Index Compression is Good, Especially for Random Access

Stefan Büttcher and Charles L. A. Clarke

David R. Cheriton School of Computer Science
University of Waterloo, Ontario, Canada

ABSTRACT

Index compression techniques are known to substantially decrease the storage requirements of a text retrieval system. As a side-effect, they may increase its retrieval performance by reducing disk I/O overhead. Despite this advantage, developers sometimes choose to store index data in uncompressed form, in order to not obstruct random access into each index term's postings list.

In this paper, we show that index compression does not harm random access performance. In fact, we demonstrate that, in some cases, random access into a term's postings list may be realized more efficiently if the list is stored in compressed form instead of uncompressed. This is regardless of whether the index is stored on disk or in main memory, since both types of storage – hard drives and RAM – do not support efficient random access in the first place.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Performance evaluation (efficiency and effectiveness)

General Terms

Experimentation, Performance

Keywords

Index Compression, Random Access, Main Memory, RAM

1. INTRODUCTION

Many search engines employ index compression techniques, such as the byte-aligned vByte method [19] to decrease the storage requirements of their index structures. If the majority of the index data are stored on disk, then this also improves the search engine's retrieval performance, due to reduced disk I/O overhead [14][19]. This even holds if the index access pattern is completely random, since in that case the smaller index leads to a reduced disk seek latency [22][24].

CIKM'07, November 6–8, 2007, Lisboa, Portugal.

In the context of in-memory indices, however, where all index data are held in RAM, it is sometimes argued that index compression methods are not advisable, especially when it is imperative that the search engine can carry out random access operations into the inverted lists stored in the index. Random access into a term's postings list is necessary when computing set intersections in Boolean retrieval [6] and in order to apply the MaxScore [21] heuristic in ranked retrieval. It is usually realized through some sort of a binary search operation (or variations thereof, such as interpolation search [15] or galloping search [7]). Compressed postings lists, in which each posting is stored as a delta value relative to its predecessor, are not directly amenable to binary search, but need to be decompressed (at least partially) prior to a random access operation.

Brewer [2], therefore, suggests that index compression should only be used if the index access pattern is largely sequential. Héman [11] describes methods for keeping the decompression overhead low so that the performance degradation – compared to an uncompressed index – becomes negligible.

In this paper, we show that index compression not only does not harm retrieval performance, but in fact can improve performance, *especially* if index access is predominantly random. Our argument is based on the insight that there exists virtually no storage medium that provides true, penalty-free random access to large amounts of data. Accessing a random byte on disk, for example, involves a costly disk seek (moving the disk's read/write head above the disk track containing the desired information) and the disk's rotational latency (waiting until the desired piece of data has spun under the head). Similarly, despite its name, accessing a random location in RAM may result in a second-level (L2) CPU cache miss or a TLB (translation lookaside buffer) miss — events that carry a grave penalty and that can easily consume a hundred CPU cycles.

The contribution of this paper is two-fold. In a first step, we propose a new data structure, the CPSS-tree ("cache/page-sensitive search tree"), which provides highly efficient random access lookup operations and is an appropriate data structure for postings lists in an in-memory inverted index. In a second step, we show how traditional index compression techniques can be applied to the data stored in a CPSS-tree. We show that by taking into account the nature of the underlying storage medium and by applying compression only at the same granularity at which the storage medium is able to provide true random access (usually a single cache line), it is possible to increase

```
int outPos = 0, previous = 0;
for (int inPos = 0; inPos < n; inPos++) {
  int delta = uncompressed[inPos] - previous;
  while (delta >= 128) {
    compressed[outPos++] = (delta & 127) | 128;
    delta = delta >> 7;
  }
  compressed[outPos++] = delta;
}
```

```
int outPos = 0, previous = 0;
for (int outPos = 0; outPos < n; outPos++) {
  for (int shift = 0; ; shift += 7) {
    int temp = compressed[inPos++];
    previous += ((temp & 127) << shift);
    if (temp < 128) break;
  }
  uncompressed[outPos] = previous;
}
```

Figure 1: C++ code for vByte's encoding (left) and decoding (right) procedure. In each case, n postings are processed. Uncompressed postings are stored in the integer array `uncompressed`, compressed postings in the byte array `compressed`. The actual implementation used in our experiments is slightly different, using pointer arithmetic for every array access.

the performance of random index access operations beyond what is possible with an uncompressed index.

In the following section, we provide a summary of some aspects of computer architecture and index data structures that are relevant in the context of our paper. We also present an overview of related work in the area of *cache-conscious* data structures that explicitly take into account the differences between various types of memory access operations. In Section 3, we describe the hardware and software configurations under which we evaluate the various index layouts examined in this paper. In Section 4, we discuss cache-efficient in-memory index structures for uncompressed inverted lists, pointing out important aspects regarding the hierarchical memory model of today's computer systems and proposing a novel cache-conscious data structure, the CPSS-tree. Section 5, finally, describes how inverted list compression can be integrated into the index structures from Section 4 without sacrificing random access performance (and in fact improving random access performance in all hardware configurations tested in our experiments).

2. BACKGROUND AND RELATED WORK

Inverted Indices

The fundamental data structure of most text retrieval systems is the inverted index [25]. An inverted index consists of two principal components: the *dictionary* and the *inverted lists* (or *postings lists*). The dictionary provides a mapping from each term in the index to the position of its postings list. A term's postings list contains a sequence of *postings*, identifying all occurrences of the term within the text collection referred to by the index.

Postings lists may either contain exact positional information about all occurrences of a given term, or just a list of *docids*, numerical identifiers referring to the documents containing the term in question. Within the context of this paper, we assume that each postings list is simply a sequence of docids. However, the methods presented here are easily applicable to other inverted index variants, such as positional indices and frequency indices.

