Cache-Conscious Sorting of Large Sets of Strings with Dynamic Tries

RANJAN SINHA and JUSTIN ZOBEL
RMIT University

Ongoing changes in computer architecture are affecting the efficiency of string-sorting algorithms. The size of main memory in typical computers continues to grow but memory accesses require increasing numbers of instruction cycles, which is a problem for the most efficient of the existing string-sorting algorithms as they do not utilize cache well for large data sets. We propose a new sorting algorithm for strings, burstsort, based on dynamic construction of a compact trie in which strings are kept in buckets. It is simple, fast, and efficient. We experimentally explore key implementation options and compare burstsort to existing string-sorting algorithms on large and small sets of strings with a range of characteristics. These experiments show that, for large sets of strings, burstsort is almost twice as fast as any previous algorithm, primarily due to a lower rate of cache miss.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms**]: Sorting; E.5 [**Files**]: Sorting; E.1 [**Data Structures**]: Trees; B.3.2 [**Memory Structures**]: Cache Memories; D.1.0 [**Programming Techniques**]: General

1. INTRODUCTION

Sorting is one of the fundamental problems of computer science. In many current applications, large numbers of strings may need to be sorted. There have been several recent advances in fast sorting techniques designed for strings. For example, many improvements to quicksort have been described since it was first introduced, an important recent innovation being the introduction of three-way partitioning by Bentley and McIlroy [1993]. Splaysort, an adaptive sorting algorithm introduced by Moffat et al. [1996], is a combination of the splaytree data structure and insertionsort. Improvements to radixsort for string data were proposed by McIlroy et al. [1993]; forward and adaptive radixsort for strings were introduced by Andersson and Nilsson [1998] and Nilsson [1996]; a hybrid of quicksort and MSD radixsort named three-way radix quicksort [Sedgewick

Authors' address: Ranjan Sinha and Justin Zobel, School of Computer Science and Information Technology, RMIT University, GPO Box 2476V, Melbourne 3001, Australia; email: {rsinha,jz}@cs.rmit.edu.au.

1998] was introduced by Bentley and Sedgewick [1997]; and, as an extension to keys that are made up of components, three-way radix quicksort was extended by Bentley and Sedgewick [1997] to give multikey quicksort.

While these algorithms are theoretically attractive, they have drawbacks. In particular, they show poor locality of memory accesses, a problem that is of increasing significance. A standard desktop computer now has a processor running at over 1 GHz and 256 Mb or more of memory. However, memory and bus speeds have not increased as rapidly, and a delay of 20–200 clock cycles per memory access is typical. For this reason, current processors have caches, ranging from 64 or 256 kilobytes on a Celeron to 8 megabytes on a SPARC; however, these are tiny fractions of typical memory volumes, of 128 to 256 megabytes on the former and many gigabytes on the latter. On a memory access, a line of data (32 or 128 bytes say) is transferred from memory to the cache, and adjacent lines may be proactively transferred until a new memory address is requested. The paging algorithms used to manage cache are primitive, based on the low-order bits of the memory address.

Thus, some years ago, the fastest algorithms were those that used the least number of instructions. Today, an algorithm can afford to waste instructions if doing so reduces the number of memory accesses [LaMarca and Ladner 1997]: an algorithm that is efficient for sorting a megabyte of data, or whatever the cache size is on that particular hardware, may rapidly degrade as data set size increases. Radixsorts are more efficient than older sorting algorithms, due to the reduced number of times each string is handled, but are not necessarily efficient with regard to cache. The degree to which algorithms can effectively utilize cache is increasingly a key performance criterion [LaMarca and Ladner 1997; Xiao et al. 2000].

Addressing this issue for string sorting is the subject of our research. According to Arge et al. [1997], "string sorting is the most general formulation of sorting because it comprises integer sorting (i.e., strings of length one), multikey sorting (i.e., equal-length strings) and variable-length key sorting (i.e., arbitrarily long strings)." Cache misses are a significant problem for string sorting. String sets are typically represented by an array of pointers to locations in memory where the variable-length strings are held. Each string reference incurs at least two cache misses, one for the pointer and one or more for the string itself depending on its length and how much of the string needs to be read. However, in practice, current algorithms incur many more—in our experiments, for the largest data sets from 5 to 10 misses per string is typical.

We propose a new sorting algorithm, *burstsort*, which is based on the burst trie [Heinz et al. 2002]. A burst trie is a collection of small data structures, or *buckets*, that are accessed by a conventional trie. The first few characters of strings are used to construct the trie, which indexes buckets containing strings with shared prefixes. The trie is used to allocate strings to buckets, the suffixes of which are then sorted using a method more suited to small sets. In principle, burstsort is similar to MSD radixsort, as both recursively partition strings into small sets, character position by character position, but there are crucial differences. Radixsort proceeds positionwise, inspecting the first character of every string before inspecting any subsequent characters; only one branch of

the trie is required at a time, so it can be managed as a stack. Burstsort proceeds stringwise, completely consuming one string before proceeding to the next; the entire trie is constructed during the sort. However, the trie is small compared to the set of strings and is typically mostly resident in cache, and the stream-oriented processing of the strings is also cache friendly.

Using several small and large sets of strings derived from real-world data, such as lexicons of web collections and genomic strings, we compared the speed of burstsort to the best existing string-sorting algorithms. Burstsort has high initial costs, making it no faster than the best of the previous methods for small sets of strings. For large sets of strings, however, we observed that burstsort is much the fastest, typically by almost a factor of two. Using artificial data, we found that burstsort is insensitive to adverse cases, such as all characters being identical or strings that are hundreds of characters in length.

For large sets of strings, burstsort is the best sorting method. Using a cache simulator, we show that the gain in performance is due to the low rate of cache misses. Not only is it more efficient for the data sets tested, but the trend in performance is to further gains as data volumes grow and processors become faster and memory latency increases.

We review existing methods for internal sorting of strings in Section 2, and introduce our burstsort in Section 3. Implementation options and their likely impact are explored in Section 4. We describe our experiments in Section 5 and present results in Section 6, with outcomes summarized in Section 7.

2. EXISTING APPROACHES TO STRING SORTING

Many sorting algorithms have been proposed, but most are not particularly well suited to string data. Here we review string-specific methods. In each case, the input is an array of pointers to strings, and the output is the same array with the strings in lexicographic order.

Quicksort. Described by Hoare [1962], the Bentley and McIlroy [1993] variant of quicksort has become the dominant sort routine used in most libraries since the early 1990s. Quicksort was originally intended for arbitrary input and hence has some overhead for specific data-types. For our experiments, we use a stripped-down version by Nilsson [1996] that is specifically tailored for character strings, designated as *Quicksort*.

Multikey quicksort. Introduced by Bentley and Sedgewick [1997], it is a hybrid of quicksort and MSD radixsort. Instead of taking the entire string and comparing with another string in its entirety, at each stage it considers only a particular position within each string. The strings are then partitioned according to the value of the character at this position, into sets less than, equal to, or greater than a given pivot. Then, like radixsort, it moves onto the next character once the current input is known to be equal in the given character.

Such an approach avoids the main disadvantage of many sorting algorithms for strings, namely, the wastefulness of a string comparison. With a conventional quicksort, for example, as the search progresses it is clear that all the strings in a partition must share a prefix. Comparison of this prefix is redundant

[Sedgewick 1998]. With the characterwise approach, the length of the shared prefix is known at each stage. However, some of the disadvantages of quicksort are still present. Each character is inspected multiple times, until it is in an "equal to pivot" partition. Each string is re-accessed each time a character in it is inspected, and after the first partitioning these accesses are effectively random. For a large set of strings, the rate of cache misses is likely to be high. In our experiments, we have used an implementation by Bentley and Sedgewick [1997], designated as *Multikey quicksort*.

Radixsort. A family of sorting methods where the keys are interpreted as a representation in some base (often a power of 2) or as strings over a given small alphabet. Instead of comparing keys in their entirety, they are decomposed into a sequence of fixed sized pieces, typically bytes. There are two, fundamentally different approaches to radix sorting: most significant digit (MSD) and least significant (LSD) [Sedgewick 1998]. It is difficult to apply the LSD approach to a string-sorting application because it is unsuitable for variable-length keys. Another drawback is that LSD algorithms inspect all characters of the input, which is unnecessary in MSD approaches. We do not explore LSD methods in this paper.

MSD radixsort. It examines only the distinguishing prefixes, working with the most significant characters first, an attractive approach because it uses the minimum amount of information necessary to complete the sorting. The algorithm has time complexity $\Omega(n + S)$, where S is the total number of characters of the distinguishing prefixes; amongst n distinct strings, the minimum value of S is approximately $n \log_{|A|} n$, where $|A|$ is the size of the alphabet. The basic algorithm proceeds by iteratively placing strings in buckets according to their prefixes, then using the next character to partition a bucket into smaller buckets.

The algorithm switches to insertionsort or another simple sorting mechanism for small buckets. In our experiments we have used the implementation of Nilsson [1996], designated as *MSD radixsort*.

MBM radixsort. Early high-performance string-oriented variants of MSD radixsort were presented by McIlroy et al. [1993]. Of the four variants, we found program C to be typically the fastest for large data sets. It is an array-based implementation of MSD radixsort that uses a fixed 8-bit alphabet and performs the sort in place. In agreement with Bentley and Sedgewick [1997], we found it to be the fastest array-based string sort. In our experiments it is designated as *MBM radixsort*.

Forward radixsort. Developed by Andersson and Nilsson [1994] and Nilsson [1996], it combines the advantages of LSD and MSD radixsort and is a simple and efficient algorithm with good worst-case behavior. It addresses a problem of MSD radixsort, which has a bad worst-case performance due to fragmentation of data into many sublists. Forward radixsort starts with the most significant digit, performs bucketing only once for each character position, and inspects only the significant characters. A queue of buckets is used to avoid the need to

allocate a stack of trie nodes, but even so, in our experiments this method had high memory requirements. In our experiments we have used the implementations of Nilsson [1996], who developed 8-bit and 16-bit versions, designated as *Forward-8* and *Forward-16*.

Adaptive radixsort. Developed by Nilsson [1996], the size of the alphabet is chosen adaptively based on a function of the number of elements remaining, switching between two character sizes, 8 bits and 16 bits. In the 8-bit case, it keeps track of the minimum and maximum character in each trie node. In the 16-bit case, it keeps a list of which slots in the node are used, to avoid scanning large numbers of empty buckets. In our experiments we have used the implementation of Nilsson [1996], designated as *adaptive radixsort*.

3. CACHE FRIENDLY SORTING WITH TRIES

A recent development in data structures is the burst trie, which has been demonstrated to be the fastest structure for maintaining a dynamic set of strings in sort order [Heinz et al. 2002; Heinz and Zobel 2002]. It is thus attractive to consider it as the basis of a sorting algorithm. Burstsort is a straightforward implementation of sorting based on burst trie insertion and traversal. We review the burst trie, then introduce our new sorting technique.

