Redesigning the String Hash Table, Burst Trie, and BST to Exploit Cache

NIKOLAS ASKITIS, RMIT University
JUSTIN ZOBEL, NICTA, University of Melbourne

A key decision when developing in-memory computing applications is choice of a mechanism to store and retrieve strings. The most efficient current data structures for this task are the hash table with move-to-front chains and the burst trie, both of which use linked lists as a substructure, and variants of binary search tree. These data structures are computationally efficient, but typical implementations use large numbers of nodes and pointers to manage strings, which is not efficient in use of cache. In this article, we explore two alternatives to the standard representation: the simple expedient of including the string in its node, and, for linked lists, the more drastic step of replacing each list of nodes by a contiguous array of characters. Our experiments show that, for large sets of strings, the improvement is dramatic. For hashing, in the best case the total space overhead is reduced to less than 1 bit per string. For the burst trie, over 300MB of strings can be stored in a total of under 200MB of memory with significantly improved search time. These results, on a variety of data sets, show that cache-friendly variants of fundamental data structures can yield remarkable gains in performance.

Categories and Subject Descriptors: E.2 [**Data**]: Data Storage Representations; E.1 [**Data**]: Data Structures

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Hash table, burst trie, binary search tree, dynamic array, cache, strings, node clustering, pointer-less storage, Judy trie

ACM Reference Format:
Askitis, N. and Zobel, J. 2011. Redesigning the string hash table, burst trie, and BST to exploit cache. ACM J. Exp. Algor. 15, 1, Article 1.7 (January 2011), 61 pages.
DOI = 10.1145/1921703.1921704 http://doi.acm.org/10.1145/1921703.1921704

1. INTRODUCTION

Simple in-memory data structures are basic building blocks of programming, and are used to manage temporary data in scales ranging from a few items to gigabytes. For the storage and retrieval of strings, the main data structures are the varieties of hash table, trie, and binary search tree.

An efficient representation for the hash table is a *standard chain*, consisting of a fixed-size array of pointers (or slots), each the start of a linked list, where each node

This project was supported by the Australian Research Council and by the personal funds and resources of the authors.
A preliminary version of the material in this article appeared in the paper "Cache-conscious Collision-Resolution in String Hash Tables," N. Askitis & J. Zobel, *Proceedings of the SPIRE International Conference on String Processing & Information Retrieval*, Buenos Aires, Argentina, 2005, pages 92–104; and as Chapters 1 to 4 of "Efficient Data Structures for Cache Architectures," PhD Thesis (available online). Software available at http://www.naskitis.com.
Author's Addresses: N. Askitis, RMIT University, Melbourne, Australia; email: askitisn@gmail.com; J. Zobel, NICTA, University of Melbourne, Australia; email: jz@csse.unimelb.edu.au.

DOI 10.1145/1921703.1921704 http://doi.acm.org/10.1145/1921703.1921704

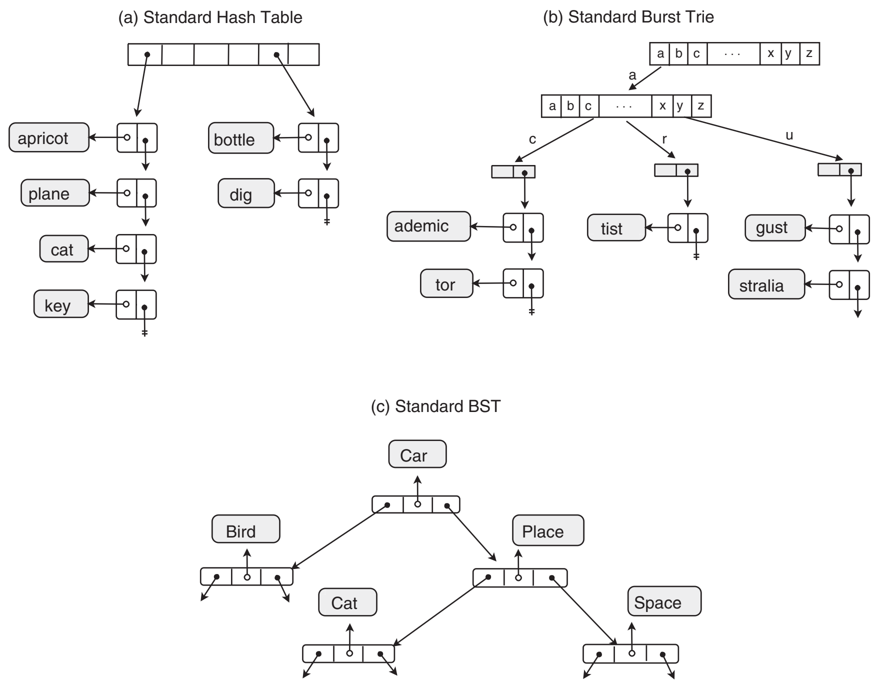


Fig. 1. The hash table (a), burst trie (b), and binary search tree (c). These standard chained data structures are currently among the fastest and most compact data structures for storage and retrieval of variable-length strings in memory. The hash table is the fastest and most compact, but it cannot maintain strings in sort order. The burst trie can maintain efficient sorted access to containers, while approaching the speed of hashing. Sorted access is necessary for tasks such as range search. The BST is a fully sorted data structure that offers good performance, on average.

in the list contains a pointer to a string and a pointer to the next node. The most efficient form of trie is the burst trie; a *standard* burst trie consists of trie nodes—each node being an array of pointers, one for each letter of the alphabet—that descend into containers represented as linked lists, where each node has a string pointer and a pointer to the next node. The use of a trie structure allows for rapid sorted access to containers; the strings within containers however, remain unsorted. In the *standard* binary search tree (BST), each node has a string pointer and two child pointers. These string data structures are illustrated in Figure 1 and are currently among the fastest and most compact tools available for managing large sets of strings in memory [Askitis and Sinha 2007; Heinz et al. 2002; Williams et al. 2001; Zobel et al. 2001; Bell and Gupta 1993; Knuth 1998; Crescenzi et al. 2003].

We use these data structures for common computing applications such as text compression, pattern matching, dictionary management, vocabulary accumulation, and document processing. Typical software systems such as databases and search engines are dependent on data structures to manage their data efficiently. Most existing data structures in computer science, such as the standard hash table, the trie, the BST and its variants including the splay tree, red-black tree, AVL tree, and ternary search trie, assume a nonhierarchical random access memory (RAM) model, which states

—All memory addresses supported by the underlying architecture are accessible.
—All memory accesses are of equal cost.

These assumptions allow data to be placed anywhere in memory, with the expectation of uniform cost. As a result, existing data structures can be fast to access, as they can minimize the number of instructions executed. However, the speed of processors have increased more rapidly than has the response time of main memory devices, leading to a performance gap that has reached two orders of magnitude on current machines. The implications are serious. A random access to memory can force the processor to wait for many hundreds of clock cycles. To alleviate these high access costs, current processors have a system of caches that sit between the processor and memory. A cache is a small high-speed memory device that operates on the assumption that recently accessed data is likely to be referenced again in the near future. A cache can thereby create the illusion of fast memory, by copying recently accessed data from main memory, in anticipation of its reuse in the near future. Main memory is consulted only when the required data is not in cache, which is known as a cache-miss.

However, the use of cache violates the principles of the RAM model. First, not all addresses in memory are accessible. A cache is a transparent layer of memory that cannot be accessed by a program and is administrated solely by the underlying hardware. Second, memory access does not have a uniform cost—a cache is at least a magnitude faster than main memory. Nonetheless, typical implementations of existing string data structures such as the standard hash table, burst trie, and the BST continue to assume uniform access costs and are, therefore, not tuned to perform well on modern cache architectures. As a consequence, these fundamental data structures are likely to incur excessive cache misses that will dominate their performance on current cache-oriented processors.

Although a cache is not accessible, a program can make better use of it—that is, become cache-conscious—by improving its *spatial* and *temporal* access locality through regular and predictable memory access patterns. Spatial locality is improved by accessing data items that are near to each other; temporal locality is improved by frequently reusing a small set of data items. To be efficient on current cache-oriented processors, data structures, therefore, need to be designed to combine computationally efficient algorithms with cache-conscious optimizations [Kowarschik and Weiß 2003; Meyer et al. 2003]. However, the best-known techniques for improving the cache efficiency of programs have limited support or effectiveness on large and dynamic string data structures, such as the standard hash table and burst trie.

In this article, we address the problem of cache-inefficiency in fundamental string data structures, by reconsidering the principles of their design. We show that simple cache-oriented redesigns of the string data structures can lead to substantial improvements in performance without compromising their dynamic characteristics. In particular, we redesign the current best alternatives—the hash table, burst trie, and BST—by exploring novel cache-conscious pointer elimination techniques that challenge the principles of the RAM model by minimizing random access to memory, at the expense of increasing the number of instructions executed. However, as the speed of processors continues to race ahead of main memory, an increase in the computational cost of a program will often be compensated if it reduces the number of cache misses incurred [Meyer et al. 2003].

The use of pointers in dynamic string data structures are the fundamental cause of cache-inefficiency, as they can lead to random memory accesses. Rao and Ross [2000] noted the importance of pointer elimination in improving the performance of integer-based tree data structures. However, pointer elimination has yet to be presented in literature as a viable and effective means at improving the cache-efficiency of dynamic string data structures. For such situations, researches have proposed techniques such as software prefetch [Karlsson et al. 2000; Callahan et al. 1991], custom memory allocation [Berger et al. 2002; Truong et al. 1998], and different kinds of pointer cache

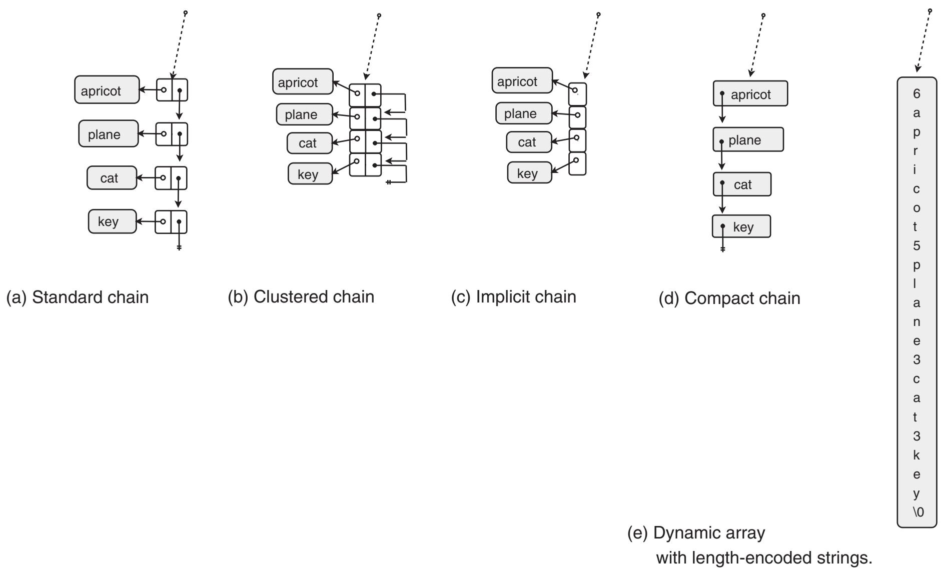

apricot
plane
cat
key
(a) Standard chain
(b) Clustered chain
(c) Implicit chain
(d) Compact chain
6 a p r i c o t 5 p l a n e 3 c a t 3 k e y \0
(e) Dynamic array with length-encoded strings.

Fig. 2. A standard chain is the typical implementation of a linked list. A clustered chain physically groups the nodes of a list together, to improve spatial access locality. However, no pointers are eliminated [Chilimbi et al. 1999]. An implicit clustered chain eliminates next-node pointers, allowing homogeneous nodes to be accessed via arithmetic offsets [Chilimbi 1999]. However, strings remain randomly allocated in memory. A compact chain, in contrast, eliminates string pointers by storing the string directly within the node. The dynamic array combines clustering and pointer elimination by storing the strings in a contiguous resizable space.

and prefetchers [Yang et al. 2004; Collins et al. 2002; Roth and Sohi 1999]. Chilimbi et al. [1999] suggested packing the homogeneous nodes of a BST into contiguous blocks of memory. Rubin et al. [1999] proposed a similar idea for linked lists. These techniques do not eliminate pointers, but improve spatial access locality by grouping nodes that are likely to be accessed contemporaneously.

However, to our knowledge, there has been no practical examination of the impact of these techniques on dynamic pointer-intensive string data structures, nor is there support for these techniques on current platforms. Chilimbi et al. [1999] note the applicability of their methods to chained hashing, but not with move-to-front on access. This is likely to be a limiting factor, as move-to-front can itself be an effective cost-adaptive reordering scheme [Zobel et al. 2001]. Nor is it clear that such methods will be of benefit in the environments that we are concerned with, where the volume of data being managed may be hundreds of times larger than cache.

Our first technique, which is called a *compact chain*, is a straightforward way of improving the spatial locality of string data structures that use *standard* chains as substructures; by storing each string directly in its node, as opposed to using a string pointer. This approach saves up to 12 bytes of space per string (assuming a typical current system architecture) and eliminates a potential cache-miss at each node access, at no cost other than the difficulty of coding variable-sized nodes. Compact chains can demonstrate the value of eliminating a pointer traversal per node, in contrast to retaining pointers and simply clustering the nodes of a linked list contiguously in memory [Chilimbi et al. 1999; Rubin et al. 1999].

Our second technique explores a more drastic measure: to eliminate the chain altogether and store the sequence of strings in a contiguous array that is dynamically resized as strings are inserted. With this arrangement, multiple strings are likely to be fetched simultaneously and subsequent fetches have high spatial locality. We illustrate the structural differences between standard, clustered, compact, and array-based linked lists in Figure 2. While our proposal, which consists of the elementary step of dumping every string in a list into a contiguous array, might be seen as simplistic, it nonetheless is attractive in the context of current architectures.

A potential disadvantage of using arrays is that, whenever a string is inserted, the array must be dynamically resized. As a consequence, a dynamic array can be computationally expensive to access. However, it is also cache-efficient, which can make the array method dramatically faster to access in practice, while simultaneously reducing space due to pointer elimination.

We experimentally evaluate the effectiveness of our pointer-elimination techniques, using sets of up to 178 million strings. We apply our compact chains to the standard hash table, burst trie, BST, splay tree, and red-black tree, forming compact variants. We then replace the chains of the hash table, burst trie, and BST using dynamic arrays, creating new cache-conscious array representations called the array hash, array burst trie, and array BST, respectively.

On practical machines with reasonable choices for parameters, our experiments show that for all array-based data structures where the array size is bounded, as in the BST and the burst trie, the analytical costs are equivalent to their standard and compact-chain representations. For all the data structures, the expected cache costs of their array representations are superior to their standard and compact equivalents, even for the array hash, where on update, the asymptotic costs are greater than its chaining equivalents, yet in practice, greater efficiency is observed.

Clustering is arguably one of the best methods available at improving the cache-efficiency of pointer-intensive data structures [Chilimbi et al. 1999]. However, to the best of our knowledge, there has yet to be a practical evaluation of its effectiveness on string data structures. We experimentally compare our compact and array data structures against a clustered hash table, burst trie, and BST. Our experiments measure the time, space, and cache misses incurred by our compact, clustered, and array data structures for the task of inserting and searching large sets of strings. Our baseline consists of a set of current state-of-the-art (standard-chained) data structures: a BST, a TST, a splay tree, a red-black tree, a hash table, a burst trie, the adaptive trie [Acharya et al. 1999], and the Judy data structure. Judy is a trie-based hybrid data structure, composed from a set of existing data structures [Baskins 2004; Silverstein 2002; Hewlett-Packard 2001].

Our results show that, in an architecture with cache, our array data structures can yield startling improvements over their standard, compact, and clustered chained variants. A search for around 28 million unique strings on a standard hash table that contains these strings, for example, can require over 2,300 seconds—using 2^{15} slots—to complete while the table occupies almost 1GB of memory. The equivalent array hash table, however, required less than 80 seconds to search while using less than a third of the space, that is, simultaneously saving around 97% in time and around 70% in space. Although this is an artificial case—we would typically allocate more slots in practice—it highlights that random access to memory is highly inefficient, and that the array hash can scale well in situations where the number of keys is not known in advance.

Similar savings were obtained for insertion. Hence, despite the high computational costs of growing arrays, our results demonstrate that cache efficiency more than compensates. The array burst trie demonstrated similar improvements, being up to 74% faster to search and insert strings, while maintaining a consistent and simultaneous reduction in space, of up to 80%. The array BST also displayed similar behaviors, being up to 30% faster to build and search using 28 million unique strings, while requiring less than 50% of the space used by the equivalent standard BST. The splay tree, red-black tree, TST, and Judy trie were in most cases, slower to access than our array BST. The clustered data structures were found to be considerably slower than the standard data structures, and were only effective at exploiting cache when there is no skew in the data distribution. Our array-based data structures, however, were superior.

These results are an illustration of the importance of considering cache in algorithm design. The standard chain hash table, burst trie, and the BST have previously been shown to be among the most efficient structures for managing strings [Heinz et al. 2002; Williams et al. 2001; Zobel et al. 2001], but we have greatly reduced their total space consumption while simultaneously reducing access time.

2. BACKGROUND

For many applications, it is necessary to efficiently maintain large volumes of data in memory. In this section, we briefly discuss the principal in-memory data structures for strings and then review cache considerations on current architectures.

Linked lists and arrays

The simplest dynamic data structure is the linked list [Newell and Tonge 1960]. It consists of a chain of nodes, where each stores a pointer to a data item, along with a pointer to the next node in the chain. Linked lists require on average and at worst O(n) comparisons per search, where n is the number of strings.

Strings can also be stored sequentially in a fixed-sized array, with an access cost of O(n) if the strings are not sorted, or an insertion cost of O(n) if the strings are sorted. Hansen [1981] describes several schemes for improving access costs of strings stored in fixed-sized nodes (or arrays). These schemes are designed to better utilize the space in a node, at the cost of some search performance. With the "middle gap" scheme, for example, strings of fixed-length are sorted and partitioned into several groups separated by unused space. When a new string is inserted into a group, the existing strings must be moved to maintain sort order (for binary search).

Trees

Trees combine the search characteristics of arrays with the dynamic characteristics of lists. In a standard BST, each node has a left and right child-node pointer, to subtrees of strings that are respectively lexicographically less than or greater than the node's string. The standard BST has an expected O($\log n$) search cost but an $O(n)$ worst case. To ensure good performance regardless of the distribution of strings, it is necessary to balance or reorganize the tree. However, the cost of reorganization can outweigh the gains in the average case, as frequently accessed strings may be moved away from the root, and reorganization can be more expensive than search.

The splay tree [Sleator and Tarjan 1985] is an example of a self-adjusting BST, which achieves good worst-case behavior through *splaying*, a technique that moves the most recently accessed node to the root of the tree. The randomized search tree [Martinez and Roura 1998] is also self-adjusting, where a heap order is maintained on nodes. When a node's weight (a random number) is greater than its parent, it is moved up the tree until the heap order is restored. Another randomized structure is a skip list [Pugh 1990], which uses a hierarchy of sorted lists above a primary sorted list, giving tree-like expected behavior, but with relatively poor efficiency for strings.

The splay tree, the randomized BST, and the red-black tree were experimentally compared against the standard BST for the common task of vocabulary accumulation [Williams et al. 2001]. For this data, the splay tree was never faster than a standard BST and was, on average, 25% slower. Furthermore, splaying on every access proved ineffective for skew data, with words such as "the"—the most frequent word in plain text—sometimes found deep in the tree. Similarly, the randomized BST was only 3% faster than the splay tree. The red-black was slower for skew data than the splay tree, with only a slight improvement when little or no skew was present. Thus, while the standard BST's worst case might prohibit its use in practice, in the average case it provides a reasonable benchmark for efficiency.

Tries

A trie is a multiway tree structure that stores sets of strings by successively partitioning them [de la Briandais 1959; Fredkin 1960; Jacquet and Szpankowski 1991]. Tries have two properties that cannot be easily imposed on data structures based on binary search. First, strings are clustered by shared prefix. Second, there is an absence of (or great reduction in the number of) string comparisons—an important characteristic, as a major obstacle to efficiency is excessive numbers of string comparisons [Williams et al. 2001]. Tries can be rapidly traversed and offer good worst-case performance, without the overhead of balancing [Szpankowski 1991].

Although fast, tries are space-intensive, which becomes a serious problem in practice [Comer 1979; Heinz et al. 2002; McCreight 1976]. There are two approaches to reducing the space of a trie. The first is to reduce the size of trie nodes by changing their structural representation. The second is to reduce the number of nodes. A simple implementation of a trie node is an array of pointers, one for each letter of the alphabet [Fredkin 1960]. A string of length k is stored as a chain of k nodes. Access to these nodes can be rapid, even for long chains. However, with tasks such as vocabulary accumulation, for trie nodes at or near the leaves the majority of pointers are likely to be unused.

A more space-efficient representation is the de la Briandais or list trie [de la Briandais 1959], which structures a trie node as a linked list that stores only used pointers. Trie nodes can also be structured as BSTs, giving a ternary search trie [Bentley and Sedgewick 1997; Clement et al. 1998]. These forms of trie nodes were shown to have logarithmic access costs but with different constants [Clement et al. 2001]. The array trie is the fastest but requires the most space. The TST is more compact but is more expensive to access, while the list trie is compact but very slow [Severance 1974].

The array trie can be modified to omit chains of nodes that descend into a single leaf, forming a compact trie [Sussenguth 1963]. Similarly, the Patricia trie [Morrison 1968; Gonnet and Baeza-Yates 1991] omits all single descendant nodes, not just those that descend directly into leaves, saving even more space but remaining larger than a TST [Sedgewick 1998; Heinz et al. 2002].

However, these techniques are insufficiently effective for large and dynamic sets of strings. In the burst-trie [Heinz et al. 2002], dynamic containers are used to store small sets of strings that share a common prefix. Containers—such as linked lists with move-to-front—can reduce the number of trie nodes by up to 80%, at little cost in speed. A similar approach [Ramesh et al. 1989] takes a fully built array trie and collapses selective nodes into static containers. The burst-trie, in contrast, is built dynamically. It starts as a single container that is populated with strings until full. Once full, the container is burst, forming smaller containers, one for each letter of the alphabet, that are parented by a new node. Strings are distributed according to their lead character, which can then be removed.

The burst-trie was experimentally compared against several data structures [Heinz et al. 2002]: a compact trie, a TST, a standard BST, a splay tree, and a standard-chain hash table, for the task of vocabulary accumulation. The burst-trie was consistently faster and more compact than the trees and tries, and could approach the efficiency of the hash table while maintaining sorted access to strings. Similar comparisons are made in the experiments reported later in this article.

Hash Tables

A hash table is a data structure that scatters keys among a set of lists that are anchored over an array of slots. In-memory hash tables are a basic building block of programming, used to manage temporary data in scales ranging from a few items to gigabytes. To store a string in a hash table, a hash function is used to generate a slot number. The string is then placed in the slot; if multiple strings hash to the same location, some form of collision resolution is needed. (It is theoretically impossible to find a hash function that can uniquely distinguish keys that are not known in advance [Knuth 1998].) There are two decisions a programmer must make: choice of hash function and choice of collision-resolution method. A hash function should be from a universal class [Sarwate 1980], so that the keys are distributed as well as possible, and should be efficient. The fastest hash function for strings that is thought to be universal is the bitwise method of Ramakrishna and Zobel [1997]; hash functions commonly described in textbooks are based on modulo and repeated multiplication and division, which are slow and are not effective at distributing strings.

Since the origin of hashing—proposed by Luhn in 1953 [Knuth 1998]—many methods have been proposed for resolving collisions. The best known are separate or standard chaining and open addressing. In chaining, also proposed by Luhn, linked lists are used to resolve collisions, with one list per slot. Thus, there is no limit on the load average, that is, the ratio of items to slots.

Open addressing, proposed by Peterson [1957], is a class of methods where items are stored directly in the table and collisions are resolved by searching for another vacant slot. However, as the load average approaches 1, the performance of open addressing drastically declines. These open addressing schemes are surveyed and analyzed by Munro and Celis [1986] and were found not be suitable for applications that manage moderate to large sets of strings. Alternative techniques include coalesced chaining [Vitter 1983], and combined chaining and open-addressing, known as pseudochaining [Halatsis and Philokyprou 1978]. However, it is not clear that the benefits of these approaches are justified by the difficulties they present.

In contrast, standard chaining is fast and flexible. For sets of strings with a skew distribution, such as occurrences of words in text, a standard-chain hash table, coupled with an effective, efficient hash function, is faster and more compact than sorted data structures such as tries and variants of BST [Williams et al. 2001]. Using move-to-front in the individual chains [Knuth 1998; Zobel et al. 2001], the load average can reach dozens of strings per slot without significant impact on access speed, as the likelihood of having to inspect more than the first string in each slot is low. Thus, a standard-chain hash table has clear advantages over open-addressing alternatives, whose performance rapidly degrades as the load average approaches 1 and which cannot easily be resized.

