Using Random Sampling to Build Approximate Tries for Efficient String Sorting

RANJAN SINHA and JUSTIN ZOBEL
RMIT University

Algorithms for sorting large datasets can be made more efficient with careful use of memory hierarchies and reduction in the number of costly memory accesses. In earlier work, we introduced burstsort, a new string-sorting algorithm that on large sets of strings is almost twice as fast as previous algorithms, primarily because it is more cache efficient. Burstsort dynamically builds a small trie that is used to rapidly allocate each string to a bucket. In this paper, we introduce new variants of our algorithm: SR-burstsort, DR-burstsort, and DRL-burstsort. These algorithms use a random sample of the strings to construct an approximation to the trie prior to sorting. Our experimental results with sets of over 30 million strings show that the new variants reduce, by up to 37%, cache misses further than did the original burstsort, while simultaneously reducing instruction counts by up to 24%. In pathological cases, even further savings can be obtained.

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

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Sorting, string, cache-conscious, cache-aware, data structure, in-memory

1. INTRODUCTION

In-memory sorting is a basic problem in computer science. However, sorting algorithms face new challenges because of changes in computer architecture. Processor speeds have been increasing at 50% per year, while speed of access to main memory has been increasing at only 7% per year [Hennessy and Patterson 2002], a growing processor-memory performance gap that appears likely to continue. An architectural solution has been to introduce one or more levels of fast memory, or cache, between the main memory and the processor. Small volumes of data can be sorted entirely within cache—typically a few megabytes of memory in current machines—but, for larger volumes, each arbitrary memory access involves a delay of up to hundreds of clock cycles.

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

Much of the research on algorithms has focused on complexity and efficiency, assuming a nonhierarchical RAM model [Aho et al. 1974], but these assumptions are not realistic on modern computer architectures, where the levels of memory have different latencies. While algorithms can be made more efficient by reducing the number of instructions, current research [LaMarca and Ladner 1999; Sinha and Zobel 2004; Xiao et al. 2000] shows that an algorithm can afford to increase the number of instructions if doing so improves the locality of memory accesses and thus reduces the number of cache misses. In particular, recent work [LaMarca and Ladner 1999; Rahman and Raman 2001; Xiao et al. 2000] has successfully adapted algorithms for sorting integers to memory hierarchies.

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)." String sets are typically represented by an array of pointers to locations where the variable-length strings are stored. 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 it needs to be read.

In our previous work [Sinha and Zobel 2004], we introduced *burstsort*, a new cache-efficient string-sorting algorithm. It is based on the burst trie data structure [Heinz et al. 2002], where a set of strings is organized as a collection of *buckets* indexed by a small *access trie*. In burstsort, the trie is built dynamically as the strings are processed. During the first phase, at most the distinguishing prefix—but usually much less—is read from each string to construct the access trie and place the string in a bucket, which is a simple array of pointers. The strings in each bucket are then sorted using an algorithm that is efficient both in terms of the space and the number of instructions for small sets of strings. There have been several recent advances made in the area of string sorting, but our experiments [Sinha and Zobel 2004] showed burstsort to be much more efficient than previous methods for large string sets. (In this paper, for reference we compare three of the best previous string-sorting algorithms: MBM radixsort [McIlroy et al. 1993], multikey quicksort [Bentley and Sedgewick 1997], and adaptive radixsort [Andersson and Nilsson 1998].) However, a shortcoming of burstsort is that individual strings must be reaccessed as the trie grows, to redistribute them into sub-buckets. If the trie could be constructed ahead of time, this cost could be largely avoided, but the shape and size of the trie strongly depends on the characteristics of the data to be sorted.

Here, we propose new variants of burstsort: SR-burstsort, DR-burstsort, and DRL-burstsort. These use random sampling of the string set to construct an approximation to the trie that is built by the original burstsort. Prefixes that are repeated in the random sample are likely to be common in the data; thus, it intuitively makes sense to have these prefixes as paths in the trie. As an efficiency heuristic, rather than thoroughly process the sample, we simply process them in order, using each string to add one more node to the trie. In SR-burstsort, the trie is then fixed. In DR-burstsort, the trie can, if necessary, continue to grow as in burstsort, necessitating additional tests but avoiding inefficiency in pathological cases. In DRL-burstsort, total cache size is used to limit initial trie size.

We have used several small and large sets of strings, as described in our earlier work [Sinha and Zobel 2004], for our experiments, which we ran on several machines of differing architecture. SR-burstsort is, in some cases, slightly more efficient than burstsort, but in other cases is much slower. DR-burstsort and DRL-burstsort are more efficient than burstsort in almost all cases, although with larger collections the amount of improvement decreases. In addition, we have used a cache simulator to examine individual aspects of the performance, and have found that in the best cases both the number of cache misses and the number of instructions falls dramatically compared to burstsort. These new algorithms are the fastest known way to sort a large set of strings.

2. BACKGROUND

In our earlier work [Sinha and Zobel 2004], we examined previous algorithms for sorting strings. The most efficient of these were adaptive radixsort, multikey quicksort, and MBM radixsort.

Adaptive radixsort was introduced by Andersson and Nilsson [1998]; it is an adaptation of the distributive partitioning scheme [Dobosiewicz 1978] to standard most-significant-digit-first radixsort. The alphabet size is chosen based on the number of keys to be sorted, switching between 8 and 16 bits. In our experiments, we used the implementation of Nilsson [1996].

Multikey quicksort was introduced by Bentley and Sedgewick [1997]. It is a hybrid of ternary quicksort and MSD radixsort. It proceeds character-wise and partitions the strings into buckets, based upon the value of the character at the position under consideration. The partitioning stage proceeds by selecting a random pivot and comparing the first character of the strings with the first character of the pivot. As in ternary quicksort, the strings are then partitioned into three sets—less than, equal to, and greater than—which are then sorted recursively. In our experiments, we used an implementation by Bentley and Sedgewick [1997].

MBM radixsort (our nomenclature) is one of several high-performance MSD radixsort variants tuned for strings that were introduced by McIlroy et al. [1993] in the early 1990s. We used programC, which we found experimentally to be the most efficient of these variants and the fastest array-based, in-place sorting algorithm for strings.

