Engineering Burstsort: Toward Fast In-Place String Sorting

RANJAN SINHA and ANTHONY WIRTH
The University of Melbourne

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 article, we introduce improvements that reduce by a significant margin the memory requirement of Burstsort: 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 such as for string management.

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. Importantly, the trade-off for the reduction in memory use is only a very slight increase in the running time of Burstsort 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%. These memory usage enhancements have enabled the copy-based approach [Sinha et al., JEA 2006] to also reduce the memory usage with negligible impact on speed.

Categories and Subject Descriptors: E.1 [**Data Structures**]; E.5 [**Files**]: Sorting/Searching

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Sorting, algorithms, cache, experimental algorithms, string management, tries

ACM Reference Format:

Sinha, R. and Wirth, A. 2010. Engineering burstsort: Toward fast in-place string sorting. ACM J. Exp. Algor. 15, Article 2.5 (March 2010), 24 pages.
DOI = 10.1145/1671973.1671978 http://doi.acm.org/10.1145/1671973.1671978

This work was supported by the Australian Research Council.
This article incorporates work previously published in "Engineering Burstsort: Towards Fast In-place String Sorting", which appears in *Proceedings of the 7th Workshop on Efficient and Experimental Algorithms*.
Author's address: R. Sinha, and A. Wirth, The University of Melbourne; email: `sinhar@unimelb.edu.au`.

DOI 10.1145/1671973.1671978 http://doi.acm.org/10.1145/1671973.1671978

1. INTRODUCTION

This article revisits the issue of sorting strings efficiently. String sorting remains a key step in solving contemporary data management problems. Arge et al. [1997] 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, so swapping strings is not as simple as swapping integers. Second, strings are compared one character (or word) 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.

1.1 Traditional Approaches to String Sorting

There have been several advances in fast sorting techniques designed for strings. Traditionally, we are taught to measure algorithm speed by the number of instructions, such as *comparisons* or *moves*: a unit-cost RAM model. Early algorithms focused on reducing instruction count under this model [Aho et al. 1974; Knuth 1998].

Standard string-sorting algorithms, as taught in typical undergraduate studies, 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 honor this rule include the following.

—*Multikey quicksort*. Bentley and Sedgewick [1997] introduced a hybrid of quicksort and radixsort named three-way radix quicksort [Sedgewick 1998]; they then extended this to produce multikey quicksort [Bentley and Sedgewick 1997].

—*Radixsort variants*. Improvements to radixsort for strings were proposed by McIlroy et al. [1993] (the so-called MBM algorithm) and Andersson and Nilsson [1998].

In this article, we compare our algorithms with these as they have been observed to be among the fastest [Sinha and Zobel 2004]. The performance of other algorithms can be obtained from the first author's earlier papers [Sinha et al. 2006; Sinha and Zobel 2004].

Other methods include three-way partitioning—an important quicksort innovation [Bentley and McIlroy 1993]—and splaysort, an adaptive sorting algorithm, introduced by Moffat et al. [1996]—a combination of the splaytree data structure and insertionsort.

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 explore these issue below.

Note that stability is ignored in this article as any sorting algorithm can be made stable by appending the rank of each key in the input [Graefe 2006; Sinha et al. 2006].

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 [Sinha and Zobel 2004], on the larger datasets there were typically 5 to 10 cache misses per string during radix-based sorting on a machine with 1MB 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: (i) using data structures that reduce the number of string accesses; (ii) improving the spatial locality of strings so that strings that are likely to be compared are kept nearer each other, and (iii) reducing or eliminating inefficient pointer-based string references.

Cache-oblivious algorithms. Frigo et al. [1999] 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 [Demaine 2002], it is nevertheless an attractive and simple model for analyzing data structures in hierarchical memory. Recent results [Brodal et al. 2007; He and Luo 2008; Bender et al. 2006] 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. Cache-oblivious algorithms are usually complex, and since the cache (or other memory) parameters are often available, we have chosen to use the cache-aware approach for Burstsort.

1.2 Burstsort

Burstsort is a technique that combines the burst trie [Heinz et al. 2002] with standard (string) sorting algorithms [Bentley and Sedgewick 1997; McIlroy et al. 1993]. It was introduced by the first author, and its variants are amongst the fastest algorithms for sorting strings on current hardware [Sinha and Zobel 2004]. 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.

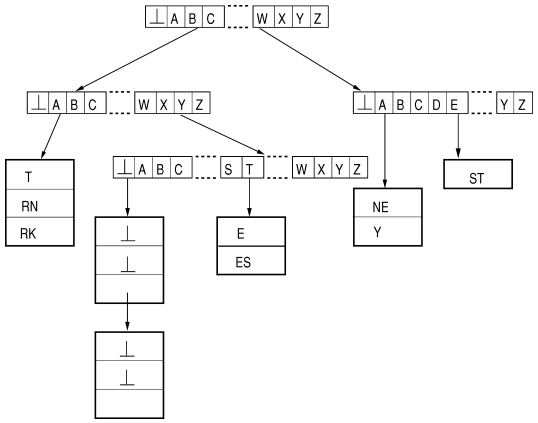


Fig. 1. Index structure of P-Burstsort. Strings inserted into the burst trie are "bat," "barn," "bark," "by," "by," "by," "by," "byte," "bytes," "wane," "way," and "west." The figure has been adapted from an earlier publication [Sinha and Zobel 2004].

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 (labeled ⊥) 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, as defined in the following text. 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.

Sorting of buckets. The standard P-Burstsort [Sinha and Zobel 2004] 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.