Moreover, in principle, there is no reason why a chained hash table could not be managed with methods designed for disk, such as linear hashing [Larson 1982] or extensible hashing [Rathi et al. 1991], which allow an on-disk hash table to grow and shrink gracefully. The primary disadvantage of hashing is that strings are randomly distributed among slots, and is thus unsuited to applications in which strings need to be available in sorted order.

Current Techniques for Exploiting Cache

As the cost of accessing memory continues to increase relative to the speed of the CPU [Patterson et al. 1997], arithmetic operations alone are no longer adequate for describing the computational cost of a data structure [Kowarschik and Weiß 2003]. For programs to remain efficient in practice, measures must be taken to exploit cache. A cache is a small high-speed memory that resides between the CPU and main memory. Its purpose is to store frequently accessed data, allowing it to be accessed rapidly on future requests [Hennessy and Patterson 2003; Handy 1998]. A cache is divided into fixed-sized blocks known as cache lines. A cache-hit occurs when the required data is found in a cache line; otherwise, it is a cache-miss, which incurs the cost of accessing main memory.

Cache, however, is typically not under the control of the programmer [Rahman 2002; Hinton et al. 2001]. Nonetheless, programs can be designed to make good use of cache by improving their locality of access. Locality of access can be improved by increasing the rate at which a program reuses recently accessed data (temporal locality), and by accessing data that is near by (spatial locality), thus utilizing cachelines and hardware prefetch (described later in the text) to reduce cache misses and consequent CPU stalls [Lebeck 1999]. The importance of minimizing TLB-misses has also been noted [Agarwal 1996; Moret and Shapiro 1994; Romer et al. 1995; Rahman et al. 2001], which is achieved through improved temporal locality and by reducing the amount of memory used by a program.

Code transformations and optimizations at compile time are the simplest means for improving access locality. These involve changing the order in which instructions in a program are executed, in particular, the iterations of a program loop, to improve cache-line utilization and to apply instruction scheduling to optimize the execution of instructions [Kerns and Eggers 1993]. Some common techniques include loop unrolling, loop skewing, and loop fusion. Detailed descriptions of these techniques can be found in compiler-based literature [Beyls and D'Hollander 2006; Allen and Kennedy 2001; Muchnick 1997; Leung and Zahorjan 1995; Manjikian and Abdelrahman 1995; Bacon et al. 1994; Carr et al. 1994]. However, these compile-time optimizations are only effective at reducing capacity misses and instruction costs. They cannot address compulsory misses [Kowarschik and Weiß 2003] and are generally no better at reducing conflict misses [Rivera and Tseng 1998]. Compulsory, capacity, and conflict misses are classifications of cache misses defined by Hill and Smith [1989].

Calder et al. [1998] introduced a compile-time data placement technique that relocates stack and global variables, as well as heap objects to improve the data access locality of programs. Although effective for stack and global variables, there was little improvement for heap objects. In addition, a training run (a program profile) is required to determine how to relocate variables to achieve good performance. This has obvious limitations on dynamic data structures that rely on heap-allocated storage and have access patterns that are typically not known in advance. Hence, compile-time techniques can do little to improve the cache performance of dynamic pointer-intensive data structures, leaving much of the cache optimization effort to the programmer [Granston and Wijshoff 1993; Loshin 1998; Burger et al. 1996].

Data prefetching is a common technique used to reduce compulsory misses and improve spatial locality by fetching data into cache before it is needed [VanderWiel and Lilja 2000]. Hardware prefetchers, such as stride prefetching [Smith 1982], work independently from the processor and require no intervention from the programmer. They typically intercept memory requests and use simple arithmetic (and often a pool of recent addresses) to predict future data accesses [Kowarschik and Weiß 2003; Intel 2007].

Although hardware prefetchers are effective at reducing compulsory misses, random memory accesses are impossible to predict. Hence, for pointer-intensive data structures such as the standard hash table, a hardware prefetcher can pollute the cache with unwanted data. More sophisticated prefetchers, such as the Markov predictor [Joseph and Grunwald 1997] or dependence-based prefetching [Roth et al. 1998], can improve prefetch precision but are often more complex and expensive [Baer and Chen 1991; 1995; Fu et al. 1992].

Hardware prefetchers work well with programs that exhibit regular or stride access patterns; arrays are prime candidates. Unfortunately, most data structures for strings use linked lists as substructures. Traversing a linked list can lead to inefficient use of cache, because nodes and their strings can be scattered in main memory, resulting in poor access locality. Furthermore, nodes are accessed via pointers which can hinder the effectiveness of hardware prefetch, due to pointer chasing [VanderWiel and Lilja 2000].

For such situations, software prefetchers [Callahan et al. 1991; Karlsson et al. 2000] have been proposed, the simplest being the use of prefetch instructions such as greedy prefetching [Luk and Mowry 1996]. These prefetch instructions are manually inserted into code by programmers or added by compilers [Mowry 1995; Lipasti et al. 1995]. In contrast to hardware prefetchers, software prefetchers have some CPU cost and often require intervention from the programmer. Roth and Sohi [1999] introduced jump-pointers, a framework of pointers that connect nonadjacent nodes in a linked list, to be used by a software prefetcher to overcome pointer-chasing. However, to be effective, an appropriate jump-point interval must be found which is nontrivial; Roth and Sohi [1999] do not describe how to adapt this interval to an application. Consequently, once installed, jump-pointers do not adapt to changes in access patterns, which can lead to inefficiency [Yang et al. 2004]. In addition, jump pointers cannot prefetch the sequence of nodes between jump intervals.

Karlsson et al. [2000] extends jump pointers by introducing a prefetch array. Prefetch arrays consists of a number of jump pointers to nodes that are located consecutively in memory. These pointers can be used to prefetch several nodes at once—by using hardware or software prefetchers. Prefetch arrays can also be used to prefetch nodes in-between jump pointers and can be effective in applications where the traversal path is not known in advance. Prefetch arrays, however, require more space than jump pointers and remain ineffective for dynamic pointer-intensive data structures such as trees with high branching factors.

Speculative execution uses a separate processor (or threads) to run ahead of the main program during a cache-miss, to dereference pointers and cache data before it is requested [Dundas and Mudge 1997; Luk 2001]. A pointer cache [Collins et al. 2002] can be useful here, as it stores frequent pointer transitions. However, overall effectiveness is poor when applied to dynamic pointer-intensive data structures, as, to be effective, knowledge of access patterns is needed, and there must be sufficient delays between cache misses and node processing.

Stoutchinin et al. [2001] observed that there tends to be a regularity in the allocation of nodes in a linked list, and thereby proposed a compiler-based software prefetch method, called *speculative induction pointer prefetching*, which modifies a compiler to detect a linked list scanning loop. Once detected, the compiler inserts code that computes a stride offset for induction pointers, which are then used to speculatively prefetch nodes whose address are computed relative to the induction pointers. This technique works well on programs that exhibit stride access patterns, but not with those that exhibit irregular memory access patterns.

There are also proposals for combining hardware and software prefetchers that attempt to overcome pointer chasing in pointer-intensive applications. Yang et al. [2004], for example, proposed programmable hardware prefetch engines, with similar techniques by Hughes and Adve [2005] for multiprocessor systems. Guided region prefetching is another example where the compiler generates prefetch hints that guide hardware prefetchers [Wang et al. 2003].

The techniques discussed earlier are concerned with the consequences of cache misses, as opposed to the cause, which is poor access locality. Chilimbi et al. [1999] showed that careful layout and relocation of nodes in pointer-intensive data structures can improve access locality without changing program semantics. Two cache-conscious relocation techniques were introduced: *clustering* and *coloring*. Clustering attempts to pack nodes that are likely to be accessed contemporaneously into blocks of memory that are sized to match the cache line. Hence, clustering is a cache-conscious heap allocator similar to *malloc*, expect that it requires the programmer to supply a pointer to an existing node that is expected to access the new node, such as a parent node. An incorrect choice of pointer will not affect program correctness, but it can affect performance.

Coloring maps contemporaneously accessed nodes to nonconflicting regions of cache, that is, it segregates nodes based on access frequency obtained from program profiles. In some cases, such as with a binary search tree, access patterns can be derived from structural topology (i.e., nodes near the root will be accessed more frequently than those near the leaves). Coloring, however, is only compatible with static tree-like data structures and incorrect usage can affect program correctness [Chilimbi 1999].

Clustering and coloring are currently only compatible with homogeneous objects, such as the fixed-sized nodes of a binary search tree; they do not support allocation or relocation of variable-length objects, such as strings. In addition, these techniques assume that the size of a node in bytes, is less than half the cache-line size. In cases where nodes are larger, their internal fields can be reorganized by separating frequently accessed fields into smaller segments that can fit into a cache line. However, this technique has limited application, as it can affect program correctness [Chilimbi 1999; Chilimbi et al. 2000].

Another important characteristic of clustering is that nodes are grouped together to improve spatial access locality without eliminating their pointers. Hence, a clustered pointer-intensive data structure can still make poor use of hardware prefetch, due to pointer chasing. Chilimbi [1999] suggests eliminating the next-node pointers of a linked list and accessing nodes using arithmetic offsets, forming an *implicit* clustered chain. However, this technique assumes homogeneous nodes and involves nontrivial programmer intervention, and was thus not pursued for coloring and clustering.

Chilimbi et al. [1999] note the applicability of their methods to chained hashing, but not with move-to-front on access. This is likely to be a limiting factor, as move-to-front is itself an effective cost-adaptive reordering scheme. Furthermore, there is currently no implementation support for clustering or coloring on current platforms. Nonetheless, implementing a clustered linked list is straightforward: The nodes of a linked list are simply allocated contiguously in memory, in order of occurrence. A clustered list is likely to make better use of cache during search, but updating a single array of nodes (that maintain next-node pointers) can be expensive. Hence, Chilimbi [1999] suggests grouping nodes into fixed-sized memory blocks that are sized to match the cache line. This forms a chain of blocks that permit efficient update, but at some cost in space and cache efficiency (the individual blocks can be scattered in memory).

Virtual cache lines, proposed by Rubin et al. [1999], is a similar technique to clustering. In this approach, the nodes of a linked list are stored within fixed-sized blocks of memory that match the size of the cache line. Access to a block will prefetch the next set of nodes in the list, which improves spatial access locality. However, the virtual cache-line technique is currently only compatible for linked lists. Its effectiveness on tree-based data structures such as the BST is unknown. In addition, nodes are assumed to be of fixed size and no pointers are eliminated; hence, variable-sized objects, such as strings, remain randomly allocated in memory, which can make inefficient use of cache.

Another similar technique to clustering is the use of hash buckets to improve the cache-efficiency of the hash join algorithm in SQL [Graefe et al. 1998]. In this approach, the two chaining hash tables that are used to perform the join are constructed with nodes that are sized to match the cache line. Hence, initial access to a node in a linked list will prefetch an entire cache line of fixed-length keys (integers), which is more efficient than accessing a single key per node.

Frias et al. [2006] proposed a cache-conscious STL list for the C++ library, which employs a double-linked list of buckets. Each bucket contains a small array of elements, pointers to next buckets and other housekeeping data. The buckets employ different algorithms to manage their elements, ranging from contiguous allocation, allocation with gaps or the use of internal lists, along with algorithms that rearrange buckets and to support efficient iteration. The authors report a considerable performance boost over conventional STL lists (though the memory efficiency of their approach, taking into account operating system overheads, is unclear for a large number of insertions).

The approach used to implement cache-conscious STL lists is akin to the general techniques highlighted earlier, in particular node clustering, virtual cache lines, and hash buckets. As such, cache-conscious STL lists assume that elements in a bucket are homogeneous; either 32-bit or 64-bit integers (there is no consideration for variable-length string keys). As a result, to support variable-length string keys, buckets will need to maintain string pointers; and as is the case for node clustering, this action will likely hinder performance when applied to dynamic string data structures, such as the chained hash table. We demonstrate this in later experiments, where we employ node clustering (i.e., a list of buckets) in several dynamic string data structures.

Truong et al. [1998] introduced a field reorganization and interleaving technique that groups the homogeneous nodes of a linked list into fixed-sized memory blocks called *arenas*, using a dynamic cache-conscious heap allocator called *ialloc*. This approach clusters the fields of nodes according to their expected frequency of access, which is determined by the programmer. Hence, a cache line will store frequently accessed components from a number of nodes, which can improve access locality. However, *ialloc* does not support variable-sized fields, such as arrays of integers or a string; such fields must remain randomly allocated in memory. In addition, pointer chasing remains an issue and the process of interleaving nodes can affect program correctness. Moreover, *ialloc* is not particularly effective for pointer-intensive data structures such as the standard-chain hash table. In this example, *ialloc* will interleave nodes from different slots which will increase the number of cache misses incurred, as cache lines will be polluted with unwanted data [Askitis 2007].

Alternative data relocation techniques include identifying frequently repeated sequences of consecutive data references in programs, and clustering them accordingly to improve the use of cache [Chilimbi and Shaham 2006]. Similar techniques involve the use of memory pools that cluster objects of the same data type to improve access locality [Lattner and Adve 2005; Zhao et al. 2005].

Berger et al. [2002] tested eight high-performance custom memory allocators and found that six were no better and often led to worse cache utilization in pointer-intensive programs, than a high-performance general-purpose allocator, such as *malloc*. The remaining two, which involved the use of memory pools, led to slight improvements in access times, but at some cost in space. Open-address hash tables have also been investigated in the context of cache [Heileman and Luo 2005], but as noted previously, open-address hash tables are not efficient or practical solutions for managing strings.

A study by [Badawy et al. 2001, 2004] tested the effectiveness of combining software prefetch on data that was cache-consciously allocated, and found software prefetching to be of little value compared to the performance gained through careful layout. The authors concluded that careful layout of data through clustering often outperformed software prefetching (such as jump pointers), particularly on machines with limited memory bandwidth. Hence, allocating or relocating data in a cache-conscious manner has been shown to be the best current way of exploiting cache in pointer-intensive programs [Hallberg et al. 2003].

A software cache can also be used in place of a software prefetcher [Aggarwal 2002], which caches recently stored (homogeneous) data items in pointer-intensive data structures, in an attempt to reduce their access cost. Access locality can also be improved by a combination of split caches, victim caches, and stream buffers, as discussed by Naz et al. [2004]. In addition, LaMarca and Ladner [1996] studied the performance of an implicit heap and showed how padding techniques that align fixed-sized heap elements to cache lines, can yield high reductions in cache misses.

Acharya et al. [1999] proposed data layout schemes that addressed—in the context of cache—the unused pointers in sparse tries [Knuth 1998]. The authors dynamically change the representation of trie nodes for very large alphabets, into either an associative array, a bounded-height B-tree, or a chained hash table, to form an adaptive trie. The adaptive trie was compared against the the ternary search trie [Bentley and Sedgewick 1997] for the task of searching a small set of strings, and was found to be faster than the TST, due to better use of cache. Crescenzi et al. [2003] also compared the adaptive trie against other string data structures, such as the Patricia trie, an open-address (linear probing) hash table, a BST, an AVL tree, and the TST, to determine their efficiency for searching small string datasets with and without skew. The adaptive trie was shown to operate efficiently with no skew in the data distribution, but was consistently slower than the TST, BST, and the open-address hash table with skew. In addition, the authors note that the adaptive trie can be more space-intensive than the TST.

Ghoting et al. [2006] studied the cache performance of the frequent-pattern mining algorithm—FPGrowth—on current processors, which uses an annotated prefix tree (or FP-tree) to compactly represent a transaction dataset [Han et al. 2000]. However, the nodes of an FP-tree are likely to be scattered in memory, which is not cache-efficient. The authors proposed replacing the FP-tree with a static cache-conscious prefix tree that stores the nodes contiguously in memory. First, a standard FP-tree is built and a single fixed-sized block of memory is allocated, sized to fit the entire tree. The FP-tree is then traversed in a depth-first order, copying its nodes sequentially into the block of memory. As a result, spatial access locality can be improved. A similar copying technique to eliminate string pointers for string sorting was also proposed by Sinha et al. [2006], who demonstrated that doing so can halve the cost of sorting a large sets of strings, due to improved spatial access locality.

Rao and Ross [1999] proposed a new indexing technique called a *cache-sensitive search tree* (or CSS-tree), which stores a directory structure on top of an existing sorted array of homogeneous keys, such as integers. The directory structure is a BST where nodes are sized to match the cache line. The key advantage offered by a CSS-tree, compared to a binary search using a single sorted array, is that the binary search is localized within nodes that are contiguously allocated, which can exploit cache. However, the CSS-tree is a static structure that is build upon a sorted array. It cannot handle updates efficiently and is not designed for variable-length strings. To support updates using fixed-length keys, the authors developed a variant called a *cache-sensitive B-tree* (CSB-tree) [Rao and Ross 2000]. The CSB-tree places all the child nodes of a given node contiguously in memory, to eliminate child node pointers. Hence, a parent node need only retain the first pointer to its child node. The rest of its children can be found through arithmetic offsets, which improves spatial access locality.

Torp et al. [1998] proposed a similar cache-efficient B-tree, called a *pointerless insertion tree* (PLI-tree), which eliminates all pointers in nodes. The PLI-tree allocates homogeneous keys in a specific order, such that child nodes can be found through arithmetic calculations. Although compact and fast, the data structure cannot handle random insertions or variable-length objects such as strings; all insertions must be in incremental order which is potentially useful for append-only operations, such as maintaining logs or time-stamped data, for example.

Cache-oblivious data structures are designed to perform well on all levels of the memory hierarchy (including disk) without prior knowledge of the size and characteristics of each level [Frigo et al. 1999; Kumar 2003]. Brodal and Fagerberg [2006], for example, theoretically investigated a static cache-oblivious string dictionary. However, their study shows no actual performance measures against well-known data structures. Similarly, a dynamic cache-oblivious B-tree [Bender et al. 2000] has been described, but with no analysis of actual performance. This is not uncommon, as there are other studies demonstrating that cache-oblivious structures can almost match the performance of optimized data structures [Bender et al. 2003], but only from a theoretical perspective. Cache-oblivious matrix algorithms have been experimentally compared against their cache-conscious variants, which, however, were found to be superior [Yotov et al. 2007]. Another example includes the cache-oblivious priority queue [Arge et al. 2002].

The cache-oblivious dynamic dictionary [Bender et al. 2004] has been compared to a conventional B-tree, but on a simulated memory hierarchy. These studies assume a uniform distribution in data and operations, which is typically not observed in practice [Bender et al. 2002]. Bender et al. [2006] have also theoretically proposed a cache-oblivious string B-tree that can handle unbounded-length strings, along with a cache-oblivious streaming B-tree that is theoretically designed to support efficient random insertion and range search of fixed-length keys [Bender et al. 2007].

Recent studies have evaluated the practicality of these data structures [Ladner et al. 2002; Brodal et al. 2002; Arge et al. 2005] and reported potentially superior performance compared to conventional data structures for fixed-length keys, but not when compared to those that have been tuned to a specific memory hierarchy [Arge et al. 2005; Rahman et al. 2001]. The data structures that we explore in this article are not cache-oblivious, as they have been designed specifically to exploit the cache between main memory and the CPU, and reside solely within (volatile) main memory. For disk-based alternatives, we present a detailed survey of existing disk-resident string data structures, including the string B-tree [Ferragina and Grossi 1999], along with the proposal and evaluation of the B-trie, an efficient disk-resident string data structure [Askitis and Zobel 2008].

3. REDESIGNING STRING DATA STRUCTURES TO EXPLOIT CACHE

In this section, we explore cache-conscious pointer elimination techniques—the compact chain and the dynamic array—and describe how to redesign the current best string data structures, with the aim of creating cache-conscious alternatives that can potentially yield substantial gains in performance, without compromising dynamic characteristics. We assume that strings are sequences of 8-bit bytes, that a character such as null is available as a terminator, and a 32-bit CPU and memory address architecture is used.

Clustered Chain

In a typical implementation of a standard chained data structure, every node accessed incurs at least two pointer traversals: one to reach the node and the other to reach its string [Esakov and Weiss 1989; Horton 2006]. As nodes and their strings are likely to be scattered across main memory, access to a node can incur up to two cache misses. Clustering is a well-known technique that improves the cache-efficiency of a data structure by storing its nodes into a contiguous block or blocks of memory, to improve spatial access locality [Chilimbi et al. 1999].

However, pointers are not eliminated, which is a key distinction from the techniques that we present in this article. Another distinction is that clustering is only compatible with homogeneous nodes. Variable-sized objects, such as strings, must remain randomly allocated in memory, which can be inefficient. Another potential disadvantage of clustering is the effectiveness of move-to-front on access, which is likely to be a limiting factor as move-to-front is itself an effective cost-adaptive reordering scheme.

To maximize the effectiveness of clustering, the entire standard chain should be stored in a single contiguous block. However, adding new nodes requires the block to be resized, which can become computationally expensive; the physical location of the block can change in memory, and, as a consequence, every node in the block must be accessed and updated. Chilimbi [1999] suggests clustering nodes into fixed-sized blocks that are sized to match the cache line. Once a block is full, a new block is allocated and the node is inserted without resizing. This technique effectively creates a chain of blocks, similar to virtual cache lines [Rubin et al. 1999]. However, unless the blocks are randomly allocated—which reduces the effectiveness of clustering—adding a new block to a set of contiguously allocated blocks can be expensive to maintain.

Details regarding these implementation issues are not generally available in the literature [Chilimbi 1999]. As a result, we implement a clustered data structure by first building it as standard chain, then converting it by accessing each chain and storing its nodes contiguously in a block of memory. We describe how to convert a standard hash table, burst trie, and BST in the discussions that follow. We also consider implicit clustering, where nodes are clustered and accessed via arithmetic offsets, as we described earlier.

Compact Chain

A straightforward way of improving the cache efficiency of a chained data structure is to store each string in its node, rather than storing it in a separate location. This halves the number of random accesses and saves 12 bytes per string: 4 bytes for the string pointer and 8 bytes imposed by the operating system for string allocation. Each node consists of 4 initial bytes, containing a pointer to the next node, followed by the string itself. The total space overhead of a compact chain is, therefore, $12s$, where s is the number of strings stored. This is half the overhead of a standard chain, at no cost. We call this variant of chaining *compact*. However, this procedure requires that nodes themselves be of variable length, which, depending on the programming language, can be difficult to implement.

The cache advantages of compact chains are obvious: Each node will involve only a single memory access; spatial locality is improved, and the reduction in total size will improve the likelihood that the next node required is already cached. Nonetheless, compact nodes are randomly allocated and remain chained. As a consequence, they cannot maximize the use of cache. However, chains are computationally inexpensive to maintain. Only pointers need to be manipulated for update operations, such as inserting a new node or when move-to-front is initiated. In practice, there are no cases—apart from ease of implementation—where a standard chain would be preferable. Hence, compact chains can be readily applied to improve the cache and space efficiency of dynamic data structures for strings that use linked lists as substructures.

The Dynamic Array

Compact chains are simple and effective at improving the cache and space efficiency of existing chained data structures, but the random allocation of their nodes and use of pointers can hinder the effectiveness of hardware prefetchers. Also, cache-line utilization is not optimal, as 4 bytes of cache-line space is wasted (the next-node pointer) per node.

We, therefore, propose an alternative—to eliminate the chain altogether and store the strings in a contiguous dynamic array. Each array can be seen as a resizable bucket. The strings in a bucket are stored contiguously, which guarantees that access to the start of the bucket will automatically fetch the next 64, 128, or 256 (cache line) bytes of the bucket into cache. With no pointers to traverse, pointer chasing is eliminated and with contiguous storage of strings, hardware prefetching schemes are highly effective. Access locality is, therefore, maximized, creating a cache-conscious alternative to compact and standard chains.

Ghoting et al. [2006] showed that copying the nodes of an FP-tree into a fixed-sized memory block can greatly increase the spatial access locality of data-mining applications. Similarly, the use of copying to eliminate string pointers for string sorting was proposed by Sinha et al. [2006], who demonstrated that doing so can halve the cost of sorting a large set of strings. It is plausible that similar techniques can lead to substantial gains for dynamic chained data structures for strings.

While our proposal, which consists of the elementary step of dumping every string in a list into a contiguous resizable array, might be seen as simplistic, it is nonetheless attractive in the context of current architectures. Furthermore, by eliminating chains, the space overheads imposed by pointers and their subsequent memory allocation requests are eliminated. The space saved can be substantial for large sets of strings, prompting better usage of cache, as there is less competition for cache lines. Similarly, the TLB hit rate will also improve, as fewer pages of memory are required.

Traversing a Dynamic Array

The simplest way to traverse a bucket (a dynamic array) is to inspect it one character at a time, from beginning to end, until a match is found. Each string in a bucket must be null terminated, and a null character must follow the last string in a bucket to serve as the end-of-bucket flag. However, this approach can cause unnecessary cache misses when long strings are encountered; note that, in the great majority of cases, the string comparison in the matching process will fail on the first character. Instead, we have used a skipping approach that allows the search process to jump ahead to the start of the next string. With skipping, each string is preceded by its length; that is, they are length-encoded [Aho et al. 1974]. The length of each string is stored in either 1 or 2 bytes, with the lead bit used to indicate whether a 7-bit or 15-bit value is present. It is generally not sensible to store strings of more than 2^{15} characters, as mandatory string-processing tasks, such as hashing, will utterly dominate search costs.