2.1 Burstsort

Any data structure that maintains the data in order can be used as the basis of a sorting method. Burstsort is based on this principle. A trie structure is used to place the strings in buckets by reading, at most, the distinguishing prefix; this structure is built incrementally as the strings are processed. There are two phases; first is insertion of the strings into the burst trie structure; second is an in-order traversal, during which the buckets are sorted.

The trie is built by *bursting* a bucket once it becomes too large; a new node is created and the strings in the bucket are inserted into the node, creating new child buckets. A fixed threshold—the maximum number of strings that can be held in a bucket—is used to determine whether to burst.

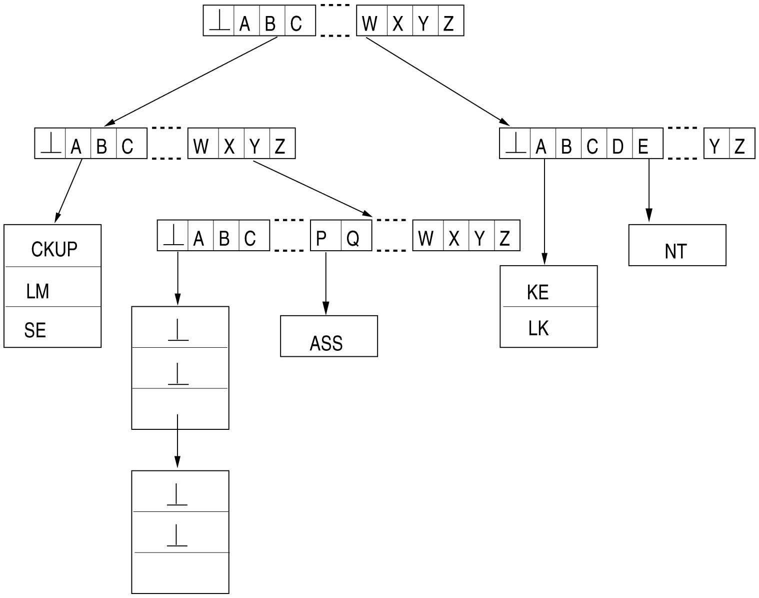


Fig. 1. A burst trie of four nodes and five buckets.

Strings that are completely consumed are managed in a special "end-of-string" structure.

During the second, traversal phase, if the number of strings in the bucket is more than one, then a sorting algorithm that takes the depth of the character of the strings into account is used to sort the strings in the bucket. We have used multikey quicksort [Bentley and Sedgewick 1997] in our experiments.

The set of strings is recursively partitioned on their lead characters. When a partition is sufficiently small, it is then sorted by a simple in-place method. However, there is a key difference between radixsorts and burstsort. In the first, trie-construction phase the standard radixsorts proceed character-wise, processing the first character of each string, then reaccessing each string to process the next character, and so on. Each trie node is handled once only, but strings are handled many times. In contrast, burstsort proceeds string-wise, accessing each string once only to allocate it to a bucket. Each node is handled many times, but the trie is much smaller than the data set, and thus the nodes can remain resident in cache.

Figure 1 shows an example of a burst trie containing eleven records whose keys are "backup," "balm," "base," "by," "by," "by," "by," "bypass," "wake," "walk," and "went," respectively. In this example, the alphabet is the set of letters from A to Z and, in addition, an empty string symbol ⊥ is shown; the bucket structure used is an array. The access trie has four trie nodes and five buckets in all. The left-most bucket has three strings, "backup," "balm," and "base," the second bucket has four identical strings "by," the fourth bucket has two strings—"wake" and "walk," the right-most bucket has only one string "went."

Experimental results comparing burstsort to previous algorithms are shown later. As can be seen, for sets of strings that are significantly larger than the available cache, burstsort is up to twice as fast. The gain is largely because of significantly reduced numbers of cache misses compared to previous techniques.

2.2 Randomized Algorithms

A randomized algorithm is one that makes random choices during its execution. According to Motwani and Raghavan [1995], "two benefits of randomized algorithms have made them popular: simplicity and efficiency. For many applications, a randomized algorithm is the simplest available, or the fastest, or both."

One application of randomization for sorting is to rearrange the input in order to remove any existing patterns, to ensure that the expected running time matches the average running time [Gupta et al. 1994]. The best-known example of this is in quicksort, where randomization of the input lessens the chance of quadratic running time. Input randomization can also be used in cases, such as binary search trees, to eliminate the worst case when the input sequence is sorted.

Another application of randomization is to process a small sample from a larger collection. In simple random sampling, each individual key in a collection has an equal chance of being selected. According to Olken and Rotem [1995],

> Random sampling is used on those occasions when processing the entire dataset is unnecessary and too expensive . . . The savings generated by sampling may arise either from reductions in the cost of retrieving the data . . . or from subsequent postprocessing of the sample. Sampling is useful for applications which are attempting to estimate some aggregate property of a set of records.

3. BURSTSORT WITH RANDOM SAMPLING

In earlier work [Sinha and Zobel 2004], we showed that burstsort is efficient in sorting strings because of the low rate of cache miss compared to other string-sorting methods. Cache misses occur when the string is fetched for the first time, during a burst, and during the traversal phase when the bucket is sorted. Our results indicated that the threshold size should be selected such that the average number of cache misses per key during the traversal phase is close to one.

Most cache misses occur while the strings are being inserted into the trie. One way in which cache misses could be reduced during the insertion phase is if the trie could be built beforehand, avoiding bursts and allowing strings to be placed in the trie with just one access, giving—if everything has gone well—a maximum of two accesses to a string overall, once during insertion and once during traversal. This is an upper bound, as some strings need not be referenced in the traversal phase and, as the insertion is a sequential scan, more than one string may fit into a cache line.

We propose building the trie beforehand using a random sample of the strings, which can be used to construct an approximation to the trie. The goal of the sampling is to get as close as possible to the shape of the trie constructed by burstsort, so the strings evenly distribute in the buckets, which can then be efficiently sorted in the cache. However, the cost of processing the sample should not be too great, or it can outweigh the gains. As a heuristic, we make just one pass through the sample, and use each string to suggest one additional trie node.