Previous improvements. A reduction in the number of bursts results in a reduction in string accesses. To that end, in the sampling-based Burstsort

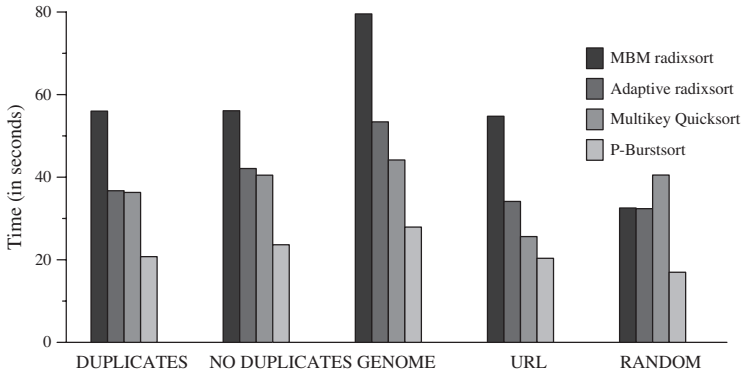


Fig. 2. Time (in seconds) to sort with MBM radixsort, adaptive radixsort, multikey quicksort, and P-Burstsort on each of the five collections (see Section 5) on a Pentium IV machine with a small 512 KB L2 cache. The bucket threshold used for P-Burstsort is 32,768.

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 [Sinha and Zobel 2005]. 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 [Sinha et al. 2006], 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 it is a major focus in this article; we outline the contributions in the next section.

1.3 Our Contributions

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

(1) First, we redesign the buckets of P-Burstsort so that the memory requirement of the index structure is drastically reduced.
(2) 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 [Sinha et al. 2006] variants of Burstsort and is discussed further in Section 2. As a consequence of these changes, memory usage is just 1% greater than an in-place algorithm.

(3) 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 access strings frequently.

1.4 Article Organization

The remainder of the article 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 space expensive than an in-place approach. This work is enhanced with a brief discussion of the benefits of moving various bookkeeping fields from the trie nodes to the buckets. Furthermore, we also apply these techniques to the copy-based approach to reduce its memory usage. In Section 3, we show how buffering a bucket's strings, before sorting them, can lower the running time by a further 30%. The impact of the layout of the trie nodes in memory is discussed in Section 4. In Section 5, we outline the experiments that we performed, and then analyze the results in Section 6. We conclude the article and set out future tasks in Section 7.

2. BUCKET REDESIGN

The primary aim in this article is to reduce the memory used by P-Burstsort, especially by the buckets. The bucket used in the original P-Burstsort [Sinha and Zobel 2004], is an array that grows dynamically, by a factor of 2, from a minimum size of 2 up to a maximum size of 8,192. 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, where L is measured in words (for storing the pointers). Given that the bucket is of size $L/2^i$, the expected wasted space is $L/2^{i+1}$. The probability that the bucket is of size $L/2^i$ is $1/2^i (i \geq 1)$, since the distribution is uniform. Therefore, the expected wasted space is roughly:

$$\sum_{i=1}^{\infty} \frac{L}{2^{2i+1}} = \frac{L}{8} \sum_{j=0}^{\infty} \frac{1}{4^i} = \frac{L}{8} \cdot \frac{4}{3} = \frac{L}{6} \tag{1}$$

Of course, on most real-world 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. Our first modification to P-Burtsort is to replace each bucket with an array of pointers to sub-buckets; we call this the BR variant. Let k be the maximum number of sub-buckets that can be allocated. Each sub-bucket is allocated as needed: Starting at size 2, it grows exponentially, on demand, until it reaches size L/k, at which point the next sub-bucket is allocated. When the total amount of sub-bucket space exceeds L (or when there are k sub-buckets allocated) the node and sub-buckets are burst (as in P-Burstsort).

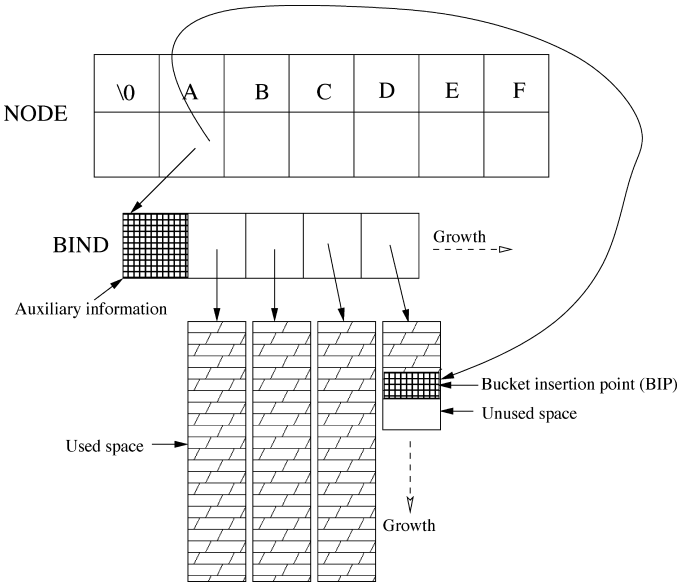


Fig. 3. P_{+BR+MF}-Burstsort: This figure shows the index structure of P-Burstsort after incorporating the bucket redesign (BR) and moving field (MF) optimizations.

Naturally, there is a trade-off between the time spent (re)allocating memory, and the space wasted by the buckets.

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 array of pointers to the sub-buckets, which grows dynamically. It has some auxiliary fields that maintain information about the current state of the sub-bucket structure. The BIND array in fact 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).

We refer to this variant as P_{+BR}-Burstsort. 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 to the number of times strings are accessed.

$O(\sqrt{L})$ bucket growth. From a theoretical point of view, exponentially growing buckets make sense 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, e.g., Exercise 3.2.3 in Levitin [2007]). With values of L and k being 8,192 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 addition to maintaining two pointers to the BIND structure and the next available position in the bucket, the fields in the trie node include information such as the current growth level of the sub-bucket, available free space in that sub-bucket, and whether it is the last sub-bucket. However, these fields are accessed only if a string is being inserted into the bucket and can be moved to the sub-bucket. In the moving field (MF) approach, we move these fields to the unused space in the bucket, just at the BIP in the last allocated sub-bucket. This approach makes Burstsort more scalable (due to compact nodes) during trie traversal, while simultaneously saving memory.