Growing a Dynamic Array

We explored two methods of growing buckets: *exact-fit* and *paging*. In exact-fit, when a string is inserted, the bucket is resized by only as many bytes as required. This conserves memory but means that copying may be frequent. Resizing a bucket involves creating a new bucket that can fit the old bucket and the string required. The old bucket is then copied, character by character, into the new bucket; the new length-encoded string is appended followed by the end-of-bucket flag. The old bucket is then destroyed.

In paging, bucket sizes are multiples of 64 bytes, thus ensuring alignment with cache lines. As a special case, buckets are first created with 32 bytes, then grown to 64 bytes when they overflow, to reduce space wastage when the bucket contains only a few strings. When grown, the old bucket can be copied into the new, a word (4 bytes) at time. Paging should reduce both the copying and computational overhead of bucket growth, but it uses more memory. The value of 64 bytes was chosen to match the L1 cache-line size found in current Intel Pentium processors [Shanley 2004].

Limitations of Dynamic Arrays

In contexts where lists are stored and searched, dynamic arrays are an attractive option. A potential disadvantage, however, is that these arrays must be of variable size; whenever a new string is inserted in a bucket, it must be resized to accommodate the additional bytes. Hence, depending on the size of the array, the frequency of growth, and the growth scheme used, array growth may become computationally expensive. Another potential disadvantage is that move-to-front, which in the context of chaining requires only a few pointer assignments, involves copying large parts of the array.

A further potential disadvantage of using buckets is that such contiguous storage appears to eliminate a key advantage of nodes—namely, that they can contain multiple additional fields. However, sequences of fixed numbers of bytes can easily be interleaved with the strings, and these sequences can be used to store the fields. For example, a 4-byte data field can be stored before each string. The impact of these fields is likely to be much the same for all kinds of chaining data structures, of which we investigate later.

3.1. Cache-Conscious Hash Tables

To develop a cache-efficient hash table, we focus on how collisions are resolved. A collision occurs when more than one string is hashed to a particular slot. The best method for resolving collisions for dynamic sets of strings is chaining; simply append the new string at the end of the chain. With move-to-front on access, the chaining hash table or standard hash table, is the fastest and most compact data structure available when sorted access is not required [Zobel et al. 2001]. The use of standard chains can, however, result in poor use of cache, making this computationally efficient data structure vulnerable to severe performance penalties on current cache-oriented processors. We can potentially improve the cache-efficiency of the standard hash table by converting it into a clustered hash table. That is, we visit every slot and store its nodes, in order of occurrence, contiguously in main memory.

We propose to replace the standard chains of a hash table with compact chains, to yield a more cache-and space-efficient alternative that we call a *compact* hash table. Although compact chains eliminate string pointers, nodes remain scattered in main memory. We eliminate these chains by using dynamic arrays. This will create a hash table with a cache-conscious collision resolution scheme that we call an *array* hash. Note, however, that this method does not change the size of the hash table; we are not proposing extendible arrays [Rosenberg and Stockmeyer 1977]. The array hash is an attractive option for current cache-oriented architectures, as it can maximize cache usage while simultaneously reducing space. As discussed earlier, strings within dynamic arrays are expensive to reorder, so move-to-front on access is likely to become a performance bottleneck. The structural differences between a standard, compact, and array hash table are shown in Figure 3.

The Array Hash Table

The procedures taken to insert, retrieve, and delete strings in the standard or compact hash tables are straightforward. The string is first hashed (using the bitwise hash function [Ramakrishna and Zobel 1997], say) to acquire a slot. The slot is then accessed and its chain is traversed, checking each node until a match is found. When a match is found, the node is moved to the front of the chain through pointer manipulation. Alternatively, the node can be deleted. If the slot is empty, or no match is found, then the search fails, and the string can be inserted. In this case, the string is encapsulated in a standard or compact node, which is then added to the end of the chain. The steps taken to insert, search, and delete in an array hash are slightly more complex and are described in the following text.

To Search for a String

A search begins by first hashing the string (using the bitwise hash function) to acquire a slot. If the slot is assigned to a bucket, the search proceeds as outlined in Section 3: The first length-encoded string is compared, character by character. On mismatch, the

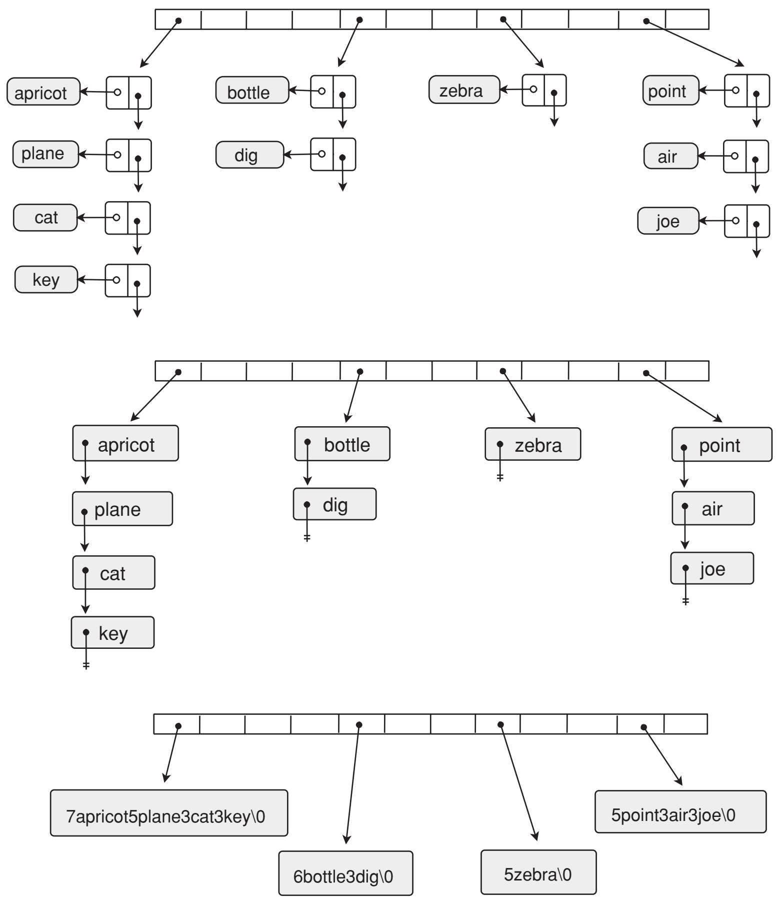


Fig. 3. The application of compact chains and dynamic arrays to the standard chain hash table. In a compact hash table, string pointers are eliminated, allowing the strings themselves to be represented as nodes. In the array hash table, all pointers—apart from the slot pointers—are eliminated. Strings are stored contiguously in main memory to make good use of both cache and hardware prefetch. Strings are appended in order of occurrence and are length-encoded to permit word-skipping, which is cache-efficient.

next length-encoded string is accessed, and so forth, until a match is found or the end-of-bucket flag is seen, on which, the search fails. On a match, the string can be moved to the start of the array. However, due to the potentially high computational costs involved, move-to-front is optional.

To Insert a String

We select a slot by hashing the string using the bitwise hash function. If the slot is empty, then a new bucket is created and assigned. The bucket is sized according to the growth policy used. The string is then length-encoded and copied into the bucket, followed by the end-of-bucket flag. Otherwise, if the slot has a bucket assigned, we search for the string as described earlier. On search failure, the bucket is grown according the growth policy used, and the string is length-encoded and appended (more details on array resizing can be found in the subsection before Section 3.1).

To Delete a String

We select a slot by hashing the string using the bitwise hash function. Assuming that the slot is not empty, we search for the string as described earlier. If found, we create a new bucket that is sized to match the space required by the old bucket, minus the length of the string to delete. If deleting the string results in an empty bucket, then the slot pointer is nulled and the deletion process is complete. Otherwise, we scan the original bucket and copy across all strings except for the string to delete. The old bucket is then destroyed and the new is assigned. This process is akin to insertion, and is thus cache-efficient. However, the dynamic arrays used by the array hash table are effectively unbounded in size. This is an important observation; it means that the worst-case cost of deletion can exceed that of the equivalent linked list [Askitis and Sinha 2010].

The worst-case deletion occurs when the first string from a slot (that is under heavy load) is continually deleted (resembling a stack-like operation). In this case, we would be forced to scan and resize the entire array, whereas we need only delete the head node of the equivalent linked list (the list will not be traversed beyond the head node, which is efficient). We can, however, address this worst-case deletion cost by employing a type of lazy deletion; we can resize the array periodically, which will improve performance, but at the expense of space. This worst-case deletion cost does not apply to the burst trie, which we describe next, as dynamic arrays are bounded in size.

3.2. Cache-Conscious Burst Tries

The standard chained burst trie is currently one of the fastest and most compact data structures available for vocabulary accumulation when sorted access to strings is required [Heinz et al. 2002]. Although computationally efficient, the burst trie uses standard chains as containers, which is neither cache- nor space-efficient. To improve the efficiency of the burst trie, we replace the standard chains with compact chains, to create a more efficient alternative called a *compact* burst trie. Similarly, we can eliminate these chains by representing containers as dynamic arrays. This gives us a cache-conscious burst trie or *array* burst trie.

The standard, compact, and array burst tries maintain array-based trie nodes that map directly to the 128 characters of the ASCII table. Each trie node is therefore 512 bytes long (assuming 4-byte pointers). The first 32 pointers and the last pointer, however, map to nonprinting characters, which can be ignored. We, therefore, reserve the first pointer of each trie node to store a string-exhaust flag, which is explained later. Hence, our implementations of the burst trie are compatible with an alphabet of 94 characters. We could eliminate unused pointers, but we found that the amount of space saved in practice was too small to justify the increased cost in trie node access.

In a typical implementation, trie nodes are randomly allocated, which is not cache-efficient. Our trie nodes are, therefore, maintained in a dynamic array, in order of allocation. This improves access locality and eliminates memory allocation overheads. Containers in a standard or compact burst trie occupy 5 bytes of memory. The first 4 bytes store a pointer to the start of the chain. The next byte maintains housekeeping information: the string-exhaust flag. The string-exhaust flag indicates whether all the characters of a string have been removed during traversal. Once a trie node or a container removes the last character of the string used to traverse the burst trie, its string-exhaust flag is set. The physical ordering of these two fields is important. For efficiency, pointers should always start at a word boundary (an address divisible by 4). If a pointer is stored between two words, and assuming a 32-bit system bus, two bus cycles are required to fetch the pointer into cache [Hennessy and Patterson 2003], which is inefficient.

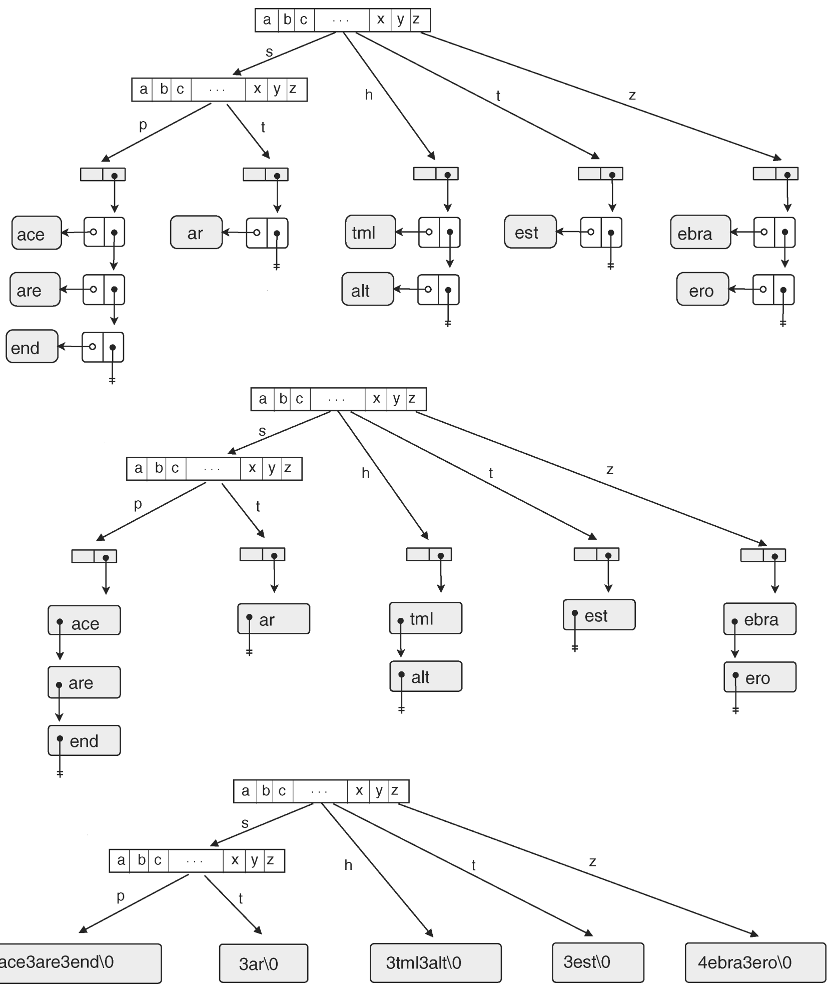


Fig. 4. The strings “space,” “spare,” “spend,” “star,” “html,” “halt,” “test,” “zebra,” and “zero” are stored in a standard burst trie (top), a compact burst trie (middle), and an array burst trie (bottom).

Alignment is not an issue for the containers of an array burst trie because pointers are eliminated. In an array burst trie, each container reserves the first 2 bytes for housekeeping: to maintain the string-exhaust flag and a flag to indicate whether a string has been inserted into the container. To implement a clustered burst trie, we convert a fully built standard-chain burst trie by accessing all containers and storing their nodes, in order of occurrence, contiguously in main memory. We now describe how to insert and search for strings in a standard, compact, and array burst trie; the structural differences, which are shown in Figure 4.

Initialization

The burst trie begins as an empty container with no trie nodes. The container is populated with strings until a threshold is met. Three thresholds—ratio, limit, and trend—are evaluated by Heinz et al. [2002], with the simplest and most effective being a limit, which bursts a container once it stores more than a fixed number of strings. In the discussions that follow, we assume that there is at least one trie node present (a root trie).

To Search for a String

Search proceeds as follows. The first character of the string (the lead character) is used as an offset into the root trie node to acquire a pointer. The pointer is followed to fetch the next node. Whenever a pointer from a trie node is traversed, the lead character of the string is deleted or consumed. If a null pointer is encountered, then the search fails. Otherwise, the process is repeated until a container is acquired.

It is possible to exhaust a string before acquiring a container, that is, to delete all of its characters. When this occurs, the first pointer (which represents the string-exhaust flag) from the last trie node accessed is inspected. The search is successful if the flag is set; otherwise, it is a failure. Similarly, the string can be exhausted on initial access to a container. When this occurs, the string-exhausted flag is accessed within the container, and, if set, the search is successful; otherwise, it is a failure.

If the string is not exhausted, then a container is acquired and searched. In a standard or compact burst trie, the search involves comparing every string in the container via a linked list traversal. On match, the node containing the required string is moved to the front of the list, and the search is successful. Otherwise, the list is exhausted and the search fails.

In an array burst trie, the container is a dynamic array and the search proceeds as described in Section 3: The first length-encoded string is accessed and compared, character by character. On mismatch, the next length-encoded string is accessed and so forth, until the end-of-bucket flag is encountered, on which the search fails. On a match, move-to-front is optional, due to the high computational costs involved.

To Insert a String

An insertion can only proceed on search failure, which occurs in one of four ways: The string is exhausted during trie traversal, the string is exhausted on initial access to a container, a null pointer is encountered, or the string is not found in a container. The two cases that fail due to string exhaustion are resolved by setting the string-exhaust flag in the acquired trie node or container, completing the insertion process.

When the search fails due to a null trie-node pointer, we assign the pointer to a new empty container. This will immediately consume the lead character of the string, which could exhaust the string. In this case, the container remains empty and its string-exhaust flag is set, completing the insertion process.

Otherwise, the string is inserted into the container, which is also how the last of the four cases is handled. For a standard or compact burst trie, the string is first encapsulated in a standard or compact node, respectively, and then added to the end of the chain. In an array burst trie, the string is appended to the container through array resizing (more details on array resizing can be found in the subsection before Section 3.1). After a string is inserted into a container, the container size must be checked to determine whether a burst is required.

To Delete a String

To delete a string, we first search for it as described earlier. If the search leads to a string-exhaust flag, the flag is cleared and deletion is complete. If a null pointer is encountered during search, then the string does not exist in the burst trie. Otherwise, a container is acquired and searched. Assuming that the string is found, deleting it in a standard or compact container is straightforward. In an array burst trie, however, a new container is allocated and sized to match the old container, minus the length of the string to delete. The old container is then scanned and its strings, apart from the one to delete, are copied into the new container. The old container is then deleted and the new is assigned. Empty containers are deleted, unless their string-exhaust flag is set. Deleting an empty container will clear its parent pointer, which may cause the parent trie-node to become leafless, in which case, it too is deleted; the deletion of trie nodes can propagate up to the root node.

Lazy deletion can also be incorporated; trie nodes can be flagged as deleted, allowing them to be efficiently reused. We can also apply lazy deletion to the containers of the array burst trie, whereby we simply slide the remaining strings in the container to the start of the string to delete, effectively overwriting it. Although more efficient, this approach comes at the expense of maintaining unused space in containers, which can be reclaimed through periodic container resizing.

Bursting a Container

A container is burst once its size exceeds a selected threshold. The choice of threshold and the definition of a container's size is described in the following text. To burst a container, it is first detached from its parent pointer. The parent pointer is then assigned to a new trie node. The strings in the container are then accessed and distributed into at most A new containers (A being the alphabet size, which in our case is 94), according to their lead character, which is removed prior to storing the strings in the containers. Bursting can, therefore, exhaust up to A strings. Once the bursting phase is complete, the original container is deleted and the new trie node allocated has its pointers assigned to the new containers.

Choosing a Container Size

A container is burst once it exceeds a certain size or limit, being the number of strings maintained. Choosing a limit requires some care. Large containers—a limit of over 100 strings, for example—are not burst as often, and can, therefore, reduce the net number of nodes created, saving space but at a cost of access time. Yet to some degree, the impact on time depends on the distribution of strings. Consider, for example, a large compact or standard-chain container that is accessed under heavy skew. With move-to-front, the majority of searches are likely to terminate on the first node. Similarly, under heavy skew, large containers in an array burst trie are likely to remain efficient, even without move-to-front, because on initial access an entire cache line of strings is prefetched, with the next few lines being likely candidates for hardware prefetch.

When little to no skew is present, however, move-to-front is rendered ineffective, and as a consequence, large compact or standard-chain containers will become expensive to access. The performance of large array-based containers, however, are likely to remain competitive due to their high spatial access locality. Smaller compact or standard-chain containers will be more efficient to access in this case, but will also greatly increase the number of trie nodes and containers allocated, which will consume a lot of space. In addition, as the number of trie nodes increase, temporal access locality can be reduced, as previously cached containers are likely to be systematically flushed out of cache by trie nodes.

An alternative scheme for the array burst trie is to change the limit of a container to capture its physical size, rather than the number of strings it can store prior to bursting. The advantage of this approach is that containers can match the size of a cache line, and thus incur at most only a single L2 cache-miss on access. Long strings, however, can complicate matters, especially when they cannot fit within fixed-sized containers. In this case, a surplus of trie nodes is likely to be generated.

Choosing a good limit for the burst trie is, therefore, not a straightforward task and is difficult to determine analytically. A small limit is likely to reduce access time but at the expense of space. A large limit will save space, but at a likely cost in access time.

Key factors to consider when deciding on a limit include the expected distribution of strings and the type of burst trie (standard, compact, or array) used. For the task of vocabulary accumulation, Heinz et al. [2002] suggested a limit of 35 strings for the standard burst trie, which they derived experimentally. We also derive a set of limits from our experiments, which should provide good performance in both time and space, for a variety of string distributions.

3.3. Cache-Conscious Binary Search Trees

Compact chains can be applied to all types of binary search tree. In this section, we concentrate on the standard BST, which is among the fastest and simplest of tree structures [Williams et al. 2001; Bell and Gupta 1993]. We replace the standard nodes of a binary search tree with compact nodes, yielding a cache- and space-efficient variant called a *compact* BST. Search and insert in a compact and standard BST is straightforward. The root node is compared against the string, and, on mismatch, the left pointer is followed if the string is lexicographically less than string in the current node; the right pointer is followed otherwise. This process continues down the tree until a match is found or until an empty or null pointer is encountered. In this case, the search fails and the string can be inserted by first encapsulating it as a compact or standard node, which is assigned to the empty pointer. To convert a fully built standard BST into a clustered BST, we contiguously allocate subtrees based on the structural topology of the BST, as described by Chilimbi [1999]. That is, we first store the root subtree—the root node and its left and right child (if any)—into a block of memory. The block is sized to match the cache line. We then traverse the tree in a depth-first manner and store the subtrees acquired into the block. Once the block overflows, a new block is allocated and the traversal continues until all subtrees are packed into blocks of memory.

The application of a dynamic array to the standard BST, however, requires greater care, as, for efficiency purposes, not all pointers can be eliminated. An array-based representation of a BST is described by Knuth [1998]. This algorithm takes an existing BST and collapses the nodes (including their keys) into a preorder traversal, in a single contiguous array, which is also known as a linear representation [Jonge and Tanenbaum 1987]. The single array is guaranteed to be accessed in a left-to-right manner, which improves access locality and thus cache-efficiency. Furthermore, with nodes stored in a preorder traversal, all left child-node pointers are now redundant and can be removed, saving space. This algorithm, however, assumes fixed-length keys. It can be adapted to store variable-length strings, but as a consequence, it can become expensive to update and, in some cases, to search; although the left child is guaranteed to be located after its parent, access to it involves a linear scan, and, to maintain preordering of nodes, a large section of the array may need to be shifted to store a new string in the correct position.

Our array-based approach differs in that we do not build and then collapse a standard BST into an array. Instead, we propose the novel approach of storing the entire BST in a single dynamic array, which is a key distinction to a clustered BST. On insertion, a string is represented as a compact node, with both left and right child-node pointers stored before the string. The node is then appended to the end of the array, which is grown (when full) using paging; we add 10MB of space per call—a value found through preliminary trials that achieves a good balance between time and space—to reduce the computational costs of copying. On search, the first node in the array (which is always the root node) is accessed and binary search proceeds as described. Although our array BST is not as space-efficient as Knuth's algorithm, retaining both child-node pointers permits efficient update and search. Moreover, with nodes being appended in order of occurrence, node access is guaranteed to proceed from left to right in the dynamic

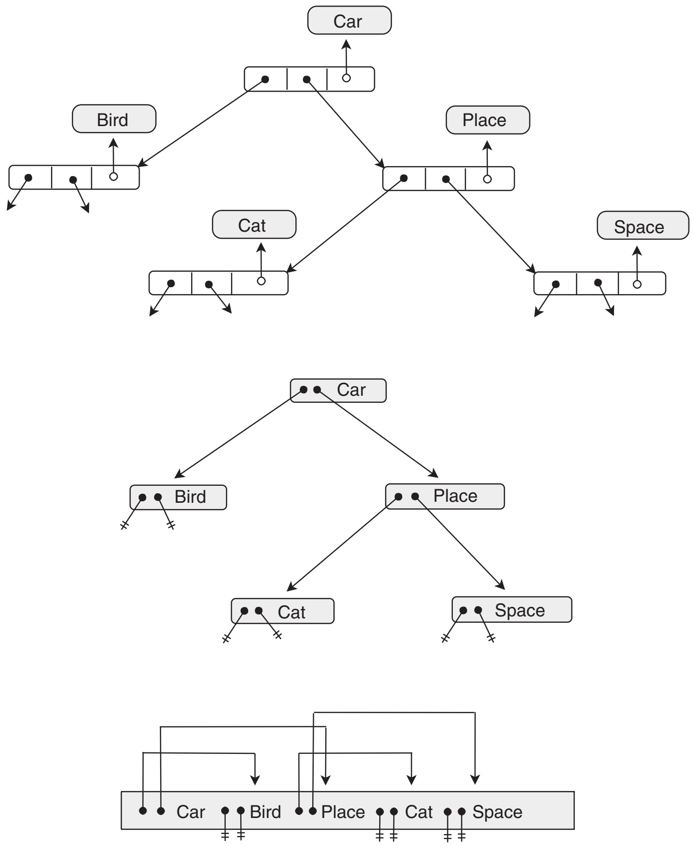


Fig. 5. The strings "Car," "Bird," "Place," "Cat," and "Space" are stored in a standard binary search tree (top), a compact binary search tree (middle), and an array binary search tree (bottom). The strings are appended in order of occurrence in the array BST to ensure left-to-right access on traversal, which is cache-efficient.

array. Figure 5 illustrates the structural difference between a standard, compact, and array BST.