Burst tries. A form of trie that is efficient for handling sets of strings of any size [Heinz et al. 2002; Heinz and Zobel 2002], it resides in memory and stores strings in approximate sort order. A burst trie comprises three distinct components: a set of strings, a set of buckets, and an access trie. A *bucket* is a small set of strings, stored in a simple data structure such as an array or a binary search tree. (Choice of bucket structure is considered later.) The strings that are stored in a bucket at depth d are at least d characters in length, and the first d characters in the strings are identical. An *access trie* is a trie whose leaves are buckets. Each node consists of an array (whose length is the size of the alphabet) of pointers, each of which may point to another trie node or to a bucket, and a single empty-string pointer to a bucket. A burst trie using lists and arrays for buckets is shown in Figures 1 and 2 respectively. Strings in the burst trie are “bat”, “barn”, “bark”, “by”, “by”, “by”, “by”, “byte”, “bytes”, “wane”, “way”, and “west”.

A burst trie can increase in size in two ways. First is by *insertion* when a string is added to a bucket. Insertion is straightforward. Let the string to be inserted be $c_1, \ldots, c_n$ of n characters. The leading characters of the string are used to identify the bucket in which the string should reside. If the bucket is at a depth of $d = n+1$, the bucket is under the empty-string pointer. The standard insertion algorithm for the data structure used in the bucket is used to insert the strings into the buckets. For an array, a pointer to the string is placed at the leftmost free slot.

The second way to increase in size is by *bursting*, the process of replacing a bucket at depth d by a trie node and a set of new buckets at depth $d+1$; all the strings in the original bucket are distributed in the buckets in the newly created node. A bucket is burst whenever it contains more than a fixed number L of

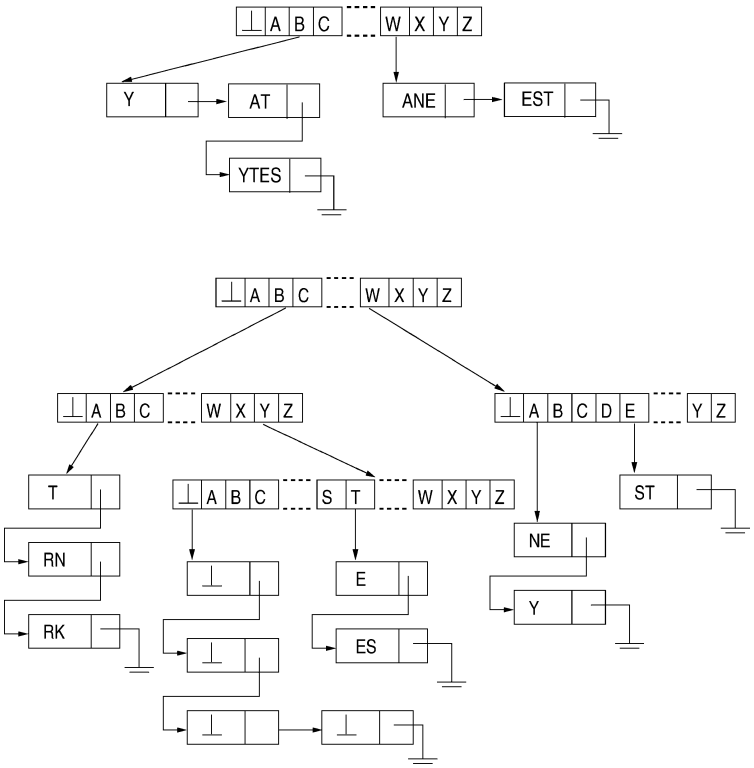

Fig. 1. Burst trie using lists, with four trie nodes, five buckets, and twelve strings. Upper: before any bursts. Lower: after all bursts.

strings. That is, when the number of strings in a bucket exceeds L, a new trie node is created, which is linked into the trie in place of the bucket. The $(d+1)$th character of the strings in the bucket is used to partition the strings into buckets pointed to by the node. (In our implementation the string is not truncated, but doing so could save considerable space, allowing much larger sets of strings to be managed [Heinz et al. 2002].) Repetitions of the same string are stored in the same list, and do not subsequently have to be sorted as they are known to be identical. Though the bucket may be an unordered structure, the buckets themselves are in sort order, and due to their small size can be sorted rapidly.

Burstsort. Our burstsort algorithm is based on the general principle that any data structure that maintains items in sort order can be used as the basis of a sorting method, simply by inserting the items into the structure one by one then retrieving them all in-order in a single pass.[1]

The data structures used in burstsort are a source array of strings, source array of pointers to strings, array-based trie nodes, and buckets that are an array of pointers to strings. Burstsort uses the burst trie data structure to

[1]Our implementations is available under the heading "String sorting", at the URL `www.cs.rmit.edu.au/~jz/resources`.

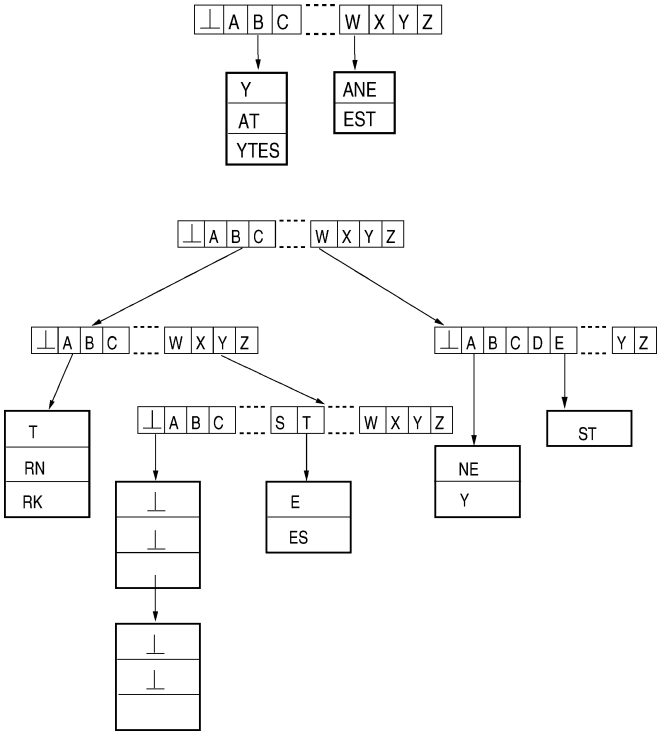

Fig. 2. Burst trie using arrays, with four trie nodes, five buckets, and twelve strings. Upper: before any bursts. Lower: after all bursts.

divide the strings into buckets, which are then sorted using other methods. As for other trie sorts, placing the string in a bucket requires reading of at most the distinguishing prefix, and the characters in the prefixes are inspected once only. Also, in many data sets the most common strings are short; these strings are typically stored at an empty-string pointer and are collected while traversing the access trie without being involved in bucket sorting.

Burstsort has similarities to MSD radixsort, but there are crucial differences. The main one is that memory accesses are more localized. During the insertion phase, a string is retrieved to place it in a bucket, then again when the bucket is burst (a rare event once a reasonable number of strings have been processed). Trie nodes are retrieved at random, but there are relatively few of these and thus most can be simultaneously resident in the cache. In contrast to this depth-first style of sorting, radixsort is breadth-first. Each string is refetched from memory per character in the string.

Burstsort proceeds as follows:

(1) Each string is inserted in turn into a burst trie, which is grown as necessary to maintain the limit L on bucket size. Strings are referenced by pointers.
   —A pointer to each string that is entirely consumed by trie traversal is placed in an end-of-string bucket and not accessed again until the traversal phase.

—Other strings are placed in buckets, the management of which is discussed in the next section.
—When a bucket overflows (that is, capacity limit L is exceeded), it is burst as described above.

(2) When all strings have been inserted, the burst trie is traversed depth first and from left-to-right in each node, observing the following conditions:
—If the size of a bucket is one, the string can be output immediately.
—If the bucket is under the empty-string pointer, all strings in the bucket are identical, so it can be traversed and output immediately.
—Other buckets with more than one string must be sorted, starting at character position d for a bucket at depth d, and can then be output. We have used multikey quicksort in our experiments; it is not tightly integrated with burstsort and there is scope for further improvement.

With a typical set of strings, most leaf nodes in the access trie would be expected to have a reasonable number of buckets, in the range 10–100 for an alphabet of 256 characters.

Considering the asymptotic computational cost of burstsort, observe that standard MSD radixsort uses a similar strategy. Trie nodes are used to partition a set of strings into buckets. If the number of strings in a bucket exceeds a limit L, it is recursively partitioned; otherwise, a simple strategy such as insertionsort is used. The order in which these operations are applied varies between the methods, but the number of them does not.

Thus burstsort and MSD radixsort have the same asymptotic computational cost. That is, both algorithms are $O(n \log n)$, which is the cost of inspecting the characters in the distinguishing prefix of every string. This is also the cache complexity of radixsort and burstsort, but in practice for burstsort this worst case is unlikely to be observed, as we now discuss.

The efficiency of burstsort is due to the way it uses cache, and in particular due to the number of L2 cache misses. The principle kinds of cache miss are L1 (level 1), which involve only a few machine cycles; L2 (level 2), which involve many more cycles; and TLB (translation lookaside buffer), which are less frequent and, on many machines, less costly than an L2 miss. Of these, L2 misses are the most significant factor in processing time, and in the following discussion we focus on L2 misses. Some cache characteristics of machines used in our experiments are listed in Table I.

The number of active locations that need to be cached is dependent on the pattern of accesses to the trie nodes and the buckets; typically this pattern is skew. Assuming that all active locations fit in cache, which is a reasonable assumption for most collections, cache misses in the initial phase of sorting are due to a scan traversal of the strings, scan traversal of the pointers to strings, and, in insignificant numbers, initial accesses to nodes and buckets; the number of cache misses per string during scan traversal is low. Bursting involves further cache misses due to scan traversal of source and destination buckets and due to accesses to strings. In the largest collections, these accesses to strings during bursting account for around one cache miss per key. During the final, traversal phase, the strings in nonempty string buckets are reaccessed, resulting in up

Table I. Architectural Parameters of the Machine Used for the Experiments

Workstation	Pentium	Pentium	UltraSPARC	Power Mac G5
Processor type	P III Xeon	P IV	USPARC III	PowerPC 970
Clock rate (MHz)	700	2000	750	1600
L1 cache (KB)	16	8	64	32
L1 block size (Bytes)	32	64	32	64
L1 associativity	4	4	4	2
L1 miss latency (cycles)	6	7	12	8
L2 cache (KB)	1,024	512	8,192	512
L2 block size (Bytes)	32	64	512	128
L2 associativity	8	8	1	8
L2 miss latency (cycles)	109	285	174	324
Memory Size (MB)	2,048	2,048	4096	256

to one cache miss each, and there is a scan traversal of the buckets and source array of pointers. In total, the expected number of cache misses is around 2–3 when the active locations fit within cache.

In contrast, radixsort involves cache misses for all of these reasons, but in addition incurs a potential cache miss for each character in the prefix of each string. The burst trie is small and may well be largely cache resident; thus trie traversal does not tend to incur cache misses; the set of strings is large, and thus each access to a string in radixsort may well have a cache penalty.