2.1 In-Place Algorithm

All string sorting algorithms would require space for the strings and for the pointers to the strings. We might define an algorithm as being in place if it used $O(\log n)$ extra space. Although memory efficient, Burstsort in fact requires $O(n)$ additional memory, but, as we will now see, the constant is small. Each string is associated with a bucket, which, in turn, is connected to a path of trie nodes back to the root. Given a maximum bucket size of L, we have seen that, on average, $L/6$ pointers are wasted, and there are approximately L/k other pointers in a particular bucket. On average, the number of buckets needed is $2n/L$, so there are roughly $n/3$ extra pointers. Each leaf node in the trie is connected to 128 buckets, so there are roughly $n/(64L)$ leaves and thus approximately $2n/(127L)$ nodes in the trie. Given that each node has about 128 pointers, the number of extra pointers in the trie is about $2.1n/L$. Certainly, it seems that $2n$ is a reasonable approximation to approximate the amount of extra memory (in bytes) used by the Burstsort index structure.

Using a sub-bucket structure, the amount of wasted space in each bucket is, on average, $L/(2k)$. Since there are approximately $2n/L$ buckets, there are n/k wasted pointers, and thus there are $4n/k$ extra bytes. We assume that $k = 32$ and L = 8,192, so the number of extra bytes is roughly $n/8 + n/1{,}024$, which is roughly $1/16$ of that used in the original version of Burstsort.

2.2 Application to Copy-Based Approach

The memory reduction techniques can also be applied to the copy-based approach [Sinha et al. 2006]. The copy-based approach has been successfully applied to the task of string and integer management [Askitis and Zobel 2005; Askitis and Sinha 2007; Nash and Gregg 2008] and a reduction of memory usage would further benefit them. In the copy-based approach, the buckets store string suffixes and thus can be much larger than if they were storing just pointers, as in P-Burstsort. The bucket size in P-Burstsort is limited to a large extent by the number of cache lines, due to the poor spatial locality of the

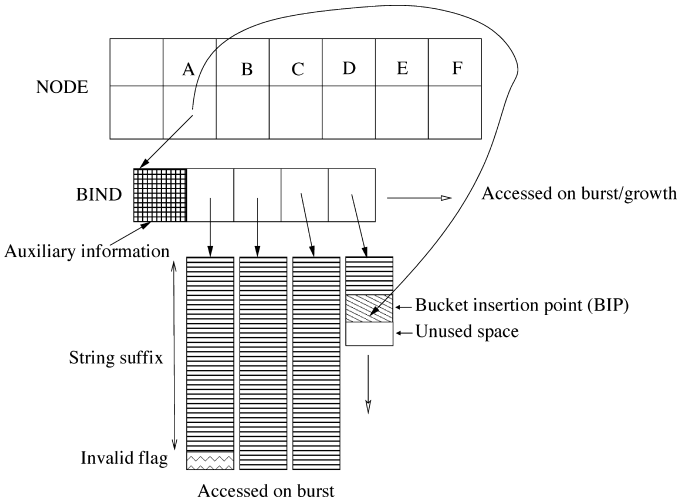


Fig. 4. C_{+BR+MF}-Burstsort: This figure shows the index structure of C-Burstsort after incorporating the bucket redesign (BR) and moving field (MF) optimizations.

fetched strings. In contrast, copying suffixes to buckets significantly improves the spatial locality; thus, it is able to benefit from the size of cache and not restricted by the number of cache lines. The buckets in the copy-based Burstsort can be much larger and waste unused space in the buckets. Consequently, our memory reduction approaches have a greater potential to reduce the memory usage in the buckets.

Figure 4 shows the structure used for the copy-based approach. Each symbol in the trie node contain two fields, a pointer to the BIP structure and a pointer to the BIND structure. The BIND structure, contains the current and maximum number of sub-buckets that can be indexed by the BIND structure. Note that the information pertaining to the maximum number of sub-buckets is unnecessary in the BIND structure, as it remains constant across all buckets. Thus, the size of the 4-byte field can be reduced further in all Burstsort variants that incorporate the bucket redesign technique.

The BIP structure at the end of the current sub-bucket is composed of three fields. They are available space in bytes (fspace), current growth level (glevel), and whether it is the last sub-bucket (lsubu). The fspace field is of length 12 bits, glevel is of length 3 bits and the lsubu is of 1 bit. The fspace field is the amount of free space in the sub-bucket within the current growth level. The final sub-bucket is managed differently to the rest, so we use 1 bit to keep track of the last sub-bucket. Thus, the BIP structure requires only 2 bytes to store enough information to implement the memory reduction techniques in the copy-based approach.

In the copy-based approach, extra processing is required in the boundaries within the sub-buckets. This is a key difference between inserting pointers to strings and the string suffixes. Pointers are of fixed size while the string suffixes are of variable-length and thus care needs to be taken near the boundary

of sub-buckets. During each string insertion, the suffix (portion of the string that is not yet consumed by the trie) is inserted followed by the 2-byte BIP structure.

The insertion of a string suffix near the end of a sub-bucket causes one of four different boundary conditions: no space in the sub-bucket for the BIP structure, space for 1 byte of the BIP structure, not enough space left for the string suffix, or space for both string suffix and the BIP structure. The approach used for each of these four cases are as follows.

(1) No space for BIP structure. The string suffixes are copied to the existing sub-bucket while the 2-byte BIP is written to a newly allocated sub-bucket.
(2) Not enough space for string. A portion of the string is copied to the current sub-bucket and the rest of the string and the BIP structure is copied to the new sub-bucket.
(3) Space for 1 byte for the BIP structure. If there is only 1 byte left, then that portion is marked as invalid so that it is not processed during either the bursting or the traversal phases. The BIP structure is copied to the new sub-bucket.
(4) Enough space for string and BIP structure. Both the string and the BIP structure are copied to the current sub-bucket.

Figure 4 shows that the first sub-bucket follows the third case, where there is only a single byte for the BIP structure. That byte is marked to indicate that it is invalid.

In our implementation, during bursting the sub-bucket is copied to a temporary array that is large enough to hold the entire bucket. This copying is a code artifact and is not essential; it would be slightly more efficient if the existing sub-buckets are directly traversed for processing during this phase. In Section 6, we shall discuss the impact of these techniques on the copy-based approach.

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 longer strings such as URLs. Moreover, with large cache lines, the lines imported may not only contain bucket-string content but also other data that is not useful.

Sinha et al. [2006] observed that when C-Burstsort actually copies the strings into buckets, it 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.

