Engineering Burstsort: Towards Fast In-Place String Sorting

Ranjan Sinha and Anthony Wirth

Department of Computer Science and Software Engineering,
The University of Melbourne, Australia
rsinha@csse.unimelb.edu.au, awirth@csse.unimelb.edu.au

Abstract. Burstsort is a trie-based string sorting algorithm that distributes strings into small buckets whose contents are then sorted in cache. This approach has earlier been demonstrated to be efficient on modern cache-based processors [Sinha & Zobel, JEA 2004]. In this paper, we introduce improvements that reduce by a significant margin the memory requirements of burstsort. Excess memory has been reduced by an order of magnitude so that it is now less than 1% greater than an in-place algorithm. These techniques can be applied to existing variants of burstsort, as well as other string algorithms.

We redesigned the buckets, introducing sub-buckets and an index structure for them, which resulted in an order-of-magnitude space reduction. We also show the practicality of moving some fields from the trie nodes to the insertion point (for the next string pointer) in the bucket; this technique reduces memory usage of the trie nodes by one-third. Significantly, the overall impact on the speed of burstsort by combining these memory usage improvements is not unfavourable on real-world string collections. In addition, during the bucket-sorting phase, the string suffixes are copied to a small buffer to improve their spatial locality, lowering the running time of burstsort by up to 30%.

1 Introduction

This paper revisits the issue of sorting strings efficiently. String sorting remains a key step in solving contemporary data management problems. Arge *et al.* [3] note that "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)". Compared to sorting fixed-length keys (such as integers), efficient sorting of variable-length string keys is more challenging. First, string lengths are variable, and swapping strings is not as simple as swapping integers. Second, strings are compared one character at a time, instead of the entire key being compared, and thus require more instructions. Third, strings are traditionally accessed using pointers; the strings are not moved from their original locations due to string copying costs.

C.C. McGeoch (Ed.): WEA 2008, LNCS 5038, pp. 14–27, 2008.

1.1 Traditional Approaches to String Sorting

Standard string sorting algorithms, as taught in a typical undergraduate education, start by creating an array of pointers to strings; they then permute these pointers so that their order reflects the lexicographic order of the strings. Although comparison of long string keys may be time-consuming, with this method at least the cost of moving the keys is avoided. The best existing algorithms for string sorting that honour this rule include multikey quicksort [6] and variants of radix sort [2], including the so-called MBM algorithm [16].

Traditionally, we are taught to measure algorithm speed by the number of instructions, such as *comparisons* or *moves*. However, in recent years the cost of retrieving items from main memory (when they are not in cache), or of translating virtual addresses that are not in the *translation lookaside buffer* (TLB) have come to dominate algorithm running times. The principal principle is *locality of reference*: if data is physically near data that was recently processed, or was itself processed not long ago, then it is likely to be in the cache and may be accessed quickly. We leave further details to Section 1.3. Note that stability is ignored in this paper as any sorting algorithm can be made stable by appending the rank of each key in the input [11,20].

1.2 Burstsort

Burstsort is a technique that combines the burst trie [13] with standard (string) sorting algorithms [6,16]. It was introduced by the first author, and its variants are amongst the fastest algorithms for sorting strings on current hardware [21]. The standard burstsort algorithm is known as P-burstsort, P referring to *pointer*. In P-burstsort, sorting takes place in two stages: (i) the strings are inserted into the trie structure, effectively partitioned by their prefixes into *buckets*, (ii) the trie is traversed in-order and the contents of each bucket (strings with a common prefix) are sorted and pointers to the strings are output in lexicographic order.

The trie. As shown in Figure 1, each trie node contains a collection of pointers to other nodes or to buckets. There is one pointer for each symbol in the alphabet so a node effectively represents a string prefix. Note that each node contains an additional pointer (labelled $\perp$) to a special bucket for strings that are identical to the prefix that the node represents.

The trie starts off with one node, but grows when the distribution of strings causes a bucket to become too large. Whenever a bucket does become *too large*, it is *burst*: the trie expands in depth at that point so that there is now a child node for each symbol of the alphabet, each node having a full collection of pointers to buckets.