Moreover, string sorting using pointers to strings—as is necessary with radixsort—is not TLB efficient as each access to a string may be a TLB miss. Burstsort does well, as it reduces the number of accesses to the strings. In the worst case it is possible that each access to a trie node or a bucket could be a TLB miss, but it is not easy to construct an example in which such behavior occurs.

To demonstrate the generality of our algorithms, in our experiments we include results from several machines with varying cache architectures. As these results show, burstsort gives improvements over previous algorithms under all of these architectures.

4. IMPLEMENTATION OPTIONS

The implementation of burstsort used for our original work [Sinha 2002] was strongly influenced by design choices that had proven effective for burst tries. However, these are not necessarily ideal for sorting, where for example random access to stored strings is not required. We therefore identified and evaluated a range of implementation options. These were

—data structure used to represent buckets;
—size of the root trie node;
—bucket capacity;
—bucket sorting method.

In detail, these options are as follows.

Bucket representation. In our original work, we used linked lists to represent buckets. During the insertion phase, linked lists are highly efficient. First,

the list nodes for all the strings can be allocated as a single array, and the array of pointers can be copied to the array of nodes in one pass. The strings themselves are not accessed during this process, which requires only a tiny fraction of the total sorting time. Second, during the insertion phase a linked list need only be accessed when a bucket is burst, which is a relatively rare occurrence; the great majority of strings do not participate in a burst operation. As each inserted string can be placed at the start of a linked list representing a bucket, the existing nodes in the list are not accessed. That is, there are few random accesses, and a linked list allows extremely cheap insertion.

However, in subsequent experiments it became apparent that linked lists lead to inefficiencies in the traversal phase. In particular, sorting a bucket requires that the list be traversed and the string pointers copied to an array (a fragment of the original array of pointers can be used). With a bucket capacity of 8,192—a figure that gave the greatest overall efficiency in preliminary experiments—a large fraction of total time was spent in bucket sorting. Also, the "insert at start of list" strategy means that burstsort is not stable.

Alternatives to linked lists were considered in the context of burst tries; for string management, it was found that a binary tree is the most efficient option. However, these options are not of value for sorting, as searching is not a factor. (Burst tries are used for management of distinct strings; for sorting, copies must be kept, as additional data may be associated with each string.)

Another alternative is to use arrays. In the simplest implementation of this approach, when the first string is to be placed in a bucket, an array of pointers is dynamically created. Additional strings are placed sequentially in the array. When it is full, it is burst as before. However, such an approach has serious drawbacks. If buckets are small, the size of the trie becomes unacceptable. If they are large, vast quantities of space are consumed: most buckets never approach the fixed capacity. A variation of this approach is to grow the arrays dynamically, up to the capacity, before bursting. These issues are discussed further below.

Managing buckets in this way is more costly than with linked lists: insertion into a bucket requires that both the start and the last-used position in the array must be accessed, or that counters be maintained within trie nodes; and, as the bucket grows, reallocation of memory and copying of pointers is required, leading to possible memory fragmentation. However, bucket sorting during the traversal phase is likely to be significantly more efficient, and the sort is stable.

An issue with the array representation is how to manage sets of identical strings, as there is no bound on the number of such strings and reallocating arrays is potentially an $O(n^2)$ costs. We chose to use linked lists of arrays, with a tail pointer to avoid traversal. The strings are only added at the end in each of the arrays, so stability is maintained.

Size of root node. In adaptive radixsort, the size of nodes dynamically switches between 2^8 and 2^{16} pointers, depending on the number of strings to be managed. With the larger node, pairs of letters are consumed at once, saving some operations, and the cost of inspecting null pointers can be avoided; the additional pointer at each level is required when end-of-string is observed. In our

experiments we observed that this strategy was only occasionally successful, as it could lead to costly stack operations that had little benefit if the number of observed pairs of letters was small. However, the simple heuristic of allowing the root node to be 2^{16} pointers has the potential to yield some benefit: this node can be maintained statically in the sort routine, and at run time the number of nodes allocated dynamically is somewhat reduced.

Bucket capacity. The bucket capacity is a parameter that balances the size of the trie against the cost of bucket sorting. A large trie—the consequence of small buckets—incurs memory management costs and poor cache behavior; large buckets are expensive to sort. We test a range of bucket capacities (from 16 to 32768 strings) in our experiments.

The impact of the bucket capacity depends on the data structure used to represent buckets. Varying the capacity for a linked-list representation is straightforward. For an array representation, how the array grows also needs to be considered. There are several possibilities. One is to allocate all at once: all nonempty buckets are the size of the threshold. This results in dramatic memory wastage for a large threshold, though it may reduce dynamic memory management. We found that this approach is not effective.

Another possibility is to grow buckets linearly: the bucket size is increased by one, or a small constant size, for each element placed in that bucket. This scheme in principle minimizes memory use, but in practice leads to fragmentation and $O(m^2)$ reallocation costs, due to copying, for a bucket of m slots.

A compromise option is to grow buckets exponentially: initially the buckets are small, then are multiplied in size until the threshold size is reached. The overhead per string is capped by the size multiplier, and in practice should be much less than this theoretical limit. Only a small number of distinct bucket sizes are created, reducing fragmentation, and dynamic memory management costs should not be excessive. Compared to the all-at-once approach, however, an extra check is required at each insertion.

In this approach, empty buckets are represented by a null pointer. When an item is inserted, an array of 16 pointers is created. When this overflows, the array is grown, using the `realloc` system call, by a factor of 8. The bucket is burst when the capacity $L = 8192$ is reached. These parameters were chosen by hand tuning on a set of test data, but the results are highly insensitive to the exact values.

In our experiments, we use the exponential approach. The memory requirements were no more than 10% greater than the memory requirements needed for the list version, and, as can be seen in the experimental results reported below, the method is extremely efficient. In our implementation, a level counter was added to each pointer in the trie structure to keep track of the bucket size. A static array was maintained which had the amount of elements to allocate at each level. Two exponential schemes were tested: starting at 16 pointers and growing by a factor of 8; and starting at 8 and growing by a factor of 4.

Bucket-sorting mechanism. Once all the strings have been placed in the buckets in the trie nodes, the buckets must be sorted. In this phase, the strings

Table II. Statistics of the Data Collections Used in the Experiments

	Data set					
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Duplicates						
Size Mb	1.013	3.136	7.954	27.951	93.087	304.279
Distinct words ($\times 10^5$)	0.599	1.549	3.281	9.315	25.456	70.246
Word occurrences ($\times 10^5$)	1	3.162	10	31.623	100	316.230
No duplicates						
Size Mb	1.1	3.212	10.796	35.640	117.068	381.967
Distinct words ($\times 10^5$)	1	3.162	10	31.623	100	316.230
Word occurrences ($\times 10^5$)	1	3.162	10	31.623	100	316.230
Genome						
Size Mb	0.953	3.016	9.537	30.158	95.367	301.580
Distinct words ($\times 10^5$)	0.751	1.593	2.363	2.600	2.620	2.620
Word occurrences ($\times 10^5$)	1	3.162	10	31.623	100	316.230
Random						
Size Mb	1.004	3.167	10.015	31.664	100.121	316.606
Distinct words ($\times 10^5$)	0.891	2.762	8.575	26.833	83.859	260.140
Word occurrences ($\times 10^5$)	1	3.162	10	31.623	100	316.230
URL						
Size Mb	3.03	9.607	30.386	96.156	304.118	—
Distinct words ($\times 10^5$)	0.361	0.924	2.355	5.769	12.898	—
Word occurrences ($\times 10^5$)	1	3.162	10	31.623	100	—

in each bucket are copied back to the original array and sorted using an algorithm that is suited to small numbers of strings. We have tested a range of sorting routines, including insertionsort, shellsort, MBM radixsort, and multikey quicksort. We found that multikey quicksort and MBM radixsort were the most efficient. Multikey quicksort is used in all our experiments; the comparison to MBM radixsort is reported below.

Other issues. Some implementation details have unpredictable impact on performance. Consider the fact that a bucket is burst when it reaches a certain size: this means that it is necessary to know bucket size. If the size is not stored, the bucket must be fully traversed at each string, an unacceptable cost; thus a counter must be held. If space is created for an array of counters in each trie node, the nodes occupy more space, but if they are stored in a header node in each bucket, as would be required with the list representation of buckets, an extra pointer access is required. It is not clear which approach is likely to be more efficient, and in practice it is likely to depend on the data set size and the relative cost of a memory access. We do not believe that experiments on a single machine or even single architecture can identify which approach is superior.

5. EXPERIMENTS

We have used three kinds of data in our experiments, words, genomic strings, and web URLs.[2] The words are drawn from the large web track in the TREC

[2]Some of these data sets are available under the heading "string sets," at the URL `www.cs.rmit.edu.au/~jz/resources`.

project [Harman 1995; Hawking et al. 1999], and are alphabetic strings delimited by nonalphabetic characters in web pages (after removal of tags, images, and other nontext information). The web URLs have been drawn from the same collection. The genomic strings are from GenBank. For word and genomic data, we created six subsets, of approximately 10^5, 3.1623×10^5, 10^6, 3.1623×10^6, 10^7, and 3.1623×10^7 strings each. We call these Set 1, Set 2, Set 3, Set 4, Set 5, and Set 6 respectively. For the URL data we created Set 1 to Set 5. Only the smallest of these sets fits in cache. In detail, the data sets are as follows.

Duplicates. Words in order of occurrence, including duplicates. Most large collections of English documents have similar statistical characteristics, in that some words occur much more frequently than others. For example, Set 6 has just over 30 million word occurrences, of which just over 7 million are distinct words.

No duplicates. Unique strings based on word pairs in order of first occurrence in the TREC web data.

Genome. Strings extracted from genomic data, a collection of nucleotide strings, each typically thousands of nucleotides long. The alphabet size is four characters. It is parsed into shorter strings by extracting n-grams of length nine. There are many duplications, and the data does not show the skew distribution that is typical of text.

Random. An artificially generated collection of strings whose characters are uniformly distributed over the entire ASCII range and the length of each string is randomly generated but less than 20. The idea is to force the algorithms to deal with a large number of characters where heuristics of visiting only a few buckets would not work well. This is the sort of distribution many of the theoretical studies deal with [Rahman and Raman 2000], although such distributions are not especially realistic.

URL. Complete URLs, in order of occurrence and with duplicates, from the TREC web data, average length is high compared to the other sets of strings.

Some other artificial sets were used in limited experiments, as discussed later.

The aim of our experiments is to compare the performance of our algorithms to other competitive algorithms, in terms of running time and L2 cache misses. The results for L2 cache misses include the three kinds of misses: compulsory, conflict, and capacity. We primarily report results on a small-cache machine but show that other machines lead to similar results.

The implementations of sorting algorithms described earlier were gathered from the best source we could identify, and all of the programs were written in C. We are confident that these implementations are of high quality. In preliminary experiments, we tested many sorting methods that we do not report here because they are much slower than methods such as MBM radixsort. These included UNIX quicksort, splaysort, and elementary techniques such as insertionsort.