On a related note, in CPL-Burstsort [Sinha et al. 2006] a predetermined fixed number of characters were copied for each string in a bucket during the insertion phase. When these characters were completely processed during the bucket sorting phase, subsequent characters were copied from the source array to the bucket for further processing. This approach of copying fixed number of characters was later extended to cover the entirety of the string collection [Ng and Kakehi 2007] (not only to small buckets as in Burstsort) and was observed to be efficient in comparison to traditional MSD radixsort algorithms.

In Section 6, we show that this string buffer (SB) modification increases the sort speed, especially for collections requiring significantly long prefixes to distinguish the strings (i.e., with large distinguishing prefixes), such as URLs. Such collections require the most accesses to the strings and thus stand to benefit most.

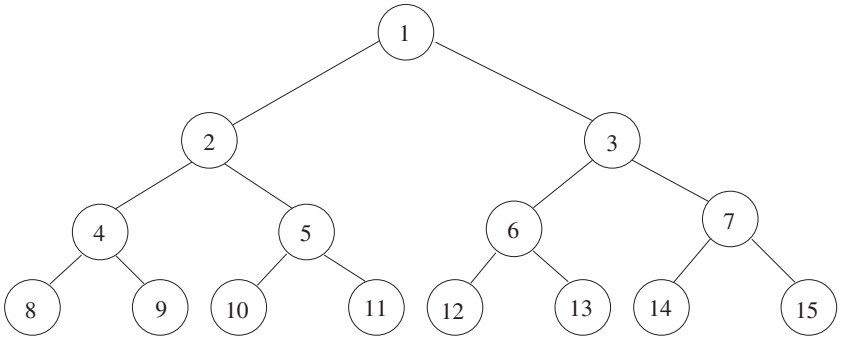
The only disadvantage is the number of instructions involved in copying suffixes and pointers to these buffers and then back to their source arrays. The SB approach may perform poorly for collections not requiring a large distinguishing prefix, such as the random collection, in which the strings may only be accessed once during bucket sorting anyway. But even for such collections, the SB approach is expected to scale better and be less dependent on the cache line size.

4. TRIE NODE ALLOCATION

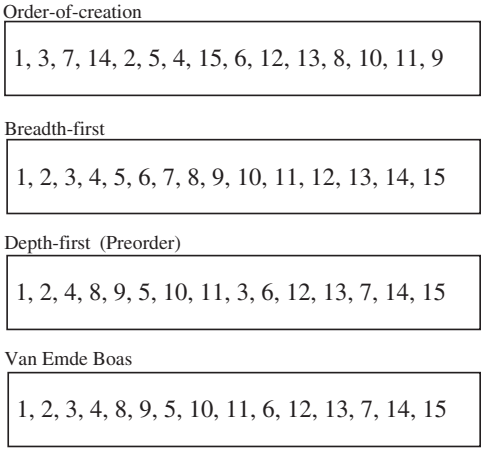
All strings traverse the trie node structure prior to being inserted into their respective buckets. Each of these trie nodes are allocated when a bucket is burst. For most string collections, the trie node allocations are arbitrary and are not designed to account for spatial locality. Hence, while traversing the trie structure, each access to a trie node may result in L1/L2 cache misses and TLB cache misses. Thus, it is necessary to improve the locality and thus improve the scalability of this structure. This can be achieved by using cache-conscious trie layout approaches that increases the spatial locality of these trie nodes to reduce cache misses.

The naïve approach shows poor locality in practice. A better approach is to index several memory blocks, each of which can hold multiple trie nodes. For example, in our experiments, we store 100 trie nodes per block. This helps to reduce the number of cache misses while providing a means to tune the locality, number of calls to memory allocators and memory usage. While this reduces the number of pages that the trie nodes now resides in, it does not yet address locality issues between the trie nodes (such as parent and its children). A parent might be in a location (either in the same or a different block) that could well be far removed from its children.

The next step is to investigate the practicality of further improving the locality of trie nodes along the traversal path to improve its scalability. The question



(a) Tree nodes



(b) Tree layout in memory

Fig. 5. Tree layout approaches: Four layouts for tree node allocation were used. These include the trie nodes allocated in Order-of-creation (naïve), Depth-first, Breadth-first, and Van Emde Boas layout.

is how and when to improve the locality of the trie nodes. In a nonsampling-based approach, the only solution is to dynamically rearrange the trie nodes to improve locality during the insertion phase. But this needs to be done in the midst of the insertion phase, as there are no trie nodes prior to the insertion phase and any modification to the trie layout after the insertion phase has negligible benefit.

To select the trie layout to use, we compare four different trie layout strategies. For this purpose, we create a trie node structure that indexes 100 million strings from a relatively larger genome dataset. These trie nodes are allocated into blocks containing a maximum of 100 trie nodes. As shown in Figure 5, these trie nodes are then rearranged in one of the three layouts: depth-first, breadth-first, and the Van Emde Boas layout [Brodal et al. 2004; Ostlin and Pagh 2003].

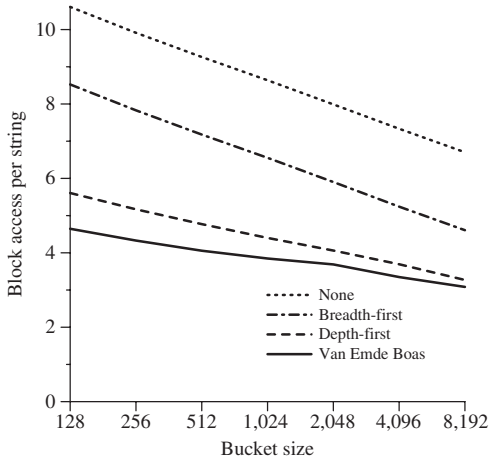


Fig. 6. Choice of trie layout: The average number of block accesses per string due to using three node layout strategies, Depth-first, Breadth-first, and Van Emde Boas. These results are from the insertion of 1,000,000 strings on the 100MB genome collection. These results are also compared with the average number of accesses per string to a trie node, assuming a block size of one. The size of the bucket is varied from 128 to 8,192. Successive accesses to the same block are counted as a single access; each block contains 100 trie nodes.