Cache efficiency. Although the use of a trie is reminiscent of radixsort, in burstsort each string is only accessed a small number of times: when inserted (cache-efficient), whenever its bucket is burst, or when its bucket is sorted. In practice this low dereferencing overhead makes the algorithm faster than radixsort or quicksort.

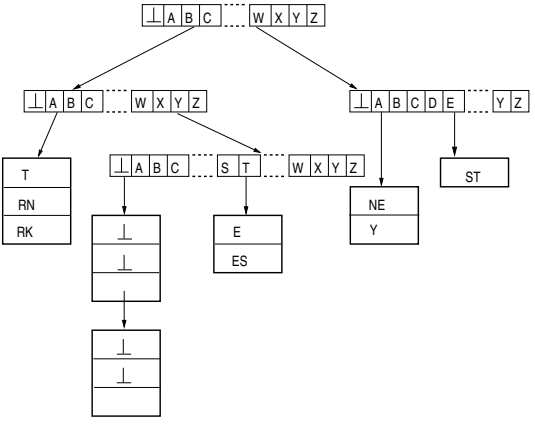


Fig. 1. Index structure of P-Burstsort. Strings inserted into the burst trie are "bat", "barn", "bark", "by", "by", "by", "by", "byte", "bytes", "wane", "way", and "west". The figure has been adapted from [21].

Sorting of buckets. The standard P-burstsort [21] uses a fast multikey quicksort for sorting the buckets, and on real-world collections has been observed to be almost twice as fast as previous algorithms. The original burstsort paper also proposes that buckets be grown *exponentially*, starting with a small bucket, then growing by a certain factor whenever the bucket is full, until a maximum size, when the bucket is burst. This wastes less memory than using a fixed size for the bucket, but increases the memory management load.

Figure 2 reminds the reader of the significant gains in running time of P-burstsort compared to three of the best string sorting algorithms: adaptive radixsort, multikey quicksort and MBM radixsort.

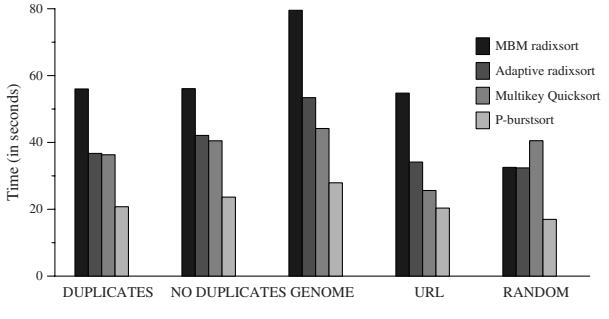


Fig. 2. Time (in seconds) to sort with adaptive radixsort, multikey quicksort, MBM radixsort, and P-Burstsort for all five collections (see Section 4) on a Pentium IV machine with a small 512 KB L2 cache. The bucket threshold used for P-burstsort is 32768.

Previous improvements. A reduction in the number of bursts results in a reduction in string accesses. To that end, in the sampling-based burstsort variants we used a small sample of strings (selected uniformly at random) to create an initial approximate trie, prior to inserting all the strings in the trie [22]. Although this approach reduced the number of cache misses, we believe there remains scope for investigation into more sophisticated schemes.

In P-burstsort, strings are accessed by pointers to them; only the pointers are moved, as is the case in traditional string sorting methods. It is vital that the locality of string accesses is improved, especially during bursts and when the buckets are being sorted. Hence, in the copy-based approach [20], strings were actually copied into the buckets from the start to improve string locality, admittedly at the cost of higher instruction counts. However, we found that the performance improves significantly, largely due to reduced cache and TLB misses.

Memory use. The priority in the earlier versions of burstsort was to increase the speed of sorting. Analysing the memory demand has so far been largely ignored, but is a major focus in this paper; we outline the contributions below.

1.3 Related Work

String sorting. There have been several advances in fast sorting techniques designed for strings. These have primarily focused on reducing instruction count, assuming a unit-cost RAM model [1,14]. For example, three-way partitioning is an important quicksort innovation [7]. Splaysort, an adaptive sorting algorithm, introduced by Moffat *et al* [17], is a combination of the splaytree data structure and insertionsort. Improvements to radixsort for strings were proposed by McIlroy *et al* [16], and by Andersson and Nilsson [2]. Bentley and Sedgewick [6] introduced a hybrid of quicksort and radixsort named three-way radix quicksort [18]; they then extended this to produce multikey quicksort [6].