The main disadvantage of the array BST—including Knuth's representation—is the overhead of deletion. Deleting a node involves one of three steps. First, the candidate node has no children and thus can be simply deleted. Second, the candidate node has a single child and is simply replaced by its child. Third, the candidate node has two children. In this case, the candidate node is replaced by its leftmost child from its right subtree. In a standard and compact chained BST, these steps require tree traversals and pointer manipulations. In an array BST, deleting a string involves both pointer manipulation and array copying; the array must be resized. However, resizing is cache-efficient, which is likely to compensate for the high computational costs involved. Hence, updating an array BST is expected to be as efficient as updating a chained BST, as we demonstrate later.

4. SPACE SAVED BY ELIMINATING POINTERS

We consider the space saved when applying our compact and array-based techniques to the standard hash table, burst trie, and BST, which we show in Table I. Every pointer eliminated saves 12 bytes of memory: 4 bytes for the pointer and 8 bytes of allocation overhead [LaMarca and Ladner 1996].

Table I. A Comparison of the Space Overhead of the Hash Table, Burst Trie, and BST, using Standard Chains, Clustered Chains, Compact Chains, and Dynamic Arrays

	Standard	Clustered	Compact	Array
BST	$28s$	$20s + 8(20s/C)$	$16s$	$8 + 8s$
Burst trie	$24\bar{s} + 520t + 13b$	$16\bar{s} + 520t + 21b$	$12\bar{s} + 520t + 13b$	$520t + 10b$
Hash (exact)	$4m + 24s$	$12m + 16s$	$4m + 12s$	$12m$
Hash (page)	—	—	—	$4m+$ $\bar{m}(B - l + 8)$

The number (s) of strings stored, the number ($\bar{s}$) of strings stored in containers, the total number (m) of slots allocated for the hash table, the number ($\bar{m}$) of slots used by the hash table (not null/empty), the number (t) of tries, and the number (b) of buckets used by the burst trie or the number of slots used the hash table. The block size (B) represents the pagesize used to grow a dynamic array, and (C) represents the block size used to cluster nodes.

Hash tables consume memory in several ways: space allocated for the strings and for pointers, space allocated for slots, and overhead due to operating system overheads and space fragmentation. For a standard chain, each string requires two pointers and (in a typical implementation) two *malloc* system calls. A further 4 bytes are required per slot. The space overhead is, therefore, $4m + 24s$ bytes, where m is the total number of slots and s is the number of strings inserted. In a compact chain, string pointers are eliminated, reducing the overhead to $4m + 12s$ bytes. In a clustered chain, the 8-byte allocation overhead per node is eliminated, but each slot incurs an initial 8-byte overhead for allocating of an array of nodes. Hence, the space overhead of a clustered hash table is up to $12m + 16s$ bytes. In an implicit chain, the next-node pointers are eliminated, but with the requirement of allocating a null 4-byte pointer at the end of each chain to serve as a delimiter. Hence, the space overhead of an implicit hash table is up to $16m + 12s$ bytes. The memory consumed by the array hash table is slightly more complicated to model. First, consider exact-fit. Apart from slot pointers, all other pointers are eliminated. The space overhead is then notionally $4m$ bytes plus 8 bytes per allocated array—that is, up to $12m$ bytes in total—but the use of copying means that there is an unknown amount of space fragmentation; fortunately, inspection of the actual process size shows that this overhead is small. The array uses length-encoding, so once a string exceeds 127 characters in length, an additional byte is required.

For paging, we assume that the block size is B bytes. When the load average is high, on average, each slot has one block that is half empty, and the remainder are fully used; thus, the overhead is $m(12 + B/2)$ bytes. When the load average is low—that is, $s < m$—most slots are either empty, at a cost of $4m$ bytes, or contain a single block, at a cost of $B - l + 8$, where l is the average string length. For short arrays, we allow creation of blocks of length $B/2$. Thus, the wastage is around $4m + \bar{m}(B - l + 8)$ bytes, where $\bar{m}$ represents the number of slots that are used (i.e., that point to an array).

For the standard binary search tree, each string stored has three pointers and issues two malloc system calls. The space overhead is, therefore, $28s$. In a clustered BST, the 8-byte allocation overhead is eliminated for every node. However, nodes are clustered into fixed-sized blocks which incur an allocation overhead. Thus, the space overhead of a clustered BST is $20s + 8(20s/C)$, where C is the size, in bytes, of the blocks used to cluster nodes. In a compact BST, the space overhead is $16s$ bytes and with an array, it is reduced to $8s + 8$ bytes, which is smaller than the standard BST by a factor of more than 3. We do not consider the space requirements of an implicit BST because eliminating child-node pointers can compromise performance, as noted in Section 3.3.

For the standard burst trie, we assume that t and b are the number of trie nodes and buckets, respectively. A trie node of 128 pointers requires 512 bytes and a call to malloc. We store trie nodes contiguously in arrays to eliminate allocation overheads and to improve TLB efficiency. However, in the measurements shown in Table I, we assume that trie nodes are individually allocated; hence, each trie node occupies 520 bytes of space.

A container in a standard or compact burst trie contains a 4-byte pointer to the start of its linked list and reserves a byte for housekeeping. The total overhead for the standard burst trie is, therefore, $24\bar{s}+520t+13b$, where $\bar{s}$ is the number of strings stored in containers. In a clustered burst trie, the 8-byte allocation overhead incurred by each node in a container is eliminated, except for the initial 8 bytes incurred from allocating the single block of memory used to house the nodes. Hence, the space required by the clustered burst trie is $16\bar{s}+520t+21b$. In an implicit burst trie, the next-node pointers are eliminated, but with the requirement of allocating a 4-byte pointer at the end of each container, to serve as a list delimiter. Hence, the space overhead for the implicit burst trie is $12\bar{s}+520t+25b$. In a compact burst trie, string pointers are eliminated, resulting in an overhead of $12\bar{s}+520t+13b$, while, for the array burst trie, the space overhead is reduced to only $520t+10b$, at no cost to performance.

5. EXPERIMENTAL DESIGN

To evaluate the efficiency of the hash table, burst trie, and BST using their standard, clustered, compact, and array representations, we compare the time required for construction and search, as well as the amount of memory consumed, against other string data structures: the standard hash table, burst trie, BST, splay tree, red-black tree, TST, the adaptive trie [Acharya et al. 1999], and the Judy data structure; we used the Judy-SL variant, which is designed for string keys [Hewlett-Packard 2001]. The elapsed or total time required by these data structures was averaged over a sequence of ten runs. After each run, main memory was flooded with random data in order to flush system caches.

In earlier work, we (and Ranjan Sinha) proposed the HAT-trie [Askitis and Sinha 2007], which extends the concepts presented in this article by changing the structural representation of containers in a burst trie from a linked list to an array hash table. Although some space is wasted as a result due to the requirement of storing a fixed-number of slots per container, containers can grow sufficiently large without incurring high cache or instruction costs. However, we did not compare the HAT-trie to other trie structures such as the adaptive trie or the Judy trie, and in particular the array burst trie, which we propose in this article. Moreover, a promising variant that combines the array burst trie with adaptive containers that change to array hash tables when full—which we call a HAT^{+}-trie—is likely to be more efficient. For brevity, however, we do not experimentally evaluate our data structures to the HAT-trie or HAT^{+}-trie in this article; we instead leave this for upcoming work [Askitis and Sinha 2010].

Measurements of space include an estimate of the overhead imposed by the operating system, which, in our case, was 8 bytes per system call. We compared our measure of space with the total memory usage reported by the operating system under the `/proc/stat/` table; we use this to investigate the extent of overhead caused by memory fragmentation. Although fragmentation did occur (as one would expect from frequent array resizing), the overhead was found to remain small and fairly consistent to our calculations of space consumption [Askitis and Zobel 2005; Askitis 2007; 2009; Askitis and Sinha 2010].

We used PAPI [Dongarra et al. 2001] to measure the actual number of instructions, TLB, and L2 cache misses of search (we did not count L1 misses, as they have only a small performance penalty). We do not report the cache performance of construction, as the results were similar to those for search. In addition, experiments involving deletion were omitted, as the results were found to be similar to those of construction.

Table II. Characteristics of the Datasets Used in our Experiments

Dataset	Distinct Strings	String Occs	Average Length	Volume (MB) of Distinct	Volume (MB) Total
DISTINCT	28,772,169	28,772,169	9.59	304.56	304.56
TREC	612,219	177,999,203	5.06	5.68	1,079.46
URLS	1,289,459	9,999,425	30.93	45.93	318.87

The datasets used for our experiments are shown in Table II. They consist of null-terminated variable length strings acquired from real-world data repositories. The strings appear in order of first occurrence in the data; they are, therefore, unsorted. When the same dataset is used for both construction and search, the process is called a *self-search*.

The TREC dataset is the complete set of word occurrences, with duplicates, in the first of the five TREC CDs [Harman 1995]. This dataset is highly skew and contains only a small set of distinct strings. The DISTINCT dataset contains almost 29 million distinct words (i.e., without duplicates) extracted from documents acquired in a Web crawl and distributed as the "large web track" data in TREC. The URLS dataset, extracted from the TREC Web data, is composed of nondistinct complete URLs.

To measure the impact of load factor on the hash tables, we varied the number of slots used. We commenced with 2^{15} slots, which we doubled to 2^{27} or until a minimum execution time was observed. For the standard and compact burst tries, we varied the container threshold (the number of strings a container needs to trigger a burst) by intervals of 10 from 30 to 100. For the array burst trie, we extended the sequence to include 128, 256, and 512 strings. Both compact and standard chaining methods are most efficient when coupled with move-to-front on access, as reported by Zobel et al. [2001]. We enabled move-to-front for the chained hash tables and burst tries but disabled it for the array-based data structures, a decision that we justify in later discussions.

The primary machine used for our experiments was a 2.8GHz Pentium IV. We conducted experiments on other machine architectures, namely a 3GHz Intel Xeon processor and a 700MHz Intel Pentium III processor. The Xeon processor showed similar performance to the Pentium IV and was thus omitted. We include results from the Pentium III processor, however, as it had no hardware prefetch mechanism—we were unable to disable hardware prefetch on our primary machine. We also include results from a Sun UltraSPARC processor. Some of the characteristics of these machines are summarized in Table III.

Our experiments (on our primary machine) were conducted using a 32-bit Linux operating system that we kept under light load (single user mode). We are confident—after extensive profiling—that our implementations are of high quality. Our data structures were implemented in C and were compiled using `gcc` version 4.1.1, with all optimizations enabled: `-fomit-frame-pointer -O3`. We found that the bitwise hash function [Ramakrishna and Zobel 1997] with a mask instead of a modulo was a near-insignificant component of the total costs of hashing. Williams et al. [2001] reported the inefficiency of using the default string compare (`strcmp`) library routine provided by the Linux operating system, and showed that their own implementation achieved speed gains of up to 20%. We, thus, do the same for our implementations.

6. RESULTS

In the environment of a computer with a cache hierarchy, the behavior of an algorithm in practice is not easy to predict analytically. Relationships between factors such as data distribution, search versus insertion rates, data volume versus cache capacities, and

Table III. Characteristics of the Machines Used in Our Experiments

	Intel Pentium IV	Intel Pentium III
CPU speed	2.8GHz	700MHz
No. CPUs	1	4
L1/L2 size (KB)	8/512	16/1,024
L1/L2 cache-line (B)	64/128	32/64
TLB entries	64	32
Main memory (MB)	2,048	2,048
Page size (KB)	4,096	4,096
Avg. mem. latency	95ns	160ns
Linux kernel	2.6.12 x86	2.6.12 x86
Hardware prefetch	yes	no

The Pentium IV (highlighted) was our primary machine.

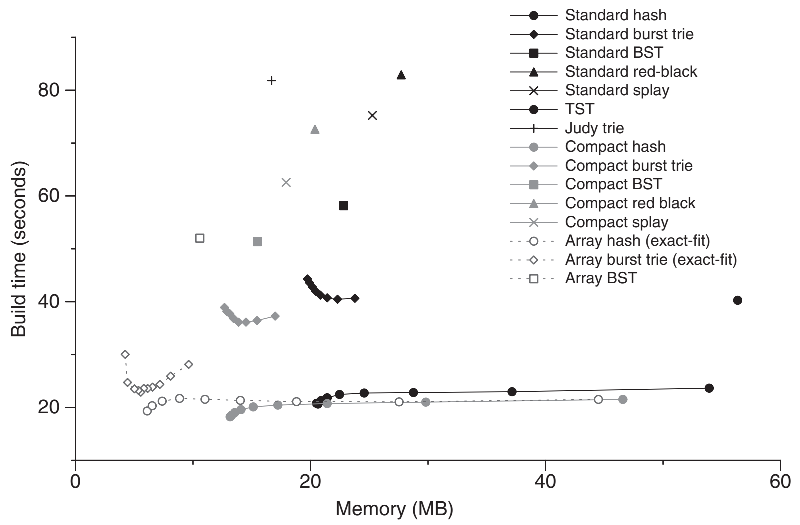


Fig. 6. The time and space required to build the data structures, using the TREC dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence is extended to include 128, 256, and 512 strings. The hash tables commenced with 2^{15} slots, doubling up to 2^{23}.

CPU, memory speed, and bus capacity mean that modeling of expected performance is far from straightforward.

We have, therefore, focused on use of extensive experiments, in which all these factors are independently varied, to explore the behavior of our algorithms in practice. In this section, we report on the outcomes of these experiments.

6.1. Skew Data

A typical use for string data structures is to accumulate the vocabulary of a collection of documents. In this process, in the great majority of attempted insertions the string is already present, and some strings are much more common than others. Figures 6 and 7 show the relationship between the time and memory requirements of our data structures, for construction and self-search. The dataset used in these experiments was TREC.

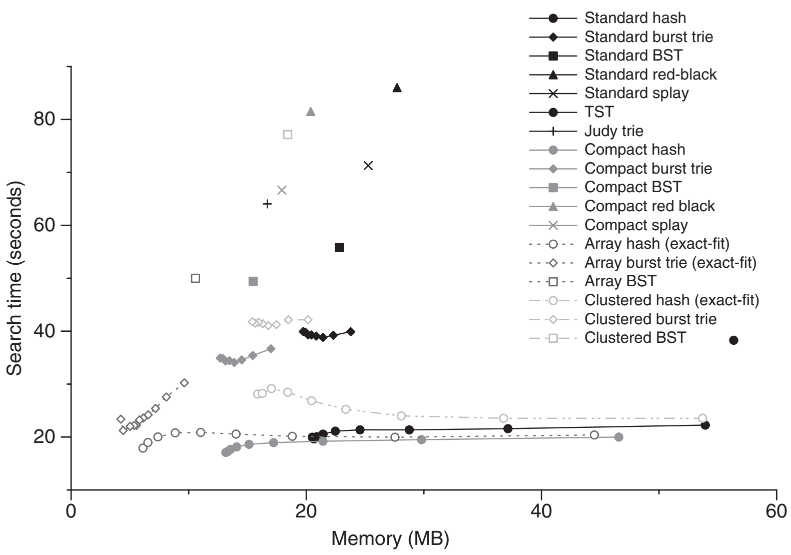


Fig. 7. The time and space required to self-search the data structures, using the TREC dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller buckets (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence is extended to include 128, 256, and 512 strings. The hash tables commenced with 2^{15} slots, doubling up to 2^{23}.

6.2. Standard-Chain Data Structures

The Judy data structure required the least amount of space of all standard-chained data structures but was almost the slowest to access under skew, being only slightly faster than the red-black tree. As reported by Williams et al. [2001], the standard BST was the fastest tree to construct and self-search under skew. The red-black and splay trees were inefficient due to the maintenance of a balanced and self-adjusting tree structure, respectively. The L2, TLB, and instruction costs for self-searching Judy and the variants of BST are shown in Figure 8. There is a strong correlation with the number of cache misses incurred and the overall search time. For example, the red-black tree incurred the highest L2 and TLB misses and was, therefore, the slowest to access. Although Judy executed fewer instructions than the standard BST, it incurred more cache misses, which caused its poor performance.

As reported by Heinz et al. [2002], the standard hash table and burst trie were the fastest standard-chained data structures to build and search. Although the standard burst trie required almost twice as much time to build and self-search than the standard hash table, it was competitive with regards to space, and maintained sorted access to containers. The TST required the most space but could match the construction and self-search speeds of the standard burst trie, as result of its relatively low L2 and TLB misses (Figure 8).

6.3. Clustered-Chain Data Structures

We compare the self-search performance of the three fastest standard-chain data structures—the hash table, burst trie, and BST—with clustering [Chilimbi 1999]. Clustering stores the nodes of a linked list in a contiguous block of memory, to improve spatial access locality without eliminating pointers. The time and space required to self-search a clustered hash table, burst trie, and BST are shown in Figure 7.

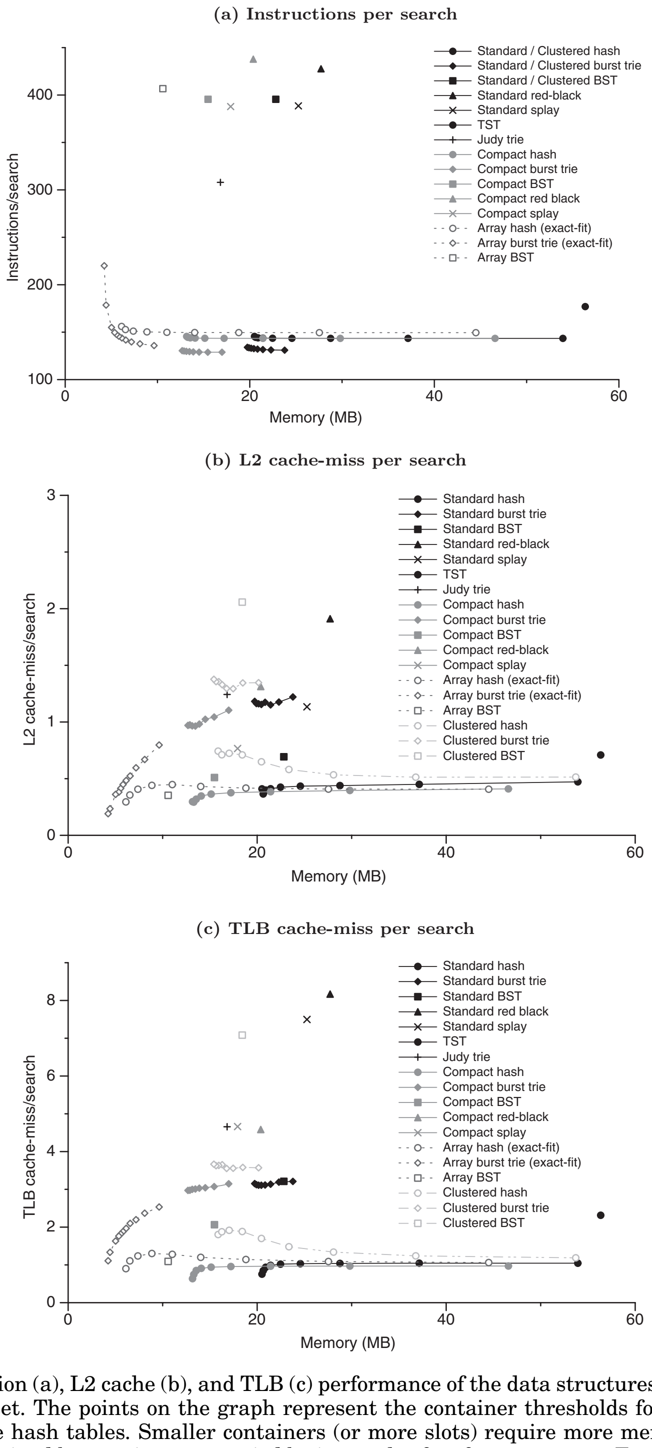


Fig. 8. The Instruction (a), L2 cache (b), and TLB (c) performance of the data structures when self-searched using the TREC dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence was extended to include 128, 256, and 512 strings. The hash tables commenced with 2^{15} slots, doubling up to 2^{23}.

Our results show that, although the clustered data structures require less space than their standard representations—due to the elimination of the 8-byte allocation overhead per node—the clustered hash table, burst trie, and BST were, in all cases, slower to access than their standard representations. Consider the memory access patterns of traversing a clustered chain from the slot of a hash table or within a container of a burst trie. With move-to-front on access, the first node in the block may not necessarily be the first node accessed. As a result, searching a clustered list can lead to random memory accesses resulting in poor use of cache. When the clustered list spans multiple cache lines, these random accesses are likely to become more expensive. This is reflected in the self-search costs of Figure 7 and the cache costs of Figure 8. For example, as the average load factor increases, the clustered hash table incurs more cache misses than the equivalent standard hash table.

If we disable move-to-front on access in a clustered list, the cost of access increases even further, as now the list must be traversed, in order of allocation, until the desired node is found. Although the number of random accesses is reduced, the number of pointer dereferences increases, which can further hinder the effectiveness of hardware prefetchers. For clustering to be effective under skew access, it is necessary to reduce both random accesses and the number of pointer dereferences, by physically moving nodes to the start of the list. However, this approach can substantially increase the computational cost of search—particularly for large lists—which can outweigh the benefits (which we confirmed through preliminary trials on the clustered hash table and burst trie).

Traversing a standard list also incurs random memory accesses. However, nodes can be allocated anywhere in memory, which gives the operating system some flexibility at improving access locality. That is, the operating system can dynamically group frequently accessed pages in memory to improve cache utilization; it cannot do this as effectively once the programmer enforces contiguous storage of nodes that are accessed via pointers.

6.4. Compact-Chain and Array-based Data Structures

We implement the compact-chain and dynamic array representations for the hash table, burst trie, and BST, and include a compact-chain representation of the red-black and splay tree. The time and space required to construct and self-search these data structures are presented in Figures 6 and 7.

Hash Tables

The compact-chain hash table was the fastest data structure to construct under skew access. The compact hash achieved its best construction time of 18.2 seconds and a self-search time of 17.1 seconds, using 2^{15} slots and 13.1MB of memory; a space overhead of about 98 bits per string (only 612,219 strings are distinct in the TREC dataset, which require a total of 5.6MB). The equivalent array hash was marginally slower to construct and self-search, requiring under 19.2 and 17.9 seconds, respectively. Despite its high cache-efficiency, the array hash required more time as a result of executing more instructions (Figure 8), caused by the absence of move-to-front during search, and array resizing during construction.

The array hash, however, was considerably smaller in size, requiring only 6.1MB of memory; a space overhead of about 6 bits per string. This efficiency is achieved despite a load average of 37 strings per slot. Increasing the number of slots (reducing the load average) had no positive impact on speed. Having a high number of strings per slot is efficient so long as the number of cache misses is low; indeed, having more slots can reduce speed, as the cache efficiency is reduced because each slot is accessed less often. Thus, the usual assumption—that load average is a primary determinant of speed—does not always hold. The cache performance of the compact and array hash tables are shown in Figure 8. In all cases, the compact and array hash tables showed consistent and simultaneous reductions in L2 and TLB misses over the standard hash table.

The standard hash table was markedly inferior in both time and space, achieving its best construction and self-search time of 20.6 and 19.6 seconds, respectively, using 2^{16} slots and 20.6MB of memory; a space overhead of about 195 bits per string. Given that the standard hash table was the fastest-known data structure to construct and self-search under skew access [Zobel et al. 2001], we have strong evidence that our new structures represent a significant improvement.

BSTs

Compared to the performance of the standard hash table and burst trie, the standard BST was markedly inferior. However, this was not the result of poor cache utilization. Figure 8 shows that the standard BST is more cache-efficient—in both L2 and TLB—than the standard burst trie. Although unbalanced, frequently accessed nodes are likely to be located near the root of the tree. Furthermore, with no structural modifications during search, frequently accessed tree paths are likely to persist longer in cache. However, the standard BST was computationally expensive to self-search—executing almost 400 instructions per search, which is almost three times more than the burst trie (Figure 8).

The compact BST further reduced L2 and TLB misses, which made it faster to build and self-search than the standard BST but remained computationally expensive relative to the standard burst trie. The array BST showed even further reductions in L2 and TLB misses, and approached the cache-efficiency of the array hash table and array burst trie. However, having executed more instructions, the array BST was marginally slower to access than the compact BST. Nonetheless, both the array and compact BSTs displayed consistent improvement over the standard BST, which is currently the fastest tree structure under skew access, with size falling from 23MB to 15MB for the compact BST, to about 10MB for the array BST, at no cost.

Burst Tries

The array burst trie showed strong improvements in both time and space over its compact and standard chained variants. These improvements are consistent for both construction and self-search. Comparing the cache performance of the array burst trie with its chained variants (Figure 8), we see a sharp decline in L2 and TLB misses as the container threshold or capacity increases from 30 to 512 strings.

At its best, the array burst trie with a container threshold of 256 strings, required 21.2 seconds to self-search and 4.4MB of space. This is about 1.6MB smaller than the array hash, which is less space than required by the strings alone. The compact burst trie, at its best, required 34.1 seconds to self-search with a container threshold of 60 strings and 13.7MB of space; an overhead of about 104 bits per string. The standard burst trie was markedly inferior, requiring 39.1 seconds to self-search and 20.6MB of space; an overhead of about 195 bits per string. Considering that the standard burst trie was reported to be the most efficient data structure for the task of vocabulary accumulation [Heinz et al. 2002], our array burst trie, being up to 46% faster while imposing no space overhead per string, is a substantial advance.