To obtain the number of accesses to a trie node and to each block of trie node, an additional 1,000,000 strings is used to traverse the trie structure to locate the buckets. In the Van Emde Boas Layout, the child is kept contiguously with its parent.

A block is reaccessed only if the trie node has a child in that block, otherwise a new block is accessed; only accesses to a new block is counted during the trie traversal. The number of block accesses are compared with varying bucket sizes, as shown in Figure 6. A larger bucket size results in fewer trie nodes being created and fewer number of trie node accesses during traversal.

Note that in this experiment, all trie nodes were allocated prior to these nodes being rearranged in a different layout. For the problem of sorting, the trie layout rearrangement needs to be done within the insertion phase. We can observe from Figure 6 that the Van Emde Boas layout results in the fewest number of accesses to new blocks, which can then translate to page blocks and cache lines. The Van Emde Boas layout is used in the trie layout experiments. We rearrange the trie nodes to this layout after a predetermined percentage of strings are inserted into the trie. We term this as an approximate Van Emde Boas layout (AVEB-layout), as soon after this rearrangement, the trie nodes are again allocated in occurrence order of the strings.

There are two key issues to consider: How many such rearrangements are required, and when these should be done. Too many rearrangements can make this step expensive, while too few may not enhance the scalability. In the following text, we elaborate on the steps required to convert a trie layout to the AVEB-layout.

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

	Size (MB)	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 string collection is divided into equal parts; for example, each part could be 20% of the collection. After each such part is inserted into the trie structure, the trie is rearranged into the AVEB-layout. If the part size is 20%, the number of times that the trie nodes are rearranged is 5. Temporary space is allocated to maintain the rearranged trie nodes. After each rearrangement, the pointer to the root node points to the newer memory, thus eliminating the need to copy the nodes back to the old memory space. During and after each rearrangement, additional cache misses may be incurred to fill the cache, but this is relatively insignificant, as it is a function of the number of trie nodes, which is small.

In practice, the trie nodes on the upper levels are created at the beginning and may have little to benefit from after the first few rearrangements. The lower levels of the trie node structure have a greater potential to benefit from these rearrangements. Finding an appropriate frequency of such rearrangements needs to be ascertained empirically and is considered in Section 6.

5. EXPERIMENTAL DESIGN

Our experiments measure the time and memory usage of string sorting. In addition, we use cache simulators such as `cachegrind` [Seward 2001] 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 I, are similar to those used in previous works [Sinha and Zobel 2004, 2005; Sinha et al. 2006].

The duplicates and no duplicates collections were obtained from the Wall Street Journal subcollection of TREC Web data [Harman 1995]. 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 nine characters), extracted from nucleotide strings from the Genbank collection [Benson et al. 2003]. The URL collection is obtained in order of occurrence from the TREC Web data.

Algorithms under consideration. The performance of the new Burstsort enhancements is compared to that of the original P-Burstsort [Sinha and Zobel

Table II. Architectural Parameters of the Machines Used for Experiments

Workstation	Pentium	Pentium	PowerPC
Processor Type	Pentium IV	Core 2	PowerPC 970
Clock Rate	2,800MHz	2,400MHz	1,800MHz
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

2004], adaptive radixsort [Andersson and Nilsson 1998], a fast multikey quicksort [Bentley and Sedgewick 1997], and MBM radixsort [McIlroy et al. 1993].

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 8,192 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 maximum alphabet size of 128 symbols was used in the trie nodes for all collections. The alphabet size of the collections ranged from a minimum of 4 symbols for the genome collection to a maximum of 95 symbols for the random collection.

Machines. The experiments were conducted on a 2800MHz Pentium IV Machine with a relatively small 512KB L2 cache and 2,048MB memory. The operating system was Linux with kernel 2.6.7 under light load. The highest compiler optimization O3 was used in all the experiments. We also used a more recent dual-core machine with relatively large 4,096KB L2 cache as well as a PowerPC 970 architecture. Further details of the machines are shown in Table II.

All codes were implemented in the C language. 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 300MB datasets, the standard deviation is small. On the PowerPC, owing to the smaller memory, we used a smaller dataset with 10 million strings [Sinha and Zobel 2004].

6. 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 III shows this extra memory usage, including unused space in buckets, memory allocation overheads and associated index structures for standard Burstsort and the variants introduced here. Note that the used memory within the buckets for the pointers are not considered 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 as much as a factor of 35 reduction. Table III also confirms that increasing bucket sizes result in smaller indexes in P_{+BR+MF}-Burstsort, unlike in P-Burstsort (except for the Random collection).

Table III. 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
8,192	None	94.37	109.37	53.37	47.37	23.85
	+BR+MF	10.37	12.37	4.37	3.37	3.85
16,384	None	94.37	100.37	57.37	50.37	24.85
	+BR+MF	5.37	6.37	2.37	3.37	1.85
32,768	None	94.37	95.37	48.37	50.37	24.85
	+BR+MF	3.37	3.37	1.37	3.37	0.85
65,536	None	90.37	92.37	55.37	61.37	25.85
	+BR+MF	2.37	2.37	1.37	3.37	0.85

Note that the used memory within the buckets for the pointers are not considered here.

Table IV. Number of Dynamic Memory Allocations of P-Burstsort and P_{+BR+MF}-Burstsort

	Collections				
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

The bucket threshold is 32,768, while each sub-bucket for P_{+BR+MF}-Burstsort contains 256 pointers.

The BIND and BIP structures require additional maintenance and dynamic memory allocation. Table IV 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 V).

We also observe that the MF technique speeds up the sorting of the URL collection in Table V due to the relatively large number of trie nodes in that 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 aid in speeding up sorting.

Growing buckets efficiently. It is important to use an efficient approach for allocating the array-based buckets. The bucket can be grown using the `realloc` function call or using the `malloc` and `memcpy` function call (referred to as Manual). To use as little memory as possible, the buckets can be grown as an exact fit, that is, allocating only as much memory as needed but at the cost of frequent calls to the memory allocators.