3.1 Sampling Process

(1) Create an empty trie root node r, where a trie node is an array of pointers (to either trie nodes or buckets).

(2) Choose a sample size R and create a stack of R empty trie nodes.

(3) A random sample of R strings is drawn from the input data.

(4) For each string $c_1 \ldots c_n$ in the sample,

   (a) Use the string to traverse the trie until the current character corresponds to a null pointer. That is, set $p \leftarrow r$, and $i \leftarrow 1$, and, until $p[c_i]$ is null, continue by setting $p \leftarrow p[c_i]$ and incrementing i. For example, on insertion of “michael,” if “mic” was already a path in the trie; a node is added for “h.”

   (b) If the string is not exhausted, that is, $i \leq n$, take a new node t from the stack and set $p[c_i] \leftarrow t$.

The sampled strings are not stored in the buckets; to maintain stability, they are inserted when encountered during the main sorting process. The minimum number of trie nodes created is 1 if all the strings in the collection are identical and of length 1. The maximum number of trie nodes created is equal to the size of the sample and is more likely in collections such as the random collection.

The intuition behind this approach is that, if a prefix is common in the data, then there will be several strings in the sample with that prefix. The sampling algorithm will then construct a branch of trie nodes corresponding to that prefix.

For example, in an English dictionary (from the utility `ispell`) of 127,001 strings, seven begin with “throu,” 75 with “thro,” 178 with “thr,” 959 with “th,” and 6713 with “t.” Suppose we sample 127 times with replacement, corresponding to an expected bucket size of 1000. Then the probability of sampling “throu” is only 0.01, of “thro” is 0.07, of “thr” is 0.16, of “th” is 0.62, and of “t” is 0.999. With a bucket size of 1000, a burst trie would allocate a node corresponding to the path “t” and would come close to allocating a node for “th.” Under sampling, it is almost certain that a node will be allocated for “t”—there is an even chance that it would be one of the first 13 nodes allocated—and likely that a node would be allocated for “th.” Nodes for the deeper paths are unlikely.

3.2 SR-burstsort

In burstsort, the number of trie nodes created is roughly linear in the size of the set to be sorted. It is, therefore, attractive that the number of nodes allocated through sampling be a fixed percentage of the number of keys in the set; by the informal statistical argument above, the trie created in the initial phase should approximate the trie created by applying standard burstsort to the same data. In *static randomized burstsort*, or SR-burstsort, the trie structure created by sampling is then static. The structure grows only through addition of strings to buckets. The use of random sampling means that common prefixes will in the great majority of runs be represented in the trie and strings will distribute well among the buckets.

For a set of N strings, we need to choose a sample size. We use a *relative trie size* parameter S. For our experiments, we used $S = 8192$, because this value was an effective bucket-size threshold in our earlier work. Then the sample size and the maximum number of trie nodes that can be created, is $R = N/S$.

SR-burstsort proceeds as follows: use the sampling procedure above to build an access trie; insert the strings in turn into buckets; then, traverse the trie and buckets to give the sorted result. No bursts occur. Buckets are a linked list of arrays of a fixed size (an implementation decision derived from preliminary experiments). The last element in each array is a pointer to the next array. In our experiments, we have used an array size of 32.

SR-burstsort has several advantages compared to the original algorithm. The code is simpler, with no thresholds or bursting, thus requiring far fewer instructions during the insertion phase. Insertion also requires fewer string accesses. The nodes are allocated as a block, simplifying dynamic memory management.

However, bucket size is not capped and some buckets may not fit entirely within the cache. The bucket-sorting routine is selected mainly for its instruction and space efficiency for small sets of strings and not for cache efficiency. Moreover, small changes in the trie shape can lead to large variations in bucket size: omitting a single crucial trie node because of sampling error may mean that a very large bucket is created.

3.3 DR-burstsort

An obvious next step is to eliminate the cases in SR-burstsort when the buckets become larger than cache and bucket sorting is not entirely cache resident. This suggests *dynamic randomized burstsort*, or DR-burstsort. In this approach, an initial trie is created through sampling as before, but as in the original burstsort, a limit is imposed on bucket size and buckets are burst if this limit is exceeded. DR-burstsort avoids the bad cases that arise in SR-burstsort because of sampling errors. The number of bursts should be small, but, compared to SR-burstsort, additional statistics must be maintained.

Thus, DR-burstsort is as follows: using a relative trie size S, select a sample of $R = N/S$ strings and create an initial trie; insert the strings into the trie as for burstsort; then traverse as for burstsort or SR-burstsort. Buckets are represented as arrays of 16, 128, 1024, or 8192 pointers, growing from one size to the next as the number of strings to be stored increases, as we have described elsewhere for burstsort [Sinha and Zobel 2004].

3.4 DRL-burstsort

For the largest sets of strings, the trie is much too large to be cache resident. That is, there is a trade-off between whether the largest bucket can fit in cache and whether the trie can fit in cache. One approach is to stop bursts at some point, especially as bursts late in the process are not as helpful. We have not explored this approach, as it would be unsuccessful with sorted data.

Table I. 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*	3.030	9.607	30.386	96.156	304.118	—
Distinct Words ($\times 10^5$)	0.361	0.923	2.355	5.769	12.898	—
Word Occurrences ($\times 10^5$)	1	3.162	10	31.623	100	—

Another approach is to limit the size of the initial trie to fit in cache, to avoid the disadvantages of extraneous nodes being created. This variant, DR-burstsort with limit or DRL-burstsort, is tested below. The limit used in our experiments depends on the size of the cache and the size of the trie nodes. In our experiments, we chose R so that R times node size is equal to the cache size.

4. EXPERIMENTS

For realistic experiments with large sets of strings, we are limited to sources for which we have sufficient volumes of data. We have drawn on web data and genomic data. For the latter, we have parsed nucleotide strings into overlapping nine-grams. For the former, derived from the TREC project [Harman 1995; Hawking et al. 1999], we extracted both words—alphabetic strings delimited by nonalphabetic characters—and URLs. For the words, we considered sets with and without duplicates, in both cases in order of occurrence in the original data.