Index Compression

In a docid index, every postings list can be represented by an increasing sequence of integers. Index compression techniques encode such a sequence by transforming the docids into an equivalent sequence of delta values. Let

$$L = \langle l_1, l_2, \ldots, l_n \rangle$$

be a postings list. Then

$$\Delta(L) = \langle l_1, l_2 - l_1, l_3 - l_2, \ldots, l_n - l_{n-1} \rangle.$$

The elements of the integer sequence $\Delta(L)$ tend to be rather small and can be encoded using standard integer encoding techniques.

One of the most popular methods for encoding such a sequence is the byte-aligned gap encoding method called *vByte* [19]. In vByte, every element of the encoded list is represented by an integral number of bytes, where the most-significant bit in each byte defines whether the given byte is the last one in the current codeword or whether there are more to follow. vByte's encoding and decoding routines are extremely simple (as shown in Figure 1), which at query time translates into low decompression latency and high query processing performance. Using the code fragments shown in the figure, an average vByte-compressed postings lists can be decoded in less than 10 CPU clock cycles per posting for the inverted indices used in our experiments.

Other methods, such as LLRUN [8] and interpolative coding [13] are known to produce better compression results than vByte, but require more complicated decompression operations, making them less interesting for high-performance retrieval operations.

Obviously, a Δ-encoded postings list does not allow random access, as the value of every posting depends on its predecessor. However, by adding so-called *synchronization points* [14] to each compressed list, perhaps one for every 64th posting, it is possible to realize quasi-random list access, where retrieving an individual posting only requires the prior decompression of a small number of postings in its immediate neighborhood, and not the entire list. In its most basic form, a synchronization point is a record of the form

$$(posting, position),$$

where *posting* is the raw value of a posting, and *position* is its (bit- or byte-) position in the compressed postings list. Sometimes, *position* can be omitted because it is implicit from the way in which postings lists are stored in the index.

The concept of synchronization points can be extended recursively, by maintaining second-, third-, fourth-order synchronization points, if the underlying storage medium (e.g., hard drive) does not permit true random access into the list of first-order synchronization points.

Sources of Index Access Latency: Hard Disks

Regardless of whether postings lists are kept in the index in compressed or in uncompressed form, accessing a random element of a given list usually does not come for free. The gravity of the penalty associated with such random access operations depends on the storage medium that contains the index.

If the index is stored on a hard disk drive, then the cost of a random access operation stems from two sources:

Disk seek latency — If the datum to be accessed is stored in a different track (or cylinder) than the previously accessed datum, then the disk drive's read/write head needs to be moved to the new track, an operation that usually takes somewhere between 2 ms and 20 ms (several million CPU cycles).

Rotational latency — Even when the hard drive does not need perform a disk seek in order to access the desired datum, it usually still has to wait some time, until appropriate disk sector has spun under the disk's read/write head. For a hard drive spinning with 7,200 rpm, an average rotational latency of about 4 ms (1 sec / 120 × 50%) can be expected. This effect is alleviated by the hard drive's and/or operating system's aggressive prefetching strategy, optimistically reading all data stored on the current track, ahead of time.

A detailed overview of different performance aspects of hard disk drives is given by Ruemmler and Wilkes [18]. The essential insight here is that random access does not come for free and that the cost of a random access operation depends on the size of the on-disk index (because the disk head, on average, needs to travel farther for a large index than for a small one). This phenomenon is addressed in the experiments conducted by Williams and Zobel [22], who show that a compressed on-disk index leads to better random access performance than an uncompressed one.

Sources of Index Access Latency: Main Memory

For an inverted index stored in main memory, the source of random access latency is not so obvious. It is caused by the various components of the memory hierarchy found in a typical computer system (shown in Figure 2).

Compared to the CPU itself, a computer's RAM is extremely slow, usually requiring somewhere in the order of 30–100 CPU cycles to deliver a machine word to the CPU. Many programs exhibit a strong temporal and spatial locality. That is, they tend to access memory locations that have been accessed very recently (temporal locality), and they tend to access memory locations that are close to other, recently accessed memory locations (spatial locality). In many cases, therefore, the performance gap between CPU and main memory can be closed by employing caches — very fast, but rather small memory units that contain copies of frequently accessed memory locations.

In most modern computer systems, the CPU cache is split up into two components: a first-level (L1) cache, and a second-level (L2) cache. The L1 cache operates at full CPU speed, but only has room for a few kilobytes of data. The L2 serves as a fallback system for the L1 cache, being slightly bigger, but also requiring slightly more time to deliver data to the CPU. A CPU cache takes into account spatial locality by fetching data from main memory in larger chunks, called

	Capacity	Latency
CPU (registers)	128-1024 bytes	1 cycle
L1 Cache	8-64 KB	1-3 cycles
L2 Cache	256-2048 KB	10-20 cycles
Main memory	1-8 GB	30-100 cycles
Hard drive	100-1000 GB	5-50 million cycles

Figure 2: The memory hierarchy. Performance characteristics of various components of a computer system.

cache lines, usually 32 or 64 bytes in size. Once a cache line has been fetched from RAM, access to more data in the same cache line can be realized very efficiently (this is called a *cache hit*). Accessing a memory location, however, whose cache line is not currently in the CPU cache is rather costly, because the entire cache line first needs to be loaded into the cache (this is called a *cache miss*).

In the context of random access operations, cache misses represent a major performance bottleneck, as they force the CPU to almost constantly wait for data from the comparatively slow main memory. This is almost independent of the size of the second-level cache, as a cache line is either in the L1 cache (because it is currently being processed) or not in the cache at all (because it has not been touched in a very long time).

