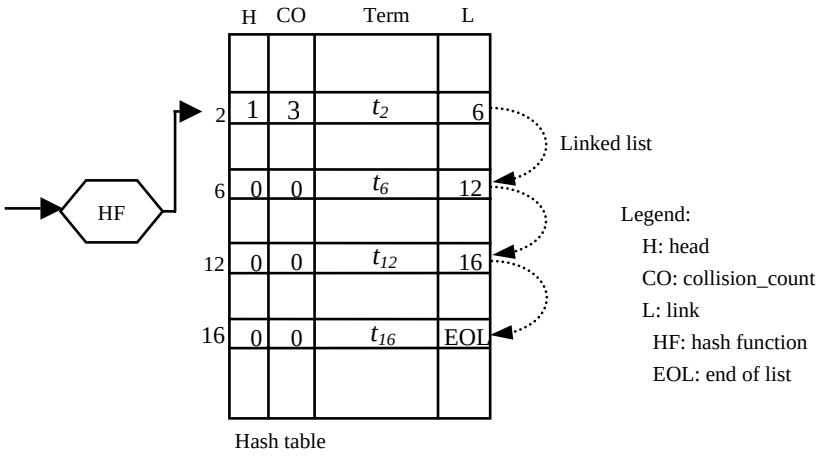

Fig. 2. Structure of the proposed link-based hash table.

Case 1: The hit state

The hit state indicates that the hashing address of a query term maps to an entry whose *head* is 1, and this term can be found in the associated list by probing the linked entries.

Case 2: The false hit state

The false hit state occurs when the hashing address of a query term maps to an entry whose *head* is 1, but this term cannot be found after all entries in the associated list have been exhaustively probed. Fig. 3(a) shows a false hit example, in which the term "engineer" cannot be found after probing the linked entries. The steps that must be performed after a false hit occurs are:

1. issue a retrieval command to the disk and
2. find a free entry (perhaps through replacement) in both the hash table and the posting block for storing the term and the returned posting list.

This new entry is linked just after the starting entry of the list for the next access, and the *collision_count* of the starting entry is incremented as shown in Fig. 3(b). In this figure, we store the term "engineer" into entry 4 and link this entry after entry 2. The *link* indicators of entries 2 and 4 are updated, and the *collision_count* of entry 2 is incremented.

Case 3: The miss state

The miss state may be either a compulsory miss or a conflict miss. A compulsory miss occurs when the hashing address of a query term maps to an empty entry in the hash table. This term is inserted into the entry, and a disk retrieval command is then issued with respect to the term's posting list. The returned posting list is cached, and the indicators *head*, and *collision_count* of this entry are set to 1, and 0, respectively.

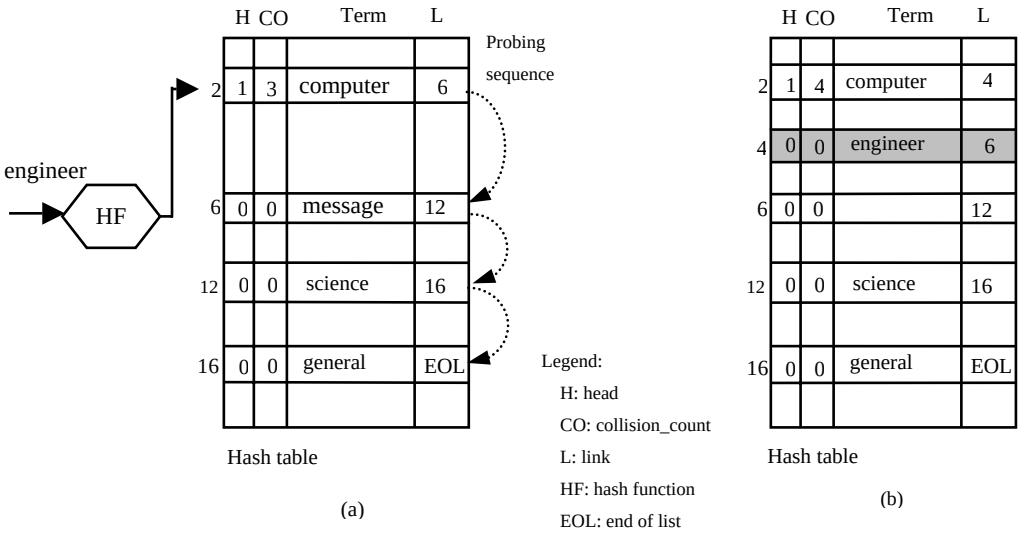

Fig. 3. A false hit example. (a) "engineer" does not exist in the list, (b) the hash table after "engineer" is inserted.