Use of large containers in the array burst trie—containers storing over 256 strings, for example—are preferable, because they can be both cache- and space-efficient. Large containers, however, are computationally expensive to access. As the size of containers increase from 128 to 512 strings, for example, Figure 8 shows a sharp increase in instructions per search, caused by the absence of move-to-front on access. The same behavior was observed during construction, with large containers being cache-efficient

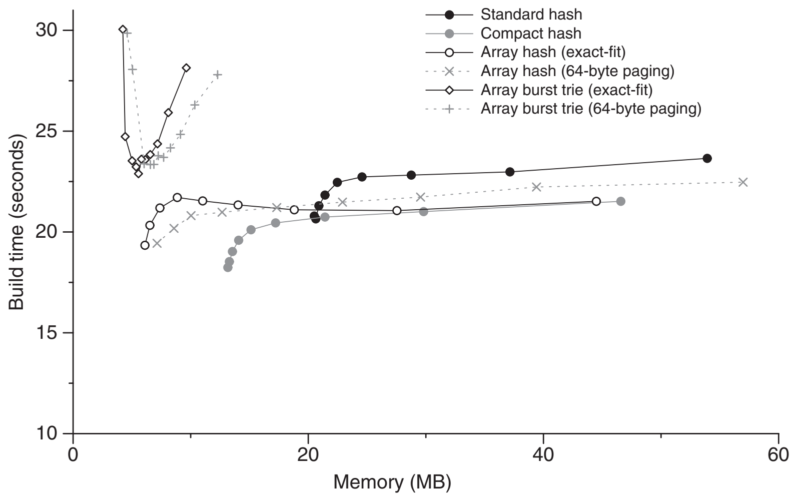


Fig. 9. A magnified version of Figure 6 showing the time and space required to build the array hash and array burst trie, using the TREC dataset with and without paging. Paging grows an array in 64-byte chunks, in contrast to exact-fit. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory.

but incurring high numbers of instructions per search, due to the absence of move-to-front and array resizing.

The high computational cost of accessing large containers could not be entirely masked by the reductions in cache misses, and, as a consequence, the array burst trie became more expensive to build and search, relative to its smaller containers. For example, when the container size increased from 256 to 512 strings in Figure 6, the time required to build increased by about 17% (from 25 to 30 seconds). However, when compared to a standard burst trie with a container threshold of 512 strings, the array burst trie was up to 66% faster (or around 58 seconds) to build and self-search, due to its high reduction in cache misses. These results show that, in order to sustain a fast and scalable burst trie, it is necessary to reduce cache misses while sustaining low numbers of instructions, relative to chaining [Askitis 2007].

6.5. Paging versus Exact-Fit Array Growth

We compare the difference in time and space for building the array hash and array burst trie, using the exact-fit and 64-byte paging techniques. The results are shown in Figure 9. The exact-fit model—where the array is resized on every insertion—is space-efficient but can be more expensive to build (as a result of excessive copying), whereas paging permits faster construction at the expense of maintaining some unused space. The relationship with speed, however, is more complex, with paging faster in some cases, but degrading relative to exact-fit, due to a more rapid increase in space consumption. The choice of array growth policy had negligible impact on the performance of search. The situations where paging would likely yield stronger improvements is with large numbers of insertions, which we consider in later experiments.

6.6. Effectiveness of Move-to-Front on Arrays

Despite its high memory usage, the compact hash table performed well under skewed access, partly due to the use of move-to-front. With dynamic arrays, move-to-front on access is computationally expensive because strings must be copied. Figure 10 compares the self-search costs with and without move-to-front on access, and includes

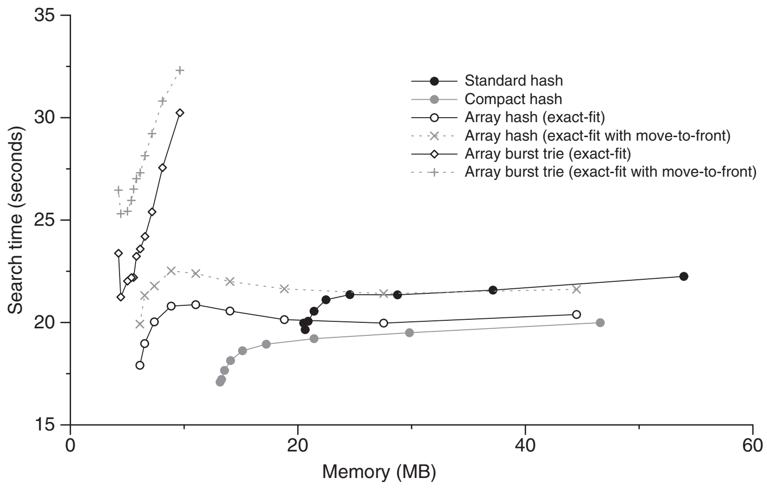


Fig. 10. A magnified version of Figure 7 showing the time and space required to self-search the array hash and array burst trie, using the TREC dataset with and without move-to-front on access. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory.

comparable figures for compact and standard chaining (both with move-to-front). The use of move-to-front reduces the speed of the array hash; even though the vast majority of searches terminate with the first string (so there is no string movement), the cases that do require a movement are costly.

Performing a move-to-front after every *k*th successful search might be more appropriate. Alternatively, the matching string can be interchanged with the preceding string, a technique proposed by McCabe [1965]. However, preliminary trials revealed that these techniques were not particularly effective at reducing the high computational costs of move-to-front. Hence, we believe that move-to-front is unnecessary for the array-based data structures, as the potential gains seem likely to be low.

With move-to-front disabled, however, frequently accessed strings are likely to incur more instructions and cache misses—in particular TLB misses—which is reflected in the instruction and TLB costs of the array hash (Figure 8). As a consequence, the array hash was slightly slower to build and self-search than the compact hash table. We also enabled move-to-front in the containers of the array burst trie and observed similar results. With move-to-front enabled, the array burst trie was consistently slower to access. In contrast to the array hash, however, the array burst trie remained faster than its chained variants, even without move-to-front, due to the use of bounded-size containers and a trie structure that removes shared prefixes—which can reduce both instruction and cache costs during search.

6.7. Skew Search on Large Data Structures

Our previous experiments involved data structures that were small in size, containing only 612,219 distinct strings. In this section, we repeat the previous skew search experiment but on much larger data structures that contained almost 29 million distinct strings. In this case, fewer frequently accessed nodes and strings are likely to reside in cache, which can impact performance. The data structures were built using the DISTINCT dataset, and then searched using the TREC dataset. The time and space required to search are shown in Figure 11.

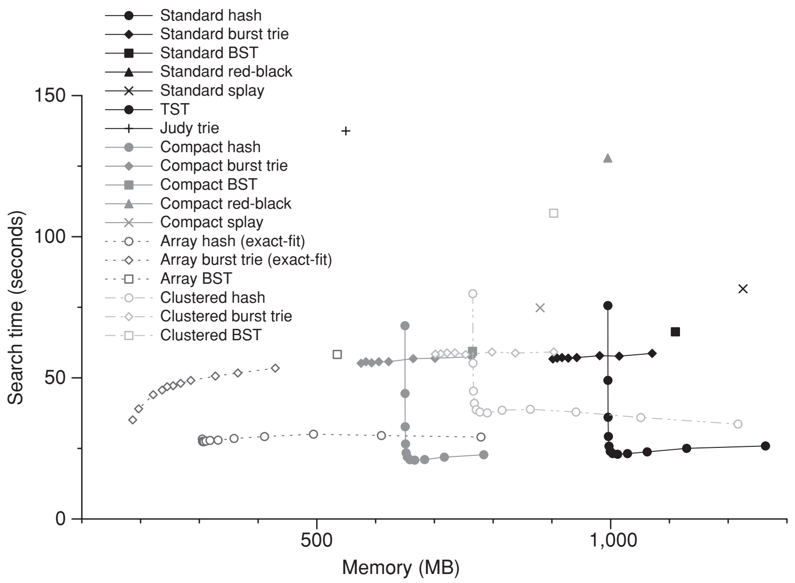


Fig. 11. A skew search using the TREC dataset, on data structures that were built using the DISTINCT dataset. The TST is not shown, as it required over 2,497MB of memory and about 1,194 seconds to search. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence was extended to include 128, 256, and 512 strings. The hash tables commenced with 2^{15} slots, doubling up to 2^{26}.

The red-black tree and Judy were the slowest data structures to access. The splay tree, however, showed considerable improvement over the red-black tree, which demonstrates that splaying is more effective under skew access than a balanced tree structure. Both the splay tree and the red-black tree, however, were space-intensive. As observed in previous experiments, the standard BST was the fastest tree to access, despite having increased in size by a factor of more than 48—from 22.8MB to over 1110MB. The TST consumed almost 2.5GB of space and thus exhausted main memory. As a consequence, virtual memory was accessed, which, resulted in a total search time of almost 1,200 seconds.

At its best, the array hash required about 27.3 seconds to search using 2^{17} slots and only 306.2MB of space. This is a space overhead of less than a bit per string. The equivalent compact and standard hash tables were up to 24% slower, requiring 32.6 and 36.1 seconds, respectively. Furthermore, the standard hash table consumed 995.6MB of memory, a space overhead of about 192 bits per string. The equivalent compact hash was more efficient, requiring 650.3 MB or about 96 bits per string. With 2^{17} slots, the average load factor was approximately 220 strings per slot. At such a high load, the chaining hash tables were too expensive to access, as they incurred high L2 and TLB misses that masked the benefits of move-to-front. Under heavy load, the cache-efficient array hash was, therefore, the fastest data structure, despite having executed more instructions due to absence of move-to-front.

Given enough space, the chained hash tables—as a result of move-to-front—could rival the cache-efficiency of the array hash, while executing fewer instructions. As a result, the chaining hash tables can become slightly faster to access than the array hash. As observed in previous experiments, however, increasing the number of slots had no positive impact on the array hash.

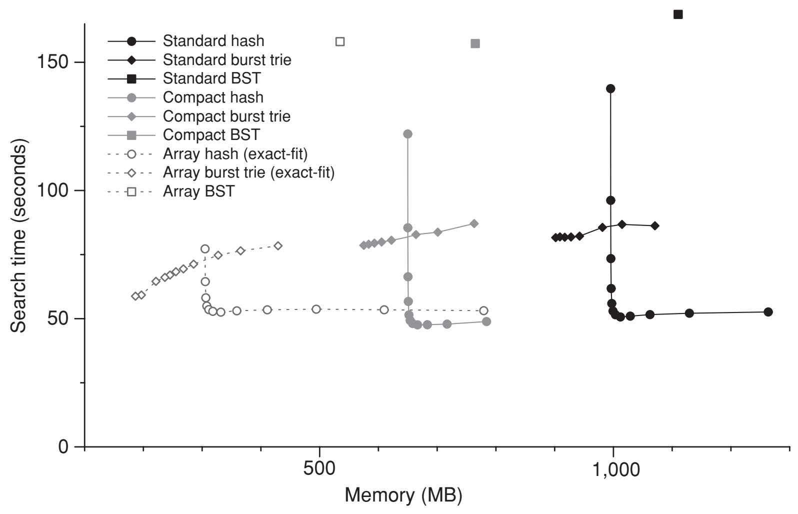


Fig. 12. A skew search using the TREC dataset on data structures that were constructed using the DISTINCT dataset on the Pentium III processor with no hardware prefetch. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence was extended to include 128, 256, and 512 strings. The hash tables commenced with 2^{15} slots, doubling up to 2^{26}.

At their best—using 2^{22} slots—the compact and standard hash tables were up to 24% faster than the array hash, requiring only 20.8 and 22.9 seconds to search, respectively. However, in order to achieve this speed (a difference of only 6.5 seconds from the array hash), the compact and standard hash table required two to three times the space of the array hash, respectively. For common string-processing tasks such as vocabulary accumulation, such an increase in space consumption will often be unacceptable, as it implies that fewer strings can be managed in-memory.

The compact and array BSTs were faster to access than the standard BST, while saving a substantial amount of space. The clustered BST, however, required almost twice the time of the standard BST. The standard burst trie was faster than all variants of BST, but remained almost twice as slow as the standard hash table, as observed in previous experiments. The clustered burst trie was the slowest burst trie to access, while the compact burst trie showed consistent gains in both time and space over the standard burst trie. The array burst trie displayed the strongest improvements, being the most space-efficient data structure—imposing no space overhead per string—while approaching the performance of the array hash table. For example, the array burst trie was up to 38% faster than its chained variants, and required at best, only 8 seconds more than the array hash while maintaining sorted access to containers.

6.8. Performance without Hardware Prefetch

We repeat the previous skew search experiment on the Pentium III processor, to observe the value of eliminating pointers on a slower machine with no hardware prefetch. In this experiment, we considered the three best data structures—the hash table, burst trie, and BST—using their standard, compact, and array-based representations. Figure 12 shows the results.

Despite the absence of hardware prefetch, the results were consistent with previous experiments. The compact BST and array BST remained faster than the standard BST

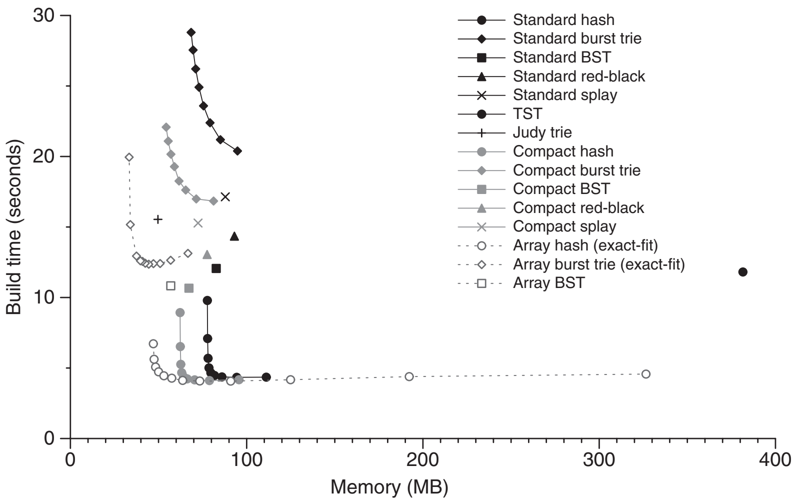


Fig. 13. The time and space required to build the data structures using the URLS dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence was extended to include 128, 256, and 512 strings. The hash tables commenced with 2^{15} slots, doubling up to 2^{24}.

while saving a considerable amount of space. The standard burst trie was faster than all three BSTs, but remained almost twice as slow as the standard hash table. The compact burst trie showed only marginal improvements in time, whereas the array burst trie showed the strongest gains.

Similarly, the array hash table was slightly slower than the compact and standard hash tables, but saved a considerable amount of space. The lack of hardware prefetch, however, meant that large arrays would incur more cache misses when traversed. As a consequence, the array hash required more time to search when under heavy load, compared to its previous performance in Figure 11. The increase in time can also be attributed to the use of a slower processor and a smaller cache-line size, which can increase the cache-miss rate [Hennessy and Patterson 2003]. Nonetheless, the array hash remained competitive to the chained hash tables, which were only faster once given enough space. These results demonstrate that the compact and array-based representations of the hash table, burst trie, and BST can yield strong gains in performance on older machines where instructions are more expensive to execute, and with no hardware prefetch.

7. URL DATA

Our next experiments used the URLS dataset, a dataset with some skew but in which the strings were much longer, of over 30 characters, on average. As in the skewed search experiments discussed previously, our aim was to find the best balance between execution time and memory consumption. Construction and self-search results are shown in Figures 13 and 14.

7.1. Hash Tables

The hash tables were the fastest data structures to build and search, but the optimum number of slots was much larger than those required during the TREC experiments, with the best load average being less than 1. To achieve its best time of 4.3 and 4.0 seconds for construction and self-search, respectively, the standard hash table

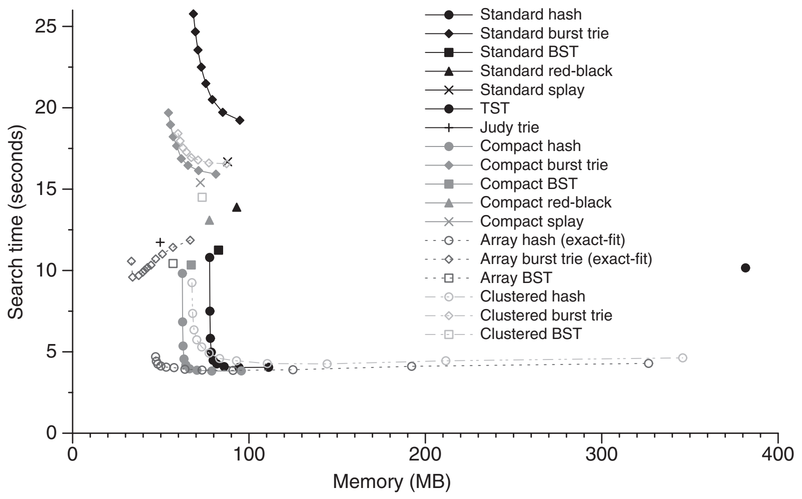


Fig. 14. The time and space required to self-search the data structures using the URLS dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence was extended to include 128, 256, and 512 strings. The hash tables commenced with 2^{15} slots, doubling up to 2^{24}.

required 2^{22} slots and 94.3MB of memory; this is a space overhead of about 300 bits per string. The compact hash was slightly faster at 4.1 and 3.8 seconds for construction and self-search, respectively, using 2^{22} slots and only 78.8MB of memory; this is an overhead of about 204 bits per string. The array hash with exact-fit achieved its fastest time of 4.1 and 3.9 seconds for construction and self-search, respectively, using only 2^{21} slots and 63.6MB of memory; this is a space overhead of about 109 bits per string, almost three times less than the standard hash.

As observed in previous experiments, the array hash was slightly slower to build and self-search than the compact hash, due to the lack of move-to-front during search and the resizing of arrays during construction. However, the increase in instructions was small and greatly compensated by the high reductions of L2 and TLB misses. As a result, the array hash was consistently faster to build and self-search than the standard hash table. The L2, TLB, and instructions costs incurred by the standard, compact, and array hash tables during self-search are shown in Figure 15. Increasing the number of slots available had little impact on the performance of the array hash, due to high reductions in L2 and TLB misses; the chaining hash tables, in contrast, could only compete in cache efficiency once given enough space.

The clustered hash table was slightly faster to self-search than the standard and compact hash, but only under heavy load, and in no case was it superior to the array hash. In the previous experiments, however, the clustered hash table was found to be consistently slower than both the standard and compact hash tables. Hence, these results suggest that the clustered hash table may become more effective at exploiting cache when there is little to no skew in the data distribution. Without skew, traversing a clustered list will effectively resemble a linear scan, which can make good use of cache. We consider this in later experiments.

7.2. Burst Tries

As in our experiments involving the TREC dataset, the array burst trie displayed strong improvements over its chained variants, approaching the speed of hashing while

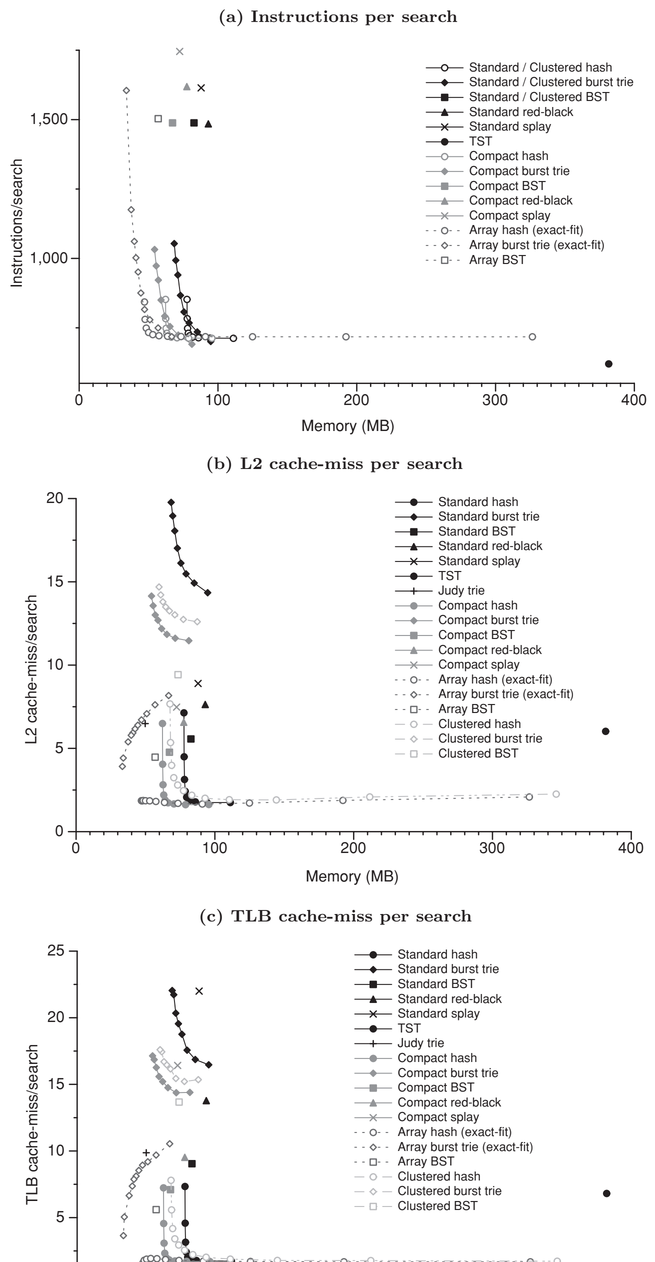


Fig. 15. The Instruction (a), L2 cache (b), and TLB (c) performance of the data structures when self-searched using the URLS dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence was extended to include 128, 256, and 512 strings. We omit the instruction cost for containers with a threshold of 512 strings. The hash tables commenced with 2^{15} slots, doubling up to 2^{24}.

requiring the least amount of space. At its best, the array burst trie took 12.9 and 9.6 seconds to construct and self-search, respectively, while consuming only 37.4MB; this is less space than required by the strings alone, which is the result of eliminating pointers and the pruning of shared prefixes in containers.

The compact burst trie also showed consistent improvements in both the time and space compared to the standard burst trie, though not as strong as the array burst trie. At its best time, the standard burst trie required 20.3 and 19.2 seconds to construct and self-search, respectively, and about 94.2MB of space; this is a space overhead of about 300 bits per string. Remarkably, the array burst trie was able to almost halve the time required by the standard burst trie, while imposing no space overhead per string.

The clustered burst trie was slightly faster to self-search than the compact burst trie, but only with large containers. Once the container size decreased, so did its performance relative to the compact burst trie. Nonetheless, the clustered burst trie remained faster to self-search than the standard burst trie, due to the reduction of L2 and TLB misses. The L2, TLB, and instruction costs incurred by the burst tries during self-search are shown in Figure 15.

As observed in the previous experiments, use of large containers in the array burst trie was not only both cache-and space-efficient, but also incurred high numbers of instructions. As the container threshold increased from 128 to 512 strings, for example, the number of instructions executed per search by the array burst trie increased from 1,175 to 2,266. The URLS dataset is not as skew at the TREC dataset, and as a consequence, the effectiveness of move-to-front is reduced. This caused the standard, clustered, and compact chains to incur high instructions costs on search, as more nodes were likely to be inspected. For example, as the container threshold increased from 128 to 512 strings, the number of instructions executed per search by the standard burst trie increased from 1,149 to 2,251—which is almost as expensive as the array burst trie. Hence, the standard and compact burst tries were slow to access due to high cache and instruction costs.

The array burst trie, however, substantially reduced the number of cache misses incurred and, as a result, was up to 60% faster than its chained variants while simultaneously saving space. Nonetheless, as its container threshold increased from 128 to 512 strings, the computational cost of accessing arrays could not be entirely masked by the reduction of cache misses. As a consequence, the array burst trie became more expensive to access relative to its smaller containers, despite a further reduction in cache misses. For example, when the container size increased from 256 to 512 strings, the array burst trie was around 25% slower (or about 5 seconds) to build but still remained greatly superior to its chained equivalents.

7.3. Variants of BST, the TST, and Judy

The standard, clustered, compact, and array BSTs performed surprisingly well compared to the tries. While not as compact, all three models—the array BST in particular—required less time to construct and self-search than all representations of burst tries; the array burst trie could only rival in speed during self-search.

Although strings in the burst tries are also stored in occurrence order, these strings share long prefixes, such as http://www, which in a burst trie, implies access to a potentially larger trie index. Trie nodes are computationally efficient but accessing a large number of them before acquiring a container can result in an increase in cache misses. With a container threshold of 50, for example, 50,627 tries nodes were created, as opposed to the 6,009 created with the TREC dataset. Consequently, as shown in Figure 15, the number of cache misses incurred by the array burst trie were, at best, only slightly less than the BSTs. Hence, although the computational cost of the BSTs was higher, the reduction in cache misses compensated. In this example, a Patricia trie could reduce the number of trie nodes created, since it is designed to omit single-descendant nodes. However, the reduction of nodes (and subsequent space consumption) is often small in comparison to alternative trie structures such as the TST [Sedgewick 1998; Heinz et al. 2002]. In addition, the space saved is offset by the complexity of its structure. The requirement of a full string comparison on leaf node access (to eliminate the possibility of a false match) for instance, is expensive compared to the comparisonless traversal of the array-based trie structure used by the burst trie [Heinz et al. 2002].

