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

Ranjan Sinha* Justin Zobel†

Abstract

Ongoing changes in computer performance 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 utilise cache particularly 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 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, due primarily to a lower rate of cache miss.

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 in 1993 by Bentley and McIlroy [5]. Splaysort, an adaptive sorting algorithm, was introduced in 1996 by Moffat, Eddy, and Petersson [14]; it uses a combination of the splaytree data structure and insertionsort. Improvements to radixsort for string data were proposed in 1993 by McIlroy, Bostic, and McIlroy [13]; forward and adaptive radixsort for strings were introduced in 1996 by Andersson and Nilsson [2, 15]; a hybrid of quicksort and MSD radixsort named three-way radix quicksort was introduced in 1997 by Bentley and Sedgewick [4, 18]; and, as an extension to keys that are made up of components, three-way radix quicksort was extended by Bentley and Sedgewick in 1997 [3] to give multikey quicksort.

While these algorithms are theoretically attractive, they have practical flaws. In particular, they show poor locality of memory accesses. This flaw 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 to 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 pro-actively 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 [12]: an algo-

*School of Computer Science and Information Technology, RMIT University, GPO Box 2476V, Melbourne 3001, Australia. rsinha@cs.rmit.edu.au

†School of Computer Science and Information Technology, RMIT University, GPO Box 2476V, Melbourne 3001, Australia. jz@cs.rmit.edu.au

rithm 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 particularly efficient with regard to cache. The degree to which algorithms can effectively utilise cache is increasingly a key performance criterion [12, 20]. Addressing this issue for string sorting is the subject of our research.

We propose a new sorting algorithm, *burstsort*, which is based on the burst trie [9]. A burst trie is a collection of small data structures, or *containers*, 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 containers, 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 position-wise, 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 string-wise, 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, 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 found that burstsort is typically the fastest 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 it has better asymptotic behaviour.

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 Quicksort was developed in 1962 by Hoare [10]. Bentley and McIlroy's variant of quicksort was proposed in the early 1990s [5] and has since then been the dominant sort routine used in most libraries.

The algorithm 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 [15] that is specifically tailored for character strings, designated as *Quicksort*.

Multikey quicksort Multikey quicksort was introduced by Sedgewick and Bentley in 1997 [3]. 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 [18]. With the character-wise approach, the length of the shared prefix is known at each stage. However, some of the disadvantages of quicksort are still present. Each char-

acter 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 Sedgewick [3], designated as *Multikey quicksort*.

Radixsort Radixsort is 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) [18]. It is difficult to apply the LSD approach to a string-sorting application because of 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 MSD radixsort 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 [15], designated as *MSD radixsort*.

MBM radixsort Early high-performance string-oriented variants of MSD radixsort were presented by McIlroy, Bostic, and McIlroy [13]. Of the four variants, we found programC to be typically the fastest for large datasets. 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 [3], we found it to be the fastest array-based string sort. In our experiments it is designated as *MBM radixsort*.

Forward radixsort Forward radixsort was developed by Andersson and Nilsson in 1994 [1, 15]. 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 [15], who developed 8-bit and 16-bit versions, designated as *Forward-8* and *Forward-16*.

Adaptive radixsort Adaptive radixsort was developed by Nilsson in 1996 [15]. 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 [15], 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 [8, 9]. 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 The burst trie is a form of trie that is efficient for handling sets of strings of any size [8, 9]. It resides in memory and stores strings in approximate sort order. A burst trie is comprised of three distinct components: a set of strings, a set of containers, and an access trie. A *container* is a small set of strings, stored in a simple data structure such as an array or a binary search tree. The strings that are stored in a container 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 containers. 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 container, and a single empty-string pointer to a container. A burst trie is shown in Figure 1. Strings in the burst trie are "bat", "barn", "bark", "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 container. Second is by *bursting*, the process of replacing a container at depth d by a trie node and a set of new containers at depth $d + 1$; all the strings in the original container are distributed in the containers in the newly created node. A container is burst whenever it contains more than a fixed number L of strings. Though the container is an unordered structure, the containers themselves are in sort order, and due to their small size can be sorted rapidly.

A question is how to represent the containers. In our earlier implementations we considered linked lists and other structures but the best method we have identified is to use arrays. In this approach, empty containers 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 container 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 practice, with our test data sets, the space overhead of the trie is around one bit per string.

