HAT-trie: A Cache-conscious Trie-based Data Structure for Strings

Nikolas Askitis **Ranjan Sinha**

School of Computer Science and Information Technology,
RMIT University, Melbourne 3001, Australia.
Email: {naskitis,rsinha}@cs.rmit.edu.au

Abstract

Tries are the fastest tree-based data structures for managing strings in-memory, but are space-intensive. The burst-trie is almost as fast but reduces space by collapsing trie-chains into buckets. This is not however, a cache-conscious approach and can lead to poor performance on current processors. In this paper, we introduce the HAT-trie, a cache-conscious trie-based data structure that is formed by carefully combining existing components. We evaluate performance using several real-world datasets and against other high-performance data structures. We show strong improvements in both time and space; in most cases approaching that of the cache-conscious hash table. Our HAT-trie is shown to be the most efficient trie-based data structure for managing variable-length strings in-memory while maintaining sort order.

Keywords: String, tree, trie, hash table, cache, in-memory.

1 Introduction

Trie-based data structures offer rapid access to strings while maintaining reasonable worst-case performance. They are successful in a variety of applications, such as text compression (Bell, Cleary & Witten 1990) and dictionary management (Aoe, Morimoto & Sato 1992), but are space-intensive. Measures need to be taken to reduce their space consumption, if they are to remain feasible for maintaining large sets of strings.

The most successful procedure for reducing the space of a trie structure has been the burst-trie (Heinz, Zobel & Williams 2002). The burst-trie is an in-memory string data structure that can significantly reduce the number of trie nodes maintained by as much as 80%, at little to no cost in access speed. It achieves this by selectively collapsing chains of trie nodes into small buckets of strings that share a common prefix. When a bucket has reached its capacity, it is *burst* into smaller buckets that are parented by a new trie node.

Buckets are internally unsorted, but they can be accessed in lexicographic order. The burst-trie can therefore provide fast sorted access to strings; buckets can be rapidly sorted on demand. It is currently the fastest and most compact in-memory tree structure that can handle large volumes of variable-length strings, in sort order. The effectiveness of the burst-trie has been demonstrated by its basis in burstsort (Sinha, Ring & Zobel 2006, Sinha & Zobel 2004), which is, to the best of our knowledge, the fastest in-memory data structure for the sorting of large sets of strings.

Although fast, the burst-trie is not cache-conscious. Like many in-memory data structures, it is efficient in a setting where all memory accesses are of equal cost. In practice however, a single random access to memory typically incurs many hundreds of clock cycles. Current processors therefore implement a hierarchy of small but fast caches between the CPU and main memory — that intercept and service memory requests whenever possible — to attempt to minimize the consultation of main memory (a cache-miss). It is of paramount importance to make the best possible use of these caches, to prevent severe performance bottlenecks that arise from excessive cache-misses. Programmers however, generally have no administrative control over these caches, yet this is usually not a problem in practice. Programs have been shown to make good use of cache through careful design that aims at improving *temporal* and *spatial* access locality. This is achieved through a variety of techniques that generally attempt to make memory access more regular or predictable.

Although space-intensive, tries can — to some extent — be cache-conscious. Trie nodes are small in size, improving the probability of frequently accessed trie-paths to reside within cache. The burst-trie however, represents buckets as linked lists which are known for their cache inefficiency. When traversing a linked list, the address of a child can not be known until the parent is processed. Known as the pointer-chasing problem, this hinders the effectiveness of hardware prefetchers (Yang, Lebeck, Tseng & Lee 2004), that attempt to reduce cache-misses by anticipating and loading data into cache ahead of the running program. This is the property underlying the efficiency of structures such as the cache-conscious array hash (Askitis & Zobel 2005), where the usual linked lists of a chaining hash table are replaced by dynamic arrays that are contiguously allocated in memory. Arrays exhibit stride access patterns that can be easily detected and exploited by hardware prefetchers.

Our motivation was to address the shortcomings of the burst-trie, by developing a cache-conscious variant that exploits the cache hierarchy used on modern processors. We introduce the *HAT-trie*, which combines the trie-index of the burst-trie with buckets that are represented as cache-conscious hash tables (Askitis & Zobel 2005). We present two variants of the HAT-trie that differ in the manner of how buckets are split: *hybrid* and *pure*. The former employs the B-trie splitting algorithm (Askitis & Zobel 2006) that reduces the number of buckets at little cost, by permitting multiple pointers to a bucket. The latter employs the *bursting* technique of the burst-trie, which is faster but usually requires more space.

We experimentally examined the performance of the HAT-tries, comparing them to a selection of data structures that are currently among the best in dynamic in-memory string management: the burst-trie, array hash, chained hash tables with move-to-front, and the *compact* binary search tree, which is a more cache-efficient variant that eliminates string pointers by storing strings within their respective nodes (Askitis & Zobel 2005). We used datasets acquired from real-world sources — containing variable-length strings with a range of characteristics — to compare space, time and cache efficiency of building and searching the data structures. To build and search, our HAT-tries were up to 80% faster than the burst-trie, with a (simultaneous) space reduction of up to 70%. The HAT-tries were more efficient than the optimized binary search tree, with the pure HAT-trie in particular, approaching, and in some cases surpassing, the performance of the cache-conscious array hash, which is currently the best for unsorted string management. These are strong results that further substantiate the importance of considering cache on pointer-intensive data structures, that are otherwise efficient.

2 Background

A trie node, named for its successful use in information re*trie*val (Fredkin 1960), is an array of pointers, one for each character in an alphabet. Each leaf node is the terminus of a chain of trie nodes representing a string (Knuth 1998). For string management, tries are fast with reasonable worst-case performance, and are valuable for applications such as data mining (Agrawal & Srikant 1994), dictionary and text processing (Aoe et al. 1992, Bentley & Sedgewick 1997), pattern matching (Flajolet & Puech 1986), and compression (Bell et al. 1990). Tries can be regarded as cache-conscious, as frequently accessed trie-paths may be cache-resident. Nonetheless, they are space-intensive, prohibiting use with large volumes of strings (Comer 1979, McCreight 1976).