For the word data 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–Set 5. In each case, only Set 1 fits in cache. The statistics of the data sets used are shown in Table I. In detail, the data sets are as follows.

- Duplicates. Words in order of occurrence, including duplicates. The statistical characteristics are those of natural language text; a small number of words are frequent, while many occur once only.
- No duplicates. Unique strings based on word pairs in order of first occurrence in the TREC web data.

- Genome. Strings extracted from a collection of genomic strings, each typically thousands of nucleotides long. The strings are parsed into shorter strings of length 9. The alphabet is comprised of four characters, “a,” “t,” “g,” and “c.” There is a large number of duplicates and the data shows little locality.
- Random. An artificially generated collection of strings whose characters are uniformly distributed over the entire ASCII range. The length of each string is random in the range 1–20.
- 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.
- Artificial A. A collection of identical strings on an alphabet of one character. Each string is one hundred characters long and the size of the collection is one million.
- Artificial B. A collection of strings with an alphabet of nine characters. The length of strings are varied randomly from one to hundred and the size of the collection is ten million.
- Artificial C. A collection of strings whose length ranges from one to hundred. The alphabet size is one and the strings are ordered in increasing length arranged cyclically. The size of the collection is one million.

The cost of bursting increases with the size of the bucket as more strings need to be fetched from memory, leading to increases in the number of cache misses and of instructions. Each correct prediction of a trie node removes the need to burst a bucket. Another situation where bursting could be expensive is use of inefficient data structures, such as binary search trees or linked lists as buckets. Traversing a linked list could result in two memory accesses for each bucket element, one access to the string and one access to the list node. To show how sampling can be beneficial as bursting becomes more expensive, we have measured the running time, instruction count, and cache misses as the size of the bucket is increased from 1024 to 131,072, or, for the artificial collections, up to 262,144.

The aim of the experiments is to compare the performance of our algorithms, in terms of the running time, number of instructions, and number of L2 cache misses.

The time measured is to sort an array of pointers to strings; the array is returned as the output. We, therefore, report the CPU times, not elapsed times, and exclude the time taken to parse the collections into strings.

The experiments were run on a Pentium III Xeon 700 MHz computer with 2 GB of internal memory, 1-MB L2 cache with block size of 32 bytes, eight-way associativity, and a memory latency of about 100 cycles. We also tested the methods on two newer architectures, a Pentium IV and a PowerPC. The details of the machine configurations are as shown in Table II. We have used the highest compiler optimization O3 in all our experiments. The total number of milliseconds of CPU time has been measured; the time taken for I/O or to parse the collection are not included as these are in common for all algorithms. For the cache simulations, we have used `valgrind` [Seward 2001].

Table II. Architectural Parameters of the Machines Used for Experiments

Workstation	Pentium	Power Mac G5	Pentium
Processor type	Pentium IV	PowerPC 970	Pentium III Xeon
Clock rate	2000 MHz	1600 MHz	700 MHz
L1 data cache (KB)	8	32	16
L1 line size (bytes)	64	128	32
L1 associativity	4-way	2-way	4-way
L1 miss latency (cycles)	7	8	6
L2 cache (KB)	512	512	1024
L2 block size (bytes)	64	128	32
L2 associativity	8-way	8-way	8-way
L2 miss latency (cycles)	285	324	109
Data TLB entries	64	256	64
Entries TLB associativity	full	4-way	4-way
Pagesize (KB)	4	4	4
Memory size (MB)	2048	256	2048

Table III. Duplicates: Sorting Time for Each Method (ms)

		Data Set					
Threshold		Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
	Multikey quicksort	62	272	920	3,830	14,950	56,070
	MBM radixsort	58	238	822	3,650	15,460	61,560
	Adaptive radixsort	74	288	900	3,360	12,410	51,870
	SR-burstsort	60	200	560	2,010	7,620	31,040
8192	Burstsort	58	218	630	2,220	7,950	29,910
	DR-burstsort	60	200	560	2,030	7,390	28,530
	DRL-burstsort	60	200	560	2,030	7,510	29,030
16384	Burstsort	60	210	630	2,270	7,970	28,490
	DR-burstsort	60	200	550	2,020	7,280	27,310
32768	Burstsort	60	210	630	2,380	8,250	28,530
	DR-burstsort	60	200	560	2,010	7,160	27,400
65536	Burstsort	60	210	640	2,480	8,590	29,620
	DR-burstsort	60	200	560	2,010	7,150	26,640
131072	Burstsort	60	220	660	2,550	9,190	31,260
	DR-burstsort	60	200	560	2,010	7,140	27,420

5. RESULTS

We present results in three forms: time to sort each data set, instruction counts, and L2 cache misses.

Times for sorting are shown in Tables III–VII. Instruction counts are shown in Figures 3 and 5. L2 cache misses are shown in Figures 2, 4, and 6; the trends for the other data sets are similar.

On duplicates, the sorting times for the burstsort methods are, for all cases but SET 1, faster than for the previous methods. These results are as observed in our previous work. The performance gap steadily grows with dataset size, and the indications from all the results—instructions, cache misses, and timings—are that the improvements yielded by burstsort will continue to increase with both changes in computer architecture and growing data volumes. Figure 2 shows the L2 cache misses in comparison to the best algorithms found in our earlier work.