The time measured in each case is to sort an array of pointers to strings; the array is returned as output. Thus an in-place algorithm operates directly on this array and requires no additional structures. For the purpose of comparing

Table III. Running Time (Milliseconds) to Sort with Each Method on Duplicates, no Duplicates, and URLs

	Data set					
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Duplicates						
Burstsort-List	68	267	830	2,800	10,040	36,250
Burstsort-Array	58	218	630	2,220	7,950	29,910
No duplicates						
Burstsort-List	71	272	1,000	3,360	11,340	43,080
Burstsort-Array	61	221	790	2,670	9,280	35,210
URL						
Burstsort-List	121	452	1,730	5,580	21,190	—
Burstsort-Array	110	395	1,530	5,070	17,950	—

the algorithms, it is not necessary to include the parsing time or the time used to retrieve data from the disk, since it is the same for all algorithms. We therefore report the CPU times, not elapsed times, and exclude the time taken to parse the collections into strings. The internal buffers of our machine are flushed prior to each run in order to have the same starting condition for each experiment.

We have used the GNU gcc compiler and the Linux operating system on a 700 MHz Pentium computer with 2 Gb of fast internal memory and a 1 Mb L2 cache with block size of 32 bytes and 8-way associativity with a memory latency of about 100 cycles. In all cases the highest compiler optimization level 03 has been used. The total number of milliseconds of CPU time consumed by the kernel on behalf of the program has been measured, but for sorting only; I/O times are not included. The standard deviation was low. The machine was under light load, that is, no other significant I/O or CPU tasks were running. For small data sets, times are averaged over a large number of runs, to give sufficient precision. For the larger data sets, times are the minimum of 10 runs—that is, the time that is most likely to be free of interference—and the standard deviation was very low. Thus occasional variations due to page faults were not included in the timing. We primarily report experiments on a specific small-cache machine but use results obtained on other machines, with different cache architectures, to explore the extent to which the improvements are machine dependent.

6. RESULTS

All timings are in milliseconds, of the total time to sort an array of pointers to strings into lexicographic order. In the tables, these times are shown unmodified. In the figures, the times are divided by $n \log n$ where n is the number of strings. With such normalization, suggested by Johnson [2002], the performance of an ideal sorting algorithm is a horizontal line.

Bucket representation. The time required to sort *no duplicates*, *duplicates*, and *URL* from Set 1 to Set 6 is shown in Table III. In these results, the size of the root node is 2^8 slots, buckets grow exponentially by a factor of 8, and capacity is 8192 strings. The results clearly separate the two versions of burstsort. Using arrays for buckets is more efficient than using lists, despite the space wastage implicit in the earlier.

Table IV. Bucket-Sorting Methods: Impact on Running Time (Milliseconds) of Different Algorithms for Sorting Buckets, in Array-Based Burstsort

	Data set					
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Duplicates						
MBM radixsort	63	249	760	2,860	10,420	38,620
Multikey quicksort	64	221	630	2,250	8,000	29,710
No duplicates						
MBM radixsort	64	240	930	3,290	11,590	43,360
Multikey quicksort	61	221	790	2,670	9,280	35,210

However, using lists instead of arrays for burstsort is more stable for varying thresholds, as can be seen from Tables V and VI for thresholds 16 and 128. (These tables are considered below.) Also, burstsort using arrays uses more instructions than does burstsort using linked lists, due to simpler code, but the former is more cache efficient as discussed later.

Bucket-sorting mechanism. Using the array implementation, we tested a range of methods for sorting bucket, including insertionsort, quicksort, MBM radixsort, multikey quicksort, and adaptive radixsort. MBM radixsort and multikey quicksort were significantly more efficient than the others for all data sets.

Timings are reported in Table IV. In contrast to MBM radixsort, multikey quicksort is not a stable sorting algorithm. As can be seen, however, multikey quicksort is by a significant margin the more efficient approach—despite the fact that it is slightly less efficient, for small data sets, in our other experiments.

Bucket capacity. In our first experiment with capacity, we tested the efficiency of four bucket sizes: 16, 128, 1024, and 8192 pointers; for the array implementation, we used factor-of-8 bucket growth. Results are shown in Tables V and VI. Note that Set 6 is omitted in some cases, and Set 5 in one case, because for small capacities the total space requirements (due to growth in the trie) exceed physical memory. Note also that these results are based on smaller numbers of runs than in the other tables, hence the reduced precision. The size 8192 was consistently the most efficient, except for the largest data sets, and we have used this value in all remaining experiments. These results suggest that there may be further improvement available by adapting the capacity to the size of the set of strings; doing so in a principled way is a subject for further research.

In our second experiment with capacity, we varied the way in which the buckets grew in the array version. In one case, the buckets grew over 4 levels and the size of each successive level differed by powers of 8, whereas in the other case, the buckets grew over 6 levels and the size of each successive level differed by powers of 4. As can be seen in Table VII, we did not observe any significant difference between the two approaches.

Size of root node. We tested the effect of using either 1 byte or 2 bytes to index the root node, that is, the node could consist of either 2^8 or 2^{16} pointers.

Table V. Bucket Capacities: Impact on Running Time (Milliseconds), List Implementation

	Data set					
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Duplicates						
1 6	190	660	1,560	5,970	23,050	—
128	70	300	860	3,440	12,560	51,340
1024	70	260	770	2,900	10,540	40,660
8192	68	267	830	2,800	10,040	36,250
16384	80	260	840	2,980	10,390	36,280
32768	70	250	790	2,920	10,110	34,760
No duplicates						
16	180	570	1,920	6,680	24,930	—
128	70	280	1,120	4,100	14,410	53,950
1024	70	260	950	3,410	12,280	46,440
8192	71	272	1,000	3,360	11,340	43,080
16384	70	270	1,030	3,420	11,410	41,190
32768	70	260	970	3,380	11,420	39,810
URL						
16	610	1,970	5,890	19,410	—	—
128	170	780	3,600	12,770	43,820	—
1024	140	480	1,710	7,720	35,590	—
8192	121	452	1,730	5,580	21,190	—
16384	120	440	2,000	6,090	22,020	—
32768	130	440	1,630	6,930	22,640	—
Genome						
16	210	770	2,610	8,750	27,980	74,430
128	90	350	1,380	4,890	17,500	64,850
1024	110	330	1,120	3,790	12,780	49,520
8192	81	316	1,120	3,740	11,710	43,350
16384	90	370	1,170	3,530	11,330	40,700
32768	120	340	1,120	3,580	11,550	40,510

The latter was consistently the most efficient, with improvements observed across all the collections of about 5%–10%.

Comparison to Previous Algorithms

Table VIII shows the running times for a wide range of sorting algorithms on *duplicates*. These are startling results. The previous methods show only moderate performance gains in comparison with each other, and there is no clear winner amongst the four best techniques. In contrast, burstsort is the fastest for all but the smallest set size tested, of 100,000 strings, where it is second only to MBM radixsort. For the larger sets, the improvement in performance is dramatic: it is more than twice as fast as MBM radixsort, and almost four times as fast as an efficient quicksort.

The rate of increase in time required per key is shown in Figure 3, where as discussed the time is normalized by $n \log n$. As can be seen, burstsort shows a low rate of growth compared to the other efficient algorithms. For example, the normalized time for MBM radixsort grows from approximately 0.00014 to approximately 0.00025 from Set 1 to Set 6, whereas burstsort does not grow at all.

Table VI. Bucket Capacities: Impact on Running Time (Milliseconds), Array Implementation

	Data set					
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Duplicates						
16	248	882	2,290	13,560	167,880	—
128	78	292	820	3,690	20,970	214,800
1024	63	223	630	2,430	9,840	45,840
8192	58	218	630	2,220	7,950	29,910
16384	60	220	640	2,290	8,080	30,390
32768	60	220	640	2,380	8,210	28,290
No Duplicates						
16	243	818	3,610	33,520	507,960	—
128	79	284	1,160	5,280	37,430	476,550
1024	66	225	800	2,970	12,190	64,110
8192	61	221	790	2,670	9,280	35,210
16384	60	220	820	2,730	9,390	33,120
32768	60	220	790	2,820	9,370	32,590
URL						
16	779	2,219	6,880	30,900	—	—
128	173	705	2,960	11,160	41,710	—
1024	139	456	1,580	6,430	27,580	—
8192	110	395	1,530	5,070	17,950	—
16384	110	390	1,760	5,420	19,080	—
32768	130	400	1,510	6,600	21,630	—
Genome						
16	220	850	3,310	17,960	171,280	—
128	100	340	1,170	4,430	17,120	67,620
1024	80	270	900	2,860	10,890	45,710
8192	70	258	870	2,830	8,990	31,540
16384	80	290	920	2,770	8,760	30,180
32768	80	280	940	2,980	9,410	30,840

There are several reasons that burstsort is efficient. In typical text the most common words are small, and so are placed under the empty-string pointer and do not have to be sorted. Only buckets with more than one string have to be sorted, and the distinguishing prefix does not participate in the sorting. Most importantly, the algorithm is cache friendly: the strings are accessed in sequence and (with the exception of bursting, which only involves a small minority of strings) once only; the trie nodes are accessed repeatedly, but are collectively small enough to remain in cache. The data structure used for buckets and the trie nodes is arrays, which are the most cache-efficient data structure. Bursting is more efficient using arrays rather than lists. Traversing a list bucket involves two random accesses for each node, one for the string and the other for the node. A bucket is traversed twice, once on bursting and once in the traversal phase.

The first line is a conventional quicksort, optimized for strings; as can be seen, it is around half the speed of multikey quicksort at all data sizes. The radixsorts and multikey quicksort are of similar efficiency, but the relative performance varies with data set size: of these methods, MBM radixsort is the fastest for Set 1, while adaptive radixsort is the fastest for Set 6.

Table VII. Bucket Growth Strategies: Impact on Running Time (Milliseconds). For "Powers of 4," the Stages are 8, 32, 128, 512, 2048, and 8192. For "Powers of 8," the Stages Are 16, 128, 1024, and 8192

	Data set					
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Duplicates						
powers of 8	64	221	630	2,250	8,000	29,710
powers of 4	59	219	630	2,250	8,030	29,780
No duplicates						
powers of 8	61	221	790	2,670	9,280	35,210
powers of 4	61	222	790	2,680	9,300	35,320
URL						
powers of 8	110	395	1,530	5,070	17,950	—
powers of 4	110	396	1,530	5,030	17,760	—
Genome						
powers of 8	70	256	870	2,840	8,970	30,550
powers of 4	70	257	880	2,890	9,140	31,810

Table VIII. Duplicates, Sorting Time for Each Method (Milliseconds)

	Data set					
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Quicksort	122	528	1,770	7,600	30,100	114,440
Multikey quicksort	62	272	920	3,830	14,950	56,070
MBM radixsort	58	238	822	3,650	15,460	61,560
MSD radixsort	72	290	1,000	3,870	14,470	56,790
Adaptive radixsort	74	288	900	3,360	12,410	51,870
Forward-8	146	676	2,030	7,590	28,670	113,540
Forward-16	116	486	1,410	5,120	19,150	74,770
Burstsort	58	218	630	2,220	7,950	29,910

An alternative view of these results is shown in Figure 3, where the time to sort the data is normalized for data set size by dividing by $n \log n$. The burstsorts show much the best asymptotic behavior, with relative time barely growing with data set size.