The standard and compact red-black and splay trees were also faster to build and self-search than the chaining burst tries but required more space and were slower to access than the BSTs. Judy also showed strong gains in performance relative to the previous TREC experiments, rivaling the speed of the standard and compact red-black and splay tree, while requiring the least amount of space of all standard-chained and compact-chained data structures. Judy also required less space than the array BST—due to the pruning of shared prefixes—and was as space-efficient as the array hash table. Nonetheless, Judy remained slower to build and self-search than the standard, compact, and array BST, the array hash, and array burst trie, as it incurred more cache misses.

The TST also rivaled the build and self-search time of the burst tries and BSTs, due to low instruction costs. However, the TST required almost 400MB of space—which is almost 4 times the space required by the standard BST. Although the clustered BST remained faster to access than the compact or standard burst tries, it was ineffective at improving the cache-efficiency of the standard BST. The cache performance of these data structures is shown in Figure 15.

Our results show that the standard burst trie is not always the fastest data structure for vocabulary accumulation, as claimed by Heinz et al. [2002]. For long strings that share long prefixes, the standard BST can be faster, though its high computational cost and order $O(N)$ worst case does not necessarily make it a practical choice. For example, as the number of distinct URLs increase to say 10 million, or with a substantial increase in the number of searches, the performance of the BST will likely deteriorate due to the excessive cost of binary search—as observed in the previous TREC experiments. The burst trie, however, is more scalable and can operate efficiently with large numbers of strings, as a result of removing shared prefixes that reduce both space and computational costs, and the use of bounded-size containers.

7.4. The Effectiveness of Move-to-Front on Arrays

Figure 16 compares the time required to self-search the array hash and array burst trie with and without move-to-front on access. There were no cases where move-to-front improved the performance of the array hash and array burst trie. As discussed in the previous TREC experiments, performing a move-to-front in an array is computationally expensive, due to string copying. Our results show that move-to-front is unnecessary for the array-based data structures, as any potential gains seem likely to be low.

7.5. Paging versus Exact-Fit Array Growth

The cost of constructing the array burst trie and array hash with and without paging is shown in Figure 17. The use of paging led to consistent improvements for the array burst trie, which was up to 11% faster (or about 1.7 seconds) than exact-fit. Exact-fit involves growing a dynamic array on every insertion by copying it into a new larger space. Paging reduces the amount of copying to save time but at a small cost in space. Paging was not particularly useful for the array hash, apart from the slight improvement observed under heavy load. In all, the savings offered by the paging technique in these experiments were small, making the more space-efficient exact-fit growth policy preferable.

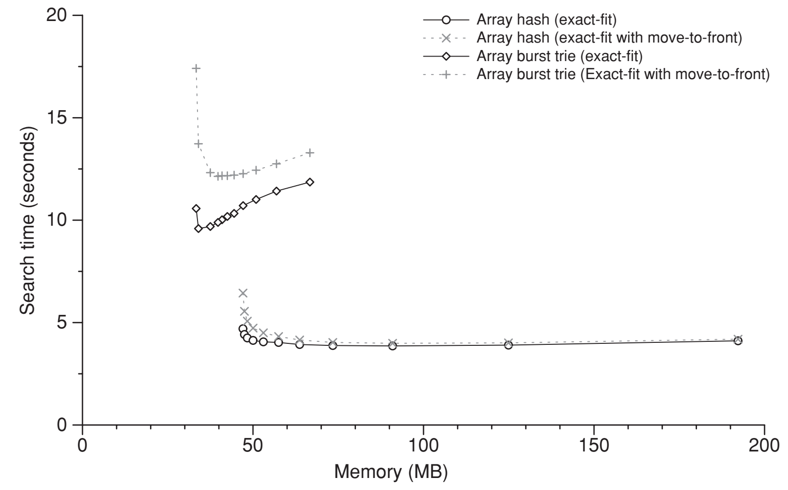


Fig. 16. The time and space required to self-search the array hash and array burst trie with and without move-to-front on access, using the URLS dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence was extended to include 128, 256, and 512 strings. The hash tables commenced with 2^{15} slots, doubling up to 2^{24}.

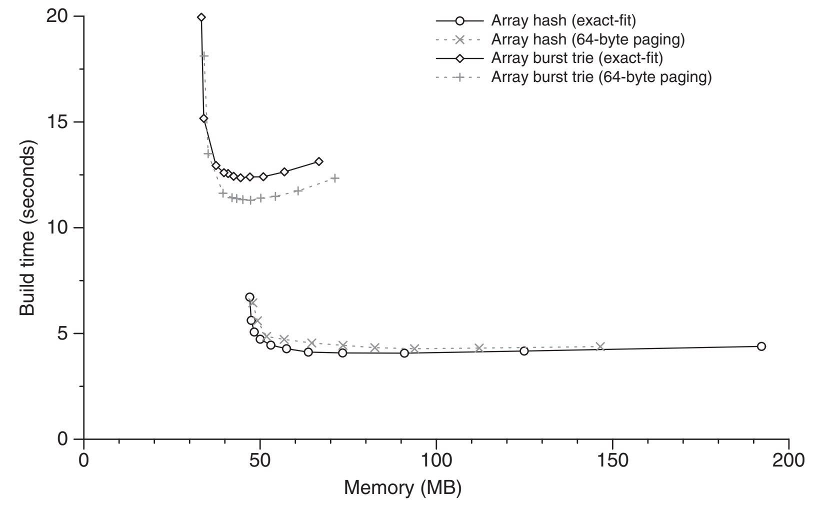


Fig. 17. The time and space required to build the array hash and array burst trie with and without paging, using the URLS dataset. Paging grows an array in 64-byte chunks, in contrast to exact-fit. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence was extended to include 128, 256, and 512 strings. The hash tables commenced with 2^{15} slots, doubling up to 2^{24}.

7.6. Performance without Hardware Prefetch

We repeat the previous skew search experiment on the Pentium III processor, to observe the value of eliminating pointers on a slower machine with no hardware prefetch. We consider the three best data structures—the hash table, burst trie, and BST—using

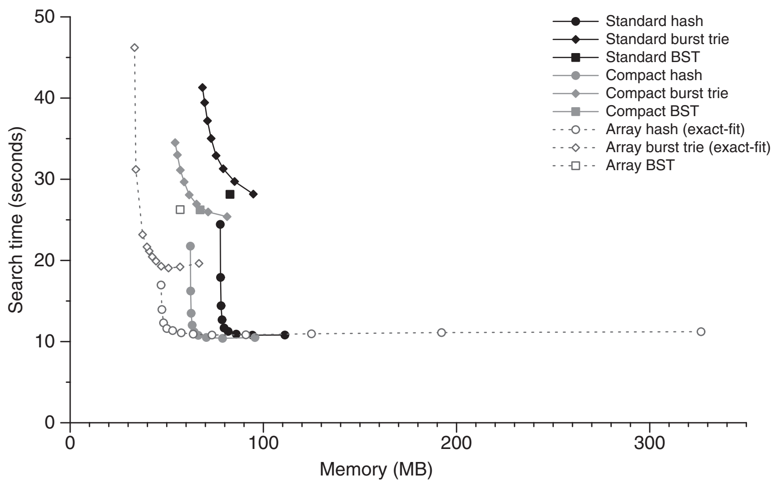


Fig. 18. The time and space required to self-search the data structures, using the URLS dataset on the Pentium III processor with no hardware prefetch. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence was extended to include 128, 256, and 512 strings. The hash tables commenced with 2^{15} slots, doubling up to 2^{24}.

their standard, compact, and array-based representations, and measure the time and space required to self-search using the URLS dataset. Results are shown in Figure 18.

The relative performance between the hash table, burst trie, and BST was consistent with previous experiments. The standard, compact, and array BSTs remained faster to access than the standard and compact burst tries, while the array burst trie displayed strong gains in performance—approaching the speed of the array hash while consuming the least amount of space. The hash tables remained the fastest data structures, with the array hash being consistently faster and smaller than the standard hash table. The compact hash was slightly faster than the array hash due to move-to-front on access, but required more space. These results show that the absence of hardware prefetch and the use of a slower processor with smaller cache lines—which can increase the cache-miss rate—has minimal impact on the relative performance between the array-based hash table, burst trie, and BST.

These array-based data structures exhibit superior performance even with no hardware prefetch as a result of good spatial access locality. On access, an array is scanned once from left to right, which maximizes the use of the respective CPU cache line that stores the portion of the array. This will likely lead to fewer cache misses, as reflected in the timings shown in Figure 18. As we access a node in a linked list, however, it will occupy only a small portion of a cache line; the remaining space is likely to contain junk data, since the nodes of a linked list can be physically scattered in main memory. As a consequence, cache-line utilization is reduced leading to more cache misses relative to the array-based data structures.

8. DISTINCT DATA

We then used the DISTINCT dataset for construction and self-search. This dataset contains no repeated strings, and thus every insertion requires that the data structure be fully traversed (i.e., the respective path of the data structure is fully traversed, such as the slot of a hash table). With no string repetitions, move-to-front on access is

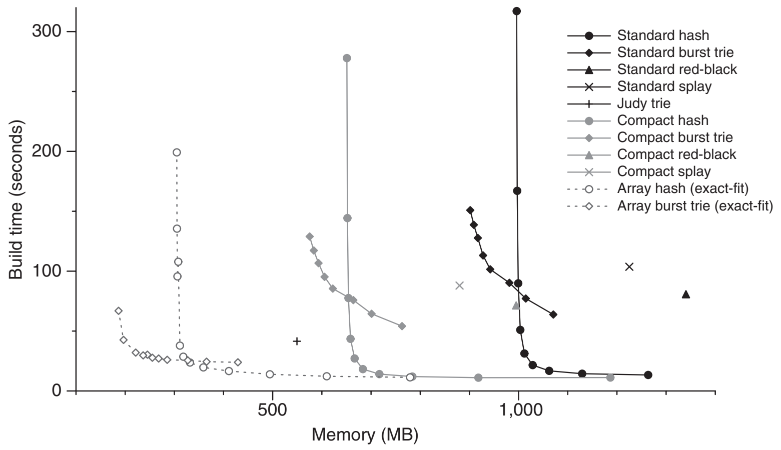


Fig. 19. The time and space required to build the data structures, using the DISTINCT dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence was extended to include 128, 256, and 512 strings. The hash tables commenced with 2^{15} slots, doubling up to 2^{26}.

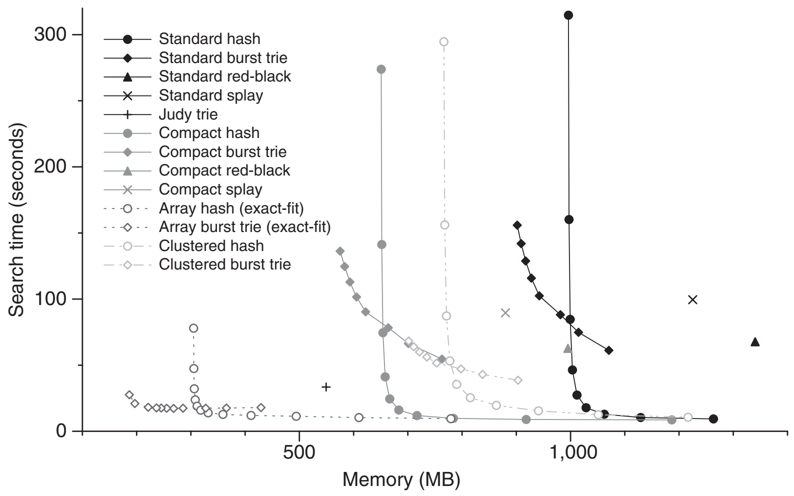


Fig. 20. The time and space required to self-search the data structures, using the DISTINCT dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence was extended to include 128, 256, and 512 strings. The hash tables commenced with 2^{15} slots, doubling up to 2^{26}.

rendered ineffective for both chains and arrays. Results for construction and self-search are shown in Figures 19 and 20.

8.1. Hash Tables

The difference in performance between the array and chaining methods is startling. The time required to construct and self-search the standard, clustered, compact, and array hash tables are shown in Table IV. This is an artificial case, but it highlights the fact that random memory accesses are highly inefficient. With only 2^{15} slots, for

Table IV. Elapsed Time (in Seconds) Required When the DISTINCT Dataset Is Used to Construct and Self-Search the Variants of Hash Table

	Num. of	Array				
	Slots	Page	Exact	Clustered	Compact	Standard
Construction	2^{15}	114.5	199.1	—	2,082.0	2,380.6
	2^{16}	106.4	135.5	—	1,055.5	1,209.0
	2^{17}	99.8	95.6	—	539.7	620.2
	2^{18}	68.3	78.8	—	277.8	317.1
	2^{19}	31.1	37.8	—	144.3	167.0
	2^{20}	19.6	28.6	—	77.5	89.8
	2^{21}	17.6	23.7	—	43.5	50.9
	2^{22}	16.3	19.6	—	27.1	31.2
	2^{23}	14.4	16.6	—	18.2	21.6
	2^{24}	12.5	13.9	—	14.0	16.7
	2^{25}	12.0	12.2	—	12.0	14.4
	2^{26}	11.3	11.4	—	11.0	13.3
	2^{27}	11.8	10.9	—	11.1	13.5
Self-search	2^{15}	—	77.9	1,310.7	2,076.5	2,370.0
	2^{16}	—	47.4	567.1	1,050.8	1,201.7
	2^{17}	—	32.1	294.6	535.4	612.7
	2^{18}	—	23.7	156.0	273.8	314.7
	2^{19}	—	19.0	87.2	141.2	160.1
	2^{20}	—	15.9	53.2	74.5	84.6
	2^{21}	—	13.9	35.4	41.1	46.3
	2^{22}	—	12.7	25.4	24.4	27.3
	2^{23}	—	11.9	19.5	16.0	17.8
	2^{24}	—	11.3	15.4	11.8	12.8
	2^{25}	—	10.3	12.5	9.7	10.4
	2^{26}	—	9.5	10.6	8.9	9.3
	2^{27}	—	8.9	10.7	8.6	13.2

example, the exact-fit array hash table was constructed in about 199 seconds, whereas the compact and standard chains required about 2,082 and 2,380 seconds, respectively. The use of paging further reduced the construction time of the array hash to only 114 seconds, a saving due to the lack of excessive copying. This speed is despite the fact that the average load factor is 878. As we anticipated in previous experiments, use of paging is effective with large numbers of insertions. As the load factor decreased, however, paging became less effective as fewer strings were inserted into each slot.

The results for self-search are similar to those for construction, with the array hash being up to 97% faster than the chained hash tables, with search time falling from over 2,370 seconds to under 80 seconds, while space simultaneously falls from 995MB to 304MB. Once again, increasing the number of slots allows the chained hash tables to be much faster, but the array hash remains competitive at all table sizes. The chained hash tables approach the efficiency of the array hash only when given surplus slots. For example, with 2^{27} slots, the compact hash table is by a small margin the fastest method, but required over 1,186MB with the equivalently-sized standard chain at 1,531MB. The array hash, however, achieved almost the same speeds using 2^{24} slots and a total of 494MB, a dramatic saving.

As anticipated from previous experiments, the clustered hash table displayed good performance with no skew in the data distribution, being up to 45% faster to self-search than the compact and standard hash tables. However, the clustered hash could only remain efficient under heavy load; as the average load factor decreased, so did its performance, until finally becoming slower than both the compact and standard hash tables. There were no cases where the clustered hash was superior to the array hash, which was up to 94% faster. Furthermore, with both pointers and nodes eliminated, a substantial amount of space is saved without any impact on performance. These results demonstrate that combining clustering with pointer elimination via the use of dynamic arrays, is by far more effective at exploiting cache than clustering alone.

The cache performance of the standard, clustered, compact, and array hash tables during self-search are shown in Figure 21. These results show a strong correlation with the self-search times reported in Figure 20. For example, the array hash sustained low L2 and TLB costs relative to its chained representations, and was thus faster to access under heavy load. In these experiments, move-to-front on access was rendered ineffective due to the absence of duplicate strings. As a consequence, searching the array hash incurred almost the same number of instructions per search as the equivalent chains. During construction, the array hash executed more instructions due to array resizing, but of which was greatly compensated with a high reduction in cache misses. As a result, the array hash could be built up to 92% faster than the chained hash tables while under heavy load—where arrays are longer and thus more expensive to resize. For uniform access distributions, the array hash is by far the most space-and time-efficient data structure when sorted access to strings is not required.

8.2. The Variants of BST, the TST, and Judy

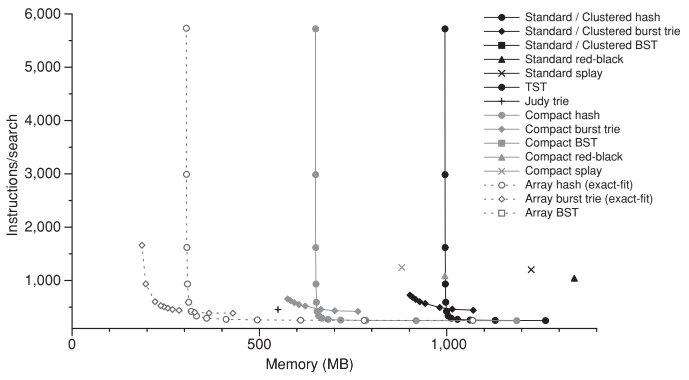
The BSTs were expensive to construct and self-search but remained competitive in space, with the array BST in particular, requiring less space than the compact and standard hash tables and burst tries. The time and space required to build and self-search the standard, clustered, compact, and array BST are shown in Table V.

At best, the array BST took 497 seconds to construct and 471 seconds for self-search, using 534MB of space. Although this is a considerable improvement over the standard BST, which required over 1,110MB of space and more than 740 seconds for both construction and self-search, the computational cost of binary search was expensive. To self-search the array BST, for example, over 34,000 instructions per search were executed. This is far more expensive than the array hash, which executed only 511 instructions per search, while using almost the same amount of space—2^{25} slots and 609MB. The clustered BST was not particularly effective at exploiting cache, showing only small gains in performance over the standard BST.

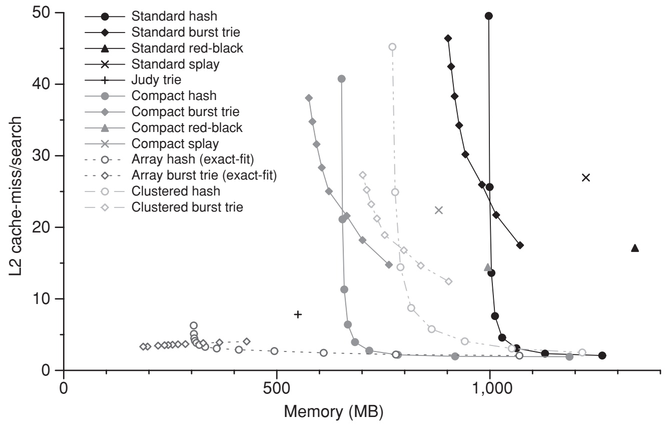
In previous experiments, the standard, compact, and array BSTs were competitive in speed, due to the presence of a skew distribution. This permitted frequently access tree paths to reside longer in cache, reducing cache misses which compensated for the use of an unbalanced structure. In these experiments, however, no access skew was present, and as a result, previously cached paths were likely to be evicted from cache prior to reuse. As a result and despite the improvements offered by the array BST, the BSTs were among the slowest data structures to access.

The standard splay and red-black trees, for both construction and self-search, performed relatively well compared to the BSTs. Splaying amortized the cost of access, reducing the number of nodes inspected per search, which in turn reduced the number of cache misses and instructions executed. The standard and compact splay trees, however, were slower to access than the standard and compact red-black trees, as a balanced structure is more effective when there is no access skew in the data distribution. The TST exhausted main memory, requiring almost 2.5GB of space, and as a consequence it required over 5,500 and 13,000 seconds to construct and self-search, respectively, due to the involvement of virtual memory. The Judy data structure was significantly faster than any of the trees, as a result of its efficient use of cache (Figure 21). Judy also required less space than the standard, clustered, and compact hash tables, but its performance remained markedly inferior in both time and space, when compared to the array hash and array burst tries.

(a) Instructions per search



(b) L2 cache-miss per search



(c) TLB cache-miss per search

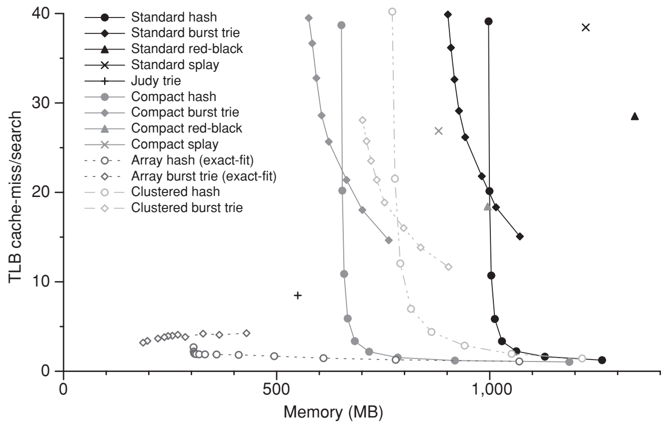


Fig. 21. The Instruction (a), L2 cache (b), and TLB (c) performance of the data structures when self-searched using the DISTINCT dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence was extended to include 128, 256, and 512 strings. The hash tables commenced with 2^{15} slots, doubling up to 2^{26}.

Table V. The Time and Space Required to Build and Self-Search a Standard, Clustered (search only), Compact, and Array BST, Using the DISTINCT Dataset

BST	Build (sec)	Self-search (sec)	Space (MB)
Array	497.1	471.1	534.7
Compact	594.8	569.3	764.9
Clustered	—	721.0	880.0
Standard	758.6	740.6	1,110.1

8.3. Burst Tries

The array burst trie showed substantial improvements over its compact and standard representations, requiring the least amount of space, at best a total space usage of 186MB—that is, 119MB less space than the original dataset—while access times were also reduced and approached those of the array hash table. With a container threshold of 128 strings, the array burst trie required only 221MB of space to achieve a self-search time of 18.1 seconds. The array hash, in contrast, needed over 311MB of space (or 2^{19} slots) to achieve a faster time. Paging was consistently faster than exact-fit, allowing the array burst trie to be constructed up to 24% faster (or around 15 seconds) with large containers, but at the cost of wasting up to 120MB of space with small containers.

The cache performance of the burst tries during self-search is shown in Figure 21. There is a strong correlation between cache performance and the time required for self-search. For example, the array burst trie was, in all cases, faster than the compact, clustered, and standard burst trie, because it sustained low L2 and TLB misses. Figure 21 shows that the cache efficiency of the array burst trie improves with an increase in container size, and yet, as shown in Figures 19 and 20, the array burst trie becomes slower to access relative to its smaller containers. This is caused by a high increase in instructions, a cost which is not entirely masked by the reduction of cache misses. The equivalent chained burst tries also incurred high instruction costs but made inefficient use of cache, which dominated their performance. Hence, as observed in previous experiments, in order to attain a fast and scalable burst trie, it is necessary to use containers that are both cache-efficient and conservative with the number of instructions they execute.

8.4. Performance without Hardware Prefetch

We repeat the previous self-search experiment to observe the impact on performance using the Pentium III, which does not employ hardware prefetch. We considered the three best data structures—the hash table, burst trie, and BST—using their standard, compact, and array-based representations. The results for the hash tables and burst tries are shown in Figure 22.

The array burst trie and array hash table were substantially faster to search than their chained representations, even without the aid of a hardware prefetcher. The array burst trie displayed the strongest gains—approaching the speed of the array hash while consuming the least amount of space.

With 2^{18} slots, the average load factor was about 110 strings per slot. In this case, the array hash took under 126 seconds to self-search, whereas the equivalent compact and standard hash tables required 498 and 587 seconds, respectively; that is, the array hash was up to 79% faster. Compared to its previous performance shown in Table IV, the array hash was only 15% slower on the Pentium III. The array BST also showed considerable improvement over its chained variants, being up to 24% faster; the standard BST required 1,918 seconds to self-search, which was reduced to 1,603 seconds by the compact BST, to a further 1,449 seconds by the array BST,

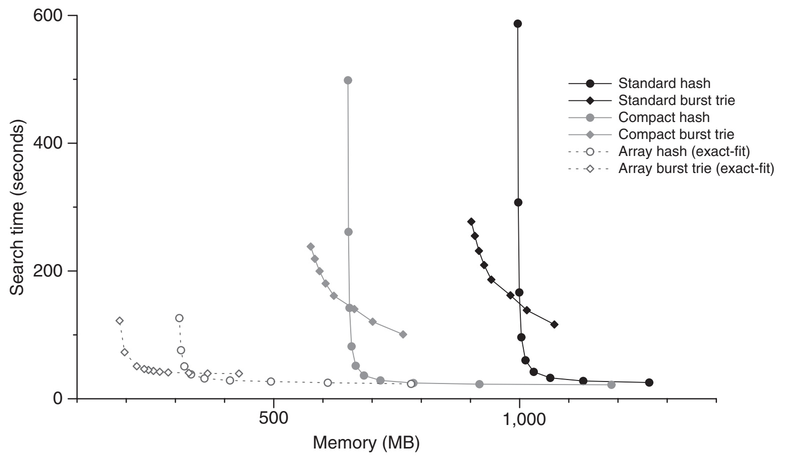


Fig. 22. The time and space required to self-search the data structures, using the DISTINCT dataset on the Pentium III with no hardware prefetch. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence was extended to include 128, 256, and 512 strings. The hash tables commenced with 2^{18} slots, doubling up to 2^{26}.

while achieving considerable reductions in space. These results demonstrate that our methods offer consistent and substantial improvements.