Table IV. Genome: Sorting Time for Each Method (ms)

		Data Set					
Threshold		Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
	Multikey quicksort	72	324	1,250	4,610	16,670	62,680
	MBM radixsort	72	368	1,570	6,200	23,700	90,700
	Adaptive radixsort	92	404	1,500	4,980	17,800	66,100
	SR-burstsort	70	240	780	2,530	10,320	44,810
8192	Burstsort	70	258	870	2,830	8,990	31,540
	DR-burstsort	70	240	770	2,470	7,960	30,870
	DRL-burstsort	70	240	770	2,460	8,410	30,680
16384	Burstsort	70	290	910	2,760	8,720	30,280
	DR-burstsort	70	240	780	2,390	7,520	27,850
32768	Burstsort	80	280	940	3,000	9,520	31,140
	DR-burstsort	60	240	770	2,390	7,560	28,780
65536	Burstsort	70	310	1,010	3,130	9,820	32,860
	DR-burstsort	70	240	770	2,400	7,520	28,710
131072	Burstsort	80	300	1,070	3,400	10,940	36,630
	DR-burstsort	70	230	770	2,400	7,570	28,740

Table V. URLs: Sorting Time for Each Method (milliseconds)[a]

		Data Set				
Threshold		Set 1	Set 2	Set 3	Set 4	Set 5
	SR-burstsort	100	360	1,310	5,350	19,420
8192	Burstsort	110	390	1,530	5,080	17,860
	DR-burstsort	110	370	1,450	4,860	17,130
	DRL-burstsort	100	370	1,450	4,850	17,610
16384	Burstsort	110	390	1,630	5,280	18,800
	DR-burstsort	110	380	1,530	4,890	17,350
32768	Burstsort	130	420	1,510	6,710	21,560
	DR-burstsort	110	370	1,380	5,890	18,670
65536	Burstsort	170	440	1,540	6,290	24,010
	DR-burstsort	110	370	1,380	5,410	19,360
131072	Burstsort	140	480	1,550	6,310	27,120
	DR-burstsort	110	370	1,340	5,330	19,830

[a] The fastest times in the burstsort family are shown in bold.

Figures 3 and 4 show the number of instructions and L2 cache misses for a bucket size of 32768. Several overall trends can be observed. The number of instructions per string does not vary dramatically for any of the methods, although it does have perturbations because of characteristics of the individual data sets. SR-burstsort consistently uses fewer instructions than the other methods, while the original burstsort requires the most. Among the burstsorts, SR-burstsort is consistently the slowest for the larger sets because of more L2 cache misses than burstsort, despite requiring fewer instructions.

For most collections, either DR-burstsort or DRL-burstsort is the fastest sorting technique and usually yield similar results. Compared to burstsort, DR-burstsort uses up to 24% fewer instructions and incurs up to 37% fewer cache misses. However, there are exceptions. In particular, DRL-burstsort has done much better than DR-burstsort on the random data. Based on this data,

Table VI. Artificial Sets: Sorting Time for Each Method (ms)

		Collection		
Threshold		Artificial A	Artificial B	Artificial C
	SR-burstsort	2,650	9,220	1,600
8192	Burstsort	2,740	10,130	1,430
	DR-burstsort	2,340	9,080	1,300
16384	Burstsort	2,510	10,110	1,460
	DR-burstsort	2,320	8,890	1,340
32768	Burstsort	2,910	10,540	1,880
	DR-burstsort	2,320	8,110	1,430
65536	Burstsort	3,760	11,210	2,610
	DR-burstsort	2,340	8,010	1,540
31072	Burstsort	5,190	11,820	3,810
	DR-burstsort	2,320	7,890	1,670
262144	Burstsort	7,900	13,200	5,660
	DR-burstsort	2,290	7,930	1,570

Table VII. Random: Sorting Time for Each Method (ms)

		Data Set					
Threshold		Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
	SR-burstsort	50	170	570	1,930	7,060	29,410
8192	Burstsort	50	180	650	2,100	6,450	23,040
	DR-burstsort	50	180	580	2,050	6,910	30,790
	DRL-burstsort	50	180	570	2,050	6,470	23,340

burstsort is by a small margin the fastest method tested. The heuristic in DRL-burstsort of limiting the initial trie to the cache size has led to clear gains, in this case, in which the sampling process is error-prone.

Some of the data sets have individual characteristics that affect the trends. In particular, with the fixed length of the strings in the genome data, increasing the number of strings does not increase the number of distinct strings. Thus, the relative costs of sorting under the different methods changes with increasing dataset size. In contrast, with duplicates, the number of distinct strings continues to steadily grow.

The sorting times shown in Tables III to VII shows that as the size of the bucket increases, burstsort becomes more expensive. On the other hand, the cost of DR-burstsort does not vary much with increasing bucket size. Table VI shows DR-burstsort can be as much as 3.5 times faster than burstsort.

As shown in Figure 5, the number of instructions incurred by DR-burstsort can be up to 30% less than burstsort. Also, interestingly, the number of instructions do not appear to vary much as the size of the bucket increases. Figure 6 similarly shows that the number of misses incurred by DR-burstsort can be up to 90% less than burstsort.

All of the new methods require fewer instructions than the original burstsort. More importantly, in most cases, DR-burstsort and DRL-burstsort require fewer cache misses. This trend means that, as the hardware performance gap grows, the relative performance of our new methods will continue to improve.