In this paper, we compare our algorithms with adaptive radixsort [2], multikey quicksort [6] and MBM radixsort [16], as they have been observed to be amongst the fastest [21]. The performance of other algorithms can be obtained from the first author's earlier papers [20,21].

Cache-aware algorithms. While the radix sorts have a low instruction count—the traditional measure of computation speed—they do not necessarily use the cache efficiently for sorting variable-length strings. In earlier experiments [21], on the larger datasets there were typically 5 to 10 cache misses per string during radix-based sorting on a machine with 1 MB L2 cache. Accesses to the strings account for a large proportion of cache misses. Approaches that can make string sorting algorithms more cache-friendly include: (a) using data structures that reduce the number of string accesses; (b) improving the spatial locality of strings, so that strings that are likely to be compared are kept nearer each other; and (c) reducing or eliminating inefficient pointer-based string references.

Cache-oblivious algorithms. Frigo *et al.* [10] introduced *cache-oblivious* algorithms, a novel design approach that respects memory hierarchies. While

(previously-mentioned) cache-aware algorithms need to be aware of machine parameters and may extract the best performance from a particular machine, they may not be portable. In contrast, the notion of cache-oblivious design suggests a highly portable algorithm. Though the cache-oblivious model makes several simplifying assumptions [9], it is nevertheless an attractive and simple model for analyzing data structures in hierarchical memory. Recent results [8,4] indicate that algorithms developed in this model can be competitive with cache-aware implementations. Finally, while there has been related work in the external memory domain, the techniques do not necessarily transfer well to in-memory algorithms.

1.4 Our Contributions

The goal of practical string sorting is to produce a fast and, ideally, an in-place algorithm that is efficient on real-world collections and on real-world machines. In this paper, we investigate the memory usage of burstsort and improve the cache efficiency of the bucket sorting phase.

First, we redesign the buckets of P-burstsort so that the memory requirement of the index structure is drastically reduced. Second, we also introduce a moving field approach whereby a field from the trie node is moved to the point in the bucket where a string is about to be inserted—and is thus shifted with each string insertion—resulting in a further reduction in memory use. These memory reduction techniques show negligible adverse impact on speed and are also applicable to the copy-based [20] variants of burstsort, though we do not evaluate the effects here. As a consequence, memory usage is just 1% greater than an in-place algorithm.

Third, the cache efficiency of bucket sorting is further improved by first copying the string suffixes to a small string buffer before they are sorted. This ensures spatial locality for a process that may well access strings frequently.

1.5 Paper Organization

The remainder of the paper is organized as follows. In Section 2 we show how a substantial redesign of the buckets results in significant reduction in memory waste, so that the algorithm is barely more expensive than an in-place approach. This work is enhanced in Section 2 with a brief discussion of the benefits of moving various bookkeeping fields from the trie nodes to the buckets. In Section 3 we show how buffering a bucket's strings, before sorting them, can lower the running time by a further 30%. In Section 4 we outline the experiments that we performed, and then analyze the results in Section 5. We conclude the paper and set out future tasks in Section 6.

2 Bucket Redesign

The primary aim in this paper is to reduce the memory used by P-burstsort, especially by the buckets. The bucket design used in the original P-burstsort [21], is an array that grows dynamically, by a factor of 2, from a minimum size of

2 up to a maximum size of 8192. While this design proved to be fast, it may lead to significant amounts of *unused* memory. For example, theoretically, if the number of pointers in the buckets were distributed uniformly in the range 1 to L, on average about $L/6$ spaces in each bucket would be unused. Of course, on most collections the distribution of the number of string pointers in a bucket would not be uniform, but there is still waste incurred by barely-filled buckets.

In the BR variant of P-burstsort, we replace each bucket with an array of pointers to sub-buckets. Each sub-bucket is allocated as needed: starting at size 2, it grows exponentially, on demand, until it reached size L/k, at which point the next sub-bucket is allocated. When the total amount of sub-bucket space exceeds L, the node and buckets are burst (as in P-burstsort). Naturally, there is a trade-off between the time spent (re)allocating memory, and the space wasted by the buckets.