In Table VI, we compare allocating memory of sizes 1MB, 2MB, 4MB, and 8MB, respectively, using several small subarrays. The aim of this experiment is to observe the performance of allocating large memory for the buckets using smaller sub-buckets as it saves memory. The time grows linearly with the array size. Using the `realloc` function call is more efficient than growing these buckets with the manual approach (using malloc and memcpy). Forthcoming work Sinha and Askitis [2010] provides a more detailed analysis of these issues.

Table V. 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

Table VI. Comparing the Performance of Using Several Small Arrays to One Large Array

		Array Size (MB)			
		1	2	4	8
Realloc					
	Exact	34	71	140	270
	Exponential	14	24	45	87
Manual					
	Exact	320	650	1,320	2,630
	Exponential	16	28	52	103

The sum of the sizes of the smaller arrays total the maximum size of the array. Each small array is grown to its maximum size of 8KB before it allocates another small array. The arrays are grown using two approaches, exact-fit (E) and exponential using a growth factor 2. The maximum size of the entire array is varied from 1MB to 8MB. This experiment is performed independently and not implemented within the sorting algorithm. The time to grow the arrays to the maximum size is shown in seconds.

While an exact-fit approach saves memory, Table VI shows that it is not competitive and can be over an order-of-magnitude slower than the exponential approach. Thus, for overall performance, we grow the buckets exponentially using the realloc function call, and to save memory, we use several small buckets instead of one large bucket.

Applying the bucket redesign and moving fields to the copy-based approach. We now study the impact of using these techniques on the copy-based approach. The buckets can be much larger for C-Burstsort as the string suffixes are copied to the buckets from the start to benefit from the cache capacity and the locality of strings in the bucket. The index structure manages a lot more data than just the pointers in P-Burstsort. We varied the size of the bucket from 32KB to 512KB to observe how the memory usage was affected.

The memory used by the index structure by both C-Burstsort and C_{+BR+MF}-Burstsort on all five collections is shown in Table VII. The memory usage of C-Burstsort increases with increasing maximum bucket size, for all the four real-world collections. For instance, for the genome collection, the memory usage of C-Burstsort increases by almost 100MB from 157.75MB to 256.96MB, as the bucket size is increased from 32KB to 512KB. Similarly, the memory usage of C_{+BR+MF}-Burstsort for the genome collection increased by 40MB from 119.48MB to 159.39MB.

Table VII. Memory Use (in megabytes) Incurred by the Index Structure of C-Burstsort and (the new) C_{+BR+MF}-Burstsort for Different Bucket Thresholds and Collections

		Collections				
Bucket Size (KB)	C-Burstsort	Duplicates	No Duplicates	Genome	URL	Random
32	None	283.51	441.23	157.75	204.11	311.02
	+BR+MF	246.77	388.81	119.48	163.12	269.84
64	None	291.39	418.34	186.87	227.26	310.67
	+BR+MF	229.61	346.30	128.87	168.91	269.89
128	None	306.81	427.08	219.04	247.76	310.11
	+BR+MF	224.52	323.39	137.18	177.79	269.93
256	None	318.09	440.34	226.17	264.70	308.68
	+BR+MF	226.47	315.74	154.75	187.84	269.97
512	None	333.56	445.29	256.96	286.30	305.34
	+BR+MF	231.25	315.48	159.39	199.55	269.98

The memory usage for the random collection remains relatively stable across all bucket sizes for both algorithms due to its unique properties. The slight increase in memory usage by C_{+BR+MF}-Burstsort from 269.84MB to 269.98MB is due to the additional space required for the BIND structure that has to index a larger number of sub-buckets. These results demonstrate that the space required for the index structure for C_{+BR+MF}-Burstsort is smaller for all the collections and bucket sizes.

Significantly, the bucket redesign lowered the memory usage relative to standard C-Burstsort. The minimum reduction factor was 12.96%, the maximum 30.67%. Note that a larger bucket size helps to improve the scalability, and also allows it to benefit from the much larger L2 caches currently available. Thus, these results clearly indicate that the bucket redesign techniques can successfully be applied to the copy-based Burstsort to reduce the memory usage further.

In Figure 7, the percentage difference in memory usage by C-Burstsort is compared to that of C_{+BR+MF}-Burstsort for all the five collections. The percentage difference rises for all the real-world collections with increasing bucket size. The drop for one dataset in the genome collection is due to the inherent characteristics of that dataset. A plausible explanation is that 256KB is the "right" size for this data: less and too many buckets are burst, more and too little space in the buckets is used.

The percentage difference for the random collection, as noted earlier in the text, remains relatively stable with increasing bucket size to its unique properties.

We consider the effect of these memory-reduction techniques on the running time of C-Burstsort in Table VIII. The memory usage of C-Burstsort and C_{+BR+MF}-Burstsort is shown in parentheses. In C-Burstsort, the space is consumed by the trie structure, string suffixes in the buckets and the empty space at the end of the buckets. In C_{+BR+MF}-Burstsort, space is consumed by the trie structure, BIND structure, string suffixes in the sub-buckets and the empty space at the end of the last sub-bucket. The primary difference in memory usage between these two algorithms is in the unused space allocated in the

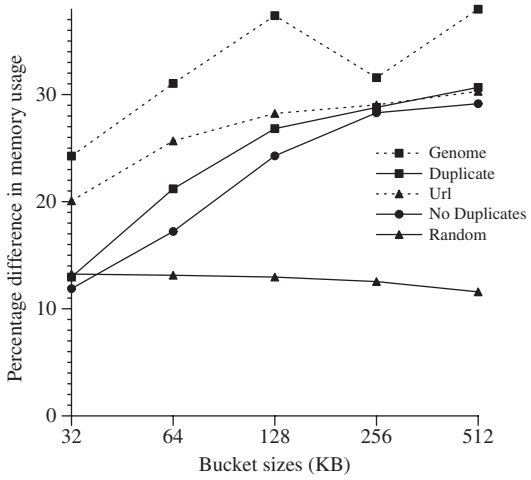


Fig. 7. Percentage difference in the memory usage as a function of the bucket size. The copy-based algorithms C-Burstsort and C_{+BR+MF}-Burstsort are compared; the new C_{+BR+MF}-Burstsort incurs less memory usage. The bucket sizes are varied from 32KB to 512KB.

