Cache-Efficient String Sorting Using Copying

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

David Ring
Palo Alto, dbring@pacbell.net

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

Abstract. Burstsort is a cache-oriented sorting technique that uses a dynamic trie to efficiently divide large sets of string keys into related subsets small enough to sort in cache. In our original burstsort, string keys sharing a common prefix were managed via a bucket of pointers represented as a list or array; this approach was found to be up to twice as fast as the previous best string sorts, mostly because of a sharp reduction in out-of-cache references. In this paper we introduce C-burstsort, which copies the unexamined tail of each key to the bucket and discards the original key to improve data locality. On both Intel and PowerPC architectures, and on a wide range of string types, we show that sorting is typically twice as fast as our original burstsort, and four to five times faster than multikey quicksort and previous radixsorts. A variant that copies both suffixes and record pointers to buckets, CP-burstsort, uses more memory but provides stable sorting. In current computers, where performance is limited by memory access latencies, these new algorithms can dramatically reduce the time needed for internal sorting of large numbers of strings.

1 Introduction

Sorting is a core problem in computer science that has been extensively researched over the last five decades. It underlies a vast range of computational activities and sorting speed remains a bottleneck in many applications involving large volumes of data. The simplest sorts operate on fixed size numerical values. String sorts are complicated by the possibility that keys may be long (making them expensive to move or compare) or variable in length (making them harder to rearrange). Finally, sorts that operate on multi-field records raise the issue of stability, that is, conservation of previous order when keys are equal in the field currently being sorted. The favored sorting algorithms are those that use minimal memory, remain fast at large collection sizes, and efficiently handle a wide range of data properties and arrangements without pathological behavior.

Several algorithms for fast sorting of strings in internal memory have been described in recent years, including improvements to the standard quicksort (Bentley and McIlroy, 1993), multikey quicksort (Bentley and Sedgewick, 1997), and variants of radix sort (Andersson and Nilsson, 1998). The fastest such algorithm is our *burstsort* (Sinha and Zobel, 2004a), in which strings are processed depth-first rather than breadth-first and a complete trie of common prefixes is used to distribute strings amongst buckets. In all of these methods, the strings themselves are left in place, and only pointers are moved into sort order. This pointerized approach to string sorting has become a standard solution to the length and length variability of strings, as expressed by Andersson and Nilsson (1998):

> "to implement a string sorting algorithm efficiently we should not move the strings themselves but only pointers to them. In this way each string movement is guaranteed to take constant time."

In this argument, moving the variable-length strings in the course of sorting is considered inefficient. Sorting uses an array of pointers to the strings, and, at the completion of sorting, only the pointers need to be in sort order.

However, evolutionary developments in computer architecture mean that the efficiency of sorting strings via pointers needs to be reexamined. Use of pointers is efficient if they are smaller than strings and if the costs of accessing and copying strings and pointers are uniform, but the speed of memory access has fallen behind processor speeds and most computers now have significant memory latency. Each string access can potentially result in a cache miss; that is, if a collection is too large to fit in cache memory, the cost of out-of-cache references may dominate other factors, making it worthwhile to move even long strings in order to increase their spatial locality. It is the cost of accessing strings via pointers that gives burstsort (which uses no fewer instructions than other methods) a speed advantage.

There has been much recent work on cache-efficient algorithms (LaMarca and Ladner, 1999, Jimenez-Gonzalez et al., 2003, Rahman and Raman, 2000, 2001, Wickremesinghe et al., 2002, Xiao et al., 2000), but little that applies to sorting of strings in internal memory. An exception is burstsort (Sinha and Zobel, 2004a), which uses a dynamically allocated trie structure to divide large sets of strings into related subsets small enough to sort in cache. Like radix sorts and multikey quicksort, burstsort avoids whole string comparisons and processes keys one byte at a time. However, burstsort works on successive bytes of a single key until it is assigned to a bucket, while the other sorts must inspect the current byte of every key in a sublist before moving to the next byte. This difference greatly reduces the number of cache misses, since burstsort can consume multiple bytes of a key in one memory access, while the older sorts may require an access for each byte. Furthermore, burstsort ensures that final sorting will be fast by expanding buckets into new subtries before they exceed the number of keys that can be sorted within cache.

In tests using a variety of large string collections, burstsort was twice as fast as the best previous string sorts, due largely to a lower rate of cache misses. On average, burstsort generated between two and three cache misses per key, one on key insertion and another during bucket sorting; the remaining fractional miss was associated with bursting, which requires accessing additional bytes of the keys, and varied with patterns in the data.

There are two potential approaches to reducing cache misses due to string accesses. The first is to reduce the number of accesses to strings, which is the rationale for burstsort. The second approach is to store strings with similar prefixes together, then to sort these suffixes. By gathering similar strings into small sets, it is feasible to load the whole of each set into cache and thus sort efficiently. That is, if strings with similar prefixes could be kept together, then accesses to them will be cache efficient. Such an approach

was found to work for integers in our adaptation of burstsort, where the variable-length suffix of an integer that had not been consumed by the trie is copied into the bucket (Sinha, 2004). In this paper we explore the application of this approach to strings.

The novel strategy we propose is to copy the strings into the buckets. That is, instead of inserting pointers to the keys into the burst trie buckets, we insert the unexamined suffixes of the keys. This eliminates any need to re-access the original string on bucket sorting, and by copying only the tails of the keys into a contiguous array, we make the most efficient use of limited cache space. Finally, with all prefix bytes reflected in the trie structure, and all suffix bytes transferred to buckets, we can free the original keys to save memory. Balancing these advantages, we can expect higher instruction counts reflecting the complications of moving variable length suffixes, and increased copying costs when the suffixes are longer than pointers.

There are several challenges, principally concerning memory usage and instruction counts. How efficiently can strings be copied to the buckets? Are the advantages of spatial locality offset by additional instructions? Is the memory usage excessive? How will the bucket be managed, as the strings are of variable length? What is the node structure? Are there any auxiliary structures? Are there any characteristics of the collection, such as the average string length, that alter best-case parameters?

We show that all of these challenges can be met. In our new string sorting algorithms based on copying, memory usage is reasonable—the additional space for the strings is partly offset by elimination of pointers—and cache efficiency is high. Using a range of data sets of up to tens of millions of strings, we demonstrate that copy-based *C-burstsort* is around twice as fast as previous pointer-based burstsorts and four to five times faster than multikey quicksort or forward radixsort, while generating less than two cache misses per key.

While C-burstsort provides fast sorting of string keys, it is not suitable for processing records, since it is not stable and discards the original keys. We therefore describe a record-oriented variant, *CP-burstsort*, which transfers both string tails and record pointers to buckets. CP-burstsort uses substantially more memory than C-burstsort or our previous pointer-based burstsorts, but provides stable sorting at more than twice the speed of a stable radix sort or 70%–90% of the speed of C-burstsort.

2 Sorting strings in internal memory

Many of the earliest sorting algorithms—such as shellsort and quicksort—rearrange data in-place with small constant space overhead. Sorting with these algorithms involves swapping of fixed-size items such as integers. Variable-length strings cannot be readily sorted in this way, so the commonest approach to sorting of strings is to store them in a contiguous array of characters and index them with an array of fixed-size pointers. Sorting can then proceed by rearranging the pointers and leaving the strings themselves unaltered. To our knowledge, all string sorting algorithms described so far use such an array of pointers, although it is not strictly necessary in algorithms such as mergesort.

Historically, as illustrated in the quotation given earlier, such use of pointers was potentially advantageous, as copying of strings could be expensive. However, pointer-based methods are not necessarily cache-efficient, as each access can be a cache miss. Here we briefly review recent developments in caching and sorting; a more detailed overview of sorting is given by Sinha and Zobel (2004a).

Caching

During the last couple of decades there have been gradual but significant changes in computer architecture. First, the speed of processors has been increasing by about 50% per year, closely following Moore's law. This is due to the fact that the density of transistors is increasing by about 35% per year and the size of die has been increasing by about 10% to 20% per year (Hennessy and Patterson, 2002). Second, the density of DRAM is increasing by 40% to 60% per year, and as a result the size of main memory has been expanding rapidly. However, memory access speeds are increasing only slowly, by only 7% per year (Hennessy and Patterson, 2002). This latency gap is widening rapidly, and this trend appears likely to continue.

The latency gap has affected the performance of computer programs, especially those that deal with large amounts of data (LaMarca and Ladner, 1999, Hennessy and Patterson, 2002). However, many programs tend to reuse the data or instructions that have been recently accessed. As programs do not access all code or data uniformly, having those frequently accessed items closer to the processor is an advantage. This *principle of locality* has led to solutions such as the use of small memories, or *caches*, between the processor and main memory. That caches have become a popular solution can be gauged from the fact that almost all processors come with at least one cache and most have two. A common approach is to have a large off-chip L2 cache and a small on-chip L1 cache. Current processors have caches ranging from 64 or 256 kilobytes on a Intel Celeron to 8 megabytes on a Sun SPARC.