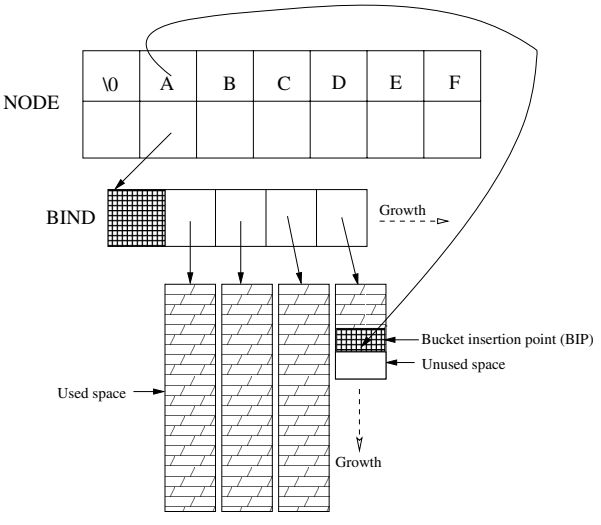


Fig. 3. P_{+BR+MF}-Burstsort: This figure shows the index structure of P-burstsort after incorporating the bucket redesign (BR) and moving field (MF) optimizations

In addition, an index structure is created to manage the sub-buckets; this auxiliary structure must be small so its own creation does not outweigh the benefit of smaller (sub-)buckets. The first component of the index structure, the *bucket index* (BIND), is the dynamically-growing array of pointers to the sub-buckets. It has some auxiliary fields that maintain information about the current state of the sub-bucket structure. The BIND array grows cell-by-cell, only on demand, therefore not wasting space. The other component is at the *bucket insertion point* (BIP), where the next string suffix pointer would be inserted, and therefore moves ahead one word with each insertion (see Figure 3).

In this P_{+BR}-Burstsort variant, only pointers to the strings are copied to the buckets, and the trie node contains two fields: a pointer to the BIND and a

pointer to the BIP. The only bucket that does not grow in this BR manner is the special bucket ($\perp$) for short (consumed) strings, whose BIND index can grow arbitrarily large.

Note that the BIND structure is only accessed during the creation of the bucket, adding a new sub-bucket, and during bursting. These occurrences are relatively rare, compared with the number of times strings are accessed.

$O(\sqrt{L})$ bucket growth. From a theoretical point of view, exponentially-growing buckets make sense principally when the maximum bucket size is unknown. Given that we have a bound on the bucket size, from a worst-case, or uniform distribution, point of view, buckets that grow following the sequence $\sqrt{L}, 2\sqrt{L}, \ldots, L$ seem to make more sense (see for example Exercise 3.2.3 in Levitin's text [15]). With typical values of L and k being 8192 and 32, our sub-bucket data structure does not quite match this, but it is close to a practical application of this principle.

Moving fields from trie nodes to bucket. It is advantageous to keep the trie nodes as compact as possible so that they are mostly cache-resident. In the Moving Field (MF) approach, we copy the latter field fields to the unused space in the bucket, just at the bucket insertion point (BIP). This approach makes burstsort more scalable (due to compact nodes), while simultaneously saving memory.

3 Buffer-Based String Sorting

In P-burstsort, string suffixes are not moved from their original locations: the aim is to sort the pointers to the strings, rather than the strings themselves. Nevertheless, a significant proportion of the cache misses of burstsort occur when the contents of the buckets are sorted. To be compared, the string suffixes must be fetched into cache as required: on average each string must be compared $\Theta(\log(L/k))$ times. In practice, the fetching of string suffixes observes poor spatial locality, especially for strings that have large prefixes in common. Moreover, with large lines in cache, those imported may not contain only bucket-string content, but other data that is not useful.

It was observed, by Sinha *et al.* [20], that actually copying the strings into buckets improves spatial locality, and cache efficiency. In that spirit, we introduce a small buffer to copy the string suffixes into during the bucket-sorting phase to improve their spatial locality and effectively use the cache lines. In a single pass of the bucket of pointers, we fetch all the string suffixes. We then create pointers to the new locations of the string suffixes, while keeping track of the pointers to the actual strings. The extra buffer storage is reused for each bucket and thus its effect on total memory used is negligible. Once the strings are sorted, a sequential traversal of the pointer buffer copies the original pointers to the source array in sorted order.