Space can be conserved by reducing the number of trie nodes and by changing their structure. The compact trie (Bell et al. 1990) omits chains of tries that descend into a single leaf. Similarly, the Patricia trie (Sedgewick 1998) omits all chains without branches, not just those that lead to leaves. Tries can be represented as linked lists (Severance 1974), eliminating unused pointers. The ternary search tree (Bentley & Sedgewick 1997, Sedgewick 1998) can save space by using 3-way trie-nodes for less than, equal to, and greater than comparisons; reducing node size for sparse data. Trie compression (Al-Suwaiyel & Horowitz 1984), trie compaction (Maly 1976), and heuristics (Comer 1979) have also been applied. These techniques, however, save space at the expense of time and do not take the memory hierarchy into account. Acharya et al. (Acharya, Zhu & Shen 1999) addressed this issue by developing cache-efficient algorithms that choose between several representations of trie nodes. Although fast, space efficiency relative to previous methods is unclear for large sets of strings.

The burst-trie (Heinz et al. 2002) successfully reduced the number of trie nodes at little cost, by collapsing trie-chains into buckets that share a common prefix. Buckets — represented as linked lists with move-to-front on access (Knuth 1998) — are then selectively burst into smaller buckets that are parented by a new trie. Although computationally efficient, the burst-trie was not designed to exploit cache; linked lists are not cache-conscious structures (Yang et al. 2004). Consequentially, performance is jeopardized when used on modern cache-oriented processors.

Current processors attempt to reduce memory latency by prefetching items that are likely to be requested in the near future. Hardware prefetchers (Collins, Sair, Calder & Tullsen 2002, Yang et al. 2004) generate prefetch requests from information accumulated at runtime. The simplest implementation being a hardware-based stride prediction table (Fu, Patel & Janssens 1992), which works well with array-based applications. Similarly, software prefetchers such as software caching (Aggarwal 2002) and jump-pointers (Roth & Sohi 1999), attempt to hide latency by fetching data before it is needed. Nonetheless, the effectiveness of prefetching remains poor when applied to pointer-intensive data structures. The binary search tree, where nodes and their associated strings are likely scattered in main memory, is an example.

Careful data layout (Chilimbi, Hill & Larus 1999) and cache-conscious memory allocation (Badawy, Aggarwal, Yeung & Tseng 2001, Hallberg, Palm, & Brorsson 2003) have shown the best results at improving the cache usage of pointer-intensive data structures. These techniques address the cause of cache misses, being poor access locality, rather than the manifestation of cache misses. The cache-sensitive search tree (Rao & Ross 1999) and the cache-conscious B^+-tree (Rao & Ross 2000) are such examples, that exploit cache by changing the layout of nodes to eliminate (almost all) pointers to random memory. Nodes are contiguously allocated and are accessed through arithmetic offsets, improving spatial locality by making good use of hardware prefetchers (Berg 2002). These data structures are fast but remain expensive to update.

Based on the success of copying strings to array-based buckets in string sorting (Sinha et al. 2006), Askitis and Zobel (2005) replaced the linked lists of the chaining hash table (previously the best method for string hashing (Zobel, Williams & Heinz 2001)) with re-sizable buckets (arrays), forming the cache-conscious array hash. Arrays require more instructions for search, but compensate by eliminating the pointer-chasing problem through contiguous memory allocation. Performance gains of up to 96% over chained hashing were observed for both search and update costs, with space overheads at best, reduced to less than two bits per string, at no impact to speed. It is plausible that similar techniques can also lead to substantial gains for pointer-intensive data structures, such as the burst-trie.

Cache-oblivious data structures have received considerable attention in recent literature. These data structures aim to perform well on all levels of the memory hierarchy — including disk — without prior knowledge of the size and speed of each level (Frigo, Leiserson, Prokop & Ramachandran 1999, Kumar 2002). Brodal and Fagerberg for example, have recently proved the existence of a static (one that can not change once constructed) cache-oblivious string dictionary (Brodal & Fagerberg 2006). However, the study is from a theoretical perspective, and shows no empirical performance of speed and memory consumption, against well-known data structures; which is of value in practice. This is not uncommon, with studies demonstrating (in theory) that cache-oblivious structures can almost match the performance of optimized data structures (Bender, Brodal, Fagerberg, Ge, He, Hu, Iacono & Lopez-Ortiz 2003).

Similarly, the dynamic cache-oblivious B-tree (Bender, Demaine & Farach-Colton 2000) has been described in theory, but with no discussion of actual performance. The cache-oblivious dynamic

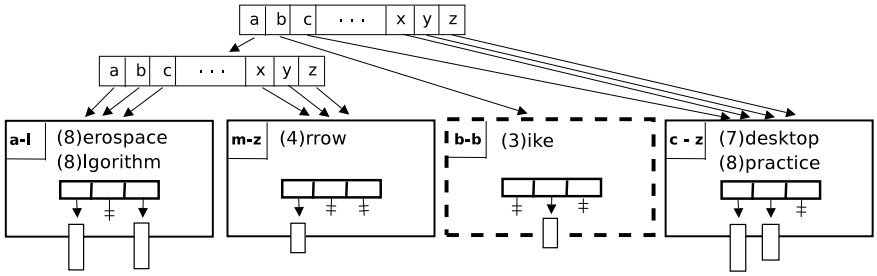

Figure 1: A hybrid HAT-trie, where buckets are split using B-trie splitting. A pure HAT-trie maintains only pure buckets that are burst, not split. Pure buckets are shown in dashed boxes. Strings are length-encoded (shown as brackets).

dictionary (Bender, Duan, Iacono & Wu 2002), which is a simplification of the cache-oblivious B-tree, is developed and evaluated against a B-tree, but only on a simulated memory hierarchy. Another example includes the cache-oblivious priority queue (Arge, Bender, Demaine, Holland-Minkley & Munro 2002). All of these data structures however, assume a uniform distribution in data and operations (Bender, Demaine & Farach-Colton 2002), which is typically not observed in practice.

There have been recent studies evaluating the practicality of these data structures (Ladner, Fortna & Nguyen 2002, Brodal, Fagerberg & Jacob 2002, Arge, Bender, Demaine, Leiserson & Mehlhorn 2004) with empirical results showing superior performance to conventional data structures, but not for those that have been tuned (both in memory consumption and time) to a specific memory hierarchy (Arge, Brodal & Fagerberg 2004). The data structures we present in this paper are not cache-oblivious, as they have been designed specifically to exploit cache between main memory and the CPU, and reside solely within (volatile) main memory. We omit comparisons of existing cache-oblivious data structures for our full paper.

3 The HAT-trie

The fastest data structures currently available for managing (storing and retrieving) variable-length strings in-memory, is the burst-trie (Heinz et al. 2002) and the chaining hash table with move-to-front on access (Zobel et al. 2001). These data structures are fast because they minimize the number of instructions required for search and update, but are not particularly cache-conscious. We know from literature that linked lists — used as buckets and as chains respectively — are the cause.