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 container in which the string should reside. If the container is at a depth of $d = n + 1$, the container is under the empty-string pointer. The standard insertion algorithm for the data structure used in the container is used to insert the strings into the containers. For an array, a pointer to the string is placed at the left-most free slot.

To maintain the limit L on container size, the access trie must be dynamically grown as strings are inserted. This is accomplished by *bursting*. When the number of strings in a container exceeds L, a new trie node is created, which is linked into the trie in place of the container. The $d + 1$th character of the strings in the container is used to partition the strings into containers 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 [9].) 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. In the context of burst tries, and in work completed more recently, we have evaluated the effect of varying parameters.

Burstsort The 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] Burstsort uses the burst trie data structure, which maintains the strings in sorted or near-sorted order. The trie structure is used to divide the strings into containers, which are then sorted using other methods. As is true for all trie sorts, placing the string in a container requires reading of at most the distinguishing prefix, and the characters in the

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

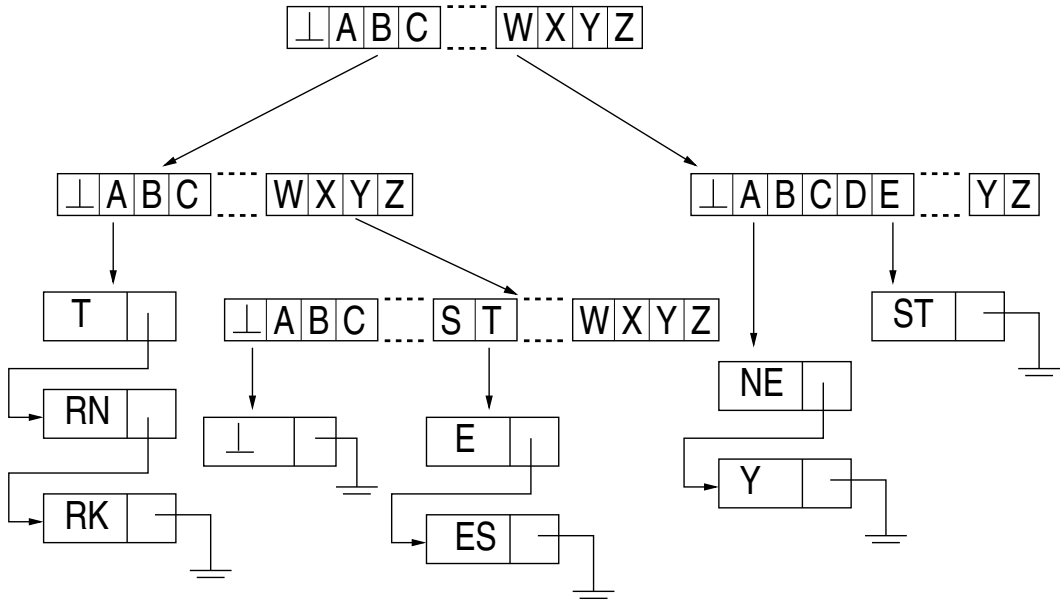

Figure 1: *Burst trie with four trie nodes, five containers, and nine strings, without duplicates.*

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 container 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 container, then again when the container 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.

With a typical set of strings, most leaf nodes in the access trie would be expected to have a reasonable number of containers, in the range 10–100 for an alphabet of 256 characters. Choosing $L = 8,192$ means that container sizes will typically be in the range 100 to 1,000, allowing fast sort with another sorting method. In preliminary experiments $L = 8,192$ gave the best results overall. Exploring the behaviour of this parameter is a topic for further research.

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 as given earlier.

4 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 project [6, 7], and are alphabetic strings delimited by non-alphabetic characters in web pages (after removal of tags, images, and other non-text information). The web URLs have been drawn from the same collection. The genomic strings are from GenBank. For word and

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

Table 1: *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	0.92354	2.355	5.769	12.898	—
Word Occurrences ($\times 10^5$)	1	3.162	10	31.623	100	—

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 thirty million word occurrences, of which just over seven 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 [17], 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. 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 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. 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 datasets, times are averaged over a large number of runs, to give sufficient precision.

5 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 normalisation, suggested by Johnson [11], the performance of an ideal sorting algorithm is a horizontal line.

Table 2 shows the running times for the algorithms on *duplicates*. These are startling results. The previous methods show only moderate performance gains in comparison to 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 2, where as discussed the time is normalised by $n \log n$. As can be seen, burstsort shows a low rate of growth compared to the other efficient algorithms. For example, the normalised 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.