Another important cache in modern processors is the *translation lookaside buffer*, or TLB, which is used to translate between virtual and physical addresses. Recently accessed entries in the page table are cached in the TLB to save expensive accesses to main memory. The TLB performance of an algorithm can be enhanced by improving the page locality of both the data structure as well the access to the data. To be efficient, an algorithm needs to maximize the impact of both types of cache (Rahman and Raman, 2001).

Quicksorts

Quicksort was introduced by Hoare (1961, 1962). The collection to be sorted is divided recursively into two partitions based on a pivot. Each partition is then recursively divided, until the partitions are small enough to be efficiently sorted by simpler methods such as insertion sort. In later passes, strings having similar prefixes are in the same partition, requiring unnecessary comparison of prefixes that are by construction known to be identical. The first pass is cache-efficient as it involves sequential passes over the array of pointers as well as the collection of strings, but successive passes may not be cache-efficient, as the string accesses are effectively random. The number of passes in quicksort is proportional to $\log N$ and thus the number of cache misses per string is expected to increase logarithmically.

Several optimizations, such as three-way partitioning and adaptive sampling, have been recently introduced in the Bentley and McIlroy (1993) variant of quicksort. This variant has been used since the early 1990s in most libraries.

The concept of multiple word sorting was described in the early 1960s; a practical algorithm, multikey quicksort, was introduced by (Bentley and Sedgewick, 1997). This algorithm can be regarded as a hybrid of radixsort and three-way quicksort. The keys are divided into three partitions ($p_<$, $p_=$, and $p_>$) based upon the dth character of the strings. The next character (at position $d+1$) is compared for strings in the $p_=$ partition. The strings in the other two partitions are divided according to the dth character. The character-wise approach reduces the comparison costs, as the redundant comparison of prefixes is reduced, but the disadvantages inherent in quicksort are still present due to poor spatial locality of the string accesses. The cache efficiency is similar to that of quicksort and the number of cache misses per key is expected to

increase logarithmically. We have used the implementation by Bentley and Sedgewick (1997), designated as *multikey quicksort*.

Radixsorts

For fixed-length keys, radixsorts can proceed from the least significant (LSD) or most significant (MSD) end of the key. However, LSD radixsorts are impractical for variable-length strings.

The in-principle costs of MSD radixsorts approaches the theoretical minimum, reading only the distinguishing prefix and only reading each character in the prefix once. In the worst case, however, each string may need to be reaccessed for each character, so the number of cache misses is then proportional to the size of the distinguishing prefix. Moreover, it is relatively easy to produce bad cases such as a collection that is larger than cache and containing identical strings. The number of passes is directly proportional to the length of the distinguishing prefix of each string.

The number of cache misses can be reduced by decreasing the number of passes by increasing the alphabet size, as in adaptive radixsort (Nilsson, 1996), where the size of the alphabet is a function of the number of elements that remains to be sorted in a bucket. In this approach, a linked list node is created to address each string; these nodes are shifted between buckets as the sorting progresses. With both an array of pointers to strings and a set of linked list nodes, there are considerable space overheads. The alphabet size varies from 8-bit to 16-bit. Several optimizations have been used that reduces the number of node elements that need to be looked at for the larger alphabets.

Another variant is the radixsort described by McIlroy et al. (1993), an in-place array-based method. We have used the original implementation of McIlroy et al. (1993), designated as *MBM radixsort*.

Burstsort

Burstsort is a cache-friendly algorithm that sorts large collections of string keys up to twice as fast as the previous best algorithms (Sinha and Zobel, 2004a). It is based on the burst trie (Heinz et al., 2002), a variant of trie in which sufficiently small subtrees are represented as buckets. As subtrees tend to be sparse, the use of buckets makes a burst trie much more space-efficient than a conventional trie, and leads to more efficient use of cache.

The principle of burstsort is that each string in the input is inserted into a burst trie, creating a sorted sequence of buckets that are internally unsorted. In the basic implementation, each bucket is a set of pointers to strings; within a bucket, each string has the same prefix, representing the path through the trie that was used to reach the bucket. The most efficient representation of buckets is as an array that is resized as necessary. Traversal of the burst trie, sorting each bucket in turn, yields sorted output. Some strings are entirely consumed in the trie; these are stored in special-purpose end-of-string buckets, in which all strings are known (by construction) to be identical.

During the insertion phase, then, each string is consumed byte by byte until it can be assigned to a bucket. Buckets that are larger than the fixed threshold are burst into new subtries, ensuring that each bucket remains small enough to be sorted in fast cache memory. Because burstsort processes one string fully before proceeding to the next, it exhibits better locality than approaches such as multikey quicksort or forward radixsort. In these methods, the first byte of every string is processed before the second of any string, so each string must be re-accessed for every distinguishing character. When the volume of keys being sorted is much larger than cache, burstsort generates far fewer out-of-cache references than the other algorithms (Sinha and Zobel, 2004a).

Burstsort gains speed by accessing keys serially and assigning them to buckets small enough to sort within cache. Our exploration of burstsort variants (Sinha and Zobel, 2004a) showed that the speed can be increased by reducing the number of times that keys are accessed from their original memory locations, by reducing the number of buckets that are formed, by balancing trie size against typical bucket size, and by reducing the number of times that large buckets are grown or burst. The best results are achieved by forming the fewest possible buckets and filling them as full as possible, and by growing or bursting buckets while they are still small.

Addressing these conflicting issues, we developed burstsort variants that avoid most of the bursts, by using the strategy of prebuilding trie structures based on small random samples of keys (Sinha and Zobel, 2004b). Sampling burstsorts are 10%–25% faster than the original, and generate up to 37% fewer cache misses.

However, the costs involved in growing and bursting buckets are a bottleneck. The string burstsorts described to date leave the keys in their original locations and place only pointers in buckets, leading to two problems. First, average key length can vary significantly by bucket. A burst threshold that is low enough to ensure that buckets referencing long keys can be sorted within cache may result in wasteful bursting of buckets that hold shorter keys. Second, pointers are localized in buckets, but bucket sorting requires repeated access to the keys, which remain dispersed at their original locations in memory. With low burst thresholds, there will be room to bring all the keys referenced by a bucket into cache, but this is wasteful, as only the unexamined terminal bytes of each key are actually needed. Furthermore, each key is accessed twice: once to assign it to a bucket, and again to bring it into cache memory when the bucket is sorted.

In adapting burstsort to integers (Sinha, 2004) we discovered that it was efficient to store only the distinguishing tail of each integer in the buckets; the common prefix was represented by the path through the burst trie. Within a bucket, each tail was of the same length, but lengths varied from bucket to bucket. This suggested that copying strings rather than referencing them could be advantageous, as we now discuss.

3 Copy-based burstsort

Burstsort would be more efficient if the spatial locality of the strings could be improved, so that strings are more likely to be cached. Improving locality requires moving strings so that strings sharing a prefix are near each other, and, if feasible, would reduce cache misses during both bursting and bucket sorting. Our contribution in this paper is copy-based burstsort, in which co-location of strings is successfully used to reduce sorting costs.

The novel approach that we propose is to copy the suffix of each string into its bucket, eliminating the need to refer to the strings at their arbitrary memory locations. In copy-based burstsort, or *C-burstsort*, each key is accessed just once in its original memory location. When it has been assigned to the appropriate bucket, its unexamined tail bytes are copied to the bucket, and the original key—including the prefix, which is represented by the path through the trie—is deleted. Buckets grow or burst based on the actual number of tail-of-string bytes held in them. When all keys have been inserted, each group of suffixes sharing a common prefix has been localized in a bucket and packed into a minimal contiguous block of memory.

C-burstsort has three main advantages. First, the full capacity of the cache can be utilized, whereas, in pointer-based burstsort, the number of strings in the buckets is typically limited to the number of blocks in cache. This allows more strings to be accommodated in each bucket, reducing the height of the trie. Second, bursting a bucket does not require fetching a string from memory, potentially causing a cache

miss with each access; only a scan traversal of the bucket is required to distribute the strings into the child buckets. Third, sorting the strings in the final traversal phase is more efficient; only a scan traversal of the bucket is required and the strings do not have to be fetched from their original memory locations, a process that would potentially cause a cache miss for each string.

A disadvantage of C-burstsort is that it is not suitable for sorting of sets of records, since it sorts unstably and discards the original keys. We therefore developed a record-based variant, *CP-burstsort*, which transfers both string tails and record pointers to buckets, and allows stable sorting of records. This variant requires more memory than C-burstsort or pointer-based burstsorts, but provides the fastest stable sorting.

The absolute and relative performance of these new algorithms will vary with the cache architecture and latencies of particular computers, but in most cases we expect them to provide the fastest options available for internal string-based sorting. The performance in a range of architectures is explored in our experiments. First, we describe the new copy-based variants of burstsort.