In Section 5 we show that this String Buffer (SB) modification increases the sort speed, especially for collections with large distinguishing prefixes, such as

URLs. Such collections require the most accesses to the strings and thus stand to benefit most.

The only disadvantage is the amount of work involved in copying suffixes and pointers to these buffers and then back to their source arrays. The SB approach may perform poorly for collections with small distinguishing prefix, such as the random collection, in which the strings may only be accessed once during bucket sorting anyway. But even for such collections, this approach is expected to scale better and be less dependent on the cache line size.

4 Experimental Design

Our experiments measure the time and memory usage of string sorting. In addition, we use cache simulators such as `cachegrind` [19] to measure the instruction count and L2 cache misses.

Data Collections. We use four real-world text collections: *duplicates*, *no duplicates*, *genome*, and *URL*. In addition, we also create a *random* collection in which the characters are generated uniformly at random. These collections, whose details are provided in Table 1, are similar to those used in previous works [21,22,20].

Table 1. Statistics of the data collections used in the experiments

	Size (Mbytes)	Distinct Words ($\times 10^5$)	Word Occurrences ($\times 10^5$)
Duplicates	304	70	316
No duplicates	382	316	316
Genome	302	2.6	316
Random	317	260	316
URL	304	13	100

The *duplicates* and *no duplicates* collections were obtained from the Wall Street Journal subcollection of TREC web data [12]. The *duplicates* collection contains words in occurrence order and includes duplicates, while the *no duplicates* collection contains only unique strings that are word pairs in occurrence order. The genome collection consists of fixed-length strings (of length 9 characters), extracted from nucleotide strings from the Genbank collection [5]. The URL collection is obtained in order of occurrence from the TREC web data.

Algorithms compared with. The performance of the new burstsort enhancements is compared to the original P-burstsort [21], adaptive radixsort [2], a fast multikey quicksort [6], and MBM radixsort [16].

Algorithm parameters. The buckets in burstsort are grown exponentially by a factor-of-2 starting from a size of 2 to L, where L is the bucket threshold. In our experiments we varied L from a minimum of 8192 to a maximum of $131,072$.

Note that for each individual sub-bucket, L/k is 256, where k is the number of sub-buckets. The alphabet size was restricted to 128 symbols with the random collection having 95 symbols.

Table 2. Architectural parameters of the machine used for experiments

Workstation	Pentium	Pentium	PowerPC
Processor type	Pentium IV	Core 2	PowerPC 970
Clock rate	2800 MHz	2400 MHz	1800 MHz
L1 data cache (KB)	8	32	32
L1 line size (bytes)	64	64	128
L2 cache (KB)	512	4096	512
L2 block size (bytes)	64	64	128
Memory size (MB)	2048	2048	512

Machines. The experiments were conducted on a 2800 MHz Pentium IV Machine with a relatively small 512 KB L2 cache and 2048 MB memory. The operating system was Linux with kernel 2.6.7 under light load. The highest compiler optimization O3 has been used in all the experiments. We also used a more recent dual-core machine with relatively large 4096 KB L2 cache as well as a PowerPC 970 architecture. Further details of the machines are shown in Table 2.

All reported times are measured using the `clock` function, and are the average of 10 runs. As these runs were performed on a machine under light load and on 300 megabyte data sets, the standard deviation is small. On the PowerPC, owing to the smaller memory, we used a smaller data set with 10 million strings [21].

5 Discussion

Bucket redesign and moving fields. All pointer-based string sorting algorithms must create space for the pointer array. The key memory overhead of P-Burstsort is its burst trie-style index structure. Table 3 shows this extra memory usage, including unused space in buckets, memory allocation bytes and associated index structures of the previous algorithm, and the variants introduced here.

The BR and MF modifications cause a large reduction in memory use. For the three real-world collections (Duplicates, No Duplicates, and Genome) there is at least a factor 35 reduction. Table 3 also confirms that increasing bucket sizes result in smaller indexes in P_{+BR+MF}-Burstsort, unlike in P-Burstsort (except for the Random collection).