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 containers 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.

Figure 3 shows the normalised 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

Table 2: *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

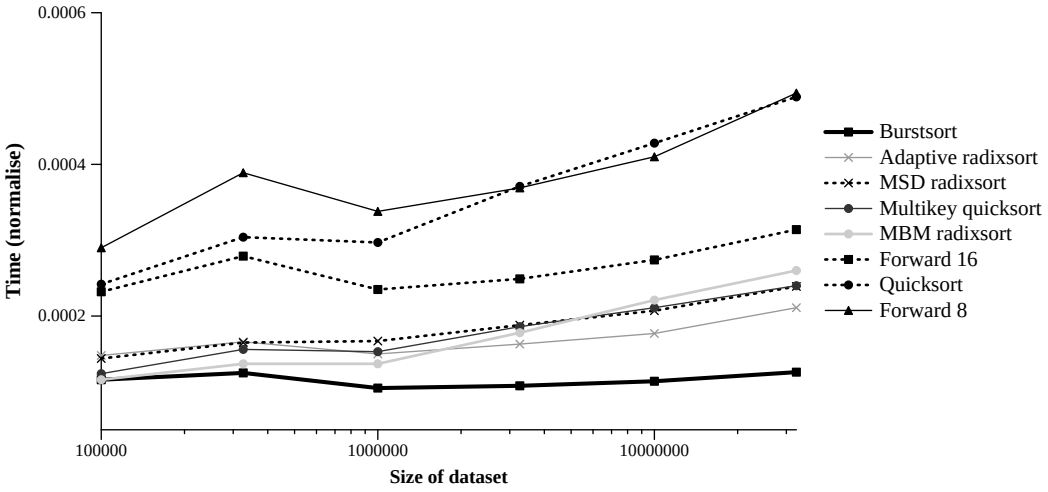

Figure 2: *Duplicates. The vertical scale is time in milliseconds divided by* $n \log n$.

set. Elimination of duplicates has had little impact on relative performance.

Table 3 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 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 4, the normalised 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 behaviour is consistent across all sets of text strings—skew or not, with duplicates 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 behaviour of the methods tested is shown in Figure 6. On the 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 normalised 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

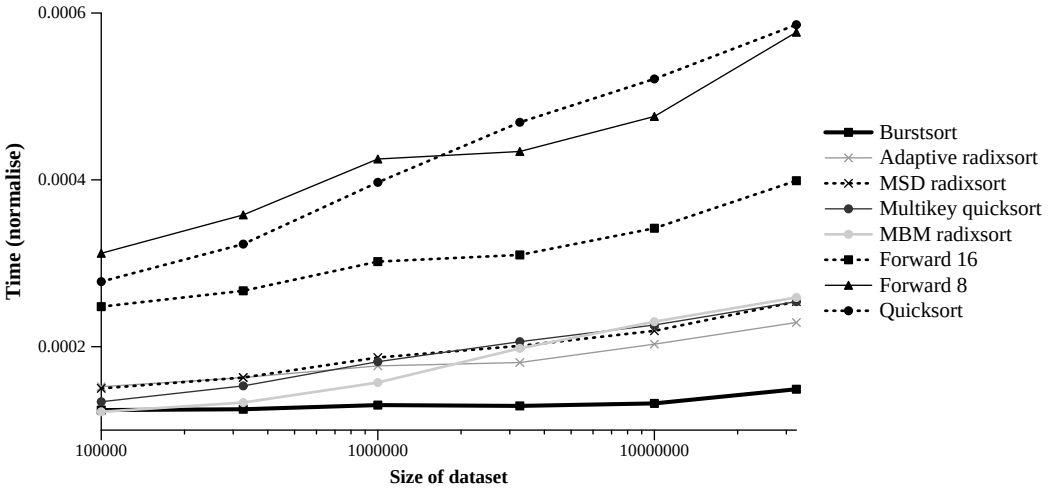

Figure 3: *No duplicates. The vertical scale is time in milliseconds divided by* $n \log n$.

Table 3: *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

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 adaptive radixsort's 910 Mb. The highest memory usage was observed by MSD radixsort, at 1,993 Mb, followed by forward-8 at 1,632 Mb and forward-16 at 1,315 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 [3],[3] of 100,187 strings (about the size of our SET 1). For this data, burstsort was again the fastest method, requiring 100 milliseconds. The times for multikey quicksort, MBM radixsort, and adaptive radixsort were 106, 132, and 118 milliseconds respectively; the other methods were much slower.