To address the bottlenecks of the burst-trie, we must significantly reduce the cost of trie traversal — being the number of trie nodes created — but more importantly, the cost of searching buckets, as these are currently the most expensive components to access. Such a reduction can only be achieved by changing the representation of buckets from linked lists, to large cache-conscious arrays. It is therefore attractive to consider structuring buckets as cache-conscious hash tables (Askitis & Zobel 2005). The advantage of using the array hash, as opposed to a simple array, is that buckets can scale much more efficiently, further reducing the number of trie nodes maintained.

This approach forms the basis of the HAT-trie, of which we studied two variants: *pure* and *hybrid*. These variants differ in the manner of how buckets are maintained and split. In the former, buckets contain strings that share only one prefix, which is removed. These buckets are classified as *pure* and are referenced by a single parent pointer. In the latter, buckets also share a single prefix, however, it is not removed; the last character of the prefix is stored along with the string. These buckets are classified as *hybrid*, and have more than one parent pointer. The hybrid HAT-trie can also maintain *pure* buckets. An example is shown in Figure 1.

The HAT-tries are built and searched in a top-down manner. A trie is accessed by following the pointer corresponding to the lead character of the query. Pointer traversal will consume the lead character unless a hybrid bucket is accessed. Short strings can be consumed (deleted) and are handled by setting an end-of-string flag in the respective trie or pure bucket. When needed, an empty pure bucket can be created to serve as an end-of-string flag. Given the situation where strings have associated data fields (which does not arise in our data), alternative approaches to storing consumed strings is to use an auxiliary data structure (Askitis & Zobel 2006). Buckets can also be modified to accommodate data fields efficiently (Askitis & Zobel 2005).

The HAT-trie begins as a single empty hybrid bucket, which is populated until full. Buckets do not maintain duplicates, hence an insertion occurs only on search failure. When a bucket is full, it is burst or split, depending on the type of HAT-trie. A pure HAT-trie, *bursts* a full bucket into at most A pure buckets that are parented by a new trie (A is the size of the alphabet). The strings are then distributed amongst the pure buckets, according to their lead character, which is removed. Strings can be consumed during the bursting procedure.

The hybrid HAT-trie however, *splits* buckets in two using the B-trie algorithm (Askitis & Zobel 2006), which we summarize. On split, a pure bucket is converted into a hybrid by creating a new parent trie with all pointers assigned to it. The old trie is pushed up as a grandparent and the splitting procedure continues as a hybrid. To split a hybrid, we find a suitable split-point that achieves good — preferably even — distribution, which may not always be possible. We access the strings in the bucket to accumulate the occurrence of every lead character. These occurrence counters are then traversed, in lexicographic order, to determine how many groups of strings to move, to acquire a distribution of at least 0.75 (number of strings moved divided by those that remain). This ratio was found to provide good space-efficiency for the b-trie. Once acquired, the split-point is chosen as the letter representing the current occurrence counter.

In the event where no strings remain, the last occurrence counter accessed is used as the split-point. This can result in the creation of an empty bucket. The two new buckets are then classified as pure or hybrid, based on split-point, and the strings are dis-

Dataset	Type	Distinct strings	String occs	Average length	Volume (MB) of distinct	Volume (MB) total
distinct	Text	28,772,169	28,772,169	9.59	304.56	304.56
trec	Text	612,219	177,999,203	5.06	5.68	1,079.46
urls	Non-text	1,287,597	9,997,487	30.93	45.82	318.56
genome	Non-text	262,084	31,623,000	9.0	3.8	316.23

Table 1: Characteristics of the datasets used in the experiments.