The BIND and BIP structures require additional maintenance and dynamic memory allocation. Table 4 shows that the number of dynamic memory allocations, in the new variants, increased by over an order-of-magnitude. The good news is that although the memory demand drops significantly, running times increase by only 10% (see Table 5).

We also observe that the MF technique speeds up the sorting of the URL collection in Table 5, due to the relatively large number of trie nodes in that

Table 3. Memory use (in megabytes) incurred by the index structure of P-Burstsort and (the new) P_{+BR+MF}-Burstsort for different bucket thresholds and collections

		Collections				
Threshold	P-Burstsort	Duplicates	No duplicates	Genome	Random	URL
8192	None	94.37	109.37	53.37	47.37	23.85
	+BR+MF	10.37	12.37	4.37	3.37	3.85
16384	None	94.37	100.37	57.37	50.37	24.85
	+BR+MF	5.37	6.37	2.37	3.37	1.85
32768	None	94.37	95.37	48.37	50.37	24.85
	+BR+MF	3.37	3.37	1.37	3.37	0.85
65536	None	90.37	92.37	55.37	61.37	25.85
	+BR+MF	2.37	2.37	1.37	3.37	0.85

Table 4. Number of dynamic memory allocations of P-burstsort and P_{+BR+MF}-Burstsort. The bucket threshold is 32768.

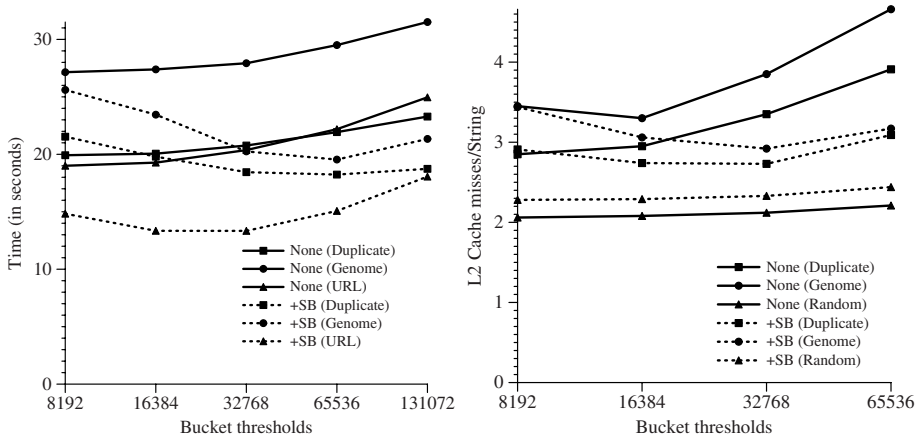
Algorithm	Duplicates	No duplicates	Genome	Random	URL
P-Burstsort	271,281	307,572	76,625	109,922	79,451
P_{+BR+MF}-Burstsort	1,936,175	1,828,805	1,948,281	1,205,117	1,293,854
Factor increase	7.13	5.94	25.42	10.96	16.28

collection. Making these trie nodes compact with the MF enhancement makes up for the cost of shifting the fields. Thus, these enhancements not only significantly reduce the memory usage but also aids in speeding up sorting.

String buffer. The SB modifications, described in Section 3, are a successful enhancement overall. Table 5 shows that for all real-world collections, which have reasonable distinguishing prefix, this approach is beneficial. Only for the Random collection, whose strings may need be fetched only once during bucket sorting, does the string buffer approach result in a slight slowdown. The average number of strings (for the Random collection) in each bucket is less than half the number of cache lines, thus the strings are mostly cache-resident as they are sorted.

Table 5. Sorting time (in seconds) as a function of algorithm modification for all five collections on the Pentium IV

	Collections				
P-Burstsort	Duplicates	No duplicates	Genome	Random	URL
None	20.76	23.64	27.92	16.98	20.36
+SB	18.44	21.16	20.25	18.31	13.33
+BR	22.59	25.25	29.88	20.80	21.03
+BR+MF	23.11	26.02	30.00	22.40	20.65
+SB+BR+MF	22.04	24.71	23.08	29.39	13.87



(a) Time (in seconds) (b) L2 cache misses

Fig. 4. Sorting time (in seconds) and L2 cache misses per string incurred by P-Burstsort and P_{+SB}-Burstsort as a function of bucket thresholds on the Pentium IV

Figure 4 and Table 5 confirms that for collections with larger distinguishing prefix, such as genome and URL, the approach is indeed the most successful and reduces running time by about 30%.