In addition to CPU cache misses, there is a second source of random access latency that is present in virtually every computer system: TLB misses. Most modern computer systems employ *paging* techniques to translate virtual memory addresses into physical ones. The information needed to perform this translation is found in the process-specific page table. In order to avoid accessing the page table for every memory access operation (which would reduce performance by 50%), most CPUs maintain a page table cache operating at full CPU speed, called the *translation lookaside buffer* (TLB), in which they keep the necessary translation information for the most recently accessed memory pages.

Like in the case of the L1 and L2 CPU caches, accessing a random memory location is likely to result in a *TLB miss*, because the page table entry for the respective page is not currently in the TLB. Compared to an L2 cache miss, a TLB miss is even more costly, because it normally entails at least two L2 cache misses (one for the page table entry, another one for the requested data) and sometimes even involves interaction with the operating system's kernel.

An overview of performance aspects of a computer's memory hierarchy is given by Smith [20] and Burger et al. [3].

Cache-Conscious Data Structures

A significant amount of effort has been put into the design of algorithms and data structures that take the performance characteristics of a computer's memory hierarchy into account and try to minimize the number of cache misses. Usually, these *cache-conscious* (or *cache-aware*) data structures are variations of existing and well-studied *cache-unaware* data structures.

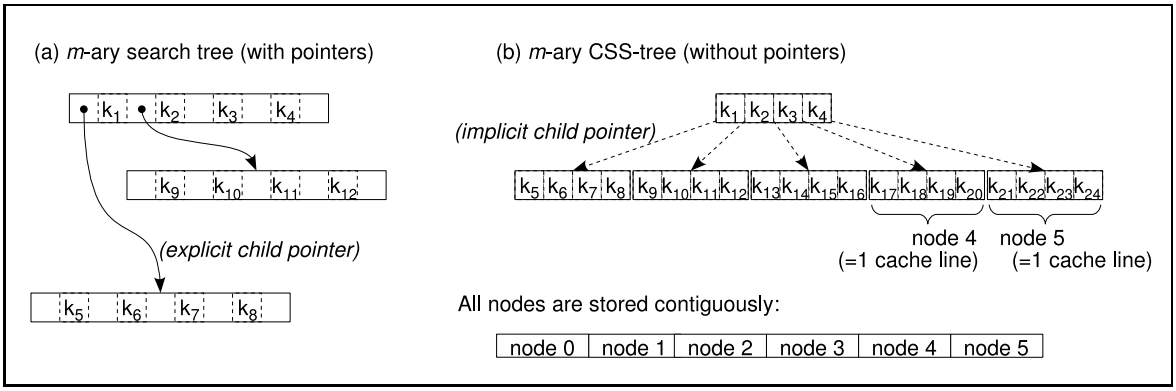

Figure 3: Cache-conscious index organization with CSS-trees. Each node in a CSS-tree occupies exactly one cache line (32 or 64 bytes). Since all nodes are stored in a contiguous fashion, explicit child pointers are no longer necessary. A child's address can be determined from its parent's address by simple arithmetic.

Closest to the work presented here is the research conducted by Rao and Ross [16]. In their paper, they discuss how m-ary search trees can be adjusted to exhibit improved memory access locality and thus better cache behavior in the presence of random access. Each node in their CSS-tree ("cache-sensitive search tree") data structure occupies exactly one CPU cache line. Nodes only contain keys, and no pointers to their children, improving space utilization and thus decreasing the number of cache misses even further. Child pointers are eliminated by storing all nodes in a large array, similar to the data organization found in the canonical implementation of HEAPSORT. An example of a CSS-tree is given by Figure 3.

The number of cache misses produced by a CSS-tree when searching for a random element in a tree containing n elements is $\log_{m+1}(n)$, a substantial improvement over the $\log_2(n)$ cache misses caused by a naïve binary search. Here, $m+1$ is the branching factor of the CSS-tree, i.e., m is the number of keys that may be stored in a single cache line (usually 8 or 16).

In a follow-up paper, Rao and Ross [17] show how the cache behavior of B$^+$-trees can be improved by eliminating child pointers and by storing all child nodes of a given inner node in a contiguous fashion. Compared to CSS-trees, their CSB$^+$-trees ("cache sensitive B$^+$-trees") have the advantage that they support efficient update operations (the contents of a CSS-tree are assumed to be static).

Cui et al. [5] propose BD-trees, a hash-based alternative to CSS-trees and CSB$^+$-trees. Hankins and Patel [10] explore how the node size of a CSB$^+$-tree affects its cache misses, TLB misses, and consequently its lookup performance.

Similar techniques are also applied in different areas of high-performance computing. Xiao et al. [23], for instance, show how the performance of MERGESORT can be improved by employing tiling and padding to partition the data in a way that better reflects the structure of the cache. Kowarschik and Weiß [12] present an overview of of cache optimization techniques in the context of numerical algorithms.

3. EXPERIMENTAL SETUP

In the remainder of the paper, we investigate various list structures that can be used to store the postings lists found in an in-memory inverted index. These structures are evaluated through performance measurements on realistic sets of data. The general procedure of our experiments is as follows:

1. An inverted index (docid index) is built from a given text collection and loaded into main memory.
2. At random, a term T and a document D are chosen.
3. A search on T's postings list is performed, returning the smallest (according to docid) document D' such that $T \in D'$ and $D' \geq D$. If $T \in D$, then $D' = D$.

For each experiment, in order to obtain reliable performance figures, steps 2 and 3 are repeated several million times for random pairs (T, D). The experiment as a whole is repeated six times, and the lowest time measured is reported, following the assumption that any deviations from this best run is due to background system activity by other processes. The performance numbers presented in this paper only include the time necessary to find D' in T's postings list, not the time it takes to produce the random pairs (T, D), nor the time required to find T's entry in the dictionary data structure (for example through a hash table lookup).