C-burstsort

In C-burstsort, the strings are initially loaded into a *segmented buffer*, or array of buffers. Segmentation reduces peak memory allocation by allowing groups of the original keys to be freed as soon as their tails are inserted into buckets. Making an arbitrary choice, which has only minimal impact on efficiency, we use fifty buffers. Statistics are collected during loading, including the number of keys, number of bytes, length of the longest key, and highest and lowest character value encountered. (The length of the largest string is used during insertion of strings into the bucket to avoid overflow without double-checking the string. The character range is used during the traversal phase; only the buckets within that range are checked.)

The strings are inserted one by one into the trie structure, and their uninspected suffixes are copied to the buckets, where the suffixes are stored contiguously. Each bucket is allowed to grow until it reaches the threshold limit for bursting. A buffer segment is freed once all strings in that segment have been inserted into the trie. For the empty-string buckets, only the counters need be incremented. During bursting, the bucket is scan-traversed and the strings are distributed and copied into new buckets. Once all the strings have been inserted, the trie nodes are traversed and pointers to the string suffixes in the buckets are created. The string suffixes in the bucket are then sorted using a suitable helper sort, such as multikey quicksort, and the strings can be output in sort order.

In C-burstsort, each bucket is represented as an array of bytes; in prior versions of burstsort, buckets were arrays of pointers. When a bucket is initially created, it is small, to avoid waste of space—an arbitrary bucket may never hold more than a few strings. Each time a bucket overflows, it is enlarged by reallocation. A limit threshold is used to put an absolute cap on bucket size; when this threshold is reached, the bucket is burst. Choice of bucket expansion strategy and of threshold is discussed later.

In detail, the phases of C-burstsort are as follows.

1. Initialization: Read all strings into the segmented buffer. Statistics are recorded as described above.
2. Insertion: The keys in the segmented buffer are processed sequentially, and each segment is freed as soon as its keys have been inserted. As each key is read, its successive bytes are used to traverse the trie structure. Starting at root, the first byte value indexes the next step in the path for strings beginning with that character. If this step is a trie node, the pointer is followed to the next node and the next character is examined, and so on until the key is exhausted or an unbranched node is found.

If a key runs out, the count for that key is incremented. There are no buckets for exhausted keys in C-burstsort, and, after insertion, the keys exist only as counts and the path of prefix character leading to a given node. Otherwise, a bucket is reached and its count is incremented. If the incremented count is one, a new bucket of size, say 2048 bytes, is allocated. The key's remaining bytes, including the terminating null, are copied to the end of the bucket. If the bucket is full it needs to be grown or burst. If the bucket's contents are projected to exceed the cache size C, or if the free burst count F (described later) is greater than 0, the bucket is burst. Otherwise it is grown.

To burst a bucket, a new trie node is allocated. The key tails in the full bucket are inserted into the new subtrie, and the full bucket is then freed. In the case that all the keys in a burst end up in a single new bucket, the burst routine automatically bursts that bucket, until a burst forms more than one new bucket or exhausts at least one key. The free burst counter F is decremented only if a burst results in a net increase in buckets. (That is, bursts that deepen the trie without branching are always free.)

To grow a bucket, the contents of the full bucket are copied to a newly allocated bucket twice as large and the old bucket is freed.

3. Traversal: Starting at the root node, the slots between the lowest and highest character values observed during loading phase are recursively scanned. When a bucket is found, it is tested to determine if it has enough free space to allocate (in cache) a tail pointer to each tail. If not—a rare case—it is burst and the new subtrie is recursively traversed. The bucket is then scanned, so that each tail pointer indicates the start of a tail. The tail pointers are then sorted with the helper sort.

   At this point, keys can be recovered in sort order directly from the trie. Prefixes are built by starting at root with a null string and appending each character traversed. For strings that are entirely represented within the trie, counts are used to indicate how often each string should be output. For other strings, the tail is appended to the prefix and the whole string is output.

A snapshot of C-burstsort after the insertion of all the strings in a collection is shown in Figure 1. Strings in the trie structure are "bat", "barn", "bark", "by", "by", "by", "by", "byte", "bytes", "wane", and "way". The string terminator symbol in all figures is denoted by #.

CP-burstsort

CP-burstsort provides stable sorting of records or other data when it is not desirable to delete the original keys. For each key, both the unexamined suffix and a pointer to the original key or record are copied to the appropriate bucket. Output is a sorted sequence of pointers to the records.

In the first phase, the data is copied into a buffer and the statistics (length of longest string and character range) of the collection are recorded. The strings are then inserted one by one into the trie structure. As each string is inserted, a pointer to the string's location in the input buffer is placed in the bucket, immediately prior to a copy of the uninspected string suffix. For the empty-string bucket, only the record pointers need be kept in the bucket. During bursting, the bucket is scan traversed and the string suffixes and record pointers are distributed and copied onto new buckets. Once all the strings have been inserted into the trie, the trie nodes are traversed and pointers to the string suffixes in the buckets are created. The string suffixes in the bucket are then sorted using a stable version of multikey quicksort. The records can then be output in sort order, by a scan traversal of the array of pointers to suffixes in the bucket. In detail, the phases of CP-burstsort are as for C-burstsort, but with the following differences.

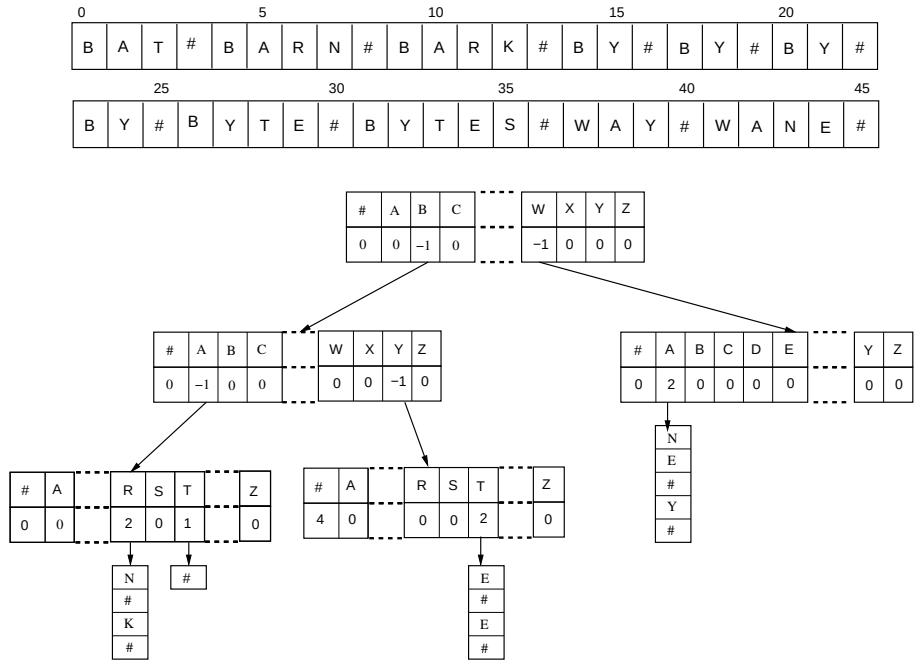

Figure 1: *C-burstsort with five trie nodes, four buckets and eleven strings. The threshold for bursting a bucket is five bytes.*

1. Initialization: Read all strings (and associated records) into a single buffer. Statistics are recorded as described above.
2. Insertion: Insertion is similar to that in C-burstsort. The only differences are that the original keys are not freed after insertion, and, instead of holding only counts, the record pointers of exhausted strings are kept in buckets; these buckets grow by doubling but do not burst.
3. Traversal: Traversal is similar to that in C-burstsort. However, alternation of record pointers and string tails must be taken into account while building the tail pointer array. Special helper sorts are needed to stably sort CP-burstsort buckets. When equal keys are found, stability is achieved by arranging their tail pointers in the order of the tails (which were inserted in original order, and moved but never re-ordered during bucket growth and bursting). Once a bucket has been sorted, the tails are no longer needed. Record pointers are copied into the positions of the sorted tail pointers, and the bucket is freed.

A snapshot of CP-burstsort after the insertion of all the strings in a collection is shown in Figure 2. Strings in the trie structure are, as above, “bat”, “barn”, “bark”, “by”, “by”, “by”, “by”, “byte”, “bytes”, “wane”, and “way”. The record pointer in Figure 2 is denoted by $\rightarrow$ and is assumed, in the figure, to occupy a single byte.

In comparing the speed of the variants of burstsort, some assumptions need to be made. For example, in some applications involving large sets of strings, the array of pointers to strings will already be present; in other applications, creating the array of pointers is an overhead. In our experiments, in order to err on the side of underestimating the improvement yielded by C- and CP-burstsort, we assume the former—the cost of creating the array of pointers is not counted.