Table VIII. Sorting Time (in seconds) of the Copy-Based Approach as a Function of Algorithm Modification for all Four Real-World Collections on the Pentium IV

	Collections			
Algorithm	Duplicates	No Duplicates	Genome	URL
C-Burstsort	16.89 (333.56)	21.80 (445.29)	16.91 (256.96)	13.10 (286.30)
C_{+BR+MF}-Burstsort	18.35 (231.25)	23.05 (315.48)	17.76 (159.39)	14.82 (199.55)

Memory usage of the index structure (in megabytes) is shown within parentheses. For both algorithms, the buckets grow to a maximum size of 524,288 bytes from a minimum size of 512 bytes using a growth factor of 2.

buckets. The space required by the entire index structure is less than the size of the collection itself due to the trie consuming the characters in the prefix and only the suffixes being stored in the buckets.

The memory usage of C_{+BR+MF}-Burstsort is significantly lower than that of C-Burstsort. For example, C-Burstsort requires over 100MB of index space as compared to C_{+BR+MF}-Burstsort and, importantly, this excess space is mostly due to the unused space in the buckets. Thus, the bucket redesign significantly reduces the space usage of the copy-based approach while being only slightly slower (between 5% and 13%).

Trie layout. The effect of the layout of the trie nodes is primarily felt during the insertion phase when for each string the trie nodes are traversed to place it in its respective bucket. During the subsequent traversal phase, the effect of the layout of the trie structure is relatively insignificant as the trie structure is traversed only once to sort the buckets. Consequently, we use the insertion phase to observe the effect of rearranging the trie nodes dynamically during this phase.

The time to insert all the strings during the insertion phase is shown for P-Burstsort in Table IX. The original trie layout (shown as Default) allocates

Table IX. Time to Insert Strings as a Function of the Trie Layout

P-Burstsort		Collections				
		Duplicates	No Duplicates	Genome	Random	URL
Default		11.56	11.12	16.54	4.89	11.05
AVEB-layout	1%	13.27	12.74	17.80	6.30	12.86
	5%	11.72	11.47	16.30	5.25	11.22
	10%	11.55	11.16	**15.96**	5.02	11.11
	20%	**11.31**	11.09	16.00	4.90	**10.98**
	40%	11.32	**11.03**	**15.96**	**4.77**	11.02
	60%	11.39	11.04	16.22	4.85	11.03

The insertion phase of P-Burstsort is compared with the Van Emde Boas trie layout which is rearranged after the insertion of a predefined percentage of strings. The time is measured in seconds. The machine used is Pentium IV. The threshold of the bucket is 8,192.

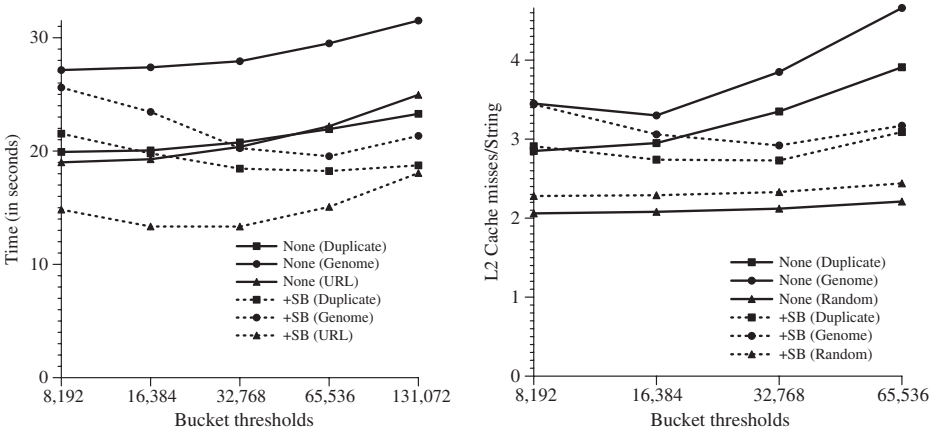
the trie nodes in occurrence order of the strings. This is compared against the dynamic rearranging of the trie nodes in a cache-friendly format; this is denoted by AVEB-layout for approximate Van Emde Boas layout. This is called *approximate* as the trie nodes, once rearranged, are then allocated in occurrence order prior to the next rearrangement.

In the AVEB-layout, the rearrangement is done after a certain fraction of the strings in the collection have been inserted. The largest dataset for each collection type has been used. We have restricted the maximum bucket size to 8,192 instead of 32,768 to allow for a larger number of trie nodes. In the experiments, the trie is rearranged for every 1%, 5%, 10%, 20%, 40%, and 60% of the strings in the collection.

The fastest times for the insertion phase for each collection are shown in italics in Table IX. The fastest times for each collection are all for the AVEB-layout, which demonstrates that the cost of rearranging and converting the trie layout could be offset by the advantage of a more cache-friendly layout. After each such rearrangement, the caches would need to be repopulated by the trie nodes as they are now in a different memory portion and thus are not cached.

The results show that using a frequency of 20% or 40% provides the fastest times for most collections. While a low frequency may show negligible impact on the performance, a high frequency can become expensive due to the frequent rearrangements and cache refilling. This is observed for the 1% case, which is the most expensive across all collections. Such a rearrangement is expected to show better relative performance on machines with a relatively high cost of a TLB miss. It would also be interesting to study the effect on a collection with a smaller alphabet size. In our experiments, we have used an alphabet size of 128; a smaller alphabet is likely to show further gains as it could benefit from cache line locality of more nodes occupying a cache line.

String buffer. The SB modifications, described in Section 3, are a successful enhancement overall. Table V 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



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

Fig. 8. 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.

half the number of cache lines, thus the strings are mostly cache-resident as they are sorted. Figure 8 and Table V confirm 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 a small reduction in the number of L2 cache misses, shown in Figure 8(b). Moreover, on machines such as PowerPC (discussed in the following text), 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 is 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. In the following text, 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 lower than that of P-Burstsort for all real-world collections (shown in Table X). Similarly, on the PowerPC, P_{+BR}-Burstsort was up to 10% faster than that of P-Burstsort (shown in Table XI). On another small cache machine (PowerPC), the SB modification reduced the running time by up to 40% (see Table XI). Thus, using a small buffer to copy string suffixes prior

Table X. Sorting Time (in seconds) for all Five Collections on the Pentium Core 2 Machine

	Collections				
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

The bucket threshold is 32,768.