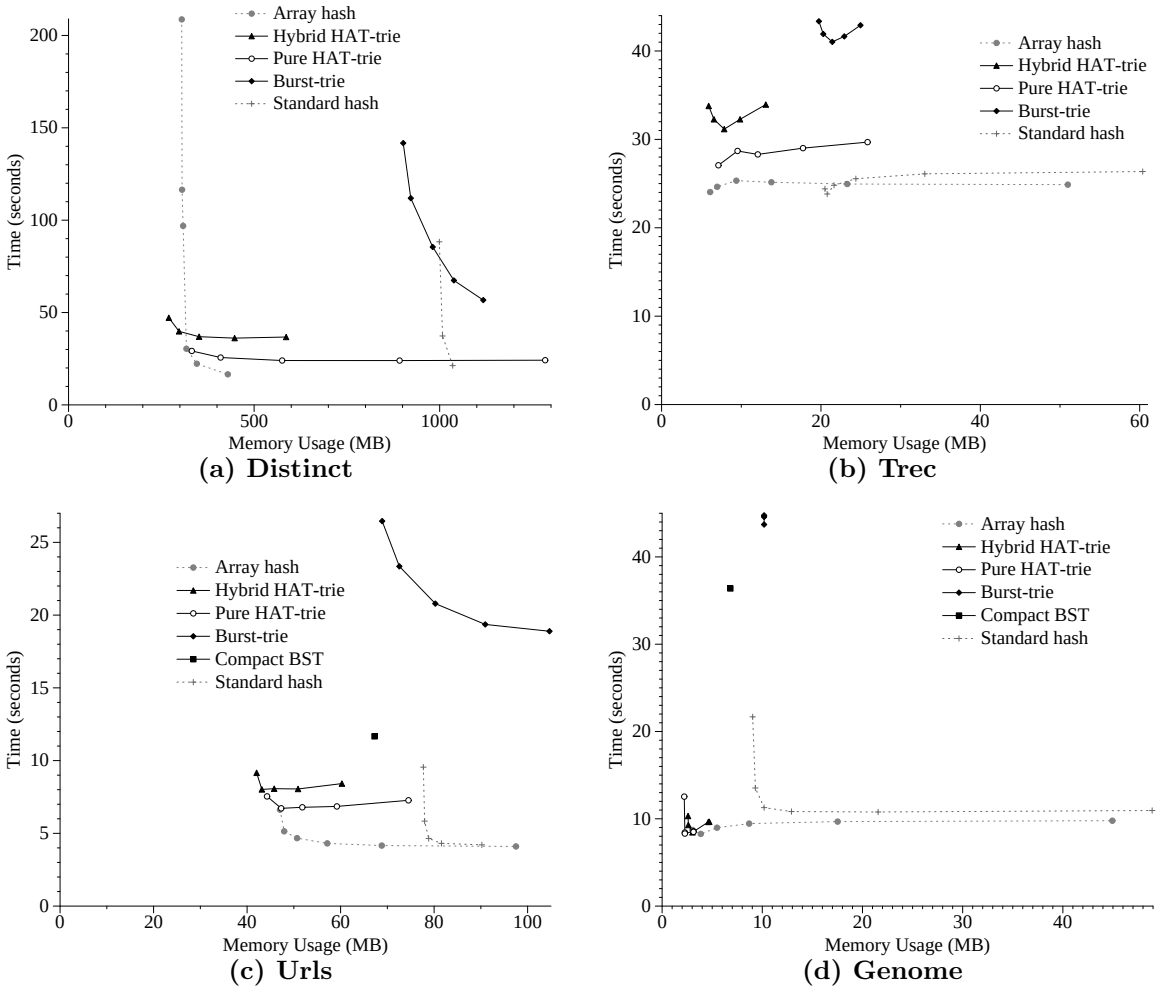

Figure 2: *Construction costs of datasets distinct, trec, urls and genome. Graph plots represent bucket thresholds (sizes). Smaller buckets require more memory.*

tributed accordingly. A string can be consumed during this process. Empty buckets are then deleted with their associated parent pointers set to null. The splitting procedure terminates only when both buckets do not exceed bucket capacity. Otherwise, the bucket that remains full is re-split.

Askitis and Zobel (2005) discuss in detail, the operations of the array hash. From an implementation perspective, we represent buckets as an array of $n+1$ word-length pointers, which are empty or point to their respective slot entries; a dynamic bucket (array) containing length-encoded strings. We reserve the first pointer for house-keeping information: the bucket type or character range, the end-of-string flag and the number of strings.

To perform an insertion or search on a bucket, the required slot is found by first hashing what remains of the query string (after traversing the tries), by using a fast bitwise hash function (Ramakrishna & Zobel 1997). Once the required slot is accessed, its array is searched on a per-character basis, with a mismatch prompting a skip to the next string in the array. Insertion only occurs when the string is not found or if the slot is empty. In such a case, the string is length-encoded and appended to the end of the array, by first growing (or initially creating) the array in an exact-fit manner, being the length of the string, which is cache-efficient.

The HAT-tries however, do impose a fixed space overhead per bucket, being the fixed number of slot pointers. Consequentially, some space is wasted when a bucket maintains only a few strings. It would be more efficient in this case, to represent an underloaded bucket as a simple array which is changed to an array hash when full. Alternatively, buckets can maintain a variable number of slots, which is altered according to frequency of access. This will save space, but at the cost of re-hashing buckets.

4 Experimental Design

We compared the speed (*elapsed* time) of construction, search and the memory requirements of our HAT-tries against that of the array hash, the binary

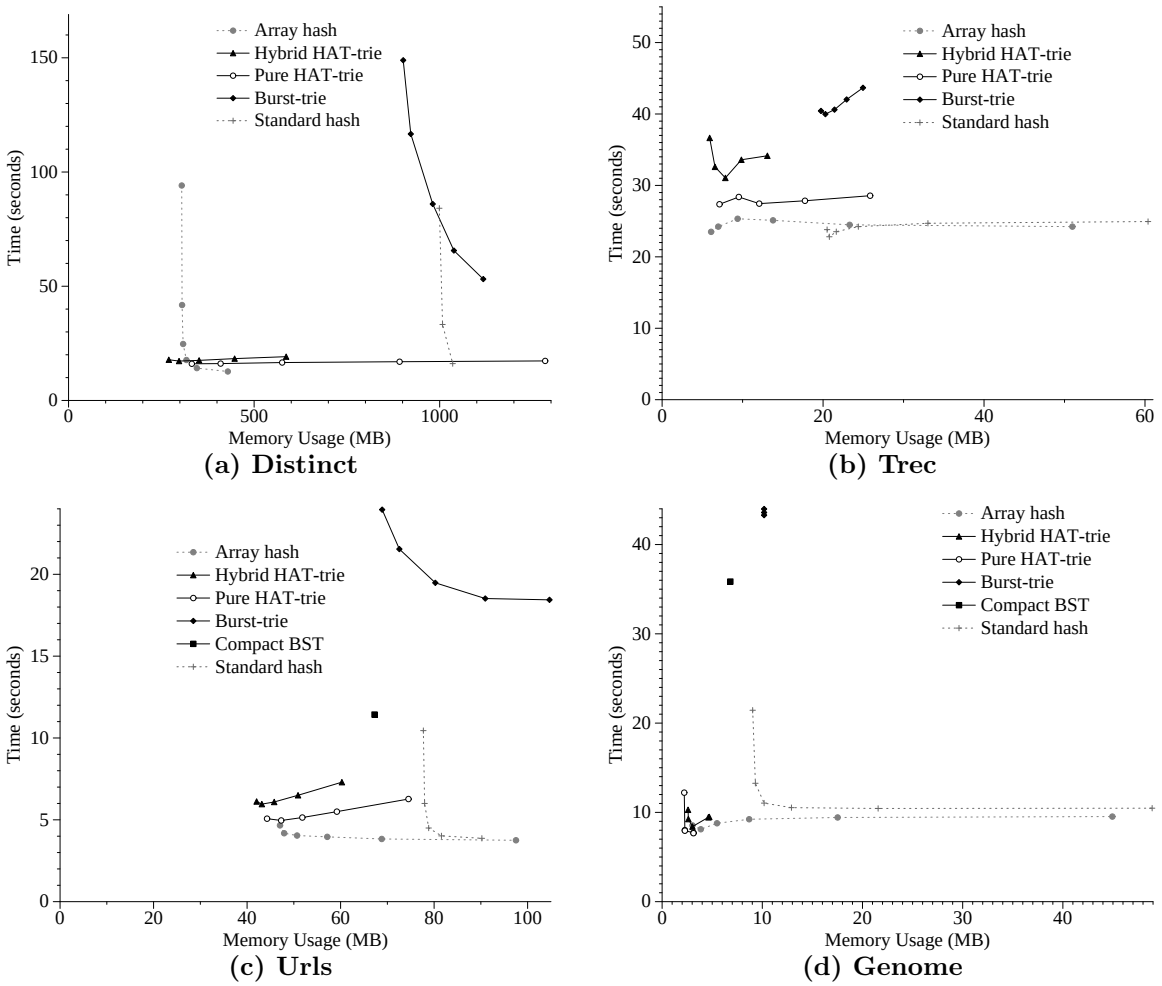

Figure 3: *Self-search costs of datasets distinct, trec, url and genome. Graph plots represent bucket thresholds (sizes). Smaller buckets require more memory.*

search tree, the burst-trie and the chaining hash table with move-to-front (which we call the *standard hash*); a group that includes the fastest and best-known data structures for in-memory string management. Our measure of memory takes into account the overhead imposed by the operating system per allocation, which is usually eight bytes; a figure which we found to be consistent after comparing our measure with that reported by the operating system under the */proc/stat/* table.