In our experiments, the index is built from the text collection known as *TREC disks 1–5*. The collection has a total size of 4.8 GB and consists of 2.1 million distinct terms in 1.5 million documents. For each term in the collection, we construct a list of docids — integers specifying the set of documents the term appears in. We remove from the index all terms that appear in less than $2^{14} = 16{,}384$ documents, since for such short postings lists the value of random access experiments would have been rather limited. The resulting index contains 3,408 terms with a total of 244 million postings (median list length: 36,003). The total size of the uncompressed index is approximately 1 GB, with small variations depending on the specific index structure chosen.

To show the general applicability of the techniques proposed in this paper, all methods are evaluated on a variety of different computer systems, purchased over the last seven years. An overview of these systems is given by Table 1.

With the exception of the Pentium III, all systems are endued with sufficiently much main memory to load the entire docid index for the TREC collection into RAM. In the experiments with the Pentium III system, because it

	AMD Opteron	Intel Core	AMD Athlon MP	Intel Pentium III
Year of purchase	2007	2006	2002	2000
CPU frequency (MHz)	2814	1667	1600	1005
Main memory (MB)	2048	1536	2048	768
L2 cache line size (bytes)	64	64	64	32
L2 miss latency (ns)	13	26	29	27
TLB miss latency (ns)	78	90	395	181
Decompression (ns per posting)	2.1	3.1	5.5	8.6
Decompression (postings per L2 miss)	6.5	8.3	5.3	3.1

Table 1: Performance characteristics of the computer systems used in our experiments. The numbers in rows "L2 miss latency" and "TLB miss latency" are rough estimates. The exact TLB miss latency is highly variable and depends greatly on the memory access pattern.

only contains 768 MB of RAM, we chose to use a slightly smaller index instead, containing postings only for the first 1 million documents in the collection. This explains why, in our experiments, the Pentium III exhibits better index performance than the Athlon MP, despite the Athlon's higher CPU frequency.

In addition to average lookup latency (measured in nanoseconds per random index access), we also report some auxiliary performance measures, such as the number of L2 cache misses and TLB misses caused by a single index access operation (on average). All such numbers were obtained from the built-in performance counters of the AMD Opteron CPU used in our experiments. To access these performance counters, we used the `oprofile`[1] Linux kernel module (version 0.9.2 for kernel 2.6.20) and its event types `L2_CACHE_MISS` and `L1_DTLB_AND_L2_DTLB_MISS`.

	Binary Search	Interp. Search
AMD Opteron	953 ns	849 ns
Intel Core	1,403 ns	1,325 ns
AMD Athlon MP	3,061 ns	2,775 ns
Intel Pentium III	2,642 ns	2,557 ns
L2 cache misses	24.0	13.7
TLB misses	11.8	7.3
Total index size	930 MB	930 MB

Table 2: Average random index access performance with an uncompressed index. Interpolation search fares slightly better than binary search, but on average still requires almost 1 μs (2,800 clock cycles) per random access on the Opteron system.

4. EFFICIENT RANDOM ACCESS FOR IN-MEMORY INVERTED LISTS

When aiming for efficient random access into the postings lists stored in an inverted file, the most obvious implementation is to store each inverted list in uncompressed form (e.g., every posting as a 32-bit integer) and to employ a binary search algorithm to find a given docid in the postings list for a given term T.

We implemented this index structure and conducted the experiment described in Section 3, using the 5-GB TREC collection. In total, for each experiment, 10 million binary search operations were performed. Each experiment was repeated six times.

The best-performing run from each experiment is reported in Table 2. It can be seen that a random list accessed, realized through binary search, takes surprisingly long. On the Opteron system, for example, the average latency of a search operation is 953 ns. Since the median list in the index comprises 36,003 postings (cf. Section 3), binary search should require no more than 16 comparison operations on average to find the desired list element (2^{16} = 65,536). Hence, according to the results of our experiments, each comparison takes approximately 60 ns, or $60 \times 2.8 = 168$ CPU clock cycles.

Of course, this rather low performance is not caused by the CPU actually executing 168 operations per comparison, but is caused by the poor cache behavior of binary search. Suppose the CPU cache line size is 64 bytes ($\hat{=}$ 16 postings). Then a binary search on a given postings list of length n will access roughly $\log_2(n/16) \approx 12$ different cache lines and thus cause 12 data cache misses. Similarly, approximately $\log_2(n/1024) \approx 6$ different pages are accessed, resulting in 6 TLB misses (assuming $n \approx 2^{16}$).

While the actual number of L2 misses and TLB misses reported by `oprofile` and shown in the table is somewhat higher than expected (almost exactly twice as high, suggesting that something might be wrong with the version of `oprofile` that we used or with the way in which we used it), it confirms the suspicion that the low performance of binary search is caused by its extremely low cache efficiency.

Employing a more sophisticated search procedure, such as interpolation search [15], which requires fewer access into the list being searched, can improve the performance of random index access operations slightly, as shown in Table 2. However, the improvement is not very dramatic, less than 15% for the four systems we tested, mainly because the speedup achieved by more benign cache behavior is partially offset by the greater computational complexity of each step in interpolation search.

CSS-Trees

It is clear that, in order to realize low-latency random index access, a different arrangement of the in-memory postings lists needs to be employed that allows the search engine to

[1] http://oprofile.sourceforge.net/ (accessed 2007-05-19)

CSS-tree node size	32 bytes	64 bytes	128 bytes
AMD Opteron	524 ns	478 ns	488 ns
Intel Core	868 ns	758 ns	735 ns
AMD Athlon MP	1,827 ns	1,588 ns	1,584 ns
Intel Pentium III	1,652 ns	1,529 ns	1,640 ns
L2 cache misses	6.0	4.1	6.6
TLB misses	3.6	3.0	2.7
Total index size	945 MB	934 MB	931 MB

Table 3: Random index access performance with CSS-trees (uncompressed).

retrieve a random posting with a smaller number of cache misses. As discussed in Section 2, Rao's CSS-tree [16] is such a data structure.