We have experimented with several other artificially-created datasets, hoping to bring out

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

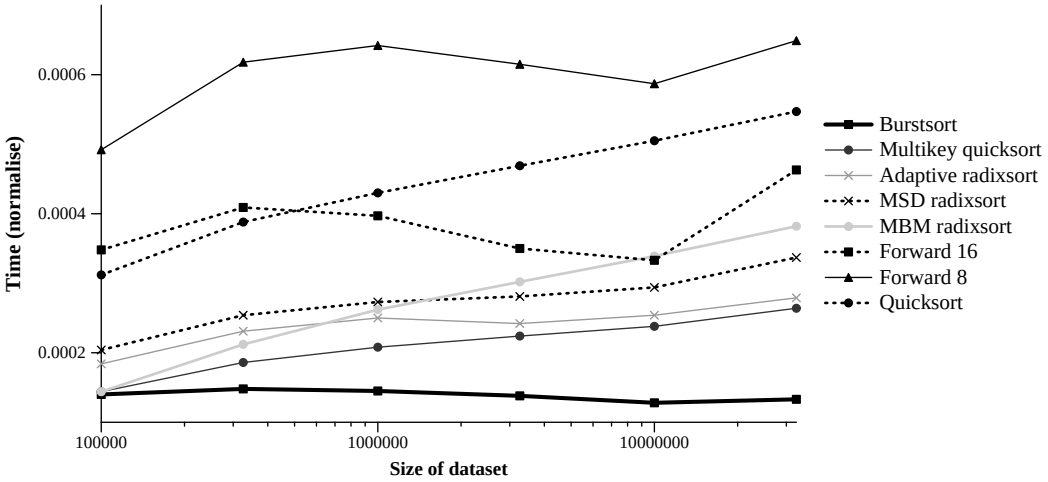

Figure 4: *Genome sorting time. The vertical scale is time in milliseconds divided by* $n \log n$.

the worst cases in the algorithms. We generated three collections ranging in size from one to ten million strings, as follows.

A. The length of the strings is one hundred, the alphabet has only one character, and the size of the collection is one million.

B. The length of the strings ranges from one to a hundred, the alphabet size is small (nine) and the characters appear randomly. The size of the collection is ten million.

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

Table 4 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 `cacheprof`, an open-source simulator for investigating cache effects in programs [19]; the cache parameters of our hardware were used. Figure 7 shows the rate of cache misses per key for *no duplicates* (the upper) and for *URL* (the lower). Similar behaviour to the *no duplicates* case was observed for the other data sets. Figure 7 is normalised by n (not $n \log n$ as in Figure 3) to show the number of cache misses per string.

For small data sets in the *no duplicates* case, burstsort and MBM radixsort shows the greatest cache

Table 4: *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

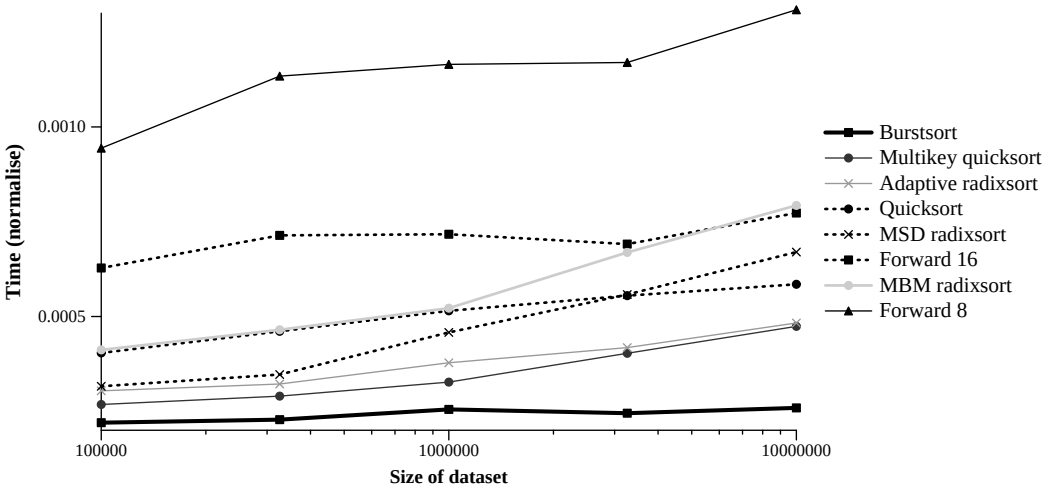

Figure 5: *URL sorting time. The vertical scale is time in milliseconds divided by* $n \log n$.