Although the binary search tree is well-known for its logarithmic average-case performance, it is not cache-conscious and in our initial experiments, we found it to be very slow. For a fairer comparison, we developed the *compact* BST, which stores strings directly within their respective nodes. This improves overall cache-efficiency (Askitis & Zobel 2005).

Our datasets consist of null-terminated variable-length strings that appear in occurrence order and are therefore unsorted; the characteristics of these datasets are shown in Table 1. The *distinct* dataset was acquired after parsing the GOV2 web crawl. Distributed as part of the TREC project, it does not contain duplicates. The highly skew *trec* dataset is the complete set of word occurrences, with duplicates, from the first of five TREC CDs (Harman 1995). The *url* dataset, also extracted from TREC web data, consists of complete URLs with duplicates. Finally, the *genome* dataset, extracted from GenBank, consists of fixed-length n-gram sequences with duplicates. Unlike the skew distributions observed in plain text however, these strings have a more uniform distribution.

The experiments were conducted on a 2.8 GHz Pentium IV machine, with 1 MB of L2 cache, 256-byte cache lines, a TLB capacity of 64 entries, 2 GB of RAM with a 4 KB page size, and a Linux operating system under light load using kernel 2.6.17. Our trie nodes used an alphabet that comprised of the first 128 characters of the ASCII table. We measured cache performance using PAPI (Dongarra, London, Moore, Mucci & Terpstra 2001) (available online), which obtains the actual number of TLB and L2 cache misses incurred during search. We restricted cache comparison to the array hash and the HAT-tries, as these were significantly better than the alternatives. We direct the reader to (Meyer, Sanders & Sibeyn 2003) for detailed information on the different types of caches used by modern processors.

The number of slots used by the array and standard hash were varied from 31,622 to 10,000,000 using the following sequence: 3.1622×10^4, 10^5, 3.16227×10^5, 10^6, 3.162277×10^6, and 10^7. For the burst-trie, we varied the bucket threshold (the number of strings needed to trigger a burst) by 25, 35, 50, 75, and 100. The HAT-tries have two parameters which are set at runtime: bucket threshold and the number of slots used. We began with a threshold of 1024 strings, doubling until 16,384. We set the number of slots used by every bucket to 1024 for the hybrid HAT-trie, and 512 for the pure HAT-trie. These settings were found to offer a good trade-off between space and speed. The pure HAT-trie requires fewer slots as it maintains only pure buckets that typically contain fewer strings.

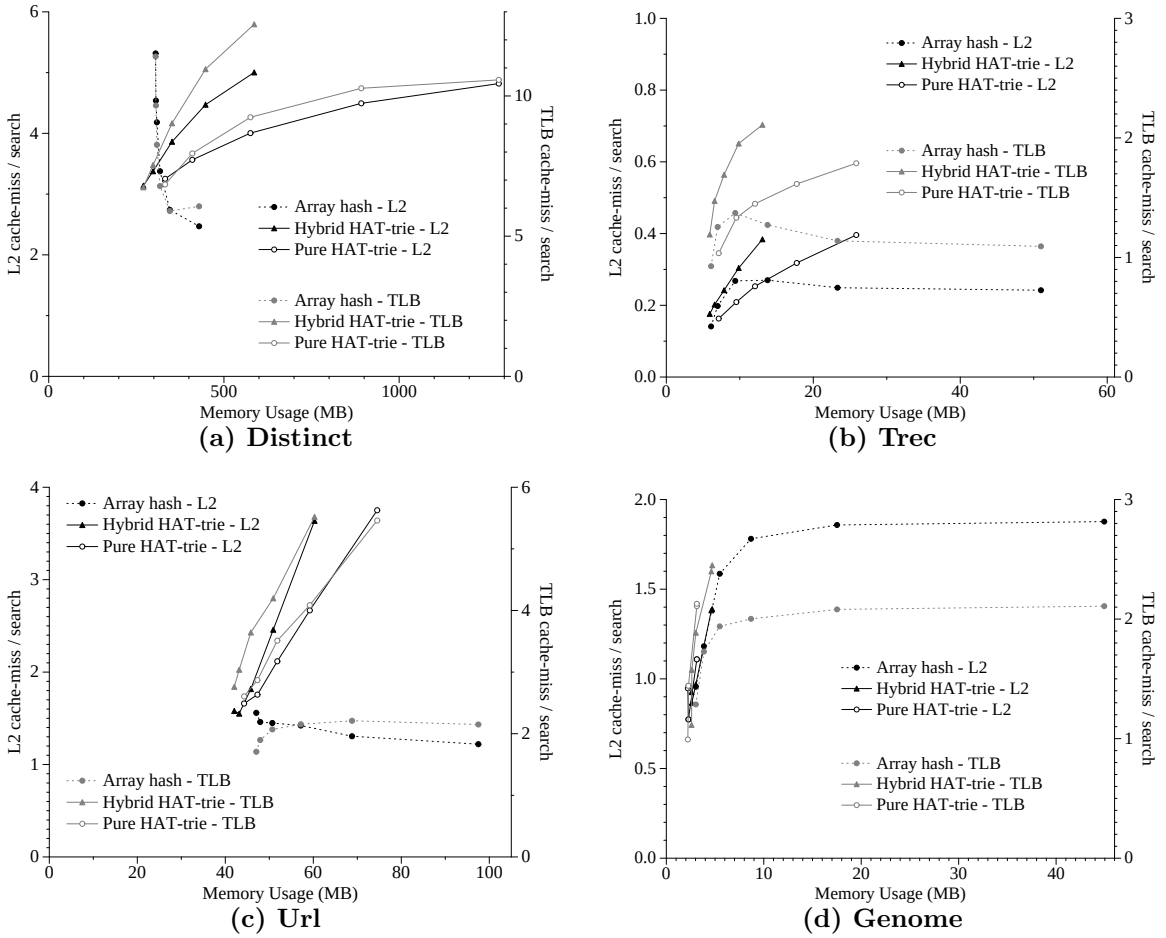

Figure 4: *The number of L2 cache-misses during the self-search. Graph plots represent bucket thresholds (sizes). Smaller buckets require more memory.*

5 Results

For all collections, we measured performance for construction and self-search. In a self-search, we search for all the words, in-occurrence order, that were used to construct the data structure. We now describe the results and observed trends for each collection.