We implemented an inverted index data structure in which each postings list is not stored as a flat array of integers, but inside a CSS-tree. We then repeated the experiment from before, picking random pairs (T, D) and performing a search for D in T's postings list.

The results we obtained for various CSS-tree node sizes (32 bytes, 64 bytes, 128 bytes) are shown in Table 3. The variations in the index size (greater node size results in smaller index) are due to a slight deviation of our implementation from Rao's originally proposed tree structure. To simplify the implementation, and to allow each tree to be constructed in a bottom-up fashion, our variant of the CSS-tree replicates tree nodes in the upper levels of the tree (but not in the two bottom-most levels). That is, the k-th key in a node at one of the higher levels of the tree is a copy of the first key in the node's k-th child node (in Rao's implementation, the k-th key in a node would numerically be between the keys in the k-th child and the $(k+1)$-st child). This deviation has the effect of increasing the storage requirements of each tree by by up to 1.5% for 32-byte tree nodes (up to 0.4% for 64-byte tree nodes). However, because the difference is so small, our results still reflect the true performance of CSS-trees on this kind of search operation.

Compared to naïve binary search, our implementation of the CSS-tree can process a request for a random posting between 73% (Pentium III) and 99% (Opteron) faster (comparing Table 3 with Table 2). The performance gain stems from a greatly reduced number of cache misses.

For a node size of 64 bytes, which is the same as the cache line size of the Opteron CPU, the average number of L2 cache misses per index access is reduced from 24 to 4.1. What is interesting, however, is that the best results are not always achieved for a node size equal to the cache line size of the CPU. Both the Intel Core and the AMD Athlon MP have 64-byte cache lines, but performance is best for a node size of 128 bytes. Similarly, the Pentium III has 32-byte cache lines, but exhibits the lowest latency for 64-byte tree nodes.

The reason for this discrepancy lies in the number of TLB misses caused by the random access operations. On the Opteron system, for example, although the number of L2 cache misses is minimized by choosing a node size of 64 bytes, TLB misses are smaller for a node size of 128 bytes. Because TLB misses are far more costly than simple data cache misses (cf. Table 1), this affects the lookup performance of CSS-trees, which were designed with the single objective of minimizing data cache misses, ignoring the TLB.

In order to achieve optimal random access performance, it seems reasonable to account for both sources of random access latency: L2 cache misses and TLB misses.

CPSS-Trees

The experience we made with CSS-trees motivated the design of a new data structure, the CPSS-tree ("cache/page-sensitive search tree"). A CPSS-tree is similar to a CSS-tree. However, instead of just minimizing the number of cache lines accessed in a search operation, a CPSS-tree also considers the cost associated with a TLB miss. Its primary objective is to minimize the number of different pages accessed by a search operation (TLB misses). Its secondary goal is to minimize the number of cache lines accessed within each page (data cache misses).

A single node in a CPSS-tree occupies an entire memory page (typically 4 KB), whereas a CSS-tree node occupies just a single cache line (32 or 64 bytes). If used to store 32-bit docids, the number of TLB misses (= number of pages touched) caused by a search operation on a list of n postings is then approximately $\log_{1024}(n)$, because each page can store up to 1,024 postings. Compared to the asymptotical number of TLB misses caused by a CSS-tree with 64-byte nodes ($\approx \log_{17}(n)$, because each node holds up to 16 keys and has up to 17 children), this constitutes a major improvement.

In order to minimize the number of cache misses per page access, the postings within each CPSS-tree node are arranged inside a local CSS-tree. The height of this tree depends on the node size in the CSS-trees stored within each node of the CPSS-tree. A node size of 32 bytes, leading to a branching factor of $32/4 + 1 = 9$, results in a tree of height 3 (i.e., at most 4 cache misses per page accessed). With 64-byte CSS nodes, the CSS-tree inside any given CPSS node has height 2 (up to 3 cache misses per TLB miss).

The general structure of a CPSS-tree is not very different from that of a cache-sensitive BD-tree [5], a hash-based indexing structure devised for similar purposes. Compared to BD-trees, however, CPSS-trees have the advantage that they allow more efficient traversals and range queries. Moreover, because keys are not stored in strictly increasing order in a BD-tree, CPSS-trees are better suited for standard query processing tasks (where postings are processed in sequential fashion) and more amenable to index compression, which is what we are aiming for.

An example of a fragment of a CPSS-tree is shown in Figure 4. In the example, each key (= posting) consumes 4 bytes, a cache line holds 64 bytes (i.e., 16 keys per CSS-node), and each memory page consists of 4,096 bytes (i.e., 64 CSS-nodes per CPSS-node). The pointers in the example are all implicit pointers. That is, the address of a child node is obtained by arithmetic operations (possible because all nodes are stored in memory in a contiguous fashion) instead of following explicit memory references.

For storage efficiency reasons, our implementation does not require the root node of a CPSS-tree to occupy an entire memory page. Whenever the root node consumed fewer than 5 cache lines, it was stored as a simple CSS-tree, directly in the inverted index's dictionary data structure. Obviously, in that case, an explicit pointer was needed to the first child node of the CPSS root node.