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.

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 the most rapidly with increase in data set size.

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 25% more instructions than adaptive radixsort. For all collections other than *URL*, the number of cache misses per key only rises from 1 to 3; for *URL* it rises from 3 to 4. No other algorithm comes close to this. 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 [12, 16, 17, 20] 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.

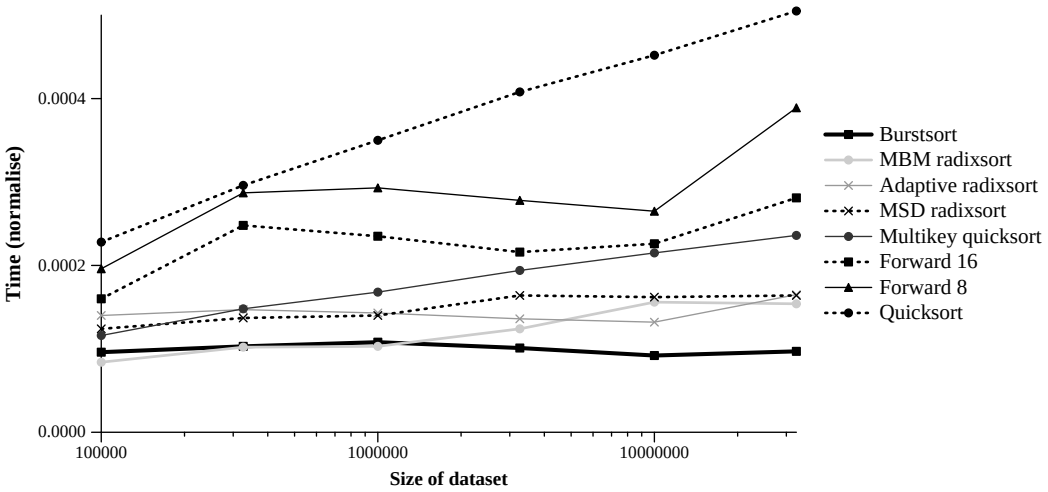

Figure 6: *Random. The vertical scale is time in milliseconds divided by* $n \log n$.

6 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 two on larger sets, and shows much better asymptotic behaviour.

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. Indeed, our experiments were run on machines that by the standards of mid-2002 had a slow processor and fast memory; on more typical machines burstsort should show even better relative performance. Our work shows that burstsort is clearly the most efficient way to sort a large set of strings.

References

[1] A. Andersson and S. Nilsson. A new efficient radix sort. In *IEEE Symp. on the Foundations of Computer Science*, pages 714–721, Santa Fe, New Mexico, 1994.

[2] A. Andersson and S. Nilsson. Implementing radixsort. *ACM Jour. of Experimental Algorithmics*, 3(7), 1998.

[3] J. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 360–369, New Orleans, Louisiana, 1997. ACM/SIAM.

[4] J. Bentley and R. Sedgewick. Sorting strings with three-way radix quicksort. *Dr. Dobbs Journal*, 1998.

[5] J. L. Bentley and M. D. McIlroy. Engineering a sort function. *Software—Practice and Experience*, 23(11):1249–1265, 1993.

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

[7] D. Hawking, N. Craswell, P. Thistlewaite, and D. Harman. Results and challenges in web search evaluation. In *Proc. World-Wide Web Conference*, 1999.

[8] S. Heinz and J. Zobel. Practical data structures for managing small sets of strings. In M. Oudshoorn, editor, *Proc. Australasian Computer Science Conf.*, pages 75–84, Melbourne, Australia, January 2002.