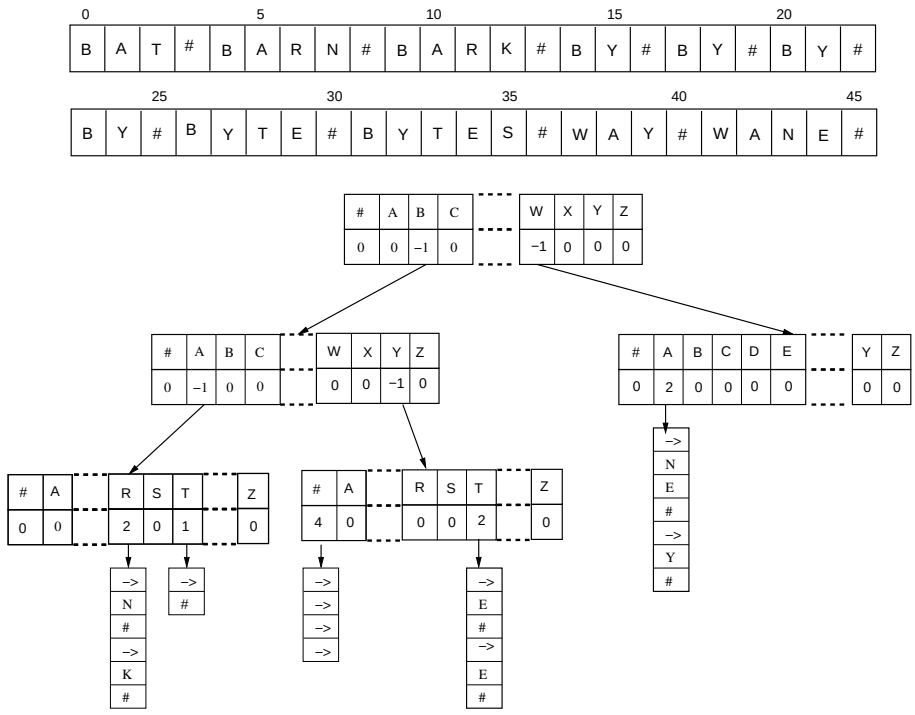

Figure 2: *CP-burstsort with five trie nodes, five buckets and eleven strings. The threshold for bursting a bucket is seven bytes.*

4 Implementation Choices

The behavior of C-burstsort may be modified or adapted to a particular system by changing the growth strategy, cache size (C), initial bucket size (S_0), free burst count (F), and choice of helper sort. We now explore implementation choices, after explaining the structures used for these copy-based burstsort variants.

Structures used by C-burstsort and CP-burstsort

The PPL elements of the tries used in C- and CP-burstsort are as shown in Figure 4. The trie node is an array of these PPL elements, which have four fields: (1) an integer field to keep track of the number of strings in a bucket; (2) a pointer to start of bucket; (3) a pointer to first unused byte of bucket; (4) a pointer to the limit at which the bucket will test as full. As a suffix is being copied into the bucket, if the position exceeds the limit, the bucket is either grown or burst.

Apart from the trie node structure, a simple array is used for the *bucket*, which stores the string suffixes and record pointers. The buckets hold string tails, and (in CP-burstsort) pointers to the original records. In C-burstsort, string tails are added to the bucket contiguously, with the first byte of each tail following the terminating null of the preceding tail. In CP-burstsort, a record pointer for each key is inserted first, followed immediately by the corresponding tail. In both methods, tail pointers are created at bucket-sorting time, in a separate array. Although the tail pointer array is distinct from the bucket, both must ideally fit into cache during bucket-sorting phase.

During the traversal phase, an array of pointers to suffixes in the bucket is created and used to sort the bucket. In C-burstsort, the strings can be retrieved by traversing this array of pointers and concatenating

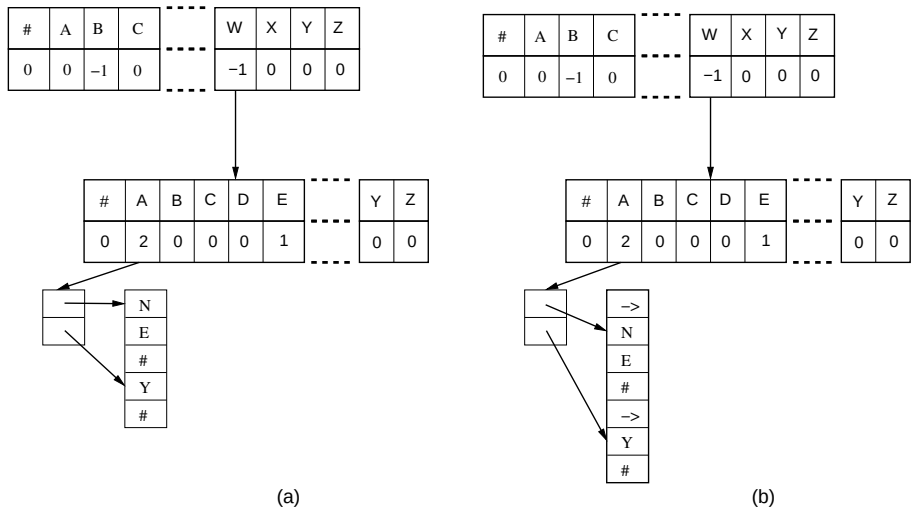

Figure 3: *Traversal phase. Left: C-burstsort. Right: CP-burstsort.*

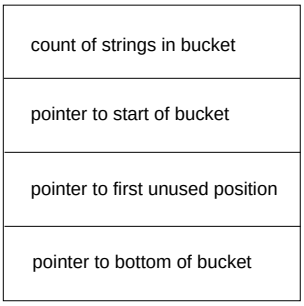

Figure 4: PPL *structure used in C-burstsort and CP-burstsort.*

the prefixes to the corresponding suffixes. In CP-burstsort, the record pointers are mapped onto the sorted tail pointers (see below) and can then be used to retrieve records in order. A snapshot of the traversal phase is shown in Figure 3. The record pointer in Figure 3 is denoted by → and is assumed to occupy a single byte.

On both Intel and PowerPC architectures, a node element occupies 16 bytes, structured as a four byte integer and three character pointers. Holding the PPL structure to 16 bytes allows four node elements to fit on a 64-byte cache line. Depending on alphabet size and character distribution, string data sets can generate several trie nodes and the effect of cache misses related to the trie structure would start to become significant. An alternative structure is to have a more compact node element. In this structure the trie node is an array of elements having two fields: a counter and a pointer to an index to bucket structure. The counter keeps track of the number of strings in the bucket. The BIS bucket index structure consists of five fields: (1) an integer field that records the size of bucket; (2) a pointer to start of bucket; (3) a pointer to first unused byte of bucket; (4) a pointer to array of pointers to string in buckets during the traversal phase; and (5) a pointer to the limit at which the bucket will test as full. As a suffix is being copied into the bucket, if the position exceeds the limit, the bucket is either grown or burst.

Including a separate BIS structure is not as cache-efficient as the more compact PPL structure, as an access to the bucket requires an access to the node element and another to the BIS structure. But a more compact trie node could lead to more page locality and thus fewer TLB misses for collections of small alphabets, and in other cases where the trie structure may become large. For some of our experiments on other architectures, this alternative structure has been used; refinement of the BIS structure for cases where the PPL structure is ineffective is a topic for future research.

Free bursts and sampling

Bursting of large buckets is an expensive operation, and significant time can be saved by correctly predicting buckets that will burst and bursting them while they are still small. Predictive bursting based on random sampling has been described for pointer-based burstsorts (Sinha and Zobel, 2004b). A simpler approach that is nearly as effective is to allow a fixed number F of free bursts, with one free burst consumed each time a burst increases the net total of buckets. This amounts to a non-random sampling scheme in which the first buckets that fill at the first bucket size (in whatever order the data are loaded) are pre-emptively burst without further growth.

In our experiments, we report times for CF-burstsort and CPF-burstsort, which are C- and CP-burstsort with free bursts, for $F = 100$. We found that 60–100 free bursts can slightly improve sorting speed for data in random order, and does not degrade performance on presorted or reversed data, while showing very little effect on memory requirements.

Fullness and burst testing

Each time a key is processed and its suffix inserted into an existing bucket, the bucket must be tested to determine if it is full. Since this test is so frequent, it needs to be as simple as possible. Whenever a bucket is created or grown, its limit pointer is set to its base pointer plus its size minus the length of the longest expected key. The fullness test simply checks whether the next free byte in the bucket is beyond this limit. If so, the bucket size is calculated. If the size equals cache size or if the free burst counter F is greater than zero, the bucket is burst. Otherwise, it is copied to a newly allocated bucket twice the old size, and the old bucket is freed.

These tests guarantee that buckets do not overflow, but in the case of a cache-sized bucket, a simple burst test does not guarantee that there is room in cache for both the string tails in the bucket and a sorting pointer to each of them. During the traversal phase, the bucket may be burst again if the size of the bucket plus the space needed for the tail pointers is greater than the cache size.