Similar results are shown in Tables IX and X, for *no duplicates* and *URL*, respectively. For the former, relative performance is almost identical to *duplicates*. Figure 4 shows the normalized running times for the algorithms on *no duplicates*. Burstsort is again the fastest for all but the smallest data set, and almost twice as fast as the next best method for the largest data set. Elimination of duplicates has had little impact on relative performance. However, the results on *URL* are surprisingly different. Conventional and multikey quicksort have both outperformed MBM radixsort, which was in many cases the most efficient of the existing methods on the other data. The burstsorts have shown even better performance for the small data sets than previously, and again have good asymptotic behavior, as illustrated in Figure 5.

Table XI shows the results for *genome*, a data set with very different properties: strings are fixed length, the alphabet is small (though all implementations allow for 256 characters), and the distribution of characters is close to uniform

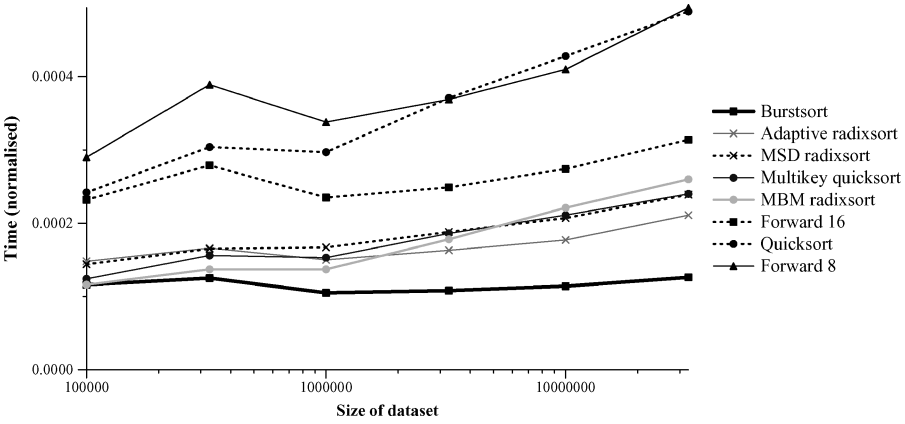

Fig. 3. Duplicates. The vertical scale is time in milliseconds divided by $n \log n$.

Table IX. No Duplicates. Running Time (Milliseconds) to Sort with Each Method

	Data set					
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Quicksort	141	561	2,380	9,600	35,890	138,790
Multikey quicksort	67	267	1,090	4,240	15,790	60,260
MBM radixsort	61	230	940	4,080	15,680	61,340
MSD radixsort	73	290	1,120	4,110	16,250	65,200
Adaptive radixsort	77	275	1,060	3,850	14,590	59,630
Forward-8	149	654	2,540	8,920	34,020	135,280
Forward-16	124	465	1,860	6,420	25,930	100,370
Burstsort	62	219	790	2,650	9,040	33,950

Table X. URL. Running Time (Milliseconds) to Sort with Each Method

	Data set				
	Set 1	Set 2	Set 3	Set 4	Set 5
Quicksort	202	802	3,090	11,160	39,760
Multikey quicksort	134	504	1,970	8,100	32,540
MBM radixsort	206	808	3,140	13,450	53,650
MSD radixsort	156	591	2,690	11,300	48,120
Adaptive radixsort	151	544	2,280	8,290	33,580
Forward-8	459	1,986	6,980	23,750	91,330
Forward-16	315	1,207	4,300	14,330	55,860
Burstsort	110	395	1,530	5,000	17,740

random. Burstsort is relatively even more efficient for this data than for the words drawn from text, and is the fastest on all data sets. For burstsort, as illustrated in Figure 6, the normalized cost per string declines with data set size; for all other methods, the cost grows.

The *URL* data presents yet another distribution. The strings are long and their prefixes are highly repetitive. As illustrated in Figure 5, burstsort is much the most efficient at all data set sizes. Taking these results together, relative behavior is consistent across all sets of text strings—skew or not, with duplicates

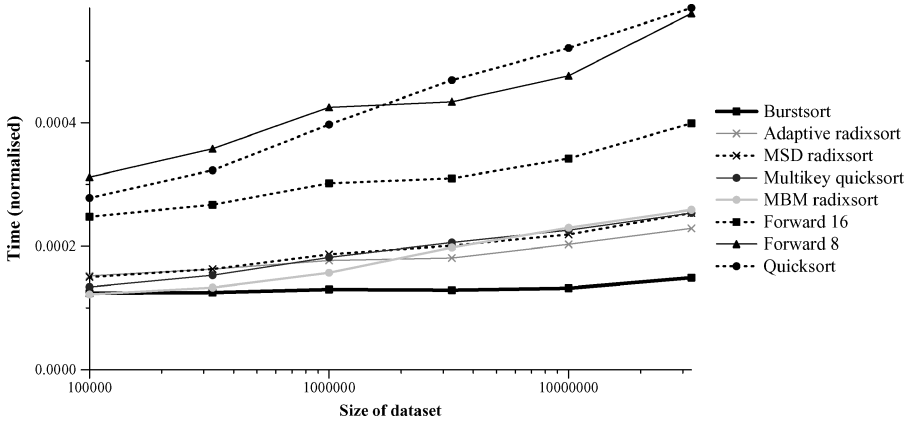

Fig. 4. No duplicates. The vertical scale is time in milliseconds divided by $n \log n$.

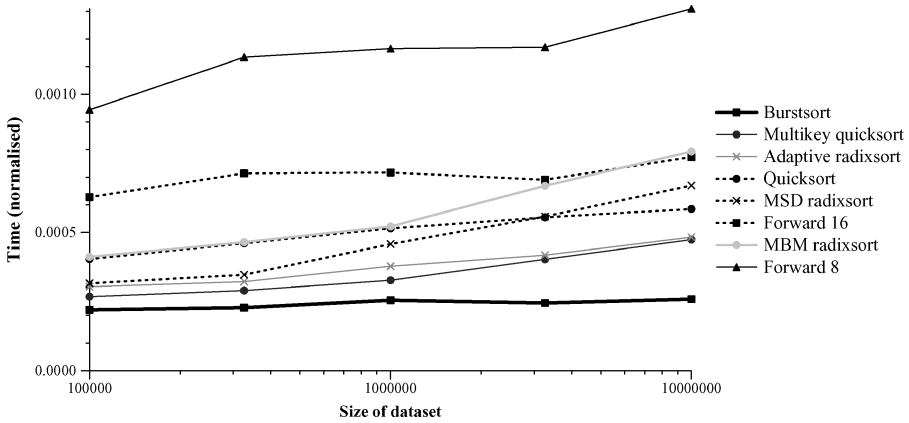

Fig. 5. URL sorting time. The vertical scale is time in milliseconds divided by $n \log n$.

or not. For all of these sets of strings drawn from real data, burstsort is consistently the fastest method.

We used the *random* data to see if another kind of distribution would yield different results. The behavior of the methods tested is shown in Figure 7. On one hand, burstsort is the most efficient method only for the largest three data sets, and by a smaller margin than previously. On the other hand, the normalized time per string does not increase at all from Set 1 to Set 6, while there is some increase for all of the other methods. (As observed in the other cases, there are several individual instances in which the time per string decreases between set x and set $x + 1$, for almost all of the sorting methods. Such irregularities are due to variations in the data.)

Memory requirements are a possible confounding factor: if burstsort required excessive additional memory, there would be circumstances in which it could not be used. For Set 6 of *duplicates* we observed that the space requirements for burstsort are 790 Mb, between the in-place MBM radixsort's 546 Mb and MSD radixsort's 907 Mb. (The space overhead of the trie is around 1 bit per string.)

Table XI. Genome, Sorting Time for Each Method (Milliseconds)

	Data set					
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Quicksort	156	674	2,580	9,640	35,330	129,720
Multikey quicksort	72	324	1,250	4,610	16,670	62,680
MBM radixsort	72	368	1,570	6,200	23,700	90,700
MSD radixsort	102	442	1,640	5,770	20,550	79,840
Adaptive radixsort	92	404	1,500	4,980	17,800	66,100
Forward-8	246	1,074	3,850	12,640	41,110	147,770
Forward-16	174	712	2,380	7,190	23,290	86,400
Burstsort	70	258	870	2,830	8,990	31,540

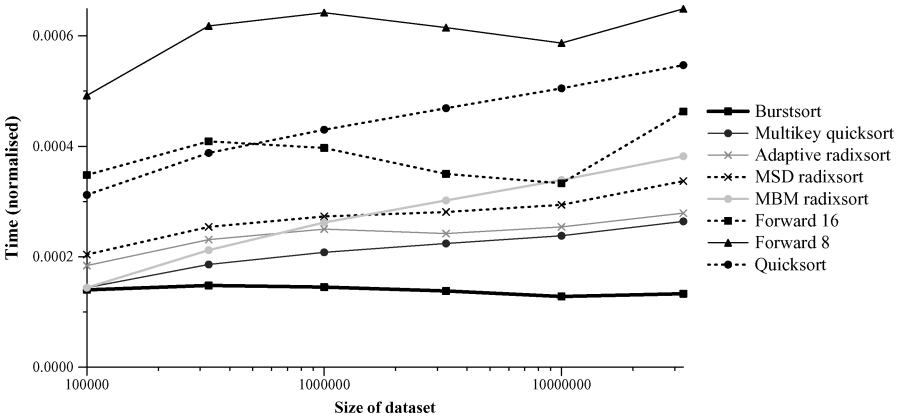

Fig. 6. Genome sorting time. The vertical scale is time in milliseconds divided by $n \log n$.

The highest memory usage was observed by forward-16, at 953 Mb, followed by forward-8 at 941 Mb and adaptive radixsort at 908 Mb. We therefore conclude that only the in-place methods show better memory usage than burstsort.

Other data. In previous work, a standard set of strings used for sorting experiments is of library call numbers [Bentley and Sedgewick 1997],[3] of 100,187 strings (about the size of our Set 1). For this data, burstsort was again the fastest method, requiring 100 ms. The times for multikey quicksort, MBM radixsort, and adaptive radixsort were 106, 132, and 118 ms, respectively; the other methods were much slower.

We have experimented with several other artificially created data sets, hoping to bring out the worst cases in the algorithms. We generated three collections ranging in size from 1 to 10 million strings, as follows.

(A) The length of the strings is 100, the alphabet has only one character, and the size of the collection is 1 million.

(B) The length of the strings ranges from 1 to 100, the alphabet size is small (nine), and the characters appear randomly. The size of the collection is 10 million.

[3]Available from `www.cs.princeton.edu/~rs/strings`.