The increase in instruction count (using cachegrind), by up to 80% is more than compensated for by small reductions in the number of L2 cache misses, shown in Figure 4 (b). Moreover, on machines such as PowerPC (discussed below), where the TLB misses are expensive, such an approach is beneficial. The SB approach adapts better to the cache capacity and enhances scalability.

We observe that the BR and MF modifications lead to lower instruction counts (by 7%) due to the reduced copying costs from using small sub-buckets (even for the Random collection). The small increase in cache misses (of 8%) by $P_{+SB+BR+MF}$-Burstsort over P_{+SB}-Burstsort for the real-world collections are due to BIND accesses and moving fields during string insertion and bursts. Combining all three modifications results in a lowering of running time in the Genome and URL collections, a small increase in the Duplicates and No Duplicates collections (of 6% and 4.5% respectively), but a poor performance in the (unrealistic) Random collection. Below, we show that the performance of these approaches on other machine architectures can simultaneously reduce memory usage and indeed improve performance.

Other machine architectures. On the Pentium Core 2 machine, the running time of P_{+BR}-burstsort is faster than that of P-burstsort for *all* real-world collections (shown in Table 6). Similarly, on the PowerPC, P_{+BR}-burstsort was up to 10% faster than that of P-burstsort (shown in Table 7). On another small cache machine (PowerPC), the SB modification reduced the running time by up to 40% (see Table 7). Thus, using a small buffer to copy string suffixes prior to bucket

Table 6. Sorting time (in seconds) for all five collections on the Pentium Core 2 machine. The bucket threshold is 32768.

Algorithm	Duplicates	No duplicates	Genome	Random	URL
Adaptive radixsort	13.63	15.04	17.75	9.72	9.01
MBM radixsort	15.57	16.01	22.38	10.61	13.53
Multikey	12.19	14.04	13.09	12.95	6.39
P-Burstsort	7.45	8.81	8.63	5.80	4.92
P_{+BR}-Burstsort	7.25	8.64	8.31	6.10	4.60
P_{+BR+MF}-Burstsort	7.26	8.75	8.23	6.28	4.54

Table 7. Sorting time (in seconds) for all five collections on the PowerPC 970 machine. The collections contain 10 million strings. The bucket threshold is 32768.

Algorithm	Duplicates	No duplicates	Genome	Random	URL
Adaptive radixsort	15.48	17.74	23.43	13.64	55.33
MBM radixsort	15.24	16.20	24.59	9.05	74.33
Multikey	14.91	17.16	21.94	18.68	58.75
P-Burstsort	10.22	11.71	16.69	7.19	48.26
P_{+SB}-Burstsort	8.24	9.24	10.48	8.35	23.53
P_{+BR}-Burstsort	9.26	10.79	14.90	7.66	45.60
P_{+BR+SB}-Burstsort	7.68	8.74	9.25	9.50	22.61

sorting is beneficial to using the cache capacity productively while reducing the TLB misses. The BR and SB techniques combine to produce the fastest times while simultaneously reducing the memory usage significantly.

These results demonstrate that the modifications work well across different machine architectures and aids in improving the speed while simultaneously enhancing scalability.

6 Conclusions and Further Work

String sorting remains a fundamental problem in computer science. It needs to be revisited because changes in computer architecture have not so much changed the objective functions, but have changed the estimates we have of them. Burstsort was already known to be fast: in this paper, its demands on main memory have been significantly reduced, without running time being compromised.

The BR enhancement enables large reductions in the bucket size, with negligible impact on sorting time, even though it requires an order-of-magnitude more dynamic allocations. The MF technique reduces the trie node memory usage by moving fields to the unused space in the bucket and shifting them with each string insertion. The success of the SB enhancement is further evidence that accessing strings in arbitrary locations (using pointers) is inefficient and there are benefits in improved spatial locality.

Now that the index structure can be reduced to around 1% of the size of the input arrays, we have produced an almost in-place string sorting algorithm that is fast in practice. Briefly, burstsort with these optimizations, is a fast and an almost in-place string sorting algorithm that is demonstrably efficient on real-world string collections, including those with large distinguishing prefixes.