A conflict miss, on the other hand, occurs when the hashing address of a query term maps to a nonempty entry (say E_0) whose *head* = 0. That is, E_0 is linked by the other entry, and there is no collision at E_0. One can immediately see that the term cannot be found in the list. This saves much probing time compared with other hashing schemes. In this case, a disk retrieval command is required. We find a free entry (say E_1) into which to insert the term (perhaps through replacement) and a free space in the posting block in which to store the returned posting list. Both the *head* and *collision_count* of E_0 are set to 1, and E_1 is linked just after E_0. With this design, two or more hashing addresses whose *head*s are both 1 may share the same linked list. Although the length of a linked list may increase accordingly, the probing length for each hashing address will not increase. For example, assume that entries E_i and E_j share the same linked list, and that both of their *head*s are 1. If the *collision_count* of E_i is C_i and that of E_j is C_j, then the longest probing length of the term colliding at E_i is exactly C_i; that is, we can "bypass" C_j entries after probing E_j (and vice versa).

Fig. 4 shows a conflict miss example. In Fig. 4(a), no entries need to be further probed. Fig. 4(b) shows a hash table into which the term "technology" has been inserted, and in which entries 2, 6, 20, and 16 form a linked list. Note that entries 2 and 6 share the same linked list, and that both of their *head*s are 1. If we search for terms colliding at entry 2 in the linked list, we can "bypass" entry 20 after probing entry 6 and continue probing from entry 16. On the other hand, if we search for terms colliding at entry 6 in the linked list, only entry 20 needs to be further probed.

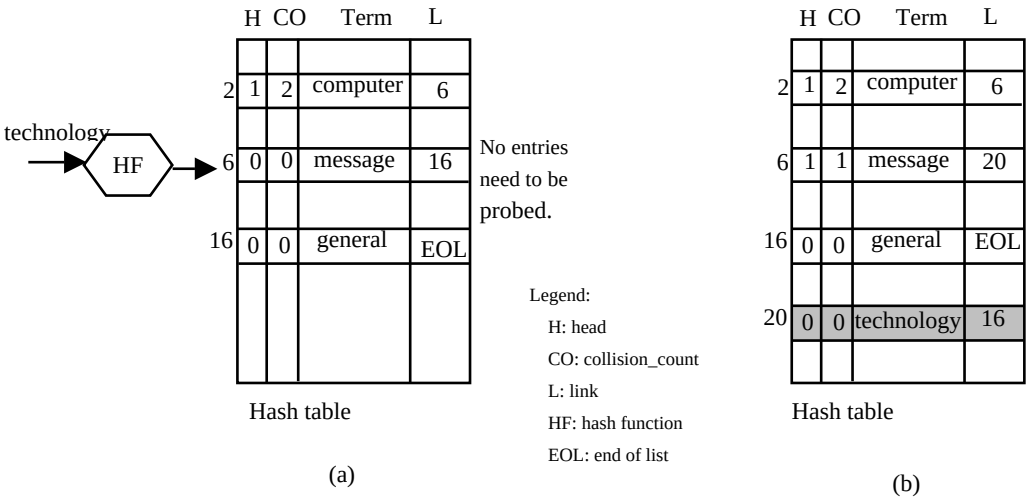

Fig. 4. A conflict miss example. (a) the hashing address of "technology" maps to entry 6, whose *head* is 0, (b) the hash table after "technology" is inserted.

2.2 Replacement Mechanism

A false hit, compulsory miss, or conflict miss will trigger replacement if the hash table is full or the posting block does not contain enough free space to store new posting lists. When the hash table is full, we use LRU, LFU, or a hybrid policy to select a victim entry. This entry and its posting list in the posting block will be freed up for replacement. However, the free space in the posting block may not be large enough to accommodate the new term's posting list. If this is the case, we will free up more than one entry and one posting list to accumulate enough free space.

Note that if we select an entry whose *head* is 0 to be the victim, then this entry can be disconnected from its associated linked-list and attached to the new linked-list in the hash table. Disconnecting an entry from its associated linked list in the hash table does not require double-link structure. This is because we can re-hash the term of this entry to get the linking information from its hashing address.