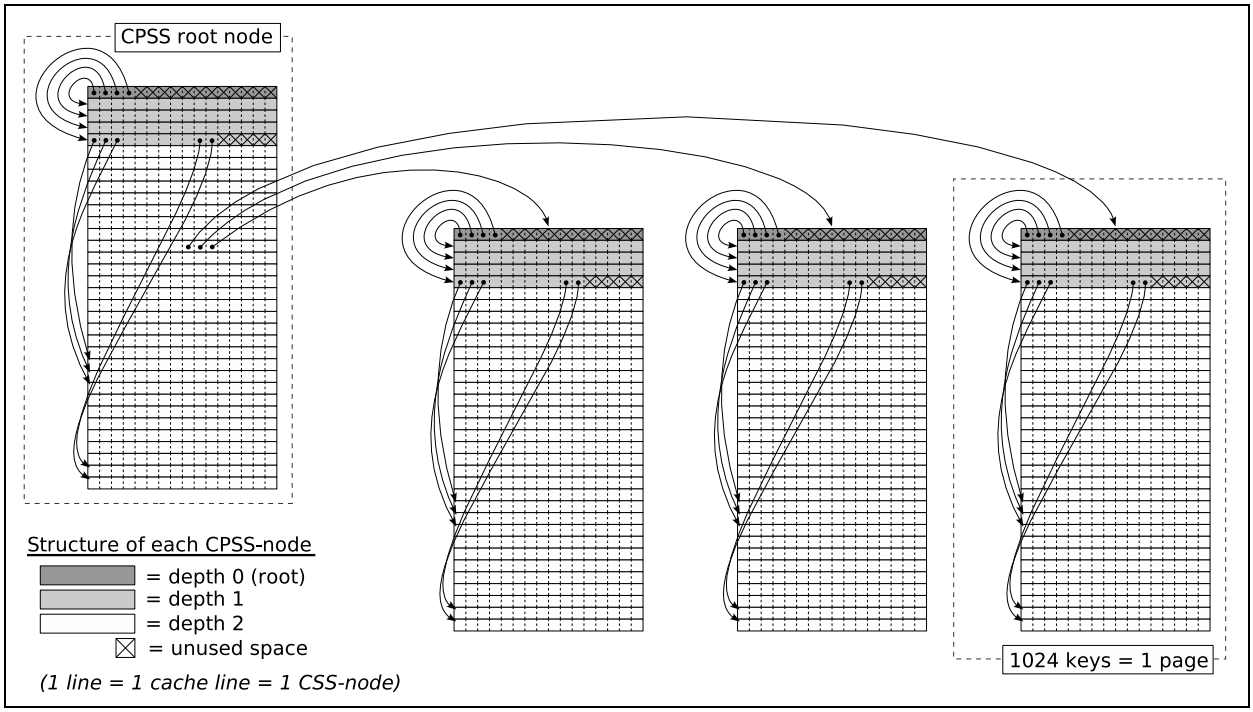

Figure 4: Improved cache-conscious index organization with CPSS-trees. Each node in a CPSS-tree (except for the root node, which may be smaller, depending on how many children it has) occupies exactly one memory page. The keys (postings) in a CPSS-node are organized in a CSS-tree (with CSS node size = cache line size for optimal performance). All child pointers are implicit.

CSS-tree node size	32 bytes	64 bytes
AMD Opteron	407 ns	401 ns
Intel Core	625 ns	608 ns
AMD Athlon MP	1,452 ns	1,374 ns
Intel Pentium III	1,265 ns	1,289 ns
L2 cache misses	7.6	5.9
TLB misses	1.4	1.4
Total index size	961 MB	957 MB

Table 4: Random index access performance with CPSS-trees (uncompressed; node size: 4 KB). The number of L2 cache misses is higher than with CSS-trees, but overall performance is better, thanks to a lower number of TLB misses per index operation.

The performance results we obtained with our implementation of the CPSS-tree are shown in Table 4. Compared to CSS-trees (cf. Table 3), the number of L2 cache misses is increased, but the number of TLB misses is decreased. Because of the great cost of TLB misses compared to data cache misses, overall performance is increased by 15% (Athlon MP), 19% (Opteron), and 21% (Core and Pentium III), respectively. The total size of the CPSS-tree-based inverted index is slightly higher than that of the CSS-tree-based one (+2.5% for a node size of 64 bytes), due to internal fragmentation in the relatively large CPSS nodes of each postings list.

Best results are obtained if the size of each CPSS node is chosen to reflect the system's page size and the size of the CSS nodes within each CPSS node is chosen to equal the cache line size of the CPU cache. For the Pentium III system, for example, 32-byte nodes (and not 64-byte nodes, as in the case of CSS-trees) lead to optimal performance.

5. COMPRESSING IN-MEMORY INDICES

Existing index compression techniques can be applied to the postings stored in a CPSS-tree in a straightforward manner. The overall structure of the tree remains the same, but postings within each leaf node are stored in compressed form instead of uncompressed.

Using vByte [19] (byte-aligned gap encoding) to compress the postings sequence found in each leaf node, the total size of the index can be reduced by up to 73% compared to an uncompressed index in which each posting consumes 32 bits. Consequently, the CPU can make better use of its caches. The number of L2 misses per search operation is reduced by 1.1, from 5.9 to 4.8; the number of TLB misses is reduced by 0.2, to 1.2 TLB misses per random index access (cf. Table 5, column "0").

For two of the systems used in our experiments, Athlon MP and Pentium III, this improvement is already sufficient to improve the index's random access performance, from 1,374 to 1,294 ns in the case of the Athlon, and from 1,265 to 1,145 ns in the case of the Pentium III. For the two other systems, however, random access latency is higher with the compressed index than with the uncompressed index. The reason for this is the increased computational load associated with decompressing the postings in each leaf node.

According to Table 1, decompressing a single vByte-encoded posting on average takes about 2 ns on the Opteron.