Bucket growth strategies

A simple way to grow buckets is to use a *static* growth factor, which is fixed for all buckets and all sizes. For example, if the growth factor is two, then whenever a bucket needs to be grown its size is doubled. An alternative is to use an *adaptive* growth factor. Different growth factors could be used for small and large buckets. A large growth factor could be used for smaller buckets and a smaller growth factor could be used for the larger buckets.

A more principled approach is to use a predictive growth factor. This can be achieved by choosing the growth factor as the ratio of number of keys in the collection to the number of keys observed. Thus, a bucket that reaches the limit early on will have a larger growth factor than a bucket that is growing more

slowly. This makes the growth factor responsive to the collection characteristics. Use of such schemes, however, creates the risk of bad performance or excessive memory usage in pathological cases, such as collections that are already sorted.

For good performance, a scheme must ensure that buckets remain small enough to be sorted within cache; minimize growth and bursting of large buckets; and keep memory overhead within reasonable bounds. The results reported in this paper use a simple scheme with a static growth factor of two.

For the copy-based schemes, we have used a combination of `malloc` and `memcpy` for allocating and growing the buckets. An alternative would be to use `realloc`, an approach that has the advantage that, if an array can be grown in place, no copying is needed; however, the housekeeping associated with `realloc` can be costly. Another alternative would be to preallocate large segmented buffers and write a dedicated memory manager, avoiding costly invocations to the `free` system call. We plan to explore these alternatives in future work.

Buckets are grown until they reach the threshold limit, which depends on the cache size and the TLB size. The limit was set to 512 KByte for a 512 KByte cache on the Pentium IV, and 512 KByte for a 1 MByte cache on a Pentium III. Where TLB misses are expensive, the size of the bucket may need to consider the size of the TLB.

Other issues

We have used segmented buffers in C-burstsort to manage the input strings. To limit memory requirements, C-burstsort reads the original keys into a segmented buffer and frees each segment as soon as its keys have been added to the burst trie. A larger number of buffers is beneficial from the point of view of reducing memory usage, but freeing a large number of buffers can be expensive.

CP-burstsort preserves the original keys or records, but as soon as each bucket is sorted, it reclaims memory by copying record pointers onto the corresponding sorted tail pointers, and then freeing the bucket. To increase the likelihood that freed objects can be re-used, both C- and CP-burstsort allocate objects in power-of-2 sizes whenever possible.

C-burstsort demonstrates the advantages of filling buckets with string tails rather than pointers during insertion phase. However, at bucket sorting time all operations occur within cache, and it is once again more efficient to manipulate pointers than the suffixes themselves. Both C- and CP-burstsort reserve space in cache to allocate a pointer to each tail, and use helper sorts that rearrange only the tail pointers. We have used multikey quicksort as the helper sort for all results in this paper.

5 Cache efficiency

A detailed cache analysis is beyond the scope of this experimental paper. However, the use of copying was partly motivated by our analysis of the costs of burstsort, and we now sketch the expected cache costs of the new variants. As we show in our experiments, it is due to reduced cache costs that the new algorithms are so efficient.

For C-burstsort, a sketch of the costs is as follows.

Insertion The source array of N strings is scan traversed. The number of cache misses is K/B, where K is the total length of strings and B is the cache line size. Accesses to the trie nodes, buckets, or bucket index can incur cache misses. This is the only phase that is not cache optimal, that is, is not $O(N/B)$. The number and size of trie nodes is relatively small and most are expected to remain in cache. The

number of active locations increases with increase in the height of the trie, which can result in the buckets and any PPL structures not being cache resident.

Bursting Involves a scan traversal of the parent bucket and sequential filling of the child buckets. This phase is cache optimal and is an improvement over pointer-based burstsorts.

Traversal The trie is traversed but only those node elements that are within the character range are accessed. Each bucket and its associated array of pointers to suffixes is traversed. The bucket traversal cost can be approximated to the size of the collection K, resulting in up to K/B misses; but note that this is a loose upper bound, as only tails of strings are stored in buckets. The scan traversal of pointers to suffix can be approximated to N/B, where N is the number of keys. The traversal phase is optimal and is an improvement over pointer-based burstsort.

Output Involves outputting the strings in sort order. This is optimal and involves only a scan traversal of the buckets and the destination array. This is an improvement on pointer-based methods, which can incur a cache miss per key in addition to the traversal of source array of pointers.

The cache performance of CP-burstsort is similar to that of C-burstsort. The main difference is in the traversal phase, where the cost is up to $(K+N)/B$ misses, a slight increase. The pointers to suffixes can be approximated to N/B, where N is the number of keys.

The copy-based burstsorts are much more TLB efficient than the pointer-based sorting methods, due to the improved spatial locality of the strings.

Insertion As the source array of strings is scan traversed once, if K is the total length of strings and P is the page size then the number of TLB misses is K/P. TLB misses can occur while accessing the trie nodes, buckets and any PPL structure. But this is dependent to a large extent on the memory management unit and where the PPL structures are allocated. In the worst case, each access to the trie nodes and buckets may incur a TLB miss, but this is highly unlikely.

Bursting In pointer-based burstsort, each access to the string could result in a TLB miss; in the copy-based methods, there will be significantly fewer misses due to the scan-based approach.

Traversal Up to K/P TLB misses are expected for the buckets, with an extra N/P misses for CP-burstsort. In pointer-based burstsort, each string access could result in a TLB miss.

Output Involves outputting the strings in sort order, which is TLB optimal and involves only a scan traversal of the buckets and the destination array.

6 Experimental design

For our experiments, we have used four real-world collections with different characteristics. These are the same collections as described in earlier work (Sinha and Zobel, 2004a,b)[1] and we follow the earlier descriptions.

These collections are composed of words, genomic strings and web URLs. The strings are words delimited by non-alphabetic characters and are selected from the large web track in the TREC project (Harman, 1995, Hawking et al., 1999). The web URLs have been selected from the same collection. The genomic strings are from GenBank (Benson et al., 1993). For word and genomic data, we created six subsets, of approximately 10^5, 3.1623×10^5, 10^6, 3.1623×10^6, 10^7, and 3.1623×10^7 strings each. We call these

[1] These data sets are available at the URL `http://www.cs.rmit.edu.au/~rsinha/papers.html`.

	Data set					
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Duplicates						
Size *MB*	1.013	3.136	7.954	27.951	93.087	304.279
Distinct Words ($\times 10^5$)	0.599	1.549	3.281	9.315	25.456	70.246
Word Occurrences ($\times 10^5$)	1	3.162	10	31.623	100	316.230
No duplicates						
Size *MB*	1.1	3.212	10.796	35.640	117.068	381.967
Distinct Words ($\times 10^5$)	1	3.162	10	31.623	100	316.230
Word Occurrences ($\times 10^5$)	1	3.162	10	31.623	100	316.230
Genome						
Size *MB*	0.953	3.016	9.537	30.158	95.367	301.580
Distinct Words ($\times 10^5$)	0.751	1.593	2.363	2.600	2.620	2.620
Word Occurrences ($\times 10^5$)	1	3.162	10	31.623	100	316.230
Random						
Size *MB*	1.004	3.167	10.015	31.664	100.121	316.606
Distinct Words ($\times 10^5$)	0.891	2.762	8.575	26.833	83.859	260.140
Word Occurrences ($\times 10^5$)	1	3.162	10	31.623	100	316.230
URL						
Size *MB*	3.03	9.607	30.386	96.156	304.118	—
Distinct Words ($\times 10^5$)	0.361	0.923	2.355	5.769	12.898	—
Word Occurrences ($\times 10^5$)	1	3.162	10	31.623	100	—

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

SET 1, SET 2, SET 3, SET 4, SET 5, and SET 6 respectively. For the URL data we created SET 1 to SET 5. In detail, the data sets are as follows.

Duplicates Words in order of occurrence, including duplicates. The characteristics are similar to most collections of English documents, that is, some words occur more often than others.

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

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

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

Alternative collections. In addition to the above collections, we have included four collections as used in previous work (Sinha and Zobel, 2004a, Bentley and Sedgewick, 1997), three of which are artificial and are designed to explore pathological cases.

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

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

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

D Collection of library call numbers consisting of 100,187 strings, about the size of our SET 1 (Bentley and Sedgewick, 1997).[2]

The aim of our experiments was to compare the performance of our new algorithms with the best algorithms from our previous work. The performance was compared in terms of the running time, instruction counts, L2 cache misses, and data TLB misses. The L2 cache and DTLB results include misses of all three kinds: compulsory, capacity, and conflict. The programs are all written in C and have been gathered from the best source we could identify. We are confident that the implementations are of high quality.

For measuring the cache performance, both simulators and hardware performance counters were used. For measuring cache effects using different cache parameters, we have used `valgrind`, an open-source cache simulator (Seward, 2001). For measuring the data TLB misses on a Pentium IV, we used PAPI (Dongarra et al., 2001), which offers an interface for accessing the hardware performance counters for processor events.