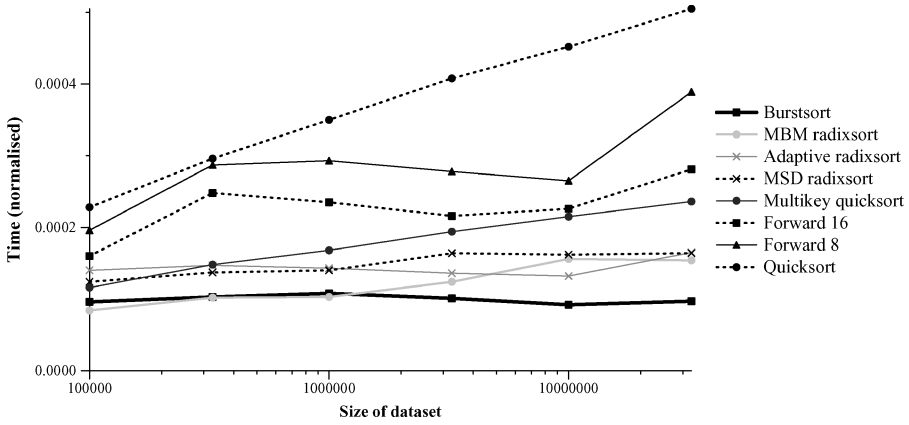

Fig. 7. Random. The vertical scale is time in milliseconds divided by $n \log n$.

Table XII. Artificial Data, Sorting Time for Each Method (Milliseconds)

	Data set		
	A	B	C
Quicksort	1,040	34,440	3,900
Multikey quicksort	11,530	18,750	5,970
MBM radixsort	18,130	40,220	19,620
MSD radixsort	10,580	26,380	5,630
Adaptive radixsort	7,870	20,060	4,270
Forward-8	12,290	38,800	6,580
Forward-16	8,140	27,890	4,450
Burstsort	2,730	10,090	1,420

(C) The length of the strings ranges from 1 to 100, and strings are ordered in increasing size in a cycle. The alphabet has only one character and the size of the collection is 1 million.

Table XII shows the running times. In each case, burstsort is dramatically superior to the alternatives, with the single exception of quicksort on Set A; this quicksort is particularly efficient on identical strings. In Set B, the data has behaved rather like real strings, but with exaggerated string lengths. In Set C, MBM radixsort—in the other experiments, the only method to ever do better than burstsort—is extremely poor.

Cache Efficiency

To test our hypothesis that the efficiency of burstsort was due to its ability to make better use of cache, we measured the number of cache misses for each algorithm and sorting task. We have used `valgrind`, an open-source simulator for investigating cache effects in programs [Seward 2001]; the cache parameters of our hardware were used. To measure the number of misses in UltraSPARC III, we have used the CPU-track utility, which uses CPU performance counters to monitor the behavior of a process. The configurations of the machine used for

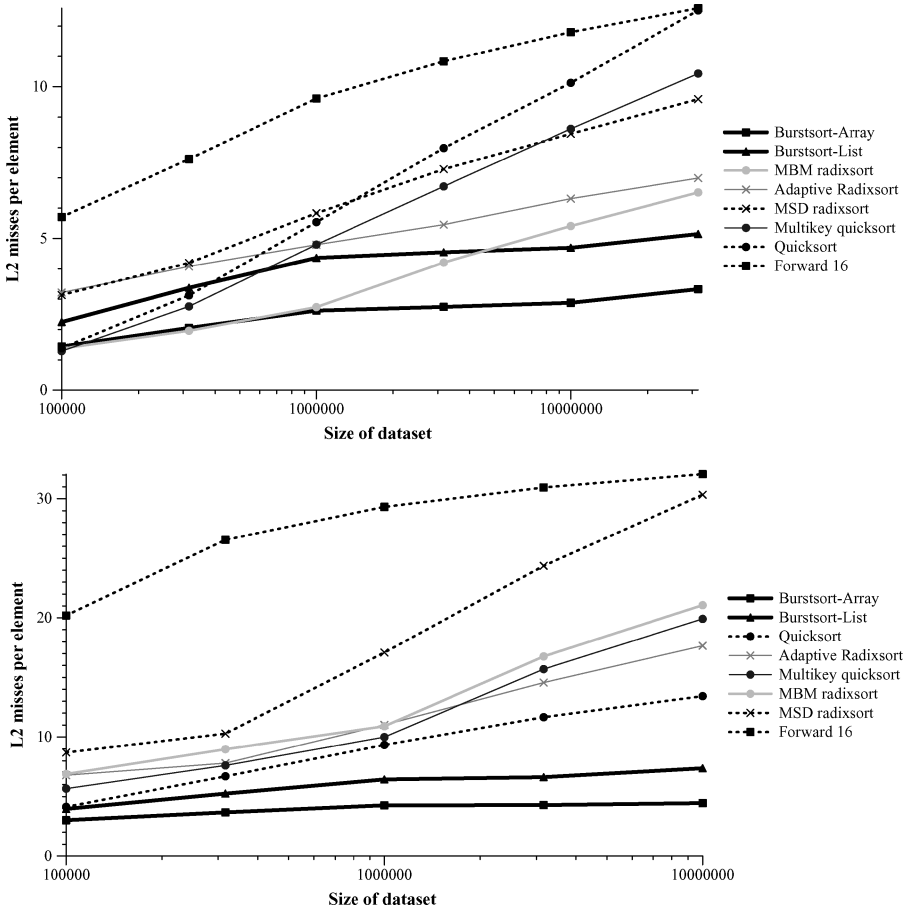

Fig. 8. Cache misses, 1 Mb cache, 8-way associativity, 32 bytes block size. Upper: no duplicates. Lower: URLs.

the experiments are presented in Table I; `calibrator`[4] was used to measure some of the machine configurations. Figures 8, 9, and 10 show the rate of cache misses per key for the collections *no duplicates*, *duplicates*, *genome*, *random*, and *URL*. Figures 8, 9, and 10 are normalized by n (not $n \log n$ as in Figure 4) to show the number of cache misses per string.

For small data sets in the *no duplicates* case, burstsort and MBM radixsort show the greatest cache efficiency, while for large data sets burstsort is clearly superior, as the rate of cache miss grows relatively slowly across data set sizes.

For the *URL* data, the difference in cache efficiency is remarkable. For all set sizes, burstsort has less than a quarter of the cache misses of the next best method. The URL collection has a large distinguishing prefix, as most of the strings have the same first few characters. Let us assume 1 million URL strings need to be sorted. The first four characters in each string are identical,

[4]See `homepages.cwi.nl/~manegold/Calibrator`.

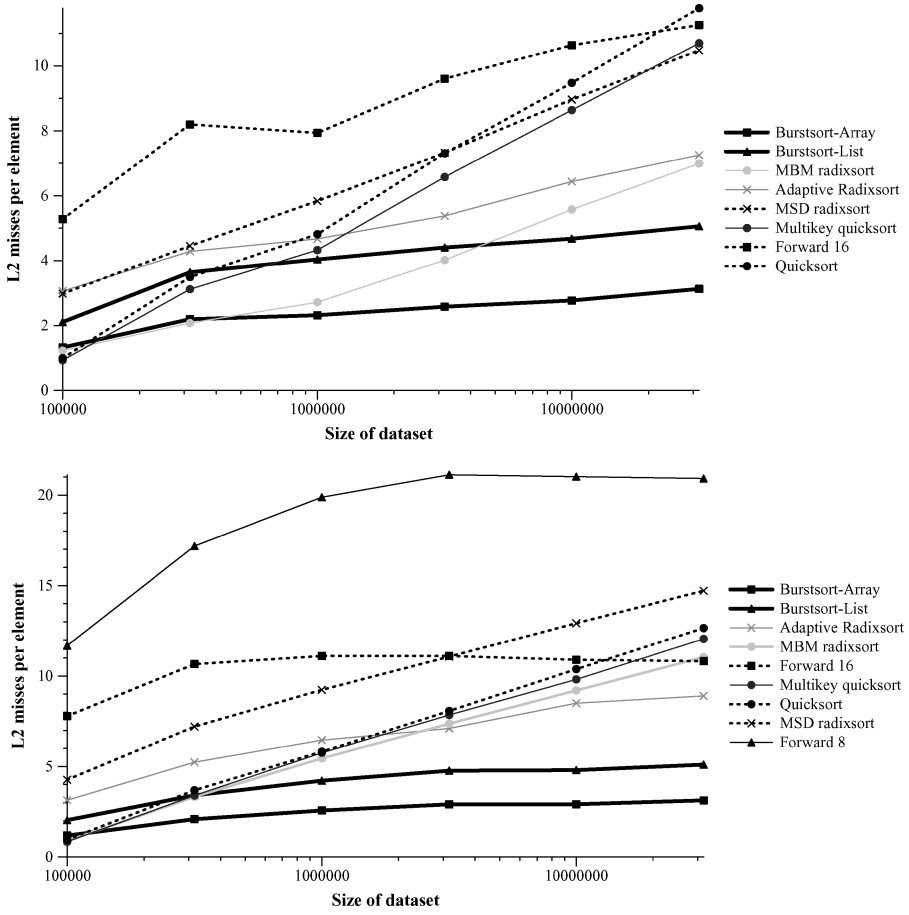

Fig. 9. Cache misses, 1 Mb cache, 8-way associativity, 32 bytes block size. Upper: duplicates. Lower: genome.

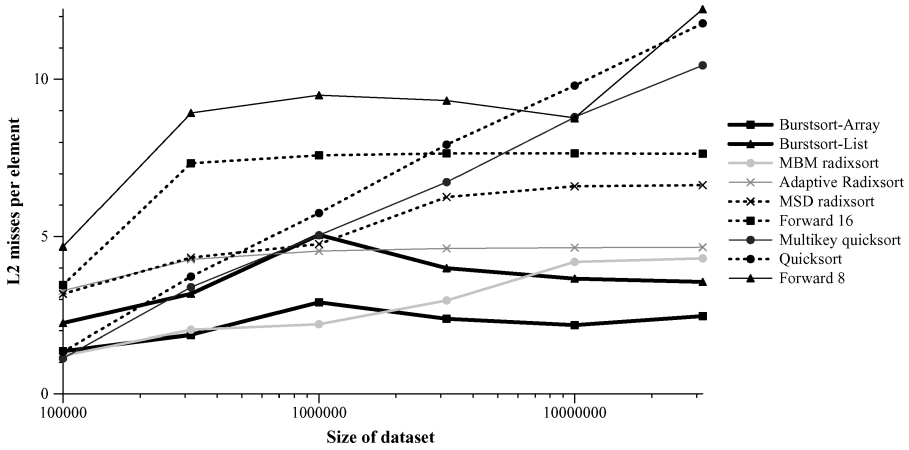

Fig. 10. Cache misses for random collection, 1 Mb cache, 8-way associativity, 32 bytes block size.