[9] S. Heinz, J. Zobel, and H. E. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM*

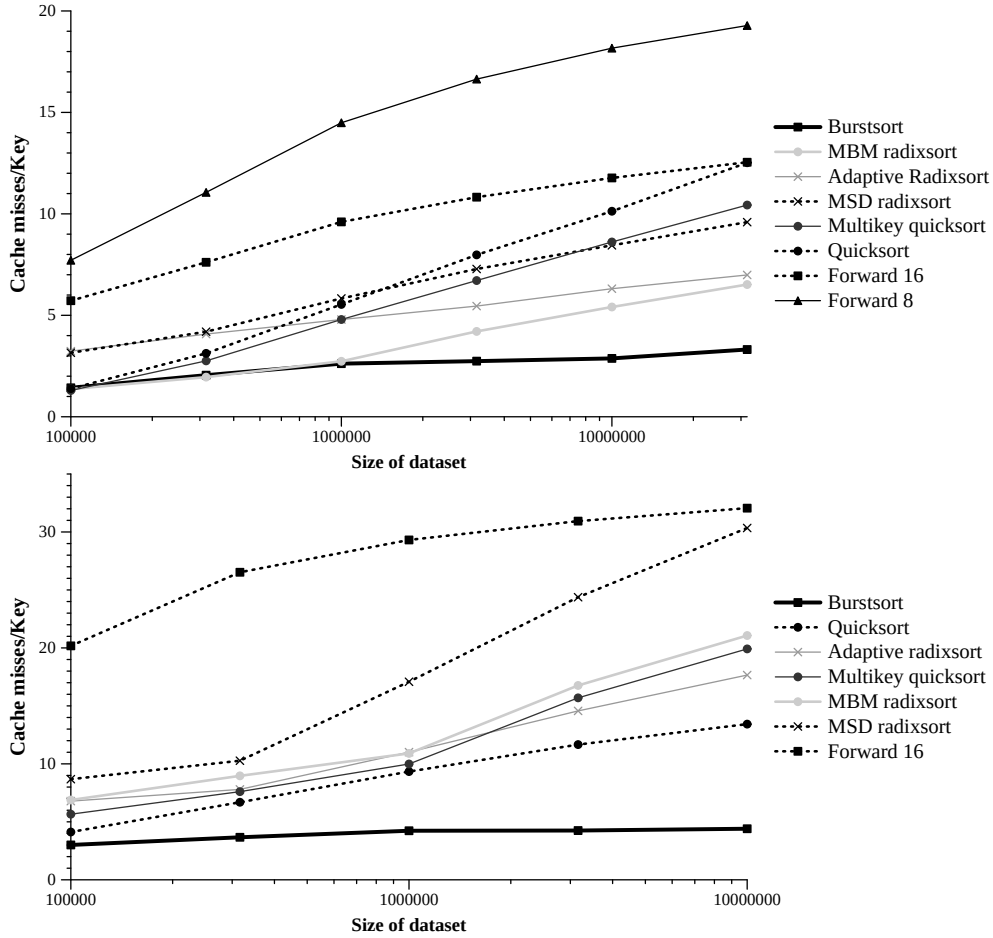

Figure 7: *Cache misses, 1 Mb cache, 8-way associativity, 32 bytes block size. Upper: no duplicates. Lower: URLs.*

Transactions on Information Systems, 20(2):192–223, 2002.

[10] C. A. R. Hoare. Quicksort. *Computer Jour.*, 5(1):10–15, 1962.

[11] D. S. Johnson. A theoretician's guide to the experimental analysis of algorithms. In Goldwasser, Johnson, and McGeoch, editors, *Proceedings of the Fifth and Sixth DIMACS Implementation Challenges*. American Mathematical Society, 2002.

[12] A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. In *Proc. Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 370–379. ACM Press, 1997.

[13] P. M. McIlroy, K. Bostic, and M. D. McIlroy. Engineering radix sort. *Computing Systems*, 6(1):5–27, 1993.

[14] A. Moffat, G. Eddy, and O. Petersson. Splaysort: Fast, versatile, practical. *Software—Practice and Experience*, 26(7):781–797, 1996.

[15] S. Nilsson. *Radix Sorting & Searching*. PhD thesis, Department of Computer Science, Lund, Sweden, 1996.

[16] N. Rahman and R. Raman. Adapting radix sort to the memory hierarchy. *ACM Jour. of Experimental Algorithmics*. To appear.

[17] N. Rahman and R. Raman. Analysing cache effects in distribution sorting. *ACM Jour. of Experimental Algorithmics*, 5:14, 2000.

[18] R. Sedgewick. *Algorithms in C, third edition*. Addison-Wesley Longman, Reading, Massachusetts, 1998.

[19] J. Seward. Cacheprof—cache profiler, December 1999. `http://www.cacheprof.org`.

[20] L. Xiao, X. Zhang, and S. A. Kubricht. Improving memory performance of sorting algorithms. *ACM Jour. of Experimental Algorithmics*, 5:3, 2000.