For all data sizes, the minimum time from ten runs has been used so that occasional variations due to page faults, which do not reflect the cache performance, are not included. But in our runs we did not observe any significant variations between the runs; the standard deviation was extremely low. The internal buffers of our machine are flushed prior to each run in order to have the same starting condition for each experiment. Time spent in parsing the strings and writing it to the source array, creating pointers to strings, retrieving strings from disk by the driver program are not included. Thus, the time measured for the pointer-based algorithms is to sort an array of pointers to strings; the array is returned as output. Thus, the timing for pointer-based methods does not include writing the strings in sort order.

We have used two relatively new processors, a Pentium IV and a PowerPC 970. Experiments on an older Pentium III processor have also been included. These machines have different cache architectures. Most of the experiments were on a 2000 MHz Pentium IV computer with 2 GByte of internal memory and a 512 KByte L2 cache with a block size of 64 bytes and 8-way associativity. The Pentium IV machine was running the Fedora linux core 2 operating system and using the GNU gcc compiler version 3.3.3. Further details on the cache architecture of the machines used for the experiments can be found in Table 2. In all experiments, the highest compiler optimization level 03 has been used. The clock function has been used to measure the times. The machine was under light load, that is, no other significant I/O or CPU tasks were running.

7 Results and discussion

Timings for each method, each data set, and each set size are shown in Tables 3 to 7. The overall picture of these results is clear: the copy-based burstsorts are much faster than the pointer-based versions. In the best case, C-burstsort is four times faster than the pointer-based burstsort and six times faster than the best of the previous sorting methods on the largest set of genome data.

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

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

Table 2: *Architectural parameters of the machines used for experiments.*

	Data set					
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Multikey quicksort	40	210	680	3,230	14,260	56,230
MBM radixsort	40	190	640	3,270	15,470	63,460
Adaptive radixsort	50	230	720	3,030	12,110	46,940
Burstsort	30	130	380	1,540	6,540	29,310
C-burstsort	30	90	270	1,060	3,570	12,470
CP-burstsort	30	110	310	1,210	4,140	15,270
CF-burstsort	30	100	260	930	3,280	12,200
CPF-burstsort	40	120	310	1,080	3,900	14,860

Table 3: *Duplicates. Running time (milliseconds) to sort with each method on a Pentium IV.*

The degree of improvement varies from collection to collection, but the trends are consistent and clear. First, C-burstsort is always faster than CP-burstsort, which in turn is faster than any previous method. Second, use of free bursts (in CF-burstsort and CPF-burstsort, with $F = 100$) consistently yields further gains. Third, the larger the collection, the greater the improvement.

These strong positive results make it clear that our new copy-based burstsorts far outperform other methods when used in typical circumstances. We now consider in detail the performance in special and pathological cases, as these illustrate the mechanisms underlying the effectiveness of copy-based sorting.

Table 8 illustrates performance on the pathological case of sorted input. The locality of strings in a sorted collection is maximal, so pointer-based sorting methods can make optimal use of cache. This is the only case in which burstsort, copy-based or otherwise, is not the best method. While the performance of the pointer-based methods improved dramatically on sorted data, by up to a factor of 10, the performance of the copy-based methods did not improve quite so dramatically. The reason appears to be the cost of copying without the relative advantages of cache-efficiency as in the case for the unsorted collections. However, we note that these results may be atypical for sorted data; on other sorted data sets, such as random strings, the

	Data set					
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Multikey quicksort	50	200	880	3,870	15,470	61,620
MBM radixsort	40	170	830	3,660	15,650	63,870
Adaptive radixsort	60	230	930	3,550	13,800	53,570
Burstsort	30	120	500	1,870	7,510	33,510
C-burstsort	30	100	380	1,380	4,620	15,650
CP-burstsort	30	110	440	1,560	5,310	19,600
CF-burstsort	30	110	380	1,270	4,450	15,260
CPF-burstsort	50	140	430	1,430	5,120	19,300

Table 4: *No duplicates. Running time (milliseconds) to sort with each method on a Pentium IV.*

	Data set					
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Multikey quicksort	50	270	1,100	4,430	17,820	68,770
MBM radixsort	50	300	1,340	5,470	23,390	91,000
Adaptive radixsort	90	350	1,370	4,690	17,980	67,360
Burstsort	40	160	590	2,200	8,720	40,600
C-burstsort	30	120	390	1,150	3,210	10,730
CP-burstsort	30	160	490	1,350	4,170	14,090
CF-burstsort	20	90	310	930	3,010	10,120
CPF-burstsort	30	100	350	1,060	3,800	13,180

Table 5: *Genome. Running time (milliseconds) to sort with each method on a Pentium IV.*

copy-based methods were considerably better.

In contrast, the burstsorts are clearly the methods of choice for the alternative collections, as shown in Table 9. However, compared to pointer-based burstsort the copy-based burstsorts are up to twice as slow for collections A and C. These collections are particularly challenging for copy-based burstsorts, as they both have long strings. The primary reason for the poorer performance is TLB misses. The trie nodes are not compact and, after the insertion of a small number of strings, the large majority of the strings have to traverse about 100 trie nodes. As there are only 64 TLB entries, each access to a node may lead to a TLB miss. If the alphabet size is reduced to 128, leading to more compact trie nodes, the performance improves dramatically, by 50%. More generally, the node size can be restricted to cover only the range of characters observed during the initial loading of strings into the segmented buffer. For this data, the node size would collapse to two symbols.

An alternative is to use more compact nodes, as is achieved using the BIS structure, as shown in Table 9. The results are similar to the pointer-based burstsort on collections A, C, and D, and are up to twice as fast as pointer-based burstsort in collection B.

The major advantage that copy-based methods have over pointer-based burstsort and the other pointer-based methods is that more strings can be accommodated in cache. In copy-based methods, the strings are stored contiguously; the number of strings in cache depends on string length and the size of cache, not on the number of blocks in cache. It is thus expected that the cache performance of C-burstsort will approach

	Data set				
	Set 1	Set 2	Set 3	Set 4	Set 5
Multikey quicksort	130	700	2,500	8,860	40,650
MBM radixsort	140	820	3,480	13,220	61,720
Adaptive radixsort	160	730	2,450	10,100	41,760
Burstsort	70	410	1,680	5,660	28,620
C-burstsort	70	230	1,420	4,320	14,090
CP-burstsort	80	260	1,500	4,560	15,390
CF-burstsort	80	240	820	2,830	12,440
CPF-burstsort	90	260	780	3,070	13,730

Table 6: *URL. Running time (milliseconds) to sort with each method on a Pentium IV.*

	Data set					
	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Multikey quicksort	40	210	970	3,900	15,120	62,360
MBM radixsort	30	130	670	2,360	8,370	37,630
Adaptive radixsort	60	220	760	2,470	9,390	41,260
Burstsort	30	100	450	1,670	5,660	26,440
C-burstsort	20	70	300	1,180	4,420	16,440
CP-burstsort	20	100	370	1,320	5,100	19,770
CF-burstsort	40	110	480	1,620	5,290	16,910
CPF-burstsort	50	150	630	1,960	5,660	20,310

Table 7: *Random. Running time (milliseconds) to sort with each method on a Pentium IV.*

that of pointer-based burstsorts as the length of strings approaches the cache line size, as observed in these results.

To show how the performance of the copy-based methods is affected by string length, we experimented with four collections of fixed-length strings. The length of the strings in each collection were 10, 30, 60, and 100 characters. The characters were chosen uniformly at random from the ASCII range. Times are shown in Table 10. As the length of strings increases in these random collections, where the characters are uniformly distributed, the cost of copying the strings into the buckets increases. An increase in string length would reduce the number of strings that can be stored in a bucket. When the average length of the strings exceeds the cache line size, the number of strings that can be stored in a bucket is less than that of pointer-based burstsort. This in turn results in creation of more trie nodes, increasing the number of cache misses during the insertion phase. But even then, the bursting and traversal phases are optimal as compared to the pointer-based burstsort. There is a trade-off between the extra cache misses incurred during the insertion phase and the cache misses saved during bursting and traversal. However, there is an underlying difficulty for burstsort due to the fact that the information density (repetition of prefixes) of these collections is low.

An alternative hybrid scheme for such collections is to copy a record pointer and a short fixed-length tail segment to buckets. If the segment is enough to complete bursting and bucket sorting, the cost of copying the remaining irrelevant bytes is avoided. If not, the next segment is paged into the bucket (probably incurring a cache miss). As shown in Table 10, this paging scheme, called CPL-burstsort, was found to be

	Data set				
	Set 2	Set 3	Set 4	Set 5	Set 6
Multikey quicksort	30	100	490	2,070	7,530
MBM radixsort	20	90	400	1,610	5,780
Adaptive radixsort	30	90	330	1,230	4,210
Burstsort	20	80	360	1,460	5,070
C-burstsort	30	120	540	2,420	8,170
CP-burstsort	40	160	670	2,840	9,490
CF-burstsort	10	100	530	2,420	8,160
CPF-burstsort	30	150	660	2,830	9,470

Table 8: *Sorted (duplicates). Running time (milliseconds) to sort with each method on a Pentium IV. Times for Set 1 were too small to measure accurately.*

	Data set			
	A	B	C	D
Multikey quicksort	6,970	14,990	2,940	90
MBM radixsort	7,430	31,310	8,060	90
Adaptive radixsort	3,660	16,830	1,760	110
Burstsort	1,300	8,910	780	60
C-burstsort	3,380	5,290	1,600	50
CP-burstsort	3,360	6,190	1,440	60
C-burstsort (BIS)	1,470	5,020	810	50
CP-burstsort (BIS)	1,460	6,280	720	50

Table 9: *Running time (milliseconds) to sort the alternative data sets with each method on a Pentium IV.*

up to three times faster than CP-burstsort on 100-byte random strings. It is expected that the improvements will be even more dramatic with increasing string length.