9. THE ADAPTIVE TRIE

We downloaded a high-quality implementation of an adaptive trie [Acharya et al. 1999], to compare its performance against our set of data structures including the cache-conscious hash table, burst trie, and binary search tree. The current implementation of the adaptive trie (for small alphabet sizes) is only compatible with lowercase alphabetic characters. We, therefore, filtered our DISTINCT and TREC datasets to remove all strings that contain either a uppercase or nonalphabetic character, forming the FILTERED DISTINCT and FILTERED TREC datasets, respectively. The FILTERED DISTINCT dataset contains 7,892,272 unique strings. We truncated the FILTERED TREC dataset to contain only the first 50 million strings from the total of 141,693,220 strings filtered. The time (in seconds) and space (in megabytes) required to build and self-search the data structures using the FILTERED DISTINCT dataset is shown in Figure 23 and Figure 24.

With no skew in the data distribution, the adaptive trie was faster to build and self-search than the trees and the standard and compact burst tries. Given enough space, however, the hash tables were superior. Similarly with a container threshold of less than 512 strings, the array burst trie was faster to build and self-search than the adaptive trie. The Judy trie and the TST were also slightly faster to build and self-search. Despite its competitive speed, however, the adaptive trie was the most space-intensive data structure, requiring over 900MB of memory, due to the space overhead of maintaining adaptive trie nodes. The space required by the adaptive trie was almost twice that of the TST and around a factor of 10 more than the array burst trie and array hash table.

Figure 25 shows the self-search performance of the adaptive trie using the FILTERED TREC dataset. We omit the cost of construction, as it was found to be similar to that of search. As claimed by Crescenzi et al. [2003], the adaptive trie performed poorly under skew access, requiring over 17 seconds to self-search, which places it among

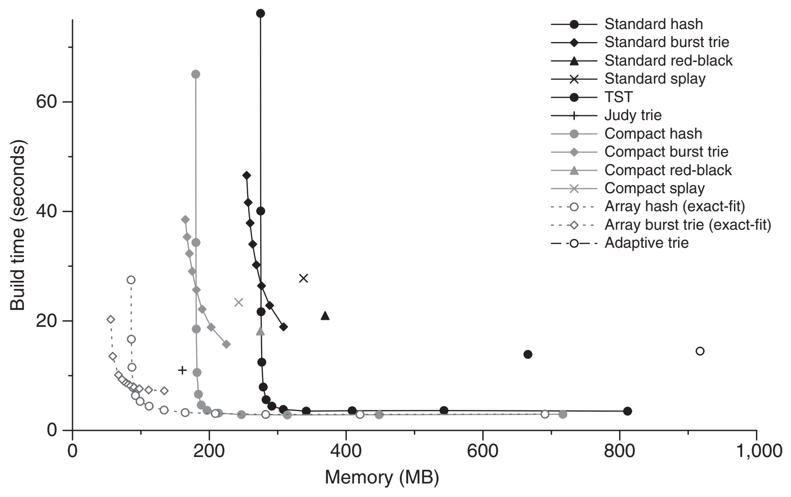


Fig. 23. The time and space required to build the adaptive trie, using the FILTERED DISTINCT dataset, described in Section 9. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence was extended to include 128, 256, and 512 strings. The hash tables commenced with 2^{16} slots, doubling up to 2^{24}.

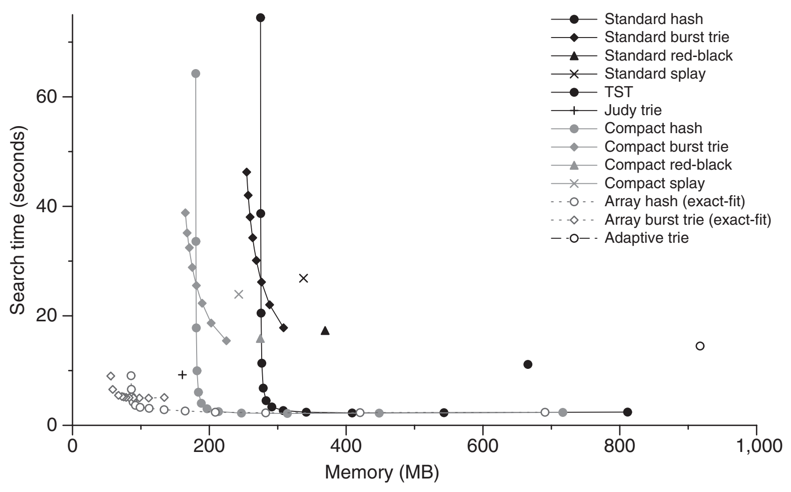


Fig. 24. The time and space required to self-search the adaptive trie, using the FILTERED DISTINCT dataset, described in Section 9. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence was extended to include 128, 256, and 512 strings. The hash tables commenced with 2^{16} slots, doubling up to 2^{24}.

the slowest data structures to access under skew, being only slightly faster than the red-black tree. Moreover, the adaptive trie required over 22MB of memory, which is more space than required by the TST. These results demonstrate that the adaptive trie is not a practical choice for managing a large set of strings in memory, due to its high space requirements and poor skew performance.

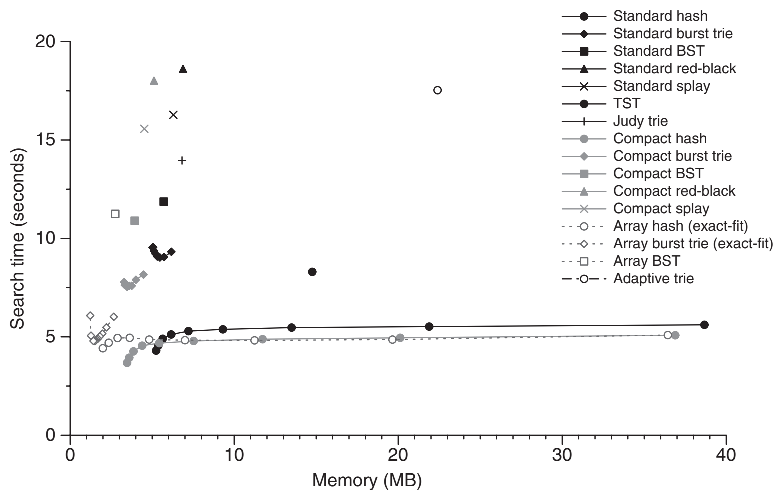


Fig. 25. The time and space required to self-search the adaptive trie, using the FILTERED TREC dataset, described in Section 9. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence was extended to include 128, 256, and 512 strings. The hash tables commenced with 2^{16} slots, doubling up to 2^{24}.

10. IMPLICIT CLUSTERED CHAINS

A clustered chain ensures that the nodes of a linked list are stored contiguously in main memory—in order of allocation—to improve spatial access locality. However, clustering does not eliminate pointers. Chilimbi [1999] suggests eliminating the next-node pointers of homogeneous nodes in a chain to form an implicit clustered chain where nodes are accessed via arithmetic offsets. As half of the pointers are eliminated, we expect better cache and space utilization over pointer-based clustering. However, implicit chains cannot support move-to-front on access, but as we show, this has little impact on performance due to a further reduction in cache misses. Several researches have also improved the cache-efficiency of software by applying similar implicit clustering techniques [Ghoting et al. 2006; Badawy et al. 2004; Luk and Mowry 1996].

In this experiment, we construct a standard hash table using the DISTINCT and TREC datasets. We then convert the standard chains that are assigned to each slot into implicit chains, to compare the time and space required to self-search the implicit hash table against the clustered, compact, standard, and array hash tables. The results are shown in Figure 26 and Figure 27.

With half the pointers eliminated, the implicit hash table is more space-efficient than the clustered hash table and is almost as space-efficient as the compact hash table. The implicit hash table requires more space than a compact hash due to the initial 8-byte allocation overhead per slot and the extra 4-byte null pointer per slot, which is used as a list delimiter.

The implicit hash table was consistently faster to self-search than the clustered hash table using the DISTINCT dataset (Figure 26). However, the relative difference between the two was small, with the implicit hash being only up to 19% faster than the clustered hash table. The implicit and clustered hash tables were faster to access than the compact and standard hash tables, but only under heavy load. Once the load factor fell below seven strings per slot (that is, using more than 2^{22} slots), both the implicit and clustered hash tables were slower to access than the compact and standard hash

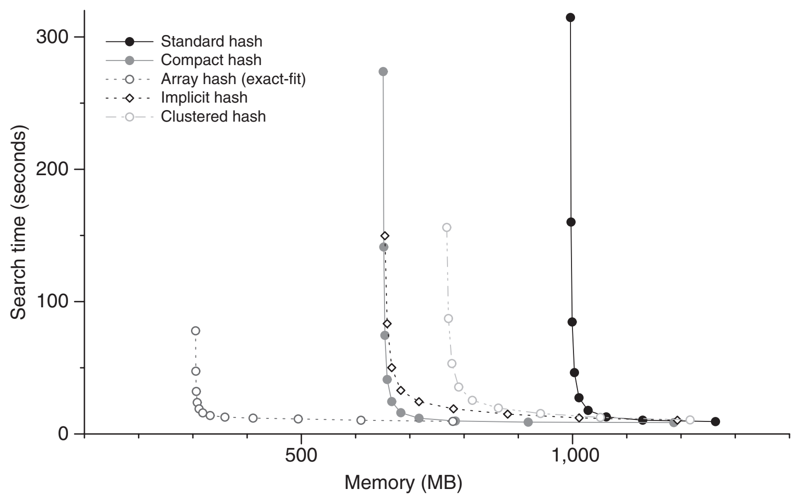


Fig. 26. The time and space required to self-search the implicit hash table, using the DISTINCT dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence was extended to include 128, 256, and 512 strings. The hash tables commenced with 2^{16} slots, doubling up to 2^{24}.

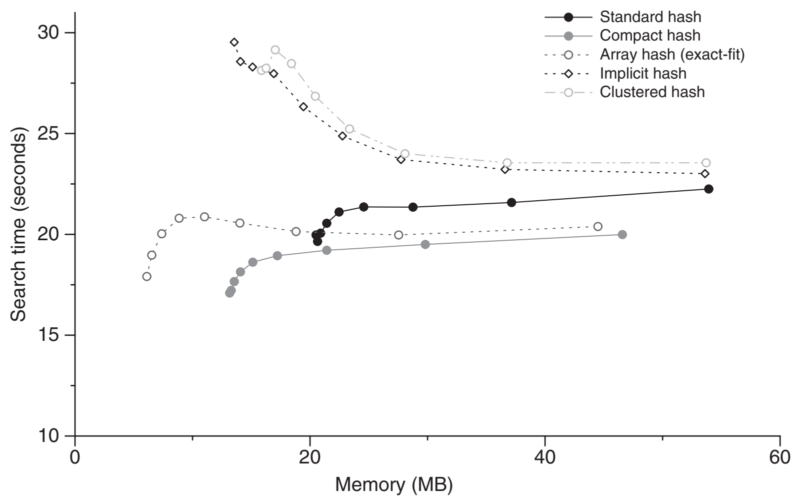


Fig. 27. The time and space required to self-search the implicit hash table, using the TREC dataset. The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence was extended to include 128, 256, and 512 strings. The hash tables commenced with 2^{16} slots, doubling up to 2^{24}.

tables. The array hash was the fastest and most compact data structure to access, being up to 93% faster than the implicit hash table when under heavy load, while simultaneously requiring less space.

These results demonstrate the value of combining clustering with pointer elimination through the use of dynamic arrays. Traversing a large implicit chain can incur up to two cache misses per string: one to access the string pointer and another to access its string.

Table VI. Elapsed time (in seconds) Required to Self-Search the Standard, Compact, Clustered, Implicit, and Array-Based Hash Tables, Using the DISTINCT Dataset

Num. of Slots	Array	Implicit	Clustered	Compact	Standard
2^{15}	77.9	1,063.0	1,310.7	2,076.5	2,370.0
2^{16}	47.4	533.8	567.1	1,050.8	1,201.7
2^{17}	32.1	279.6	294.6	535.4	612.7
2^{18}	23.7	149.7	156.0	273.8	314.7
2^{19}	19.0	83.3	87.2	141.2	160.1
2^{20}	15.9	50.1	53.2	74.5	84.6
2^{21}	13.9	32.9	35.4	41.1	46.3
2^{22}	12.7	24.3	25.4	24.4	27.3
2^{23}	11.9	18.9	19.5	16.0	17.8
2^{24}	11.3	14.9	15.4	11.8	12.8
2^{25}	10.3	12.1	12.5	9.7	10.4
2^{26}	9.5	10.3	10.6	8.9	9.3
2^{27}	8.9	9.8	10.7	8.6	13.2

In a dynamic array, in contrast, a single cache-miss will prefetch an entire cache-line of strings, and with no pointers to follow, hardware prefetch can greatly reduce the number of cache misses incurred as the array is scanned. The difference in time (in seconds) required to self-search the standard, compact, clustered, implicit, and array hash tables, using the DISTINCT dataset, is shown in Table VI.

Figure 27 shows the skew self-search performance of the standard, compact, clustered, implicit, and array hash tables, using the TREC dataset. The implicit hash table was slightly faster to access than the clustered hash table, but not when under heavy load due to the absence of move-to-front on access. Moreover, the implicit hash table showed relatively poor performance when compared to the standard, compact, and array hash tables, being up to 45% slower to access. Although the array hash did not employ move-to-front on access, it remained substantially faster than the implicit hash, due to the elimination of both string and node pointers, which led to a high reduction in cache misses.

11. VOCABULARY ACCUMULATION

Our next experiment measures the time and space required by the standard, compact, and array hash tables for the task of accumulating the vocabulary of a large text collection in memory. We only consider the hash tables in this experiment, as they are the fastest data structures. Similarly, we do not consider the clustered hash table because of its poor performance under skew access. To conduct the experiment, the hash tables were modified to maintain 4 bytes of satellite data that are stored before each string (in the array and compact-chain hash tables). The satellite data is used maintain a 4-byte string occurrence counter. When a string is initially stored in a hash table, its counter is set to 1. Thereafter, every time the string matches a query (as the hash table is being built), its counter is incremented by 1. The results of accumulating the vocabulary of the TREC dataset are shown in Figure 28, along with comparable measures of performance without satellite data.

With 4 bytes of satellite data associated with each string, fewer strings can reside in cache and as a consequence, more cache misses are likely to occur during traversal. The presence of satellite data in a standard or compact hash table had only a small impact on performance when under heavy load. The array hash with satellite data, in contrast, was up to 9% slower than the array hash without satellite data. The array hash can minimize the number of cache misses incurred by eliminating the traversal of pointers (other than slot pointers). However, this also implies that any increase in cache misses caused by the reduction of cache-line utilization (such as the

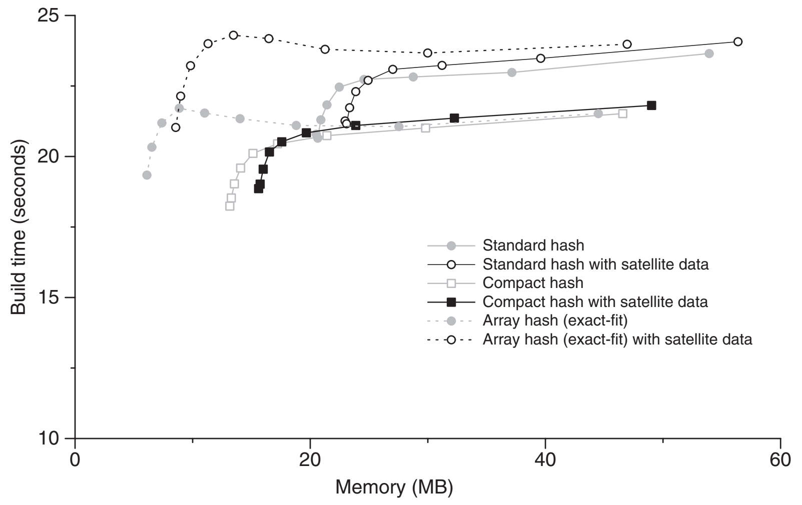


Fig. 28. The time and space required to accumulate the vocabulary of the TREC dataset. Unlike the previous TREC experiments, these hash tables associate 4 bytes of data (satellite data) per string, which are used to count the number of string occurrences encountered as the hash tables are built. Comparable figures are also presented with hash tables that have no satellite data, and hence, simply maintain strings. The points on the graph represent slots used by the hash tables. An increase in slots requires more memory. The hash tables commenced with 2^{15} slots, doubling up to 2^{23}.

space occupied by satellite data), is likely to cause a higher impact on performance than chaining. That is, the chaining hash tables incur cache misses due to pointer traversals, which are costly, but as a result, can dampen the impact on performance when satellite data is maintained, as observed. A further factor to consider is the cost of maintaining unaligned satellite entries in the array hash table. That is, by interleaving integers with strings, some integers may be stored outside a word-boundary, forcing an extra bus cycle on access, which can impact performance. In addition, unaligned access to integers may be prohibited on some computer architectures, such as a Sun UltraSPARC. In such cases, word padding is required to ensure that satellite data in buckets remain word-aligned.

Nonetheless, with only 4 bytes of satellite data per string, the relative difference in time and space between the standard, compact, and array hash tables, remained similar to previous results, with the array hash being faster and more space-efficient than the standard hash table when under heavy load. The impact of satellite data in string hash tables, however, is likely to increase with an increase in the size of satellite data. In cases where the satellite data occupies a significant portion of the cache-line size, storing it in a separate location is likely to benefit performance, particularly for the array hash.

12. ALTERNATIVE HARDWARE: SUN ULTRASPARC SERVER

We compare the performance (with respect to both time and space) of the standard chain hash table, standard-chain burst trie, the array hash table and array burst trie, along with the TST, on a Sun UltraSPARC Server running a Solaris (release 5.10) operating system. The server has 16GB of RAM with two 1.33GHz UltraSPARC III processors. Our intention here is to provide a brief demonstration on the effectiveness of our techniques on an alternative computing architecture (that is, other than an Intel). The results are shown in Figure 29, which illustrates the time taken to

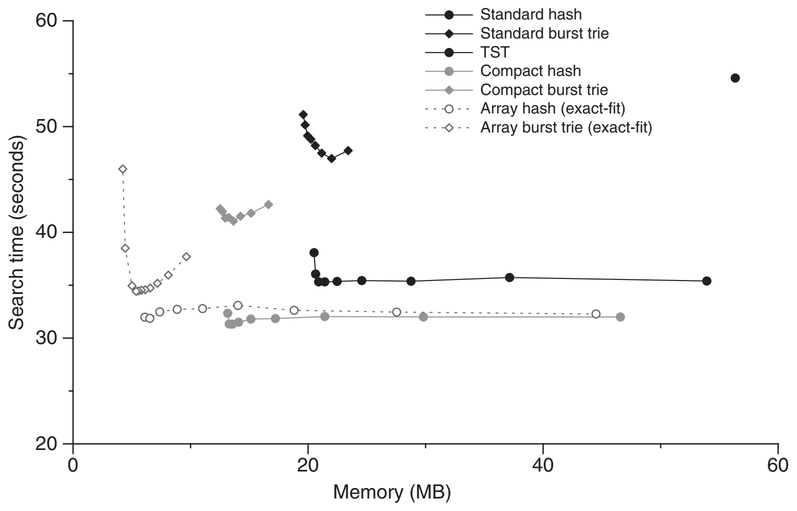


Fig. 29. The time and space required to self-search the data structures, using the TREC dataset on a Sun UltraSPARC Server (running Solaris 5.10). The points on the graph represent the container thresholds for the burst trie and the slots used by the hash tables. Smaller containers (or more slots) require more memory. The container thresholds for the chained burst tries were varied by intervals of 10 from 30 to 100. For the array burst trie, the sequence was extended to include 128, 256, and 512 strings. The hash tables commenced with 2^{16} slots, doubling up to 2^{24}.

self-search using our TREC dataset. As expected, the array burst trie and array hash were superior with respect to both time and space to their chained variants.

13. SUMMARY

In this article, we experimentally explored the performance of well-known string data structures, namely the hash table, burst trie, variants of BST, TST, the adaptive trie, and the Judy data structure. We compared the time, space, and cache performance of these data structures for the task of storing and retrieving large sets of strings in-memory. The fastest data structures were found to be the hash table, burst trie, and BST. However, our results show that the standard representation of these structures—a linked list of fixed-sized nodes consisting of a string pointer and a node pointer—is neither cache- nor space-efficient. A well-known technique for improving the use of cache is clustering, which stores the nodes of a list contiguously in memory. However, the clustered representations of the hash table, burst trie, and BST were, in most cases, inferior to their standard representations and were only effective when accessed with no skew in the data distribution.

In every case, replacing the standard chain with a compact-chain—a simple alternative where the fixed-length string pointer is replaced with the variable-length string—proved faster and smaller. Eliminating the chains altogether by storing the sequence of strings in a contiguous array that is dynamically resized as strings are inserted, showed further substantial gains in performance on both current and older machine architectures. The array hash table, for example, achieved up to a 97% improvement in speed over the standard hash table. Similarly, the array burst trie displayed strong and consistent improvements, being up to 89% faster than the standard burst trie. In most cases, the array burst trie approached the speed of hashing, while maintaining sorted access to containers. The array BST also displayed considerable improvements, being up to 36% faster than the standard and clustered BST. The compact BST was, in most cases, marginally faster than the array BST but required more space.

Compared to compact chaining, dynamic arrays can yield substantial further benefits. In the best case, the space overhead of the array hash can be reduced by a factor of around 200 to less than a single bit per string, while access speed is consistently faster than under standard, clustered, and compact chaining. The array burst trie also displayed similar benefits, with space falling from around 200 bits per string, to no space overhead, while access speed remained substantially faster than the standard, clustered, and compact burst tries. The array BST also displayed similar gains, requiring only a third of the space of the standard BST, while being faster to access. These results are an illustration of the importance of considering cache in algorithm design. The standard-chain hash table, burst trie, and (in most cases) the BST were previously shown to be the most efficient structures for managing strings, but we have greatly reduced their total space consumption while simultaneously reducing access time.

13.1. Recommendations for Parameters

Having observed the performance of the array hash and array burst trie on different string distributions and sizes, we recommend the following parameters that should offer good—but not necessarily an optimal—performance in practice. For the array hash and assuming skew in the data distribution, it is preferable to maintain a high average load factor (where $s > m$, s being the number of unique keys and m representing the number of hash slots); for example, by maintaining between 2^{15} and 2^{18} slots.

For uniform access distributions—where typically millions of distinct strings are processed—the number of slots can be increased to reduce the average load factor, though, from our experience, it is preferable to keep the average load factor above 1 (i.e., $s > m$); as an example, by maintaining between 2^{23} and 2^{24} slots when processing tens of millions of keys, which should achieve a good balance between time and space. A further reduction in load factor (where m approaches or exceeds s) may improve speed, but at a cost in space. Also, since the array hash scales well under heavy load, employing a dynamic hash table resizing scheme (i.e., to adjust the number of slots based on load) would likely yield only small gains in performance. For the array burst trie, our results show that a container threshold between of 128 to 256 strings should provide good performance for both skew and uniform data distributions.

ACKNOWLEDGMENTS

We thank the anonymous reviewers of this article and software.

REFERENCES

Acharya, A., Zhu, H., and Shen, K. 1999. Adaptive algorithms for cache-efficient trie search. In *Proceedings of the Workshop on Algorithm Engineering and Experiments*. SIAM, Philadelphia, PA, 296–311.

Agarwal, R. 1996. A super scalar sort algorithm for RISC processors. In *Proceedings of the International Conference on the Management of Data*. ACM, New York, 240–246.

Aggarwal, A. 2002. Software caching vs. prefetching. In *Proceedings of the International Symposium on Memory Management.* ACM, New York, 157–162.

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. 1974. *The Design and Analysis of Computer Algorithms*, 1st Ed. Addison-Wesley, Boston, MA.

Allen, R. and Kennedy, K. 2001. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, San Francisco, CA.

Arge, L., Bender, M. A., Demaine, E., Leiserson, C., and Mehlhorn, K. 2005. Abstracts collection. In *Proceedings of the Cache-Oblivious and Cache-Aware Algorithms Seminar*. Schloss Dagstuhl, Wadern, Germany.