However, if we select an entry whose *head* is 1 to be the victim, then the entry and its associated linked list need to be completely attached to the new linked list after the replacement operation. This is because disconnecting this entry from the list will cause its following linked entries to be lost. (this is the major drawback of the chain hashing scheme [6].) With this design, again, two entries whose heads are both 1 may share the same linked list. However, the probing length for each hashing address will not be affected because the probing length is actually determined by the *collision_count*.

2.3 Posting Block

The posting block is a large memory pool that stores cached posting lists. A good space management scheme for the posting block greatly influences the response time and

utilization of the cache. In the following, we discuss two possible space-management schemes that take variation in the posting list length into account.

One basic variable-length scheme is the *compact* scheme. In this scheme, all the elements in a posting list must be stored in a contiguous free space for sequential retrieval. When a posting list is inserted into the posting block, we have to find a sufficiently large space for it. The advantage of this scheme is that the posting list can be read sequentially without incurring any transferring overhead. However, this approach has some drawbacks. Consider that a posting list of N postings is to be stored in the posting block. Assume that the size of each free contiguous block is B_i, for $i = 1, 2, \ldots$, and that the total size of the free space in the posting block is T. If $T > N$ and $\text{Max}(B_1, B_2,\ldots) < N$, then we have to compact some blocks so that they will fit into a free space of size N. This compaction operation is quite time-consuming. In addition, although sequential retrieval can benefit from contiguous storage in this scheme, a trend in Web applications is to access the posting list in a random manner as in, for example, the fast searching process in the ranked query model [9]. The memory arrangement for sequential retrieval may not be completely suitable for random access operations.

Another variable-length space management approach is the chunk-based. In this scheme, the posting block is implemented as a set of fixed-size chunks. Each chunk stores n postings and a pointer as shown in Fig. 5. When a posting list containing N postings is inserted into the posting block, $\lceil N/n \rceil$ chunks are allocated to store this list instead of a contiguous free space. Each allocated chunk is linked to the list by the pointer of the previous chunk. This scheme obviously has the advantages of simplicity and flexibility, compared with the compact scheme. Moreover, the chunk-based scheme can support random access operations. For example, if the system wants to access the $(2n + 3)$th posting, it can bypass the first two chunks and retrieve the third posting in the third chunk directly. The drawbacks of this scheme are that it needs non-data overheads to link the chunks, and that the size of a chunk cannot be determined easily. In section 3, we will present the use of simulation to find a suitable size for the IF Cache.

Fig. 5. Data structure of a chunk.

2.4 Stop Word Filter

The purpose of the stop word filter is to prevent insignificant words from being issued during the search. In the proposed architecture, we implement a stop-word board in the filter to record the stop words. This stop-word board is like a lookup table that uses the hash function to index the entries. The contents of the stop-word board are based on the 135 stop word mentioned in [4]. When a term is searched in the cache, it is first matched with the entries in the board. If the term exists in the board, the cache access is rejected by the filter.

3. PERFORMANCE EVALUATION

In this section, we investigate two important issues, functionality and efficiency, related to the IF cache through simulation benchmarks. With respect to functionality, we evaluate the advantage of the IF cache in reducing the disk I/O. With respect to efficiency, we estimate the proper cache size.

3.1 Simulation Model

To compare the performance of various algorithms, simulations were performed on a benchmark collection obtained from the fifth disk of TREC (Text REtrieval Conference) [10]. This collection is a very large document collection distributed worldwide for use in comparative information retrieval experiments [4]. The collection contains two suites of full texts of various newspaper and newswire articles plus government proceedings, and each suite has about 130,000 documents. The data in the first suite includes material from the Foreign Broadcast Information Service (FBIS), and the data in the second suite includes material from the Los Angeles Times (LATimes). We built the inverted files for these two suites and then applied the proposed cache architecture to them, respectively. Short descriptions of these inverted files are given in Table 1.

Table 1. Statistics of two inverted files used in our experiments.