The L2 cache misses incurred for the no-duplicates and URL collections are shown in Figure 5. (Similar results to those on no-duplicates are observed for the other collections.) The cache misses for C-burstsort are as low as half that of pointer-based burstsort and as low as a tenth of that of the previous best algorithms. The string length of the URL collection is long with the average length being about 28 characters. Copying and maintaining such large strings in the bucket may reduce the advantages but even then for the largest set C-burstsort incurs fewer cache misses than pointer-based burstsort.

Copying strings into buckets keeps similar strings together. This increases the page locality, resulting in fewer TLB misses during the bursting and traversal phases. To further lower TLB misses during bucket sorting, the size of the bucket should be the smaller of the cache size and the TLB size. TLB misses can result while accessing trie nodes and bucket, which can be distributed in memory across several pages. The number of TLB misses due to trie accesses is distribution dependent and is expected to increase with the size of the collection, as shown in Figures 6 and 7. For smaller data sizes, the TLB misses are almost non-existent and rise slowly with increasing data size. The number of TLB misses incurred by C-burstsort is only one-third that of pointer-based burstsort or the best of the previous algorithms. For a TLB of 64 entries, the number of strings in the bucket ideally should be less than 64, but the threshold size in pointer-

	String length			
	10	30	60	100
Burstsort	480	520	550	610
C-burstsort	300	640	1,310	2,080
CP-burstsort	370	690	1,350	2,110
CPL-burstsort	480	620	680	740

Table 10: *Running time (milliseconds) to sort the random collection of one million strings of fixed lengths, on a Pentium IV.*

	Collections				
	Duplicates	No duplicates	Genome	URL	Random
Burstsort	2,504	2,245	2,792	3,313	95
C-burstsort	371	381	369	1,340	95
CP-burstsort	500	491	508	1,499	95

Table 11: *Count of trie nodes for the largest set size of each collection.*

based burstsort is $8,192$ (the number of blocks in cache), which in the worst case results in a TLB miss for each access to a string.

It is expected that the cost of copying string suffixes will be high in terms of the number of instructions. This is shown in Figure 8, where the number of instructions per key for C-burstsort is higher than for any other method. The number of required instructions is up to twice as much as adaptive radixsort. In older machines, such as the Pentium III Xeon, this cost in instructions may offset the savings in cache misses. This is shown in Figure 11, where C-burstsort is slower than pointer-based burstsort for the URL collection. However, as processors continue to increase in speed relative to memory, the advantage of these new copy-based approach should continue to increase.

In pointer-based burstsort, the number of pointers to strings that can be stored in a bucket is dependent on the number of blocks in cache. As the size of the cache line increases, the number of blocks in cache will decrease, thus reducing the threshold size. The effects of varying cache line size using the same threshold (of $8,192$) is shown in Figure 9. For pointer-based burstsort, the L2 cache misses shows an increasing trend as the size of cache line increases. More cache misses are incurred during the bucket sorting phase, as all the strings in the bucket may no longer be cache resident for the larger cache lines. For a line size of 256 bytes, the number of blocks is $2,048$; it would have been better to use a threshold size less than $2,048$, but this will lead to a larger number of trie nodes, which may be inefficient. Thus the performance of pointer-based burstsort is relatively more dependent on the number of blocks in cache.

In contrast, increasing the line size has the opposite effect for C-burstsort. Efficiency is less dependent on the number of blocks in cache; indeed, as the strings are stored contiguously, an increase in cache line size reduces cache misses in bucket processing. For C-burstsort, the cache misses decrease by over a factor of three; only the URL collection shows a slight upward trend for a cache line size of 256, due to the large number of trie nodes and the decreasing number of blocks.

The number of trie nodes is shown in Table 11. The number of trie nodes is directly related to the size of the bucket and the number of strings it can accommodate. Since, for all collections, the size of the string was less than the cache line size, the number of strings that can be stored in a bucket under pointer-based

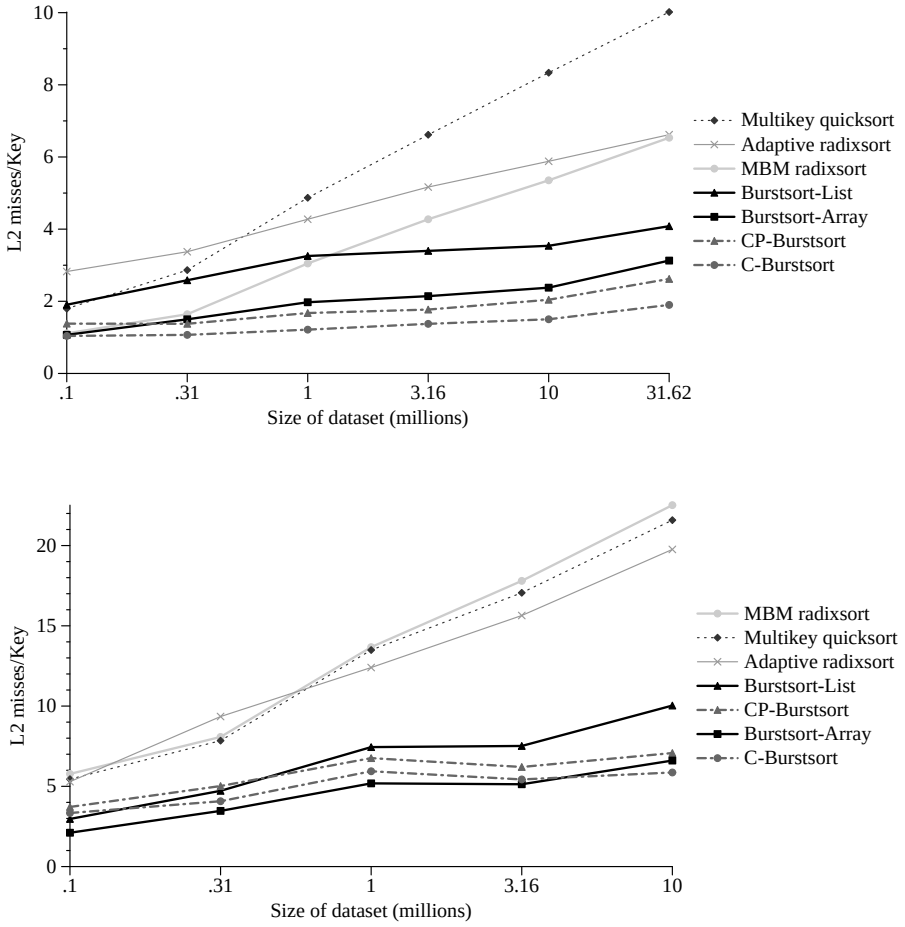

Figure 5: *Cache misses, 512 KB cache, 8-way associativity, 64 bytes block size. Upper: no duplicates. Lower: URLs.*

burstsort is less than a bucket under C-burstsort. The number of trie nodes in pointer-based burstsort was up to seven times more than in C-burstsort.

Table 12 shows the peak memory used while sorting. The results for multikey quicksort can be taken as a practical best-case for a pointer-based in-place method. On average, pointer-based burstsort uses about 40% more memory. Surprisingly, C-burstsort averages very close to multikey quicksort, using slightly more memory for the duplicates and no-duplicates collections, but somewhat less for the genome, URL and random collections. On a wider variety of data collections in random, sorted, or reverse order, we have observed that C-burstsort and multikey quicksort both use an average of 1.3 bytes of allocated memory per byte sorted. Memory usage by C-burstsort can be further reduced by using smaller buffer segments during key insertion or (with some loss of speed) by using a smaller bucket-growth factor. CP-burstsort shows the highest memory usage, since its buckets hold both string tails and record pointers and the original keys are not deleted. However, it should be compared with other stable sorts, such as the adaptive radixsorts, which also tend to have higher memory requirements.