Table XI. Sorting Time (in Seconds) for all Five Collections on the PowerPC 970 Machine. The Collections Contain 10 Million Strings

	Collections				
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

The bucket threshold is 32,768.

to bucket 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.

7. 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 function, but have changed the estimates we have of them. Burstsort was already known to be fast: In this article, its demands on main memory have been significantly reduced, with only a slight impact on running time.

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 [Sinha and Zobel 2005] 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? A likely avenue for further optimizations is to compress the trie structure using path compression and level compression [Nash and Gregg 2008; Nilsson and Tikkanen 1998]. We are incorporating some of the newer capabilities in modern processors such as out-of-order execution [Karkkainen and Rantala 2008] and SIMD processing [Zhou and Ross 2002].

ACKNOWLEDGMENTS

We thank the reviewers for their feedback.

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 J. Exp. Algorithmics 3*, 7.

ARGE, L., FERRAGINA, P., GROSSI, R., AND VITTER, J. S. 1997. On sorting strings in external memory. In *Proceedings of the ACM Symposium on Theory of Computation*. ACM, New York, 540–548.

ASKITIS, N. AND SINHA, R. 2007. Hat-trie: A cache-conscious trie-based data structure for strings. In *Proceedings of the 13th Australasian Computer Science Conference (ACSC'07)*. ACS, Sydney, Australia, 97–105.

ASKITIS, N. AND ZOBEL, J. 2005. Cache-conscious collision resolution in string hash tables. In *Proceedings of the SPIRE String Processing and Information Retrieval Symposium*. Springer, Berlin, 91–102.

BENDER, M. A., COLTON, M. F., AND KUSZMAUL, B. C. 2006. Cache-oblivious string b-trees. In *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'06)*. ACM, New York, 233–242.

BENSON, D. A., KARSCH-MIZRACHI, I., LIPMAN, D. J., OSTELL, J., AND WHEELER, D. L. 2003. Genbank. *Nucleic Acids Res. 31*, 1, 23–27.

BENTLEY, J. AND SEDGEWICK, R. 1997. Fast algorithms for sorting and searching strings. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, 360–369.

BENTLEY, J. L. AND MCILROY, M. D. 1993. Engineering a sort function. *Softw. Pract. Exp. 23*, 11, 1249–1265.

BRODAL, G. S., FAGERBERG, R., AND JACOB, R. 2004. Cache oblivious search trees via binary trees of small height. http://www.cs.technion.ac.il/_itai/Courses/Cache/Presentations/Cache%20Oblivious%20Search%20Trees.ppt.

BRODAL, G. S., FAGERBERG, R., AND VINTHER, K. 2007. Engineering a cache-oblivious sorting algorithm. *ACM J. Exp. Algorithmics 12*, 2.2, 23.

DEMAINE, E. D. 2002. Cache-oblivious algorithms and data structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets*. Springer, Berlin.

FRIGO, M., LEISERSON, C. E., PROKOP, H., AND RAMACHANDRAN, S. 1999. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS'99)*. IEEE, Los Alamitos, CA, 285–298.

GRAEFE, G. 2006. Implementing sorting in database systems. *Comput. Surv. 38*, 3, 1–37.

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

HE, B. AND LUO, Q. 2008. Cache-oblivious databases: Limitations and opportunities. *ACM Trans. Datab. Syst. 33*, 2.

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

KARKKAINEN, J. AND RANTALA, T. 2008. Engineering radix sort for strings. In *Proceedings of the String Processing and Information Retrieval Symposium*. Springer-Verlag, Berlin, 3–14.

KNUTH, D. E. 1998. *The Art of Computer Programming: Sorting and Searching* 2nd Ed. vol. 3. Addison-Wesley, Reading, MA.

LEVITIN, A. 2007. *Introduction to the Design and Analysis of Algorithms* 2nd Ed. Pearson, Upper Saddle River, NJ.

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

MOFFAT, A., EDDY, G., AND PETERSSON, O. 1996. Splaysort: Fast, versatile, practical. *Softw. Pract. Exp. 26*, 7, 781–797.

NASH, N. AND GREGG, D. 2008. Comparing integer data structures for 32 and 64 bit keys. In *Proceedings of the 7th International Workshop on Experimental Algorithms (WEA'08)*. Springer, Berlin, 28–42.

NG, W. AND KAKEHI, K. 2007. Cache efficient radix sort for string sorting. *IEICE Trans. Fundam. Electron. Comm. Comput. Sci. E90-A*, 2, 457–466.

NILSSON, S. AND TIKKANEN, M. 1998. Implementing a dynamic compressed trie. In *Proceedings of the Workshop on Algorithm Engineering*. SIAM, Philadelphia, 25–36.

OSTLIN, A. AND PAGH, R. 2003. Cache-oblivious algorithms. http://www.itu.dk/_annao/ADT03/ lecture10.pdf.

SEDGEWICK, R. 1998. *Algorithms in C*, 3rd ed. Addison-Wesley, Boston, MA.

SEWARD, J. 2001. Valgrind—memory and cache profiler. http://developer.kde.org/_sewardj/docs-1.9.5/cgtechdocs.html.

SINHA, R. AND ASKITIS, N. 2010. Copy-based approach: A deeper understanding.

SINHA, R., RING, D., AND ZOBEL, J. 2006. Cache-efficient string sorting using copying. *ACM J. Exp. Algorithmics 11*, 1.2.

SINHA, R. AND ZOBEL, J. 2004. Cache-conscious sorting of large sets of strings with dynamic tries. *ACM J. Exp. Algorithmics 9*, 1.5.

SINHA, R. AND ZOBEL, J. 2005. Using random sampling to build approximate tries for efficient string sorting. *ACM J. Exp. Algorithmics 10*.

ZHOU, J. AND ROSS, K. A. 2002. Implementing database operations using simd instructions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*. ACM, New York, 145–156.

Received February 2009; revised November 2009; accepted November 2009