Further work. With large caches now available in multicore processors, it would be interesting to see if our sampling approaches [22] can be developed further: larger caches are expected to be more tolerant of sampling errors. Can burstsort make significant speed increases by using multiple cores for sorting the buckets? In future implementations, we intend to explore the effect of trie layouts such as using an approximate van Emde Boas layout in a dynamic environment.

Acknowledgments

This work was supported by the Australian Research Council.

References

1. Aho, A., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading (1974)
2. Andersson, A., Nilsson, S.: Implementing radixsort. ACM Jour. of Experimental Algorithmics 3(7) (1998)
3. Arge, L., Ferragina, P., Grossi, R., Vitter, J.S.: On sorting strings in external memory. In: Leighton, F.T., Shor, P. (eds.) Proc. ACM Symp. on Theory of Computation, El Paso, pp. 540–548. ACM Press, New York (1997)
4. Bender, M.A., Colton, M.F., Kuszmaul, B.C.: Cache-oblivious string b-trees. In: PODS 2006: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, New York, NY, USA, pp. 233–242. ACM Press, New York (2006)
5. Benson, D.A., Karsch-Mizrachi, I., Lipman, D.J., Ostell, J., Wheeler, D.L.: Genbank. Nucleic Acids Research 31(1), 23–27 (2003)
6. Bentley, J., Sedgewick, R.: Fast algorithms for sorting and searching strings. In: Saks, M. (ed.) Proc. Annual ACM-SIAM Symp. on Discrete Algorithms, New Orleans, LA, USA. Society for Industrial and Applied Mathematics, pp. 360–369 (1997)
7. Bentley, J.L., McIlroy, M.D.: Engineering a sort function. Software—Practice and Experience 23(11), 1249–1265 (1993)
8. Brodal, G.S., Fagerberg, R., Vinther, K.: Engineering a cache-oblivious sorting algorithm. ACM Jour. of Experimental Algorithmics 12(2.2), 23 (2007)
9. Demaine, E.D.: Cache-oblivious algorithms and data structures. In: Lecture Notes from the EEF Summer School on Massive Data Sets, BRICS, University of Aarhus, Denmark, June 2002. LNCS (2002)
10. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Beame, P. (ed.) FOCS 1999: Proceedings of the 40th Annual Symposium on Foundations of Computer Science, Washington, DC, USA, pp. 285–298. IEEE Computer Society Press, Los Alamitos (1999)

11. Graefe, G.: Implementing sorting in database systems. Computing Surveys 38(3), 1–37 (2006)
12. Harman, D.: Overview of the second text retrieval conference (TREC-2). Information Processing and Management 31(3), 271–289 (1995)
13. Heinz, S., Zobel, J., Williams, H.E.: Burst tries: A fast, efficient data structure for string keys. ACM Transactions on Information Systems 20(2), 192–223 (2002)
14. Knuth, D.E.: The Art of Computer Programming: Sorting and Searching, 2nd edn., vol. 3. Addison-Wesley, Reading (1998)
15. Levitin, A.V.: Introduction to the Design and Analysis of Algorithms, 2nd edn. Pearson, London (2007)
16. McIlroy, P.M., Bostic, K., McIlroy, M.D.: Engineering radix sort. Computing Systems 6(1), 5–27 (1993)
17. Moffat, A., Eddy, G., Petersson, O.: Splaysort: Fast, versatile, practical. Software—Practice and Experience 26(7), 781–797 (1996)
18. Sedgewick, R.: Algorithms in C, 3rd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1998)
19. Seward, J.: Valgrind—memory and cache profiler (2001), `http://developer.kde.org/~sewardj/docs-1.9.5/cg_techdocs.html`
20. Sinha, R., Ring, D., Zobel, J.: Cache-efficient string sorting using copying. ACM Jour. of Experimental Algorithmics 11(1.2) (2006)
21. Sinha, R., Zobel, J.: Cache-conscious sorting of large sets of strings with dynamic tries. ACM Jour. of Experimental Algorithmics 9(1.5) (2004)
22. Sinha, R., Zobel, J.: Using random sampling to build approximate tries for efficient string sorting. ACM Jour. of Experimental Algorithmics 10 (2005)