In the experiments reported above, the timings do not include the time taken to write the strings in sort order. The time taken for writing the sorted strings onto the destination array for pointer-based and copy-based methods is shown in Table 13. For pointer-based methods, the time taken to write the strings in sort

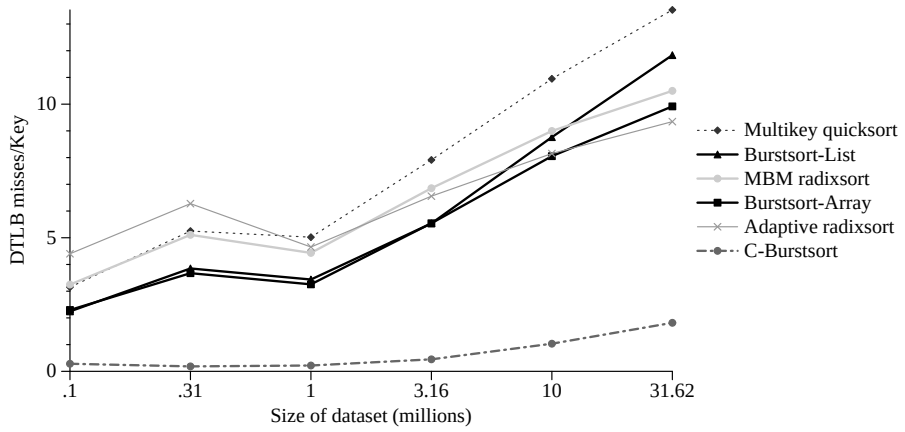

Figure 6: *Data TLB misses on a Pentium IV for the duplicate collection*

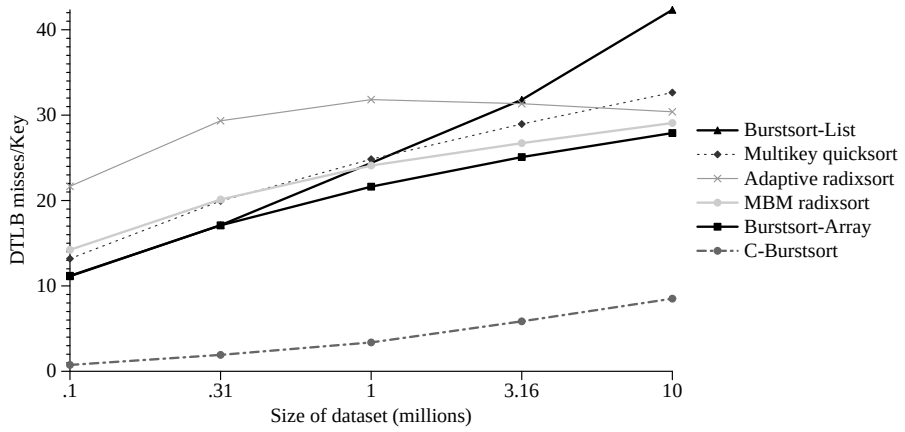

Figure 7: *Data TLB misses on a Pentium IV for the URL collection*

order by traversing the sorted array of pointers to strings involves a scan traversal of the array of pointers to strings, and may incur a cache miss per string access as the strings are accessed arbitrarily. In the case, of the copy-based methods, the time measured includes a traversal of the entire trie node and a scan traversal of the bucket and writing the strings to the destination array. As the strings are stored contiguously, this phase is a scan traversal and can be regarded as optimal.

Other architectures

While most of our experiments used the Pentium IV, we also tested our algorithms on the older Pentium III, and on the significantly different architecture of the PowerPC 970. The copy-based burstsort with the BIS structure has been used for these experiments. On the PowerPC, Figure 10 shows that for the largest collection C-burstsort is up to three times faster than pointer-based burstsort. These are dramatic improvements for such a small collection size. Similar performances were seen for the other collections.

On older processors like the Pentium III, copying costs may offset savings in cache misses. Figure 11 shows that C-burstsort is faster than pointer-based burstsort for most collections except for the URL collection. While the improvements for most collections are much smaller than on the Pentium IV or PowerPC,

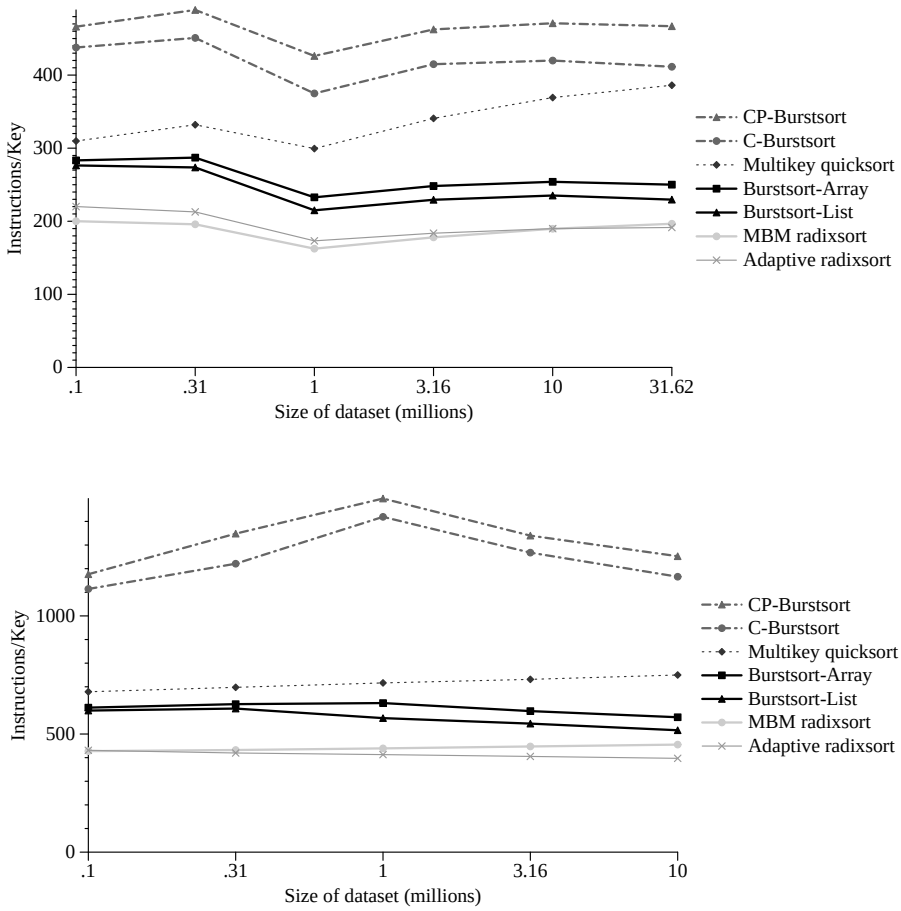

Figure 8: *Instructions per key, 512 KB cache, 8-way associativity, 64 bytes block size. Upper: duplicates. Lower: URLs.*

they are still faster than pointer-based burstsort. In other words, the trends in hardware are favoring the burstsorts compared to other sorting methods, and favor our new copy-based burstsorts compared to the original burstsort.

The normalized times for C-burstsort are shown in Figure 12. For all these collections, the normalized time has not grown at all with the increase in collection size. For the genome collection the normalized time has gone down due to the increase in duplicates.

8 Discussion

The pointer-based burstsorts described in our previous papers are fast because, with a single access to each key, they assign related strings to buckets small enough to be sorted in cache. However, their performance is limited by three problems. First, since the buckets hold only pointers, each key must again be accessed at its original arbitrary memory location at bucket sorting time; these random references contribute significantly to the cache and TLB misses generated by the pointer burstsorts. Second, when the original keys are re-accessed, the terminal bytes needed for final sorting are flanked by already processed prefix bytes and bytes from adjacent but unrelated keys; thus, the cache lines loaded contain a substantial proportion of useless

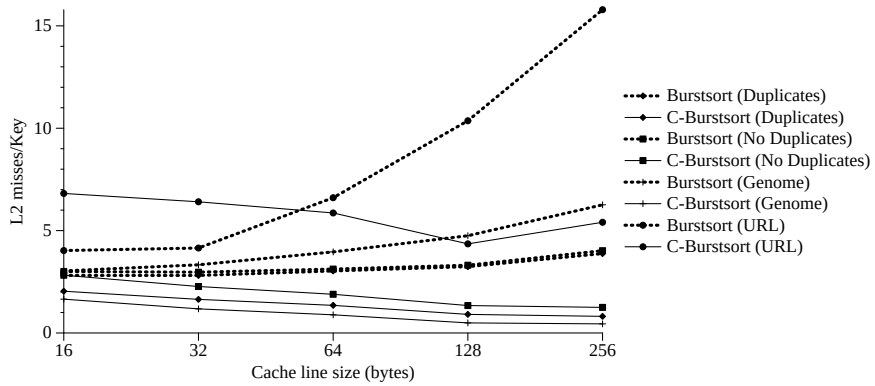

Figure 9: *Cache misses with varying cache line size on Pentium IV with 512 KB L2 cache.*

	MKQsort	Burstsort	C-burstsort	CP-burstsort
Duplicates	427	646	436	768
No duplicates	504	734	566	990
Genome	424	610	342	689
URL	344	412	324	620
Random	440	593	434	948
Sorted (duplicate)	427	644	435	