Description	FBIS inverted file	LATimes inverted file
Document count	130,471	131,896
Distinct terms	209,782	167,805
Size (Mbytes)	263	297
Average posting count in a posting list	136	192

To simulate user query behavior, we implemented a query-term generator which picks query terms from the inverted files. The occurrence of these terms follows the *Zipf*-like distribution, a distribution widely used in recent IRS studies [15]. In this distribution, the relative probability of a request for the *i*th most popular term is proportional to $1/i^{\alpha}$, where $\alpha \leq 1$. We let $\alpha = 0.8$ in the experiments because α appears to center around 0.8 for most traces in homogeneous environments [15]. In each experiment, we generated 100, 000 user queries, and the lengths of the queries were distributed evenly from one to five terms. Furthermore, we adopted the Boolean query model, in which the AND, OR, and NOT Boolean operators are uniformly inserted into the generated queries. Therefore, a user query looks like "*information* <AND> *retrieval* <AND> *cache*," where information, retrieval, and cache are query terms.

We define the hit rate of the IF cache as the fraction of the total cache accesses that are in the *hit* state. If we assume that the IF cache can store an unlimited number of query terms, we can get a hit rate of about 65% for the terms in the 100,000 queries. We call this rate the ideal hit rate and use it to denote the maximal locality that can be gained from enhancements.

In Fig. 6, we show our cache simulator and how we measure the average response time. The response time for a cache hit is measured from the time a user query is sent into the cache to the time the related posting lists are returned from the posting block. For a false-hit or miss state, the response time is composed mainly of the probing time in the hash table and the retrieval time in the disk.

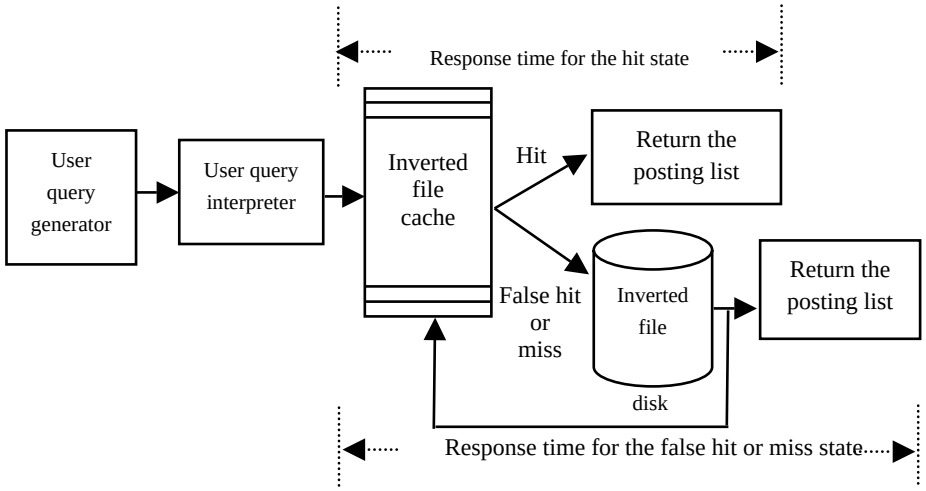

Fig. 6. Platform of the cache simulator.

3.2 Simulation Results

The experiments conducted in our evaluation focused on two aspects of performance. First, we compared the hashing performance of the proposed link-based scheme with that of both the open addressing scheme and the chained hashing scheme. Second, we compared the storage efficiency of the chunk-based scheme with that of the compact scheme. Because the IF cache is implemented in order to speed up user query processing, we used the average response time as the major performance metric in the experiments. All the experiments were performed on a PC with a 700MHz CPU, 128 MB SDRAM with 10 ns average access time, and a 2.1 GB hard disk with 9 ms average seek latency.

3.2.1 Effect of the inverted file cache

Fig. 7 shows that using the IF cache in an IRS reduces the average response time significantly. Assume that the hash table contains 4K entries with the open addressing scheme applied for the purpose of collision resolution, and that the size of the posting block is 8MB with the compact scheme applied for the purpose of space management. Each posting stored in the posting block is a 32-bit long integer (in uncompressed form). The simulation results show that the average response time decreases by about 33% for FBIS and 29% for LATimes. We suppose that these decreases in the average response time result from reduction in the disk I/O. Such reduction can increase the service capability in an IRS.

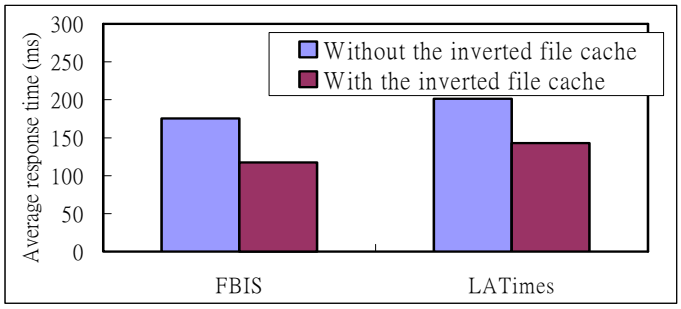

Fig. 7. Effect of the inverted file cache. Assume that the hash table contains 4K entries with the open addressing scheme used for collision resolution, and that the size of the posting block is 8MB with the compact scheme applied for the purpose of space management.