Arge, L., Bender, M. A., Demaine, E. D., Holland-Minkley, B., and Munro, J. I. 2002. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the Symposium on Theory of Computing*. ACM, New York, 268–276.

Arge, L., Brodal, G., and Fagerberg, R. 2005. Cache-oblivious data structures. In *Handbook on Data Structures and Applications*, D. P. Mehta and S. Sahni, Eds. CRC Press, Boca Raton, FL, 34–41.

ASKITIS, N. 2007. Efficient data structures for cache architectures. Ph.D. thesis, School of Computer Science and Information Technology, RMIT University, Australia. http://www.naskitis.com.

ASKITIS, N. 2009. Fast and compact hash tables for integer keys. In *Proceedings of the 32nd Australasian Computer Science Conference.* Australian Computer Society, Sydney, Australia, 101–110.

ASKITIS, N. AND SINHA, R. 2007. HAT-trie: A cache-conscious trie-based data structure for strings. In *Proceedings of the of the 30th Australasian Computer Science Conference.* Australian Computer Society, Sydney, Australia, 97–105.

ASKITIS, N. AND SINHA, R. 2010. Engineering scalable, cache and space efficient tries for strings. *Int. J. Very Large Datab. 19*, 5, 633–660. http://www.springerlink.com/content/86574173183j6565/.

ASKITIS, N. AND ZOBEL, J. 2005. Cache-conscious collision resolution in string hash tables. In *Proceedings of the String Processing and Information Retrieval Symposium*. Springer, Berlin, 91–102.

ASKITIS, N. AND ZOBEL, J. 2008. B-tries for disk-based string management. *Int. J. Very Large Datab. 18*, 1, 157–179. http://www.springerlink.com/content/x7545u2g85675u17.

BACON, D. F., GRANHAM, S. L., AND SHARP, O. J. 1994. Compiler transformation for high-performance computing. *ACM Comput. Surv. 26*, 4, 345–420.

BADAWY, A. A., AGGARWAL, A., YEUNG, D., AND TSENG, C. 2001. Evaluating the impact of memory system performance on software prefetching and locality optimization. In *Proceedings of the International Conference on Super-Computing*. ACM, New York, 486–500.

BADAWY, A. A., AGGARWAL, A., YEUNG, D., AND TSENG, C. 2004. The efficacy of software prefetching and locality optimizations on future memory systems. *J. Instruct. Level Parall. 6,* 7.

BAER, J. AND CHEN, T. 1991. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the Conference on Super-Computing*. ACM, New York, 176–186.

BAER, J. AND CHEN, T. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput. 44*, 5, 609–623.

BASKINS, D. 2004. A 10-minute description of how Judy arrays work and why they are so fast. judy.sourceforge.net/doc/shop_interm.pdf.

BELL, J. AND GUPTA, G. 1993. An evaluation of self-adjusting binary search tree techniques. *Softw. Pract. Experi. 23*, 4, 369–382.

BENDER, M., BRODAL, G. S., FAGERBERG, R., GE, D., HE, S., HU, H., IACONO, J., AND LOPEZ-ORTIZ, A. 2003. The cost of cache-oblivious searching. In *Proceedings of the Symposium on the Foundations of Computer Science.* IEEE, Los Alamitos, CA, 271–282.

BENDER, M. A., DEMAINE, E. D., AND FARACH-COLTON, M. 2000. Cache-oblivious B-trees. In *Proceedings of the Foundations of Computer Science*. IEEE, Los Alamitos, CA, 399–409.

BENDER, M. A., DEMAINE, E. D., AND FARACH-COLTON, M. 2002. Efficient tree layout in a multilevel memory hierarchy. In *Proceedings of the European Symposium on Algorithms*. Springer, Berlin, 165–173.

BENDER, M. A., DUAN, Z., IACONO, J., AND WU, J. 2004. A locality-preserving cache-oblivious dynamic dictionary. *J. Algor. 53*, 2, 115–136.

BENDER, M. A., FARACH-COLTON, M., FINEMAN, J. T., FOGEL, Y. R., KUSZMAUL, B. C., AND NELSON, J. 2007. Cache-oblivious streaming B-trees. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*. ACM, New York, 81–92.

BENDER, M. A., FARACH-COLTON, M., AND KUSZMAUL, B. C. 2006. Cache-oblivious string Btrees. In *Proceedings of the Symposium on Principles of Database Systems*. ACM, New York, 233–242.

BENTLEY, J. L. AND SEDGEWICK, R. 1997. Fast algorithms for sorting and searching strings. In *Proceedings of the Symposium on Discrete Algorithms*. ACM, New York, 360–369.

BERGER, E. D., ZORN, B. G., AND MCKINLEY, K. S. 2002. Reconsidering custom memory allocation. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications.* ACM, New York, 1–12.

BEYLS, K. AND D'HOLLANDER, E. H. 2006. Intermediately executed code is the key to find refactorings that improve temporal data locality. In *Proceedings of the Conference on Computing Frontiers*. ACM, New York, 373–382.

BRODAL, G. AND FAGERBERG, R. 2006. Cache-oblivious string dictionaries. In *Proceedings of the Symposium on Discrete Algorithms*. ACM, New York, 581–590.

BRODAL, G. S., FAGERBERG, R., AND JACOB, R. 2002. Cache oblivious search trees via binary trees of small height. In *Proceedings of the Symposium on Discrete Algorithms*. ACM, New York, 39–48.

BURGER, D., GOODMAN, J. R., AND KÄGI, A. 1996. Memory bandwidth limitations of future microprocessors. In *Proceedings of the International Symposium on Computer Architecture.* ACM, New York, 78–89.

Calder, B., Krintz, C., John, S., and Austin, T. 1998. Cache-conscious data placement. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, New York, 139–149.

Callahan, D., Kennedy, K., and Porterfield, A. 1991. Software prefetching. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 40–52.

Carr, S., McKinley, K. S., and Tseng, C. 1994. Compiler optimizations for improving data locality. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 252–262.

Chilimbi, T. M. 1999. Cache-conscious data structures–design and implementation. Ph.D. thesis, Computer Sciences Department, University of Wisconsin-Madison.

Chilimbi, T. M., Davidson, B., and Larus, J. R. 1999. Cache-conscious structure definition. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, New York, 13–24.

Chilimbi, T. M., Hill, M. D., and Larus, J. R. 1999. Cache-conscious structure layout. In *Proceedings of the Conference on Programming Language Design and Implementation.* ACM, New York, 1–12.

Chilimbi, T. M., Hill, M. D., and Larus, J. R. 2000. Making pointer-based data structures cache conscious. *Computer 33*, 12, 67–74.

Chilimbi, T. M. and Shaham, R. 2006. Cache-conscious co-allocation of hot data streams. In *Proceedings of the Conference on Programming Language Design and Implementation.* ACM, New York, 252–262.

Clement, J., Flajolet, P., and Vallee, B. 1998. The analysis of hybrid trie structures. In *Proceedings of the Symposium on Discrete Algorithms*. ACM, New York, 531–539.

Clement, J., Flajolet, P., and Vallee, B. 2001. Dynamic sources in information theory: A general analysis of trie structures. *Algorithmica 29*, 1/2, 307–369.

Collins, J., Sair, S., Calder, B., and Tullsen, D. M. 2002. Pointer cache assisted prefetching. In *Proceedings of the Annual International Symposium on Microarchitecture*. ACM, New York, 62–73.

Comer, D. 1979. Heuristics for trie index minimization. *ACM Trans. Datab. Syst. 4*, 3, 383–395.

Crescenzi, P., Grossi, R., and Italiano, G. F. 2003. Search data structures for skewed strings. In *Proceedings of the 2nd International Workshop Experimental and Efficient Algorithms*. Springer, Berlin, 81–96.

de la Briandais, R. 1959. File searching using variable length keys. In *Proceedings of the Western Joint Computer Conference*. 295–298.

Dongarra, J., London, K., Moore, S., Mucci, S., and Terpstra, D. 2001. Using PAPI for hardware performance monitoring on linux systems. In *Proceedings of the Conference on Linux Clusters: The HPC Revolution.* IEEE, Los Alamitos, CA.

Dundas, J. and Mudge, T. 1997. Improving data cache performance by pre-executing instruction under a cache miss. In *Proceedings of the Conference on Supercomputing*. ACM, New York, 176–186.

Esakov, J. and Weiss, T. 1989. *Data Structures: An Advanced Approach Using C*, 1st Ed. Prentice-Hall, Upper Saddle River, NJ.

Ferragina, P. and Grossi, R. 1999. The string B-tree: A new data structure for string search in external memory and its applications. *J. ACM 46*, 2, 236–280.

Fredkin, E. 1960. Trie memory. *Comm. ACM 3*, 9, 490–499.

Frias, L., Petit, J., and Roura, S. 2006. Lists revisited: Cache-conscious stl lists. In *Proceedings of the Workshop on Experimental Algorithms.* Springer, Berlin, 121–133.

Frigo, M., Leiserson, C., Prokop, H., and Ramachandran, S. 1999. Cache-oblivious algorithms. In *Proceedings of the Symposium on the Foundations of Computer Science.* IEEE, Los Alamitos, CA, 285.

Fu, J. W. C., Patel, J. H., and Janssens, B. L. 1992. Stride directed prefetching in scalar processors. *SIGMICRO Newsl. 23*, 1-2, 102–110.

Ghoting, A., Buehrer, G., Parthasarathy, S., Kim, D., Nguyen, A., Chen, Y., and Dubey, P. 2006. Cache-conscious frequent pattern mining on modern and emerging processors. *Int. J. Very Large Datab. 16*, 1, 77–96.

Gonnet, G. H. and Baeza-Yates, R. 1991. *Handbook of Algorithms and Data Structures: In Pascal and C*, 2nd Ed. Addison-Wesley, Boston, MA.

Graefe, G., Bunker, R., and Cooper, S. 1998. Hash joins and hash teams in Microsoft SQL server. In *Proceedings of the International Conference on Very Large Databases.* Morgan Kaufmann, San Francisco, CA, 86–97.

Granston, E. D. and Wijshoff, H. A. G. 1993. Managing pages in shared virtual memory systems: getting the compiler into the game. In *Proceedings of the International Conference on Supercomputing.* ACM, New York, 11–20.

HALATSIS, C. AND PHILOKYPROU, G. 1978. Pseudo-chaining in hash tables. *Comm. ACM 21*, 7, 554–557.

HALLBERG, J., PALM, T., AND BRORSSON, M. 2003. Cache-conscious allocation of pointer-based data structures revisited with HW/SW prefetching. In *Proceedings of the 2nd Annual Workshop on Duplicating, Deconstructing, and Debunking.* http://www.ece.wisc.edu/~wddd/2003/01_hallberg.pdf.

HAN, J., PEI, J., AND YIN, Y. 2000. Mining frequent patterns without candidate generation. In *Proceedings of the International Conference on Management of Data*. ACM, New York, 1–12.

HANDY, J. 1998. *The Cache Memory Book*, 2nd Ed. Academic Press Professional, Inc., San Diego, CA.

HANSEN, W. J. 1981. A cost model for the internal organization of B+-tree nodes. *ACM Trans. Program. Lang. Syst. 3*, 4, 508–532.

HARMAN, D. 1995. Overview of the second text retrieval Conference (TREC-2). *Inf. Process. Manage. 31*, 3, 271–289.

HEILEMAN, G. L. AND LUO, W. 2005. How caching affects hashing. In *Proceedings of the Workshop on Algorithm Engineering and Experiments.* SIAM, Philadelphia, PA, 141–154.

HEINZ, S., ZOBEL, J., AND WILLIAMS, H. E. 2002. Burst tries: A fast, efficient data structure for string keys. *ACM Trans. Inf. Syst. 20*, 2, 192–223.

HENNESSY, J. L. AND PATTERSON, D. A. 2003. *Computer Architecture: A Quantitative Approach,* 3rd Ed. Morgan Kaufmann, San Francisco, CA.

HEWLETT-PACKARD. 2001. Programming with Judy: C language Judy version 4.0. Tech. rep., HP Part Number: B6841-90001.

HILL, M. D. AND SMITH, A. J. 1989. Evaluating associativity in CPU caches. *IEEE Trans. Comput. 38*, 12, 1612–1630.

HINTON, G., SAGER, D., UPTON, M., BOGGS, D., CARMEAN, D., KYKER, A., AND ROUSSEL, P. 2001. The microarchitecture of the Pentium 4 processor. *Intel Technol. J. 5*, 1–13.

HORTON, I. 2006. *Beginning C: From Novice to Professional*, 4th Ed. Apress, Berkeley, CA.

HUGHES, C. J. AND ADVE, S. V. 2005. Memory-side prefetching for linked data structures for processor-in-memory systems. *J. Parall. Distrib. Comput. 65*, 4, 448–463.

INTEL. 2007. Intel 64 and IA-32 architectures software developer's manual. Vol. 1: Basic architecture. Tech. rep., Intel Developer's Manual. http://www.intel.com/products/processor/manuals/index.htm.

JACQUET, P. AND SZPANKOWSKI, W. 1991. Analysis of digital tries with markovian dependency. *IEEE Trans. Inf. Theory 37*, 5, 1470–1475.

JONGE, W. D. AND TANENBAUM, A. S. 1987. Two access methods using compact binary trees. *IEEE Trans. Softw. Eng. 13*, 7, 799–810.

JOSEPH, D. AND GRUNWALD, D. 1997. Prefetching using Markov predictors. In *Proceedings of the International Symposium on Computer Architecture*. IEEE, Los Alamitos, CA, 252–263.

KARLSSON, M., DAHLGREN, F., AND STENSTROM, P. 2000. A prefetching technique for irregular accesses to linked data structures. In *Proceedings of the Symposium on High-Performance Computer Architecture.* IEEE, Los Alamitos, CA, 206–217.

KERNS, D. R. AND EGGERS, S. J. 1993. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *Proceedings of the Conference on Programming Language Design and Implementation.* ACM, New York, 278–289.

KNUTH, D. E. 1998. *The Art of Computer Programming: Sorting and Searching*, 2nd Ed. Vol. 3. Addison-Wesley Longman, Redwood City, CA.

KOWARSCHIK, M. AND WEISS, C. 2003. An overview of cache optimization techniques and cache-aware numerical algorithms. In *Algorithms for Memory Hierarchies*, U. Meyer, P. Sanders, and J. F. Sibeyn, Eds. Dagstuhl Research Seminar, Schloss Dagstuhl, Germany, 213–232.

KUMAR, P. 2003. Cache oblivious algorithms. In *Algorithms for Memory Hierarchies*, U. Meyer, P. Sanders, and J. F. Sibeyn, Eds. Dagstuhl Research Seminar, Schloss Dagstuhl, Germany, 193–212.

LADNER, R. E., FORTNA, R., AND NGUYEN, B. 2002. A comparison of cache aware and cache oblivious static search trees using program instrumentation. In *Experimental Algorithmics: From Algorithm Design to Robust and Efficient Software*, R. Fleischer, B. Moret, and E. M. Schmidt, Eds. Springer, Berlin, 78–92.

LAMARCA, A. AND LADNER, R. 1996. The influence of caches on the performance of heaps. *ACM J. Exp. Algorithmics 1*, 4.

LARSON, P. 1982. Performance analysis of linear hashing with partial expansions. *ACM Trans. Datab. Syst. 7*, 4, 566–587.

LATTNER, C. AND ADVE, V. 2005. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proceedings of the Conference on Programming Language Design and Implementation.* ACM, New York, 129–142.

LEBECK, A. R. 1999. Cache conscious programming in undergraduate computer science. In *Proceedings of the Technical Symposium on Computer Science Education*. ACM, New York, 247–251.

LEUNG, S. AND ZAHORJAN, J. 1995. Optimizing data locality by array restructuring. Tech. rep. TR-95-09-01, Department of Computer Science and Engineering, University of Washington.

LIPASTI, M. H., SCHMIDT, W. J., KUNKEL, S. R., AND ROEDIGER, R. R. 1995. Spiad: Software prefetching in pointer and call-intensive environments. In *Proceedings of the Annual International Symposium on Microarchitecture*. ACM, New York, 252–263.

LOSHIN, D. 1998. *Efficient Memory Programming*, 1st Ed. McGraw-Hill Professional, New York.

LUK, C. 2001. Tolerating memory latency through software-controlled pre-execution in simultaneous multi-threading processors. In *Proceedings of the International Symposium on Computer Architecture.* IEEE, Los Alamitos, CA, 40–51.

LUK, C. AND MOWRY, T. C. 1996. Compiler-based prefetching for recursive data structures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 222–233.

MANJIKIAN, N. AND ABDELRAHMAN, T. 1995. Array data layout for the reduction of cache conflicts. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems*. ISCA.

MARTINEZ, C. AND ROURA, S. 1998. Randomized binary search trees. *J. ACM 45*, 2, 288–323.

MCCABE, J. 1965. On serial files with relocatable records. *Oper. Res. 13*, 609–618.

MCCREIGHT, E. M. 1976. A space-economical suffix tree construction algorithm. *J. ACM 23*, 2, 262–271.

MEYER, U., SANDERS, P., AND SIBEYN, J. F., EDS. 2003. *Algorithms for Memory Hierarchies, Advanced Lectures*. Dagstuhl Research Seminar, Schloss Dagstuhl, Germany.

MORET, B. AND SHAPIRO, H. 1994. An empirical assessment of algorithms for constructing a minimum spanning tree. *Monographs in Discrete Mathematic and Theoretical Computer Science 15*, 99–117.

MORRISON, D. R. 1968. Patricia: A practical algorithm to retrieve information coded in alphanumeric. *J. ACM 15*, 4, 514–534.

MOWRY, T. C. 1995. Tolerating latency through software-controlled data prefetching. Ph.D. thesis, Computer Systems Laboratory, Stanford University.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*, 1st Ed. Morgan Kaufmann, San Francisco, CA.

MUNRO, J. I. AND CELIS, P. 1986. Techniques for collision resolution in hash tables with open addressing. In *Proceedings of the Fall Joint Computer Conference.* ACM, New York, 601–610.

NAZ, A., REZAEI, M., KAVI, K., AND SWEANY, P. 2004. Improving data cache performance with integrated use of split caches, victim cache, and stream buffers. In *Proceedings of the Workshop of Memory Performance*. ACM, New York, 41–48.

Newell, A. and Tonge, F. M. 1960. An introduction to information processing language V. *Comm. ACM 3*, 4, 205–211.

PATTERSON, D., ANDERSON, T., CARDWELL, N., FROMM, R., KEETON, K., KAZYRAKIS, C., THOMAS, R., AND YELICK, K. 1997. A case for intelligent RAM. *IEEE Micro 17*, 2, 34–44.

PETERSON, W. W. 1957. Open addressing. *IBM J. Res. Dev. 1*, 130–146.

PUGH, W. 1990. Skip lists: A probabilistic alternative to balanced trees. *Comm. ACM 33*, 6, 668–676.

RAHMAN, N. 2002. Algorithms for hardware caches and TLB. In *Algorithms for Memory Hierarchies*, U. Meyer, P. Sanders, and J. F. Sibeyn, Eds. Dagstuhl Research Seminar, Schloss Dagstuhl, Germany, 171–192.

RAHMAN, N., COLE, R., AND RAMAN, R. 2001. Optimized predecessor data structures for internal memory. In *Proceedings of the International Workshop on Algorithm Engineering.* Springer, Berlin, 67–78.

RAMAKRISHNA, M. V. AND ZOBEL, J. 1997. Performance in practice of string hashing functions. In *Proceedings of the Symposium on Databases Systems for Advanced Applications*. IEEE, Los Alamitos, CA, 215–224.

RAMESH, R., BABU, A. J. G., AND KINCAID, J. P. 1989. Variable-depth trie index optimization: Theory and experimental results. *ACM Trans. Datab. Syst. 14*, 1, 41–74.

RAO, J. AND ROSS, K. A. 1999. Cache conscious indexing for decision-support in main memory. In *Proceedings of the International Conference on Very Large Databases*. ACM, New York, 78–89.

RAO, J. AND ROSS, K. A. 2000. Making B+-trees cache conscious in main memory. In *Proceedings of the International Conference on the Management of Data*. ACM, New York, 475–486.

RATHI, A., LU, H., AND HEDRICK, G. E. 1991. Performance comparison of extendible hashing and linear hashing techniques. *ACM SIGSMALL/PC Notes 17*, 2, 19–26.

RIVERA, G. AND TSENG, C. 1998. Data transformation for eliminating conflict misses. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, New York, 38–49.

ROMER, T. H., OHLRICH, W. H., KARLIN, A. R., AND BERSHAD, B. N. 1995. Reducing TLB and memory overhead using online superpage promotion. In *Proceedings of the International Symposium on Computer Architecture*. ACM, New York, 176–187.

ROSENBERG, A. L. AND STOCKMEYER, L. J. 1977. Hashing schemes for extendible arrays. *J. ACM 24,* 2, 199–221.

ROTH, A., MOSHOVOS, A., AND SOHI, G. S. 1998. Dependence based prefetching for linked data structures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 115–126.

ROTH, A. AND SOHI, G. S. 1999. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the International Symposium on Computer Architecture*. ACM, New York, 111–121.

RUBIN, S., BERNSTEIN, D., AND RODEH, M. 1999. Virtual cache line: A new technique to improve cache exploitation for recursive data structures. In *Proceedings of the International Conference on Compiler Construction*. Springer, Berlin, 259–273.

SARWATE, D. V. 1980. A note on universal classes of hash functions. *Inf. Process. Lett. 10*, 1, 41–45.

SEDGEWICK, R. 1998. *Algorithms in C, Parts 1-4: Fundamentals, Data structures, Sorting, and Searching,* 3rd Ed. Addison-Wesley, Boston, MA.

SEVERANCE, D. G. 1974. Identifier search mechanisms: A survey and generalized model. *ACM Comput. Surv. 6*, 3, 175–194.

SHANLEY, T. 2004. *The Unabridged Pentium 4: IA32 Processor Genealogy,* 1st Ed. Addison-Wesley, Boston, MA.

SILVERSTEIN, A. 2002. Judy IV shop manual. judy.sourceforge.net/doc/shop_interm.pdf.

SINHA, R., RING, D., AND ZOBEL, J. 2006. Cache-efficient string sorting using copying. *ACM J. Exp. Algorithmics 11*, 1.2.

SLEATOR, D. D. AND TARJAN, R. E. 1985. Self-adjusting binary search trees. *J. ACM 32*, 3, 652–686.

SMITH, A. J. 1982. Cache memories. *ACM Comput. Surv. 14*, 3, 473–530.

STOUTCHININ, A., AMARAL, J. N., GAO, G. R., DEHNERT, J. C., JAIN, S., AND DOUILLET, A. 2001. Speculative prefetching of induction pointers. In *Proceedings of the International Conference on Compiler Construction*. Springer, Berlin, 289–303.

SUSSENGUTH, E. 1963. Use of tree structures for processing files. *Comm. ACM 6*, 5, 272–279.

SZPANKOWSKI, W. 1991. On the height of digital trees and related problems. *Algorithmica 6*, 2, 256–277.

TORP, K., MARK, L., AND JENSEN, C. S. 1998. Efficient differential timeslice computation. *IEEE Trans. Knowl. Data Eng. 10*, 4, 599–611.

TRUONG, D. N., BODIN, F., AND SEZNEC, A. 1998. Improving cache behavior of dynamically allocated data structures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques.* IEEE, Los Alamitos, CA,–329.

VANDERWIEL, S. P. AND LILJA, D. J. 2000. Data prefetch mechanisms. *ACM Comput. Surv. 32*, 2, 174–199.

VITTER, J. S. 1983. Analysis of the search performance of coalesced hashing. *J. ACM 30*, 2, 231–258.

WANG, Z., BURGER, D., MCKINLEY, K. S., REINHARDT, S. K., AND WEEMS, C. C. 2003. Guided region prefetching: A cooperative hardware/software approach. In *Proceedings of the International Symposium on Computer Architecture*. ACM, New York, 388–398.

WILLIAMS, H. E., ZOBEL, J., AND HEINZ, S. 2001. Self-adjusting trees in practice for large text collections. *Softw. Practice Exper. 31*, 10, 925–939.

YANG, C., LEBECK, A. R., TSENG, H., AND LEE, C. 2004. Tolerating memory latency through push prefetching for pointer-intensive applications. *ACM Trans. Archit.Code Optim. 1,* 4, 445–475.

YOTOV, K., ROEDER, T., PINGALI, K., GUNNELS, J., AND GUSTAVSON, F. 2007. An experimental comparison of cache-oblivious and cache-conscious programs. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*. ACM, New York, 93–104.

ZHAO, Q., RABBAH, R., AND WONG,W. 2005. Dynamic memory optimization using pool allocation and prefetching. *ACM SIGARCH Comput. Archit. News 33,* 5, 27–32.

ZOBEL, J., HEINZ, S., AND WILLIAMS, H. E. 2001. In-memory hash tables for accumulating text vocabularies. *Inf. Process. Lett. 80*, 6, 271–277.

Received September 2007; revised November 2010; accepted November 2010