Distinct data. The *distinct* dataset shows three important properties: there is no skew, each access requires a full tree traversal, and a large number of strings need to be managed. Such a collection is not particularly trie-friendly. Figures 2(a) and 3(a) show the effects on memory and speed as bucket thresholds vary. Small buckets are split often, resulting in a large number of tries and buckets. Larger buckets are split less frequently and are therefore, more space-efficient. We can see the effect on space in Table 2, which shows the number of nodes created relative to the bucket threshold.

We can save space by using larger buckets, but at the anticipated cost of access time. The HAT-tries however, showed little (if any) variance in the speed of construction and search, as the bucket threshold changed. For instance, with a threshold of 1024 strings, the hybrid HAT-trie required about 37 seconds to build and 19 seconds to search, consuming about 586 megabytes of memory. Buckets that were sixteen times larger increased the build time by less than ten seconds, while being faster to search (by about two seconds) and (simultaneously) reducing space to 270 megabytes; which is less than was required by the dataset. The pure HAT-trie displayed even less variance in speed, with the largest buckets being just as fast to build and search, as smaller buckets. It remained consistently faster than the hybrid HAT-trie, as a result of bursting, which is cheaper and can lead to fewer strings per bucket, but at the cost of some space.

These results are in contrary to the burst-trie, which can only achieve its best time using smaller buckets, and best space with larger buckets. Either way, the burst-trie was of no contest, requiring over a gigabyte of memory to reach its optimal search time of 53 seconds. The hash tables remained the fastest data structures, but only when given enough space, which proved excessive for the standard hash, and was partially omitted in these graphs. In comparison however, the HAT-tries — the pure HAT-trie in particular — could almost match the speed and space-efficiency of the array hash, while maintaining sorted access to strings.

Despite our efforts at improving the cache-efficiency of the binary search tree, we omitted it in these graphs, as it required over 430 seconds to construct and search, using 764 megabytes of memory. Figure 4(a) shows the cache costs of the HAT-tries and the array hash during search. The array hash performed poorly under heavy load, incurring over five $L2$ cache misses and almost twelve TLB misses per search. As we add more slots to reduce the average load factor, buckets became smaller and cheaper to access, and have improved probability of cache residency. As a result, we see a sharp decline in the number of cache misses.

In contrast however, increasing memory usage (by using smaller buckets) is detrimental to the perfor-

	Slots	Threshold	Tries	Buckets	Memory (MB)
	1,024	1,024	11,826	47,479	586.34
Hybrid	1,024	2,048	6,384	23,374	447.48
HAT-trie	1,024	4,096	2,677	11,381	351.80
	1,024	8,192	1,283	5,762	297.73
	1,024	16,384	680	2,939	270.16
	512	1,024	11,826	443,187	1,284.27
Pure	512	2,048	6,384	264,529	891.97
HAT-trie	512	4,096	2,677	130,994	575.63
	512	8,192	1,283	64,001	409.72
	512	16,384	680	34,780	332.13
	-	25	564,723	6,153,817	1117.45
	-	35	406,799	5,021,979	1038.30
Burst-trie	-	50	292,000	4,058,240	981.30
	-	75	173,005	3,018,590	922.16
	-	100	128,868	2,498,127	901.74

Table 2: Node count as the bucket threshold varies with the *distinct* dataset.

mance of the HAT-tries. Although buckets remain cheaper to access when they are small in size, maintaining a large number of tries and buckets can ultimately reduce cache-efficiency, as fewer are likely to remain within cache. Furthermore, with an increase of trie nodes, cache will likely be flooded with tries on search, overwriting previously cached buckets. Consequentially, this will lead to an increase in $L2$ cache misses, as observed.

Temporal access locality is unlikely to be as efficient for the HAT-tries, as it is for the hash tables. Every trie encountered during search branches to a new location in memory. Tries (apart from those close to the root of the tree) are therefore, less likely to be visited frequently, especially as their numbers increase. The number of conversions of virtual to physical addresses (TLB misses), is likely to increase as a result. Larger buckets are therefore of value to the HAT-tries, as they reduce the number of nodes created and space consumed, yielding higher cache efficiency.

Skewed data. Our next experiment evaluates cost where some strings are accessed much more frequently than others. We repeated the previous experiment using *trec*; the results are shown in Figures 2(b) and 3(b).

With just over half a million distinct strings, the data structures were much smaller than those constructed previously. The array hash performed well, even with a load factor of about 20 strings (31,622 slots). Although buckets were large, 99% of the searches need only access the first string of each bucket. Contrary to our previous results on the distinct dataset, it is clear that under skew access, the array hash can perform at its best, for both construction and search, with large buckets. The standard hash was also fast to access under heavy load as a result of move-to-front, but required more space.

Our HAT-tries remained competitive, being able to approach the speed of the array hash — the pure HAT-trie in particular — while almost matching its space consumption. The pure HAT-trie showed little variance in speed as we varied the bucket thresholds. The hybrid HAT-trie however, was slower to search with larger buckets, due to the use of b-trie splitting, which works more efficiently with a large number of strings per bucket (only 612,219 strings are distinct in this dataset). Both HAT-tries however, surpassed the performance of the burst-trie, which reached an optimal construction and search speed of about forty seconds (at least ten seconds slower) while requiring almost three times the space. The compact BST was once again, too expensive and was omitted, requiring over 65 seconds to construct and search, while consuming about 15 megabytes of memory. It required less space than the burst-trie, as a result of eliminating string pointers.

Cache efficiency is shown with Figure 4(b). Larger buckets lead to a reduction in cache misses for the HAT-tries, which is consistent with what we observed in previous experiments. The array hash performed well with large buckets, as a result of heavy skew. That is, larger buckets (slot entries) are more frequently accessed and are likely to remain in cache longer than smaller, but more numerous buckets. The pure HAT-trie remains more cache efficient than the hybrid HAT-trie however, as a result of bursting buckets and using 512 (as opposed to 1024) slots per bucket, which can reduce TLB misses under skew. Although using large buckets is more cache-efficient, the cost of searching a hybrid HAT-trie with large buckets, is expensive. The cause is the manner of how buckets were split during construction. The b-trie algorithm is not as effective at splitting buckets, when deprived of strings (Askitis & Zobel 2006); buckets, although fewer in number, are likely to be split unevenly in this case.