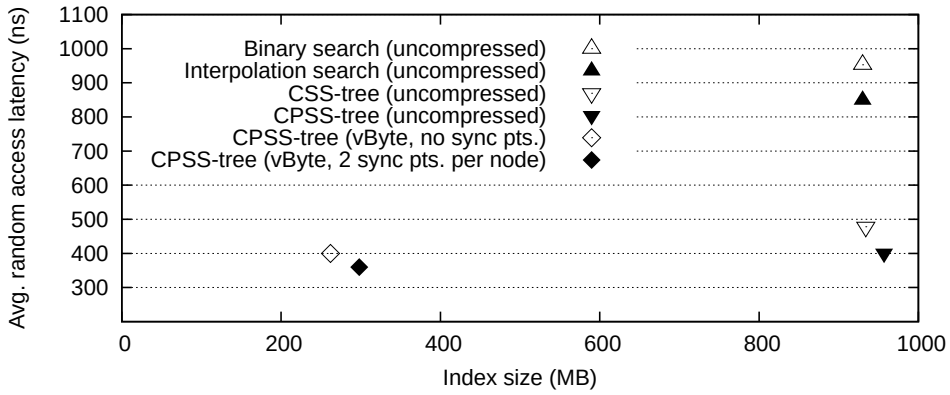

Figure 5: Space/time trade-off points for the Opteron system used in our experiments. CPSS node size: 4 KB (one page). CSS node size: 64 bytes (one cache line).

Sync pts. per leaf	0	1	2
AMD Opteron	400 ns	362 ns	360 ns
Intel Core	612 ns	551 ns	549 ns
AMD Athlon MP	1,294 ns	1,206 ns	1,194 ns
Intel Pentium III	1,145 ns	1,125 ns	1,157 ns
L2 cache misses	4.8	4.8	4.9
TLB misses	1.2	1.2	1.2
Total index size	262 MB	279 MB	298 MB

Table 5: Random index access performance with compressed CPSS-trees (vByte). Node size = cache line size (32 bytes for the Pentium III; 64 bytes for the other systems). The number of synch. points per compressed leaf node is varied between 0 and 2.

A leaf node, on average, contains around 60 compressed postings. We may expect that 50% of these need to be decompressed in order to find the desired posting, leading to an overhead of about 60 ns per search operation and having the effect that the increased cache efficiency does not immediately lead to better random index access.

Fortunately, reducing the decompression overhead is relatively simple. Each leaf node in the CPSS-tree is augmented with a sequence of synchronization points (cf. Section 2). Each synchronization point consists of an uncompressed posting (32-bit integer) and an 8-bit integer specifying the position of the compressed posting immediately following the given synchronization point. The sequence of synchronization points is stored at the beginning of each leaf node (= CPU cache line), followed by a sequence of compressed postings.

By adding a single synchronization point to each leaf node in the search tree, the decompression overhead can be reduced by 50%. By adding a second synchronization point, a further 33% reduction is possible. Table 5 shows the performance improvements measured in our experiments. For the Opteron PC, random access latency can be reduced by 10%, from 401 ns to 360 ns. For the other systems, a similar speedup can be achieved (Core: -10%; Athlon MP: -13%; Pentium III: -11%).

Inserting synchronization points into the leaf nodes increases the size of the index, by 14% in the case of two synchronization points per node. However, compared to an uncompressed index, the resulting index still is relatively compact (-69%, compared to uncompressed 32-bit postings). Figure 5 shows various space-time trade-off points for the Opteron system with 64-byte cache lines.

Other Compression Methods

Obviously, the method described above does not only work with vByte, but also with other compression methods. However, the number of compression methods that can effectively be used to decrease the random access latency is fairly limited, as their decoding throughput needs to be extremely high, even with additional synchronization points in the leaf nodes of each CPSS-tree. We tested two methods that meet this requirement:

- Simple-9 [1] is a word-aligned compression method specifically designed with the goal of high decoding performance. It achieves better compression rates than vByte and provides decoding routines that are almost as efficient as vByte's.
- Rice codes are a variant of Golomb codes [9] in which the modulus M is a power of 2 ($M = 2^k$). They can be decoded more efficiently than Golomb codes (bit-shifts instead of integer division/multiplication), especially if used in conjunction with bit buffering [4].

The results we obtained with these two methods (and our original implementation based on vByte) on the Opteron system are shown in Figure 6. Both methods, Rice and Simple-9, achieve far better compression rates than vByte. However, Rice's relatively high decoding overhead ($\approx$ 5 ns per posting) leads to an average random access latency that is almost twice as high as that of vByte. However, by inserting 5 synchronization points into each leaf node, Rice's random access performance can be reduced to under 400 ns. Remarkably, even with this many synchronization points, the total size of the index is still below what we see when vByte is used (237 MB vs. 262 MB).

With Simple-9, the situation is different. While the method does not achieve the same compression rates as Rice coding, it has the advantage of extremely low-latency decoding operations. Without any synchronization points, Simple-9's average random access latency is 406 ns, only

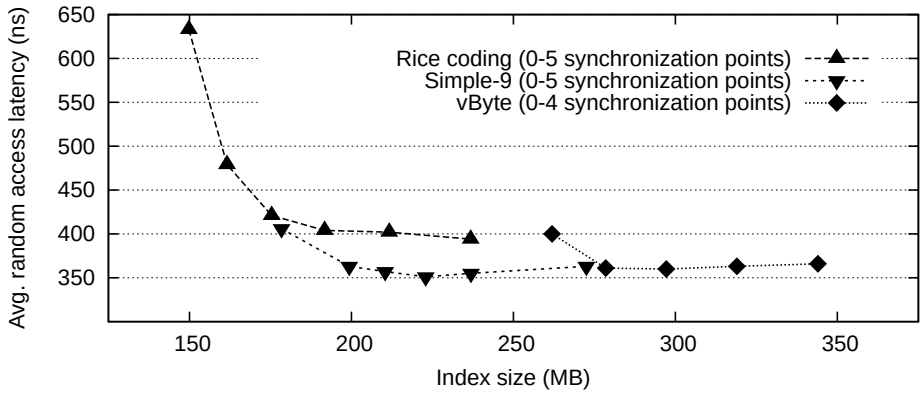

Figure 6: CPSS space/time trade-off points for different compression methods, varying the number of synchronization points per leaf node between 0 and 4 (vByte) or between 0 and 5 (Rice, Simple-9). Hardware: AMD Opteron (4-KB pages, 64-byte cache lines).