3.2.2 Effect of the hashing scheme

The performance of a hashing scheme heavily depends on the load factor of a given hash table. The load factor as defined in [7] is the ratio of the number of keys initially stored in the hash table to the number of entries in the table. The load factor provides a measure of the amount of space in the hash table that is used, and we use it to see how collision resolution in different hashing schemes affects the average response time. Fig. 8(a) shows the simulation results for the FBIS inverted file, and Fig. 8(b) shows the results for the LATimes inverted file. As shown in both figures, when the load factor increased from 0.5 to 0.9, the open addressing scheme produces the longest average response time as expected. On the other hand, the chain hashing scheme and the proposed

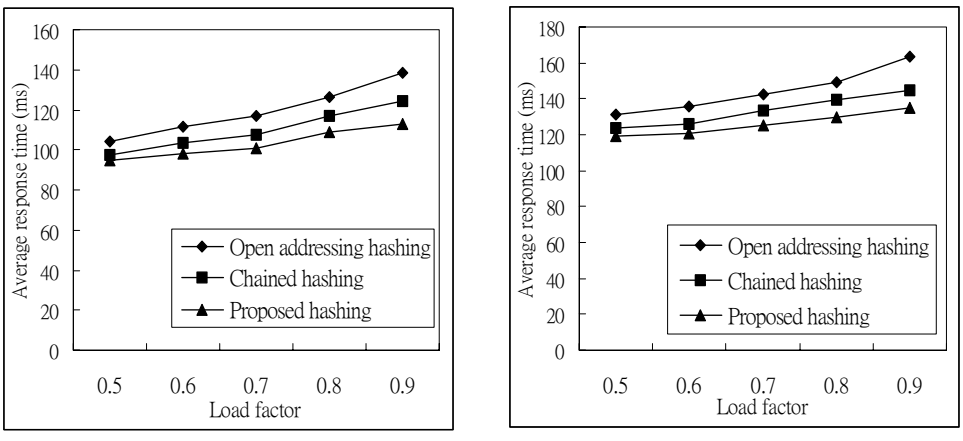

Fig. 8. Comparison of three hashing schemes under different load factors. Assume that the hash table contains 4K entries, and that the size of the posting block is 8MB.

hashing scheme produce lower average response times even though the hash table is 90 percent full. This shows that it is preferable to use the linked-list-based structure for collision resolution when the hash table involves larger load.

Fig. 9 shows the average response time versus the number of entries in the hash table. Assume that the size of the posting block for storing all the cached posting lists is 8MB, and that the compact scheme is applied for the purpose of space management. We varied the number of entries in the hash table from 4K to 20K and compared the average response times of the three hashing schemes. We found that although increasing the number of entries could lengthen the search paths in the hash table, it could also capture more user query locality and reduce the number of disk I/O operations. However, when the number of entries increased from 12K to 20K, the average response time decreased only slightly. We conclude that under the posting block size limitation (8MB), 12K hash-table entries are sufficient to capture most of the locality. Note that as the number of entries increases, FBIS achieves better performance compared to LATimes. Recall that the average posting-list length in the FBIS inverted file is shorter than that in the LATimes inverted file. Under the same posting block size, increasing the number of entries in the hash table may allow more short-posting-lists to be cached.