This results in frequently accessed buckets that contain too many strings, and are therefore, more computationally expensive to search, albeit in cache. We confirmed this after comparing the instruction and CPU cycle costs during search, which were higher for larger buckets.

URL data. The *URL* dataset is highly skew and contains long strings, some in excess of a hundred characters. URLs typically share long prefixes; most URLs start with "http://www". As a result, string comparisons can be more expensive.

The HAT-tries have the advantage of stripping away some of these prefixes during trie traversal, saving space and time. As a result, in Figures 2(c) and 3(c), the HAT-tries required the least amount of space (by using larger buckets) while simultaneously approaching the speed of the array hash. Interestingly however, the burst-trie remained expensive in both time and space. Even though prefixes are removed, the overhead imposed by linked list buckets, is too high.

The compact BST, although pointer-intensive, was almost twice as fast as the burst-trie, while requiring less space. This is due to the elimination of string pointers, which halves the number of random accesses made, the effects of which become more apparent under skew access. Its performance however, will start

to deteriorate as the number of searches increase, as observed with *trec*. As observed in previous experiments, the HAT-tries were able to perform at their best using large buckets that are more cache efficient, as shown in Figure 4(c).

Genome data. Our last experiment involves the *genome* dataset, containing fixed-length strings of strong skew. However, these strings are distributed much more uniformly than those of text. As shown in Figures 2(d) and 3(d), the HAT-tries almost matched the speed of the array hash for construction and search, while requiring less space. In contrast, for a high load factor, the cost of searching the array hash was higher, as string prefixes are not removed. Figure 4(d) shows that larger buckets allow the HAT-tries to approach and surpass the $L2$ and TLB performance of the array hash. Combined with the use of a trie-structure that strips away common prefixes (saving both space and comparison costs), the HAT-tries were superior. The burst-trie and the compact BST were greatly inferior to the array hash and the HAT-tries. The standard hash could only approach their speed, once given excessive space.

6 Conclusion

The burst-trie is currently the fastest in-memory data structure for maintaining strings in sort order, which is for example, a key requirement needed by database management software. It is not however, cache-conscious which presents a serious performance bottleneck with modern processors that typically use a hierarchy of caches to reduce costly accesses to main memory. We have introduced the HAT-trie, a variant of the burst-trie that carefully combines existing data structures to yield a fast, compact, scalable, and cache-conscious data structure, that can maintain variable-length strings in-memory and in sort order. We proposed two versions of the HAT-trie, *hybrid* and *pure*, which differ in the manner of how they split nodes.

We compared the HAT-tries to the cache-conscious array hash (which is currently the best for unsorted string management), an optimized binary search tree, the burst-trie and the chaining hash table with move-to-front on access. For both construction and search, the HAT-tries were up to 80% faster than the burst-trie, while simultaneously reducing space consumption by as much as 70%. The chaining hash table could only compete in speed once given excessive space, while the binary search tree proved to be too inefficient in most cases.

Our HAT-tries — the pure HAT-trie in particular — could approach, and in some cases surpass, the speed of the array hash while requiring less space. In general, the hybrid HAT-trie required the least space of all data structures, but was not as fast as the pure HAT-trie. Our results highlight further potential improvements, primarily with the way buckets are structured. However, to our knowledge, the current HAT-tries are the first trie-based data structures that can approach the speed and space efficiency of hash tables, while maintaining sorted access to strings. These are strong results that further substantiate the effectiveness of using dynamic arrays in the structural design of pointer-intensive data structures that are otherwise computationally efficient.

References

Acharya, A., Zhu, H. & Shen, K. (1999), Adaptive algorithms for cache-efficient trie search, *in* 'Proc. ALENEX Workshop on Algorithm Engineering and Experiments', Springer-Verlag, pp. 296–311.

Aggarwal, A. (2002), Software caching vs. prefetching, *in* 'Proc. Int. Symp. on Memory management', ACM Press, New York, pp. 157–162.

Agrawal, R. & Srikant, R. (1994), Fast algorithms for mining association rules, *in* 'Proc. VLDB Int. Conf. on Very Large Databases', Morgan Kaufmann, pp. 487–499.

Al-Suwaiyel, M. & Horowitz, E. (1984), 'Algorithms for trie compaction', *ACM trans. on Database Systems* **9**(2), 243–263.

Aoe, J., Morimoto, K. & Sato, T. (1992), 'An efficient implementation of trie structures', *Software—Practice and Experience* **22**(9), 695–721.

Arge, L., Bender, M. A., Demaine, E. D., Holland-Minkley, B. & Munro, J. I. (2002), Cache-oblivious priority queue and graph algorithm applications, *in* 'Proc. ACM Symp. on Theory of Computing', ACM Press, New York, pp. 268–276.

Arge, L., Bender, M. A., Demaine, E., Leiserson, C. & Mehlhorn, K. (2004), Abstracts collection, *in* 'Cache-Oblivious and Cache-Aware Algorithms', number 04301 *in* 'Dagstuhl Seminar Proceedings', IBFI, Germany.

Arge, L., Brodal, G. & Fagerberg, R. (2004), In handbook on data structures and applications, *in* 'Cache-oblivious Data Structures', CRC Press.

Askitis, N. & Zobel, J. (2005), Cache-conscious collision resolution for string hash tables, *in* 'Proc. SPIRE String Processing and Information Retrieval Symp.', Springer-Verlag, pp. 92–104.

Askitis, N. & Zobel, J. (2006), B-tries for disk-based string management. Manuscript in submission.

Badawy, A. A., Aggarwal, A., Yeung, D. & Tseng, C. (2001), Evaluating the impact of memory system performance on software prefetching and locality optimizations, *in* 'Proc. Int. Conference on Supercomputing', ACM Press, New York, pp. 486–500.

Bell, T. C., Cleary, J. G. & Witten, I. H. (1990), *Text Compression*, Prentice-Hall.

Bender, M. A., Demaine, E. D. & Farach-Colton, M. (2000), Cache-oblivious b-trees, *in* 'IEEE Symp. on the Foundations of Computer Science', IEEE Computer Society Press, pp. 399–409.

Bender, M. A., Demaine, E. D. & Farach-Colton, M. (2002), Efficient tree layout in a multilevel memory hierarchy, *in* 'Proc. European Symp. on Algorithms', Springer-Verlag, pp. 165–173.