6 ns more than vByte in the same configuration. With 3 synchronization points per leaf node, Simple-9 achieves a random access latency of 351 ns. This is lower than what is attainable by using vByte. The reason for Simple-9's dominance over vByte (smaller index and lower index access latency at the same time) is that its more compact index structure results in fewer cache misses — an effect that outweighs the slightly increased decoding overhead.

6. CONCLUSION

Text retrieval systems can be distinguished based on whether they keep their index structures in main memory or on disk. A search engine's index access pattern, on the other hand, can be classified according to whether postings lists are accessed in a random fashion or largely sequentially.

It has long been known that index compression can improve the performance of on-disk indices, regardless of whether they are accessed in a random fashion or sequentially [22]. In this paper, we have shown that the random access performance of in-memory indices can also be improved by applying index compression techniques. The last remaining question, therefore, is whether index compression may help increase the performance of *sequential* list operations on in-memory inverted files.

At this point, it is not clear whether this is the case. However, if the performance gap between CPU and main memory keeps widening, it might in fact be possible that index compression, because it allows the CPU to make better use of memory bandwidth, also leads to improved performance for sequential operations on in-memory indices. Novel index compression methods like Zukowski's PFOR and PFOR-delta [26], taking the super-scalar nature of modern CPUs into account, seem like the most promising candidates.

7. REFERENCES

[1] V. N. Anh and A. Moffat. Inverted Index Compression using Word-Aligned Binary Codes. *Information Retrieval*, 8(1):151–166, January 2005.

[2] E. A. Brewer. Combining Systems and Databases: A Search Engine Retrospective. In J. M. Hellerstein and M. Stonebraker, editors, *Readings in Database Systems (4th edition)*, Cambridge, Massachusetts, 2005. MIT Press.

[3] D. Burger, J. R. Goodman, and G. S. Sohi. Memory Systems. In *The Computer Science and Engineering Handbook*, pages 447–461. 1997.

[4] S. Büttcher and C. L. A. Clarke. Unaligned Binary Codes for Index Compression in Schema-Independent Text Retrieval Systems. University of Waterloo Technical Report CS-2006-40, October 2006.

[5] B. Cui, B. C. Ooi, J. Su, and K.-L. Tan. Main Memory Indexing: The Case for BD-Tree. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):870–874, 2004.

[6] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive Set Intersections, Unions, and Differences. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2000)*, pages 743–752, San Francisco, USA, January 2000.

[7] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Experiments on Adaptive Set Intersections for Text Retrieval Systems. In *Revised Papers from the Third International Workshop on Algorithm Engineering and Experimentation (ALENEX 2001)*, pages 91–104, Washington, DC, USA, January 2001.

[8] A. S. Fraenkel and S. T. Klein. Novel Compression of Sparse Bit-Strings — Preliminary Report. *Combinatorial Algorithms on Words, NATO ASI Series*, 12:169–183, 1985.

[9] S. W. Golomb. Run-Length Encodings. *IEEE Transactions on Information Theory*, IT-12:399–401, July 1966.

[10] R. A. Hankins and J. M. Patel. Effect of Node Size on the Performance of Cache-Conscious B+-Trees. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 283–294, San Diego, USA, June 2003.

[11] S. Héman. Super-Scalar Database Compression between RAM and CPU Cache. Master's Thesis. University of Amsterdam. Amsterdam, The Netherlands, July 2005.

[12] M. Kowarschik and C. Weiß. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In *Algorithms for Memory Hierarchies, Advanced Lectures*, pages 213–232, March 2002.

[13] A. Moffat and L. Stuiver. Binary Interpolative Coding for Effective Index Compression. *Information Retrieval*, 3(1):25–47, 2000.

[14] A. Moffat and J. Zobel. Self-Indexing Inverted Files for Fast Text Retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, October 1996.

[15] Y. Perl, A. Itai, and H. Avni. Interpolation Search – A log log n Search. *Communications of the ACM*, 21(7):550–553, 1978.

[16] J. Rao and K. A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings of 25th International Conference on Very Large Data Bases*, pages 78–89, Edinburgh, UK, September 1999.

[17] J. Rao and K. A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 475–486, Dallas, USA, May 2000.

[18] C. Ruemmler and J. Wilkes. An Introduction to Disk Drive Modeling. *The Computer Journal*, 27(3):17–28, 1994.

[19] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of Inverted Indexes for Fast Query Evaluation. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 222–229, Tampere, Finland, August 2002.

[20] A. J. Smith. Bibliography and Reading on CPU Cache Memories and Related Topics. *SIGARCH Computer Architecture News*, 14(1):22–42, 1986.

[21] H. Turtle and J. Flood. Query Evaluation: Strategies and Optimization. *Information Processing & Management*, 31(1):831–850, November 1995.

[22] H. E. Williams and J. Zobel. Compressing Integers for Fast File Access. *The Computer Journal*, 42(3):193–201, 1999.

[23] L. Xiao, X. Zhang, and S. A. Kubricht. Improving Memory Performance of Sorting Algorithms. *ACM Journal of Experimental Algorithms*, 5, 2000.

[24] J. Zobel and A. Moffat. Adding Compression to a Full-Text Retrieval System. *Software — Practice and Experience*, 25(8):891–903, 1995.

[25] J. Zobel and A. Moffat. Inverted Files for Text Search Engines. *ACM Computing Surveys*, 38(2):6, 2006.

[26] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE*, page 59, 2006.