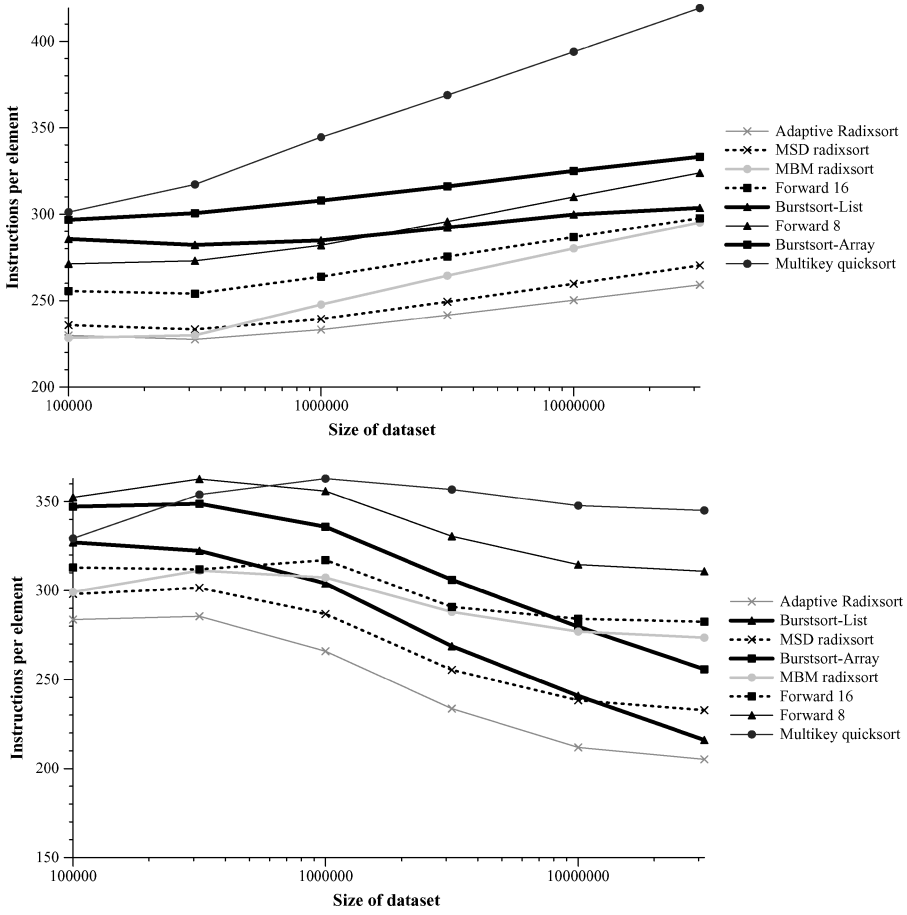

Fig. 11. Instructions per element. Upper: no duplicates. Lower: genome.

say "http". With conventional radixsorts, they would have to be accessed once for each of these characters, whereas in burstsort, as the number of strings in the bucket reaches the threshold (say 8192), the bucket is burst creating another level and all the strings end up in another bucket in the second level and it is burst again; this occurs four times in a loop. Only the 8192 strings are involved and they are fetched just once. The remaining 991,808 strings are directly inserted into the 4th level of the trie, reading four characters of each string in just one access.

We then investigated each of the sorting methods in more detail. For quicksort, the instruction count is high, for example, 984 instructions per key on Set 6 of *duplicates*; the next highest count was 362, for multikey quicksort. Similar results were observed on all data sets. As with most of the methods, there is a steady logarithmic increase in the number of cache misses per key. For multikey quicksort, the number of instructions per key is always above the radixsorts, by about 100 instructions. Although relatively cache efficient in many cases, it deteriorates most rapidly with increase in data set size.

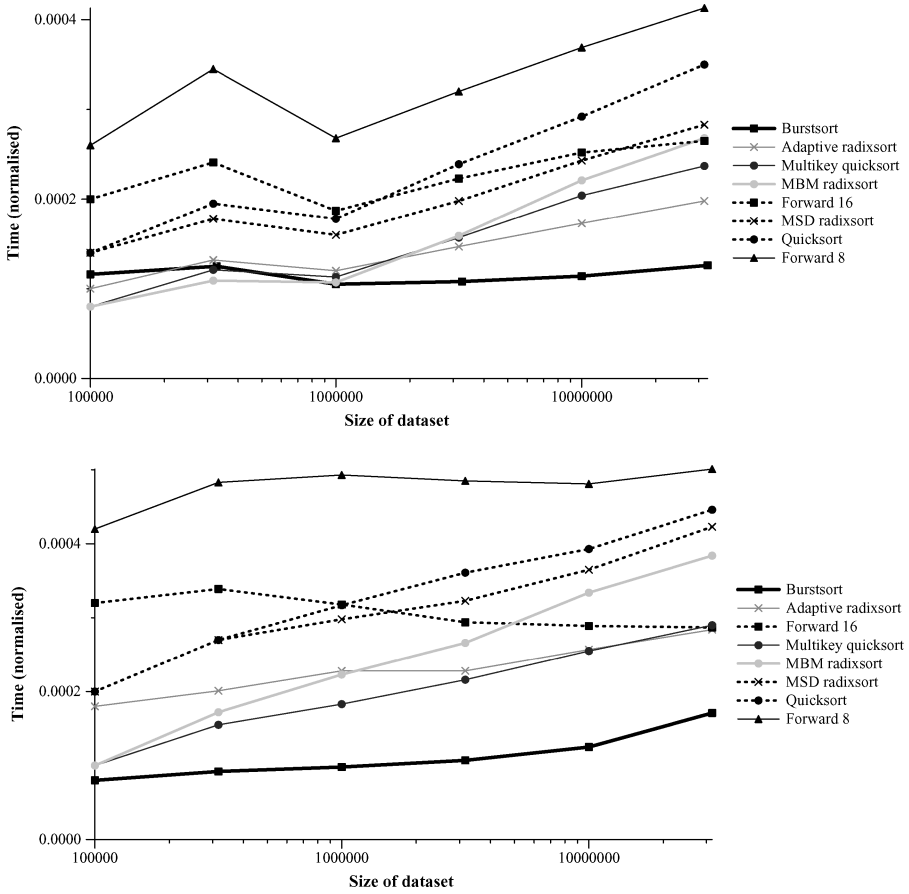

Fig. 12. Architecture used: Pentium IV; The vertical scale is time in milliseconds divided by $n \log n$. Upper: duplicates. Lower: genome.

For smaller string sets, MBM radixsort is efficient, but once the set of pointers to the strings is too large to be cached, the number of cache misses rises rapidly. MSD radixsort is very efficient in terms of the number of instructions per key, next only to adaptive radixsort, and for *no duplicates* the number of cache misses rises relatively slowly compared to the other radixsorts, again next only to adaptive radixsort. Adaptive radixsort is the most efficient of the previous methods in terms of the number of instructions per key in all collections except *random*. The rate of cache miss rises slowly. Thus, while MBM radixsort is more efficient in many of our experiments, adaptive radixsort appears asymptotically superior. In contrast, forward 8 and 16 are the least efficient of the previous radixsorts, in cache misses, instruction counts, and memory usage.

Burstsort is by far the most efficient in all large data sets, primarily because it uses the CPU cache effectively—indeed, it uses around 25% more instructions than adaptive radixsort. Instruction counts per string for each algorithms are shown in Figure 11. Quicksort is omitted due to very high number

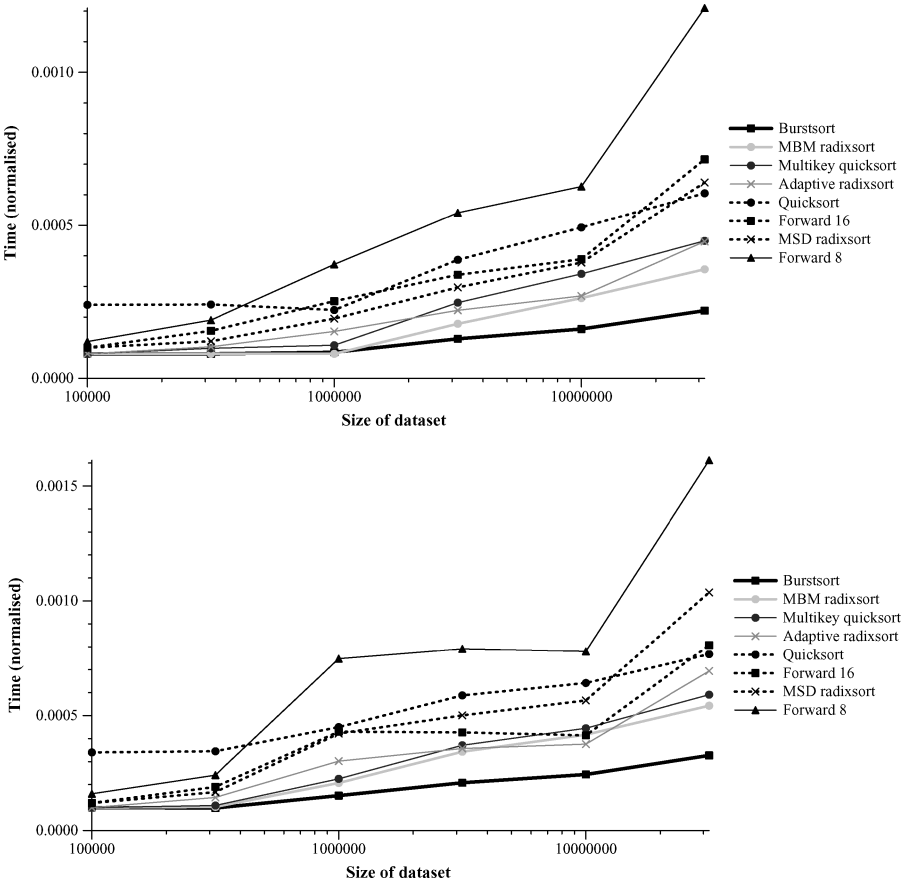

Fig. 13. Architecture used: UltraSPARC III; the vertical scale is time in milliseconds divided by $n \log n$. Upper: duplicates. Lower: genome.

of instructions; burstsort with lists is included because it has a low-instruction count, despite being slower than burstsort with arrays. As discussed above, all other burstsort results in this section are for burstsort with arrays.

These results demonstrate that cache misses are a key determinant of performance. We showed earlier that burstsort is around twice as fast as the best of the other algorithms, yet it is far from the best method by instruction count. For small sets, where most of the data fits in the cache, the effect of cache misses is small, but as the data size grows they become crucial. It is here that the strategy of only referencing each string once is so valuable.

Recent work on cache-conscious sorting algorithms for numbers [LaMarca and Ladner 1997; Rahman and Raman 2000, 2001; Xiao et al. 2000] has shown that, for other kinds of data, taking cache properties into account can be used to accelerate sorting. However, these sorting methods are based on elementary forms of radixsort, which do not embody the kinds of enhancements used in MBM radixsort and adaptive radixsort. The improvements cannot readily be adapted to variable-sized strings.