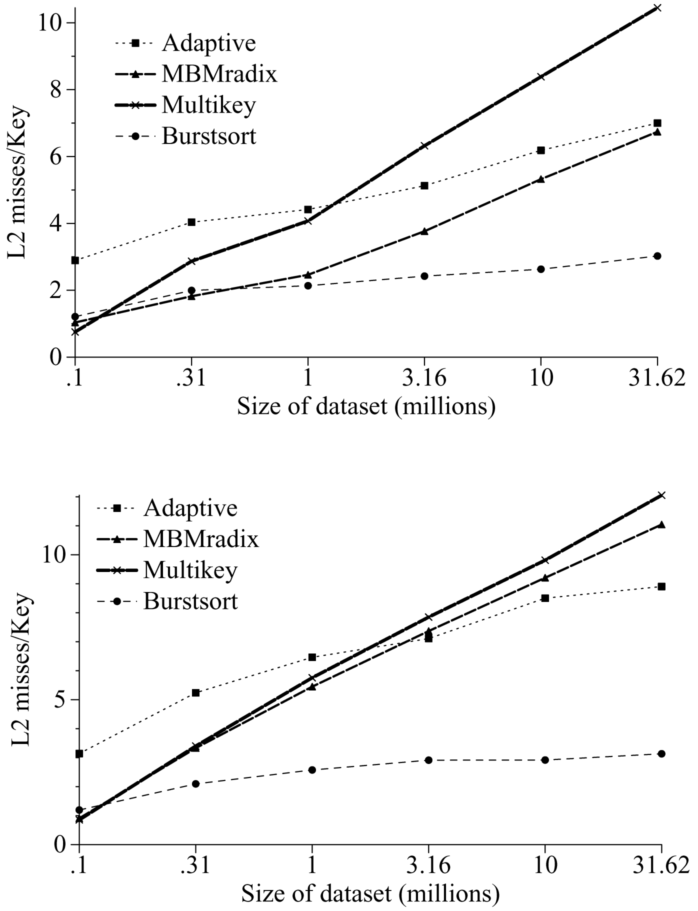
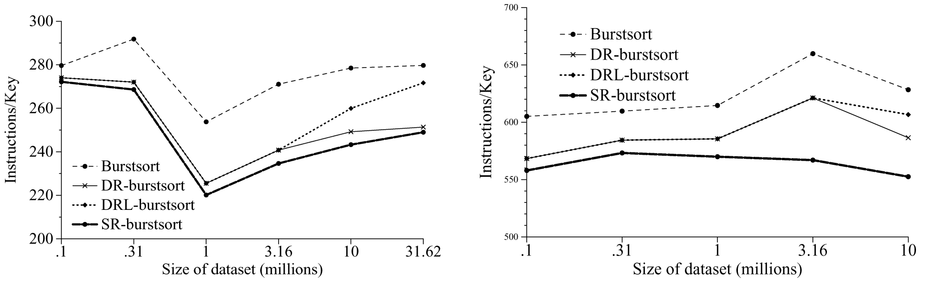
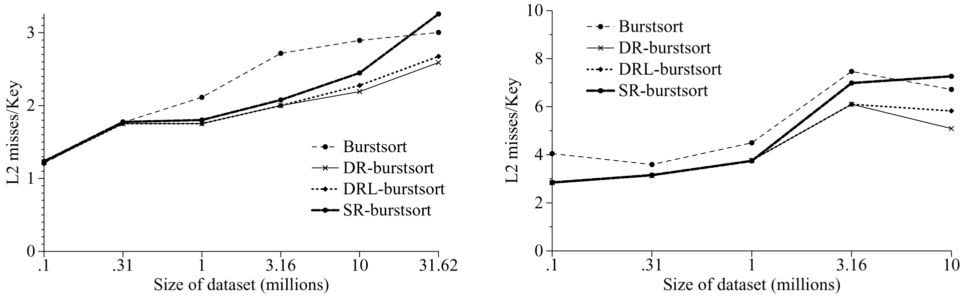


Fig. 2. L2 cache misses for the most efficient sorting algorithms, burstsort has a threshold of 8192. Upper, duplicates, lower, genome.



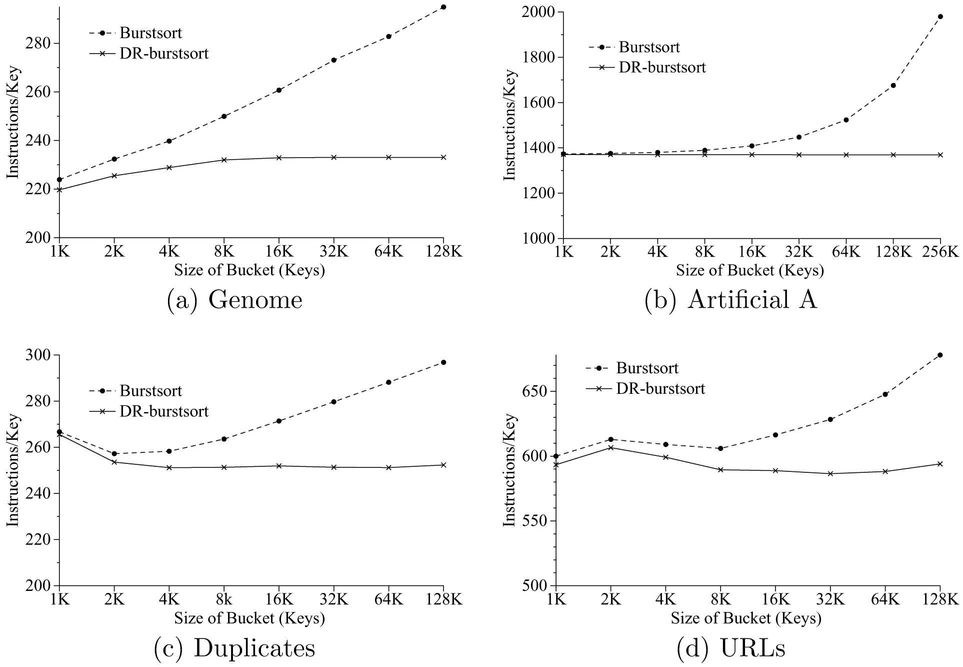
(a) Duplicates (b) URLS

Fig. 3. Instructions per key on each data set, for each variant of burstsort for a threshold of 32768.



(a) Duplicates (b) URLs

Fig. 4. L2 cache misses per key on each data set, for each variant of burstsort for a threshold of 32768.