Bender, M. A., Duan, Z., Iacono, J. & Wu, J. (2002), 'A locality-preserving cache-oblivious dynamic dictionary', *Proc. ACM-SIAM Symp. on Discrete Algorithms* **53**(2), 29–38.

Bender, M., Brodal, G. S., Fagerberg, R., Ge, D., He, S., Hu, H., Iacono, J. & Lopez-Ortiz, A. (2003), The cost of cache-oblivious searching, *in* 'IEEE Symp. on the Foundations of Computer Science', IEEE Computer Society Press, pp. 271–282.

Bentley, J. & Sedgewick, R. (1997), Fast algorithms for sorting and searching strings, *in* 'Proc. ACM-SIAM Symp. on Discrete Algorithms', Society for Industrial and Applied Mathematics, pp. 360–369.

Berg, S. G. (2002), Cache prefetching, *in* 'Tech Report UW-CSE', Uni. of Washington.

Brodal, G. S. & Fagerberg, R. (2006), Cache-oblivious string dictionaries, *in* 'Proc. ACM-SIAM Symp. on Discrete Algorithms', ACM Press, New York, pp. 581–590.

Brodal, G. S., Fagerberg, R. & Jacob, R. (2002), Cache oblivious search trees via binary trees of small height, *in* 'Proc. ACM-SIAM Symp. on Discrete Algorithms', ACM Press, New York, pp. 39–48.

Chilimbi, T. M., Hill, M. D. & Larus, J. R. (1999), Cache-conscious structure layout, *in* 'Proc. ACM SIGPLAN conf. on Programming Language Design and Implementation', ACM Press, New York, pp. 1–12.

Collins, J., Sair, S., Calder, B. & Tullsen, D. M. (2002), Pointer cache assisted prefetching, *in* 'Proc. Annual ACM/IEEE MICRO Int. Symp. on Microarchitecture', IEEE Computer Society Press, pp. 62–73.

Comer, D. (1979), 'Heuristics for trie index minimization', *ACM trans. on Database Systems* **4**(3), 383–395.

Dongarra, J., London, K., Moore, S., Mucci, S. & Terpstra, D. (2001), Using papi for hardware performance monitoring on linux systems, *in* 'Proc. Conf. on Linux Clusters: The HPC Revolution', Urbana, Illinois, USA.

Flajolet, P. & Puech, C. (1986), 'Partial match retrieval of multimedia data', *Jour. of the ACM* **33**(2), 371–407.

Fredkin, E. (1960), 'Trie memory', *Communications of the ACM* **3**(9), 490–499.

Frigo, M., Leiserson, C. E., Prokop, H. & Ramachandran, S. (1999), Cache-oblivious algorithms, *in* 'IEEE Symp. on the Foundations of Computer Science', IEEE Computer Society Press, p. 285.

Fu, J. W. C., Patel, J. H. & Janssens, B. L. (1992), Stride directed prefetching in scalar processors, *in* 'Proc. Annual ACM/IEEE MICRO Int. Symp. on Microarchitecture', IEEE Computer Society Press, pp. 102–110.

Hallberg, J., Palm, T., & Brorsson, M. (2003), Cache-conscious allocation of pointer-based data structures revisited with hw/sw prefetching, *in* '$2^n d$ Annual Workshop on Duplicating, Deconstructing, and Debunking'.

Harman, D. (1995), Overview of the second text retrieval conference (TREC-2), *in* 'Proc. Second Text Retrieval Conference', Pergamon Press, Inc., pp. 271–289.

Heinz, S., Zobel, J. & Williams, H. E. (2002), 'Burst tries: A fast, efficient data structure for string keys', *ACM trans. on Information Systems* **20**(2), 192–223.

Knuth, D. E. (1998), *The Art of Computer Programming: Sorting and Searching*, Vol. 3, second edn, Addison-Wesley.

Kumar, P. (2002), Cache oblivious algorithms., *in* 'Algorithms for Memory Hierarchies', Vol. 2625 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 193–212.

Ladner, R. E., Fortna, R. & 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', Springer-Verlag, pp. 78–92.

Maly, K. (1976), 'Compressed tries', *Communications of the ACM* **19**(7), 409–415.

McCreight, E. M. (1976), 'A space-economical suffix tree construction algorithm', *Jour. of the ACM* **23**(2), 262–271.

Meyer, U., Sanders, P. & Sibeyn, J. F., eds (2003), *Algorithms for Memory Hierarchies*, Vol. 2625 of *Lecture Notes in Computer Science*, Springer-Verlag.

Ramakrishna, M. V. & Zobel, J. (1997), Performance in practice of string hashing functions, *in* 'Proc. Int. Symp. on Database Systems for Advanced Applications', World Scientific Press, Melbourne, Australia, pp. 215–223.

Rao, J. & Ross, K. A. (1999), Cache conscious indexing for decision-support in main memory, *in* 'Proc. VLDB Int. Conf. on Very Large Databases', Morgan Kaufmann, pp. 78–89.

Rao, J. & Ross, K. A. (2000), Making b^+-trees cache conscious in main memory, *in* 'Proc. ACM-SIGMOD Int. Conf. on the Management of Data', ACM Press, New York, pp. 475–486.

Roth, A. & Sohi, G. S. (1999), Effective jump-pointer prefetching for linked data structures, *in* 'Proc. Int. Symp. on Computer Architecture', IEEE Computer Society Press, pp. 111–121.

Sedgewick, R. (1998), *Algorithms in C*, Addison-Wesley.

Severance, D. G. (1974), 'Identifier search mechanisms: A survey and generalized model', *Proc. ACM Computer Science Conf.* **6**(3), 175–194.

Sinha, R., Ring, D. & Zobel, J. (2006), 'Cache-efficient string sorting using copying', *ACM Jour. of Exp. Algorithmics* **11**(1.2).

Sinha, R. & Zobel, J. (2004), 'Cache-conscious sorting of large sets of strings with dynamic tries', *ACM Jour. of Exp. Algorithmics* **9**(1.5).

Yang, C., Lebeck, A. R., Tseng, H. & Lee, C. (2004), 'Tolerating memory latency through push prefetching for pointer-intensive applications', *ACM Trans. Architecture Code Optimisation* **1**(4), 445–475.

Zobel, J., Williams, H. E. & Heinz, S. (2001), 'In-memory hash tables for accumulating text vocabularies', *Information Processing Letters* **80**(6), 271–277.