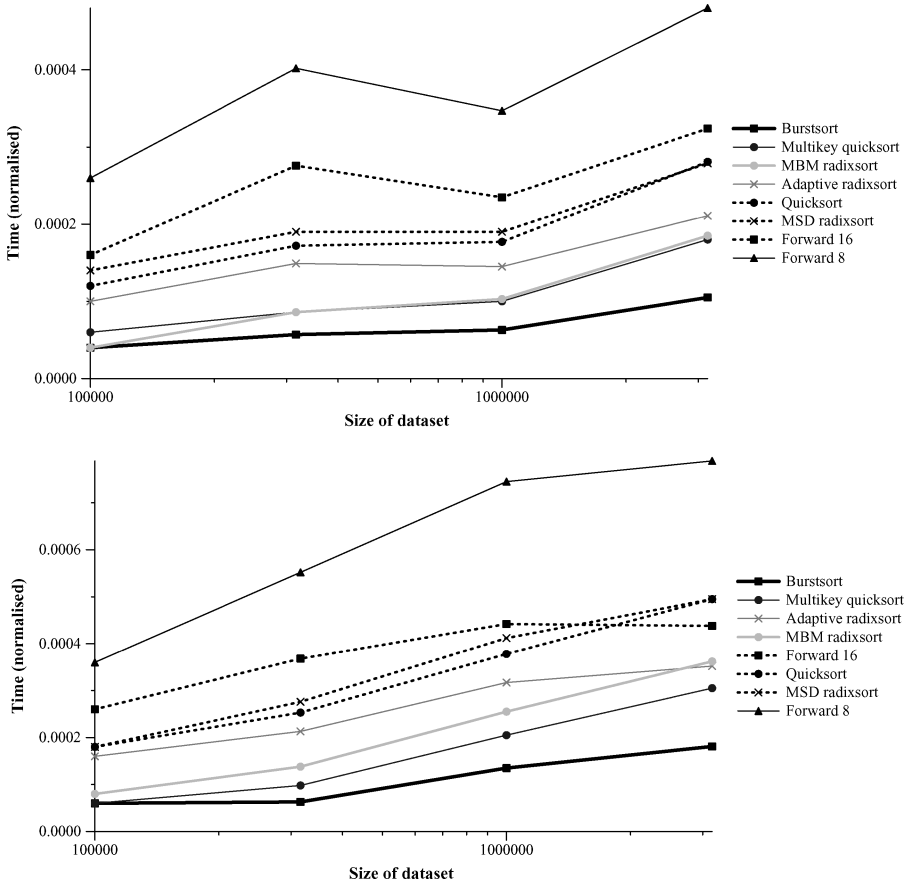

Fig. 14. Architecture used: PowerPC; The vertical scale is time in milliseconds divided by $n \log n$. Upper: duplicates. Lower: genome.

Other Architectures

Most of the experiments reported so far were undertaken on a Pentium III. Other machines have different cache architectures, and it is reasonable to suppose that the relative performance of the algorithms will change.

To explore this possibility, we ran the algorithms—using the same parameters—on three other machines, a Pentium IV, an UltraSPARC III, and a PowerPC. Typical results are shown in Figures 12, 13, and 14. As can be seen, relative efficiency is little different to that observed on the Pentium III.

L2 cache misses on the Pentium IV and the UltraSPARC III are illustrated in Figure 15. Despite the fact that the cache architectures of the two machines have significant differences, the relative rates of cache miss are virtually indistinguishable. These results are dramatic confirmation that a cache-conscious algorithm can be significantly more efficient than the alternatives, independent of the cache design.

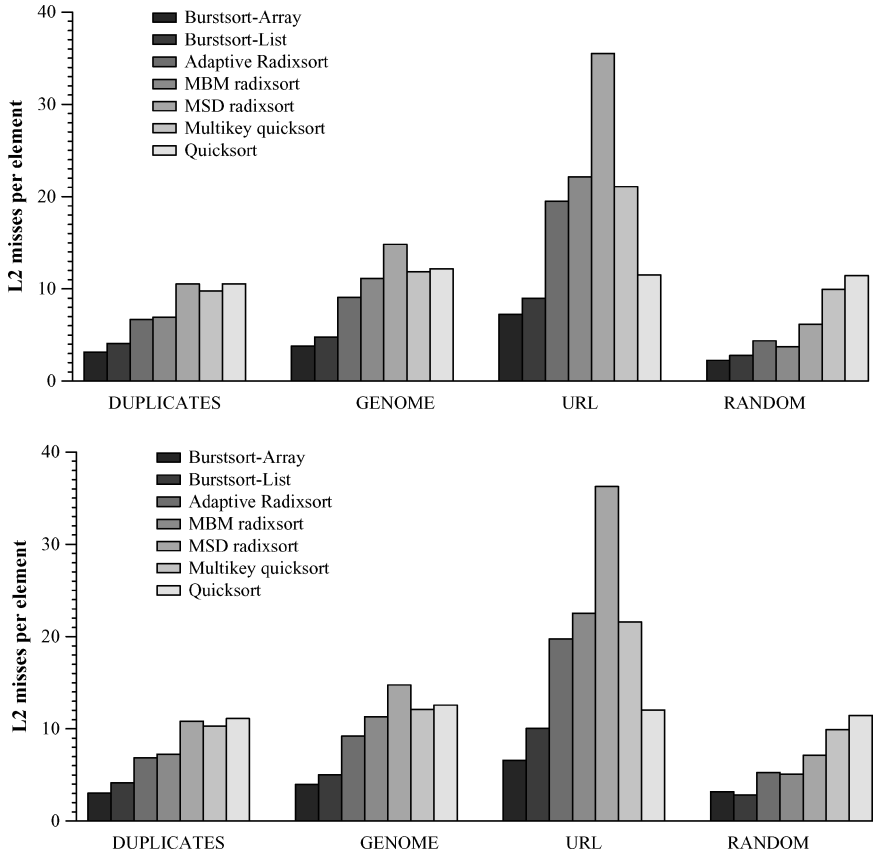

Fig. 15. L2 cache misses for the largest set size for each collection. Upper: UltraSPARC III. Lower: Pentium IV.

7. CONCLUSIONS

We have proposed a new algorithm, burstsort, for fast sorting of strings in large data collections. It is based on the burst trie, an efficient data structure for managing sets of strings in sort order. To evaluate the performance of burstsort, we have compared it to a range of string-sorting algorithms. Our experiments show that it is about as fast as the best algorithms on smaller sets of keys—of 100,000 strings—and is the fastest by almost a factor of 2 on larger sets, and shows much better asymptotic behavior.

The main factor that makes burstsort more efficient than the alternatives is the low rate of cache miss on large data sets. The trend of current hardware is for processors to get faster and memories to get larger, but without substantially speeding up, so that the number of cycles required to access memory continues to grow. In this environment, algorithms that make good use of cache are increasingly more efficient than the alternatives. It is reasonable to suppose that burstsort could be further improved by being tuned to the parameters of a specific cache architecture but, as our experiments show, it is substantially

faster than other sorting methods on a variety of processors. Our work shows that burstsort is by a significant margin the most efficient way to sort a large set of strings.

ACKNOWLEDGMENTS

This paper incorporates work previously published in "Efficient trie-based sorting of large sets of strings," R. Sinha and J. Zobel, *Proceedings of Australasian Computer Science Conference*, Adelaide, Australia, M. Oudshoorn, Ed., February, 2003, and in "Cache-conscious sorting of large sets of strings with dynamic tries," R. Sinha and J. Zobel, *Proceedings of ALENEX Workshop on Algorithm Engineering and Experiments*, Baltimore, Maryland, R. Ladner, Ed., January, 2003. This research was supported by the Australian Research Council.

REFERENCES

ANDERSSON, A. AND NILSSON, S. 1994. A new efficient radix sort. In *IEEE Symposium on the Foundations of Computer Science*. IEEE Computer Society, Santa Fe, NM. 714–721.

ANDERSSON, A. AND NILSSON, S. 1998. Implementing radixsort. *ACM J. Exp. Algorithmics 3*, 7.

ARGE, L., FERRAGINA, P., GROSSI, R., AND VITTER, J. S. 1997. On sorting strings in external memory. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*. ACM Press, El Paso. 540–548.

BENTLEY, J. AND SEDGEWICK, R. 1997. Fast algorithms for sorting and searching strings. In *Proceedings of Annual ACM–SIAM Symposium on Discrete Algorithms*. ACM/SIAM, New Orleans, LA, 360–369.

BENTLEY, J. L. AND MCILROY, M. D. 1993. Engineering a sort function. *Software—Practice and Experience 23*, 11, 1249–1265.

HARMAN, D. 1995. Overview of the second text retrieval conference (TREC-2). *Information Processing & Management 31*, 3, 271–289.

HAWKING, D., CRASWELL, N., THISTLEWAITE, P., AND HARMAN, D. 1999. Results and challenges in web search evaluation. In *Proceedings of World-Wide Web Conference*. Elsevier North-Holland, Toronto.

HEINZ, S. AND ZOBEL, J. 2002. Practical data structures for managing small sets of strings. In *Proceedings of Australasian Computer Science Conference*, M. Oudshoorn, Ed. Australian Computer Society, Melbourne, 75–84.

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.

HOARE, C. A. R. 1962. Quicksort. *Comput. J. 5*, 1, 10–15.

JOHNSON, D. S. 2002. A theoretician's guide to the experimental analysis of algorithms. In *Proceedings of the 5th and 6th DIMACS Implementation Challenges*, M. H. Goldwasser, D. S. Johnson, and C. C. McGeoch, Eds. American Mathematical Society, Providence, RI.

LAMARCA, A. AND LADNER, R. E. 1997. The influence of caches on the performance of sorting. In *Proceedings of Annual ACM–SIAM Symposium on Discrete Algorithms*. ACM Press, New Orleans, LA, 370–379.

MCILROY, P. M., BOSTIC, K., AND MCILROY, M. D. 1993. Engineering radix sort. *Comput. Syst. 6*, 1, 5–27.

MOFFAT, A., EDDY, G., AND PETERSSON, O. 1996. Splaysort: Fast, versatile, practical. *Software—Practice and Experience 26*, 7, 781–797.

NILSSON, S. 1996. Radix sorting & searching. Ph.D. Thesis, Department of Computer Science, Lund, Sweden.

RAHMAN, N. AND RAMAN, R. 2000. Analysing cache effects in distribution sorting. *ACM J. Exp. Algorithmics 5*, 14.

RAHMAN, N. AND RAMAN, R. 2001. Adapting radix sort to the memory hierarchy. *ACM J. Exp. Algorithmics 6*, 7.

SEDGEWICK, R. 1998. *Algorithms in C, 3rd ed*. Addison-Wesley Longman, Reading, MA.
SEWARD, J. 2001. Valgrind—memory and cache profiler. `http://developer.kde.org/~sewardj/docs-1.9.5/cg_techdocs.html`.
SINHA, R. 2002. Fast sorting of strings with dynamic tries. Minor Thesis, School of Computer Science and Information Technology, RMIT University.
XIAO, L., ZHANG, X., AND KUBRICHT, S. A. 2000. Improving memory performance of sorting algorithms. *ACM J. Exp. Algorithmics 5*, 3.

Received July 2003; revised September 2004; accepted October 2004