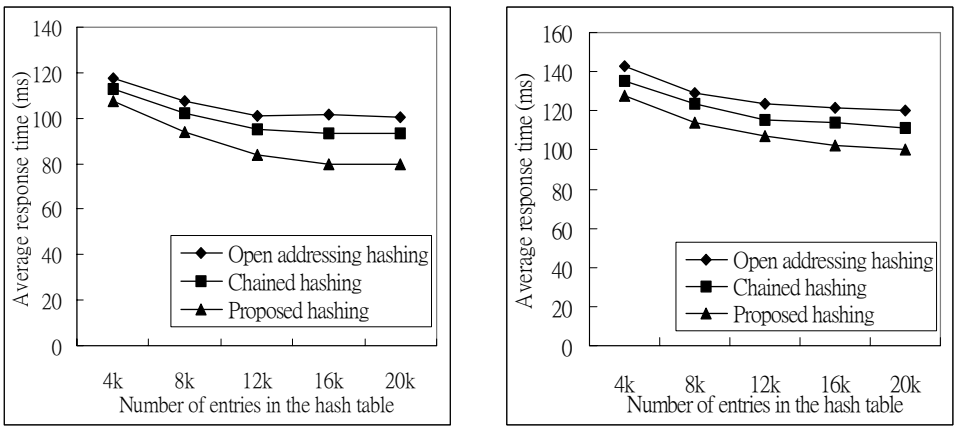

Fig. 9. Effects of the three hashing schemes: (a) the FBIS inverted file and (b) the LATimes inverted files. It was assumed that the size of the posting block was 8MB, and that the compact scheme was applied to store all the cached posting lists.

Fig. 9(a) shows a performance comparison of the three hashing schemes applied to the FBIS inverted file. The open addressing scheme produced the longest average response time due to its probing rule. The chained hashing scheme performed better than the open addressing scheme because the average probing length was reduced by the linked-list structure. The proposed linked-based hashing scheme reduced the average response time more than the other two schemes did because the drawbacks of the open addressing scheme and the chain-hashing scheme are avoided by using a completely linked-list structure. In addition, maintaining such a structure does not require complex

entry allocation or management. The simulation results show that the proposed link-based hashing scheme performed 20% better than the open addressing scheme and 14% better than the chained hashing scheme when the number of entries in the hash table reached 20k.

Fig. 9(b) shows the results obtained by running the same experiment on the LATimes inverted file. The average response time for the LATimes inverted file was longer than that for the FBIS inverted file. This was probably due to the large size of the LATimes inverted file. Again, the proposed link-based hashing scheme performed better than the other two schemes. This is consistent with the results shown in Fig. 9(a).

3.2.3 Effect of the space management scheme applied in the posting block

To examine the space management in the posting block, we compared the performance of the chunk-based scheme with that of the compact scheme. We assumed that the hash table contained 12K entries, and that the proposed hashing scheme was applied for the purpose of collision resolution. We varied the total amount of space in the posting block from 4 MB to 20 MB. For the compact scheme, we use the first-fit approach to find the appropriate amount of space. For the chunk-based scheme, we set the size of a chunk as 90 postings, with each posting being a 32-bit long integer.

Fig. 10 shows that on average, the chunk-based scheme resulted in lower response time than the compact scheme did. With the chunk-based scheme, caching a posting list only requires finding a sufficient number of free-chunks to be linked. With the compact scheme, however, caching a posting list not only requires finding a sufficiently large contiguous space, but also involves complex movements of old data. These compact operations block normal cache usage and increase the average response time.

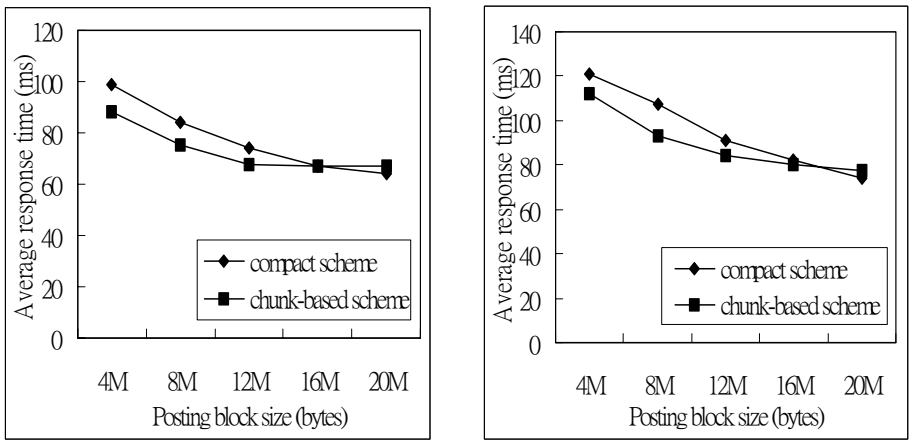

Fig. 10. Comparison of the different space management schemes: (a) the FBIS inverted file and (b) the LATimes inverted file. It was assumed that the hash table contained 12K entries, and that the proposed hashing scheme was applied for the purpose of collision resolution. We varied the size of the posting block used to store all the cached posting lists from 4MB to 20MB.