(a) Genome (b) Artificial A

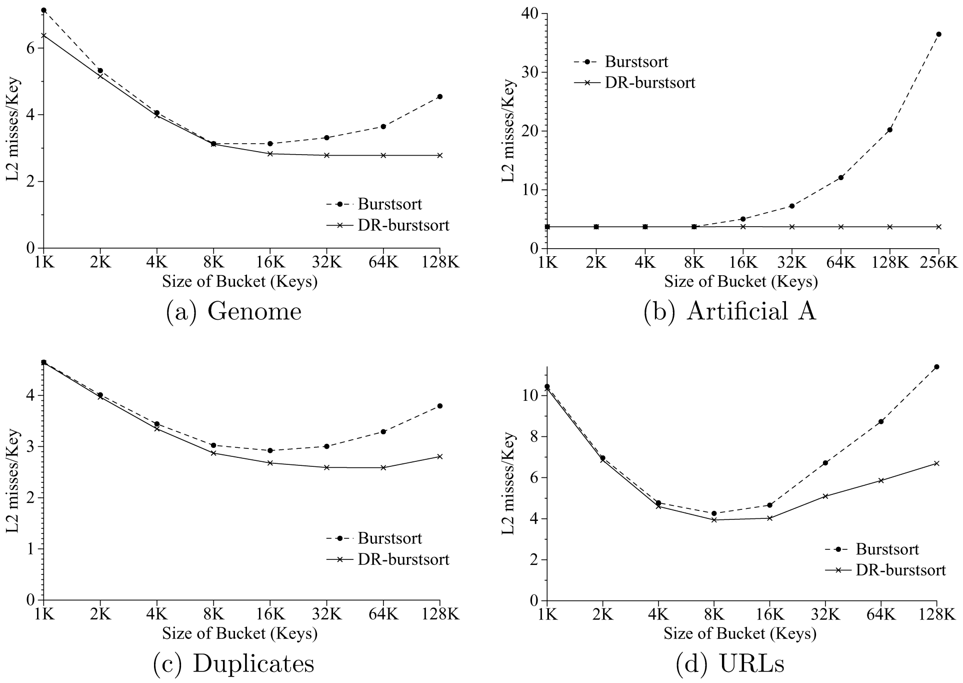
(c) Duplicates (d) URLs

Fig. 5. Instructions per key for the largest data set, for each variant of burstsort. $K = 1024$.

5.1 Other Architectures

We tested the performance of our algorithms on newer architectures, a Pentium IV and a PowerPC. The performance characteristics are similar to those found on the Pentium III. As the cost of bursting increases because of increase in bucket size, the sampling variants are effective across all machines and across all collections.

Figures 7 and 8 shows the performance of the algorithms with increasing threshold size on the Pentium IV and PowerPC, respectively. The algorithmic parameters used for the experiments are identical on both PowerPC and Pentium IV, as both have the same cache size. Because of limits on available



(a) Genome

(b) Artificial A

(c) Duplicates

(d) URLs

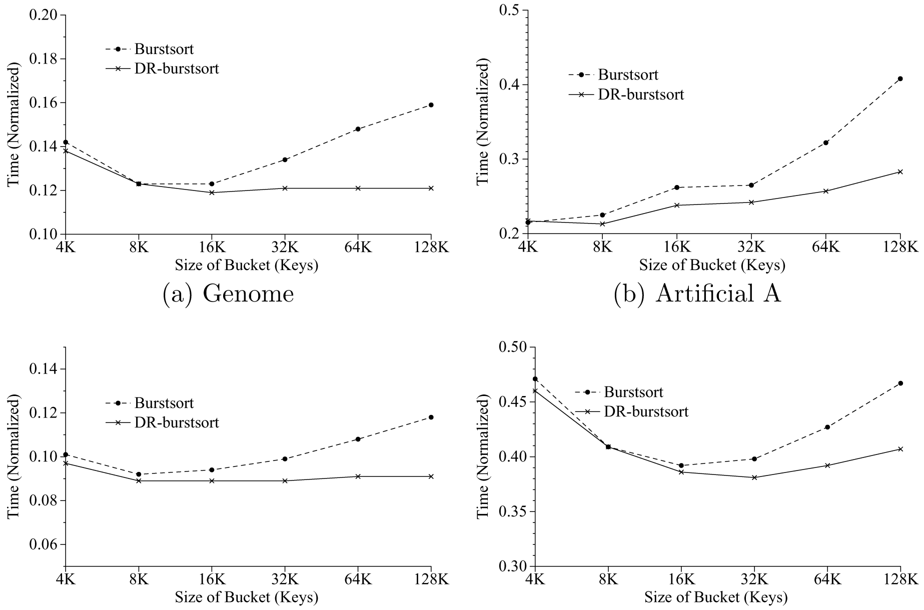
Fig. 6. L2 cache misses per key for the largest data set, for each variant of burstsort. $K = 1024$.

main memory, the size of the collection on the Pentium IV was 10 million strings, while on the PowerPC it was about 3 million strings. On both the Pentium IV and PowerPC, for most thresholds, the performance of DR-burstsort is better than that of that of burstsort for all collections. On the Pentium IV, the performance improvements of DR-burstsort over burstsort ranges from 13% for URLs to up to 42% for the random collection. Similarly, the performance improvements in PowerPC ranges from 12% for URLs to up to 42% for the random collection.

As the size of the buckets increases for the random collection, the bucket-sorting phase becomes increasingly expensive as the strings are no longer cache-resident. This is because of the strings being uniformly distributed across the buckets and not reaching the threshold size. Sampling helps by creating more trie nodes and thus smaller buckets, helping to reduce the cost for bucket, sorting.

6. CONCLUSIONS

We have proposed new algorithms—SR-burstsort, DR-burstsort, and DRL-burstsort—for fast sorting of strings in large data collections. They are variants of our burstsort algorithm and are based on construction of a small trie that rapidly allocates strings to buckets. In the original burstsort, the trie was constructed dynamically; the new algorithms are based on taking a random sample of the strings and using them to construct an initial trie structure before any strings are inserted.