Fig. 10 also shows that increasing the total amount of space in the posting block can reduce the average response time. This is because increasing the space in the posting block can improve utilization of the hash table. However, when the amount of space increases to 12 MB or more, the average response time of the chunk-based scheme decreases slightly. The reason is that the chunk-based scheme imposes an extra link-traversal cost for retrieving a cached posting list in a large space. On the other hand, the compact scheme performs even better than the chunk-based scheme when the amount of space increases to 20 MB. We conclude that in such a large space, the compact scheme requires few compact operations, and that the contiguous storage of postings saves much retrieval time.

4. CONCLUSIONS

This paper has proposed an inverted file cache architecture for information retrieval systems. In this cache, we use a link-based hashing scheme to speed up entry indexing and a chunk-based scheme to store cached posting lists. We have evaluated these proposed schemes by examining four factors affecting cache performance: the load factor, the hash table size, the posting block size and the chunk size. The simulation results show that the average response time of the test system with an inverted file cache could be reduced by 60% compared with that of the system without an inverted file cache. We believe that an information retrieval system will benefit from the proposed architecture in the following ways:

1. reduction of the average response time,
2. reduction of the number of disk I/O operations, and
3. an increase in the system service capacity.

REFERENCES

1. C. Faloutsos and D. W. Oard, "A survey of information retrieval and filtering methods," Technical Report CS-TR-3514, Department of Computer Science, University of Maryland, 1995.
2. E. Rillof and L. Hollaar, "Text database and information retrieval," *ACM Computer Surveys*, Vol. 28, 1996, pp. 133-135.
3. J. Zobel, A. Moffat, and K. Ramamohanarao, "Inverted files versus signature files for text indexing," *ACM Transactions on Database Systems*, Vol. 23, 1998, pp. 453-490.
4. I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes - Compressing and Indexing Documents and Images*, 2nd ed., Morgan Kaufmann Publishers, Inc., 1999.
5. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, The MIT Press, 1990.
6. M. J. Folk, B. Zoellick, and G. Riccardi, *File Structures*, Addison-Wesley, 1998.
7. D. E. Knuth, *The Art of Computer Programming, Vol. 3, Searching and Sorting*, Addison-Wesley, 1973.
8. U. Manber, *Introduction to Algorithms A Creative Approach*, Addision-Wesley, 1989.

9. A. Moffat and J. Zobel, "Self-indexing inverted files for fast text retrieval," *ACM Transactions on Information Systems*, Vol. 14, 1996, pp. 249-279.
10. TREC, Http://trec.nist.gov, 2001.
11. L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: evidence and implications," *IEEE INFOCOM*, 1999, pp. 126-134.

Wann-Yun Shieh (謝萬雲) received the B.S. degree in Computer Science and Information Engineering from National Chiao Tung University, Hsinchu, Taiwan, Republic of China in 1996. Currently, he is pursuing the Ph.D. degree in computer science and information engineering at National Chiao-Tung University, Hsinchu, Taiwan, Republic of China. His research interests include computer architecture, parallel and distributed systems, and information retrieval.

Jean Jyh-Jiun Shann (單智君) received the B.S. degree in Electronic Engineering from Feng-Chia University, Taichung, Taiwan, Republic of China in 1981. She attended the University of Texas at Austin from 1982 to 1985, where she received the M.S.E. degree in Electrical and Computer Engineering in 1984. She was a lecturer in the Department of Computer Science and Information Engineering, National Chiao-Tung University, Hsinchu, Taiwan, R.O.C., while working towards the Ph.D. degree. She received the degree in 1994 and is currently an Associate Professor in the department. Her current research interests include computer architecture, parallel processing, and information retrieval.

Chung-Ping Chung (鍾崇斌) received the B.E. degree from National Cheng-Kung University, Tainan, Taiwan, Republic of China in 1976, and the M.E. and Ph.D. degrees from Texas A&M University in 1981 and 1986, respectively, all in Electrical Engineering. He was a lecturer in electrical engineering at Texas A&M University while working towards the Ph.D. degree. Since 1986, he has been with the Department of Computer Science and Information Engineering at National Chiao-Tung University, Hsinchu, Taiwan, R.O.C., where he is a professor. His research interests include computer architecture, parallel processing, and parallelizing compiler.