(a) Genome

(b) Artificial A

(c) Duplicates

(d) URLs

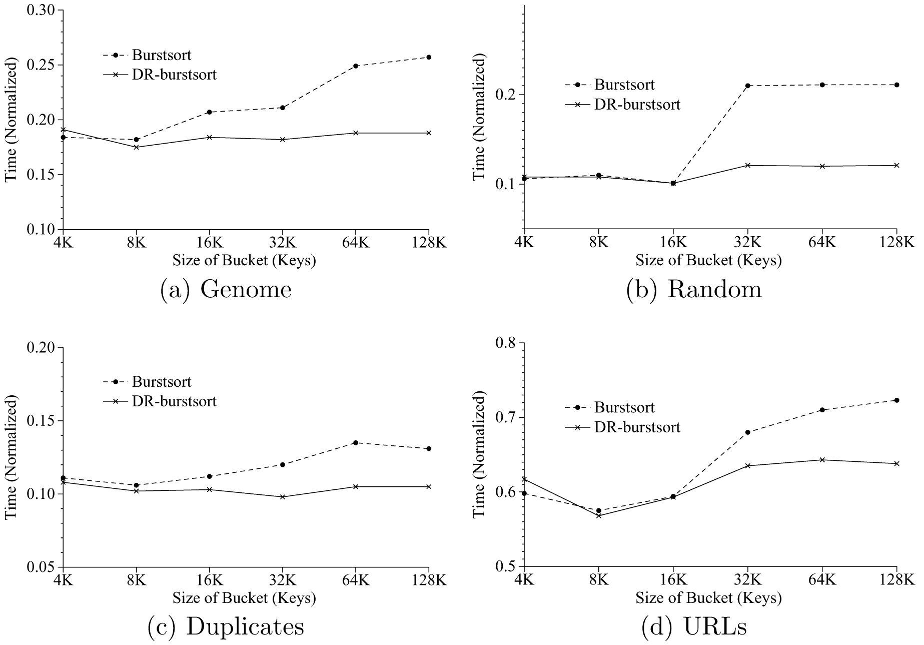
Fig. 7. Sorting time for varying thresholds on Pentium IV architecture. The vertical scale is time in seconds divided by $n \log n$. The size of the dataset for each collection was 10 million strings. $K = 1024$.

SR-burstsort, where the trie is static, reduces the need for dynamic memory management and simplifies the insertion process, leading to code with a lower instruction count than the other alternatives. Despite promising performance in preliminary experiments and the low instruction count, however, it is generally slower than burstsort, as there can easily be bad cases where a random sample does not correctly predict the trie structure, which leads to some buckets being larger than expected.

DR-burstsort and DRL-burstsort improve on the worst case of SR-burstsort by allowing the trie to be modified dynamically, at the cost of additional checks during insertion. They are faster than burstsort in all experiments with real data, because of elimination of the need for most of the bursts. The use of a limit in DRL-burstsort avoids poor cases that could arise in data with a flat distribution.

Our experimental results show that the new variants reduce cache misses even further than does the original burstsort, by up to 37%, while simultaneously reducing instruction counts by up to 24%. As the cost of bursting grows, the new variants reduce cache misses by up to 90%, while simultaneously reducing instruction counts by up to 30% and the time to sort is reduced by up to 72%, as compared to burstsort.

Other machines have been used to check whether the performance gains are observed in architectures with different characteristics. The results are consistent across all these machines. On both the PowerPC and Pentium IV,



(a) Genome

(b) Random

(c) Duplicates

(d) URLs

Fig. 8. Sorting time for varying thresholds on PowerPC architecture. The vertical scale is time in seconds divided by $n \log n$. The size of the dataset for each collection was about 3 million strings. $K = 1024$.

DR-burstsort is more efficient than burstsort, with improvements ranging from about 11 to 42%.

There is further scope for improving the performance by tuning the parameters for each machine. Preanalysis of collections to see whether the alphabet is restricted showed an improvement of 16% for genomic collections. Preanalysis would be of value for customized sorting applications. Another variation is to choose the sample size based on analysis of collection characteristics. A further variation is to recursively apply SR-burstsort to large buckets. We are testing these options in current work.

Even without these improvements, however, burstsort and its variants are a significant advance, dramatically reducing the costs of sorting a large set of strings. Cache misses and running time are as low as one-half that required by any previous method. With the current trends in computer architecture, the performance gains given by our methods will continue to improve.

ACKNOWLEDGMENTS

This paper incorporates work previously published in "Using Random Sampling to Build Approximate Tries for Efficient String Sorting," R. Sinha and J. Zobel, In *Proceedings of the 3rd Workshop on Efficient and Experimental Algorithms* (*WEA*), Angra dos Reis, Rio de Janeiro, Brazil, May 2004. This research was supported by the Australian Research Council.

REFERENCES

AHO, A., HOPCROFT, J. E., AND ULLMAN, J. D. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA.

ANDERSSON, A. AND NILSSON, S. 1998. Implementing radixsort. *ACM Jour. of Experimental Algorithmics 3*, 7.

ARGE, L., FERRAGINA, P., GROSSI, R., AND VITTER, J. S. 1997. On sorting strings in external memory. In *Proc. ACM Symp. on Theory of Computation*, P. Shor, Ed. El Paso, Tx. ACM, New York. 540–548.

BENTLEY, J. AND SEDGEWICK, R. 1997. Fast algorithms for sorting and searching strings. In *Proc. Annual ACM-SIAM Symp. on Discrete Algorithms*, M. Saks, Ed. Society for Industrial and Applied Mathematics, New Orleans, LA. 360–369.

DOBOSIEWICZ, W. 1978. Sorting by distributive partitioning. *Information Processing Letters 7*, 1, 1–6.

GUPTA, R., SMOLKA, S. A., AND BHASKAR, S. 1994. On randomization in sequential and distributed algorithms. *Computing Surveys 26*, 1, 7–86.

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

HAWKING, D., CRASWELL, N., THISTLEWAITE, P., AND HARMAN, D. 1999. Results and challenges in web search evaluation. *Computer Networks 31*, 11–16, 1321–1330.

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

HENNESSY, J. L. AND PATTERSON, D. A. 2002. *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann, San Mateo, CA.

LAMARCA, A. AND LADNER, R. E. 1999. The influence of caches on the performance of sorting. *Journal of Algorithms 31*, 1, 66–104.

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

MOTWANI, R. AND RAGHAVAN, P. 1995. *Randomized Algorithms*. Cambridge University Press, Cambridge.

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

OLKEN, F. AND ROTEM, D. 1995. Random sampling from databases—a survey. *Statistics and Computing 5*, 1 (Mar.), 25–42.

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

SEWARD, J. 2001. Valgrind—memory and cache profiler. `http://developer.kde.org/~sewardj/docs-1.9.5/cg_techdocs.html`.

SINHA, R. AND ZOBEL, J. 2004. Cache-conscious sorting of large sets of strings with dynamic tries. *ACM Jour. of Experimental Algorithmics 9*.

XIAO, L., ZHANG, X., AND KUBRICHT, S. A. 2000. Improving memory performance of sorting algorithms. *ACM Jour. of Experimental Algorithmics 5*, 3.

Received February 2005; revised December 2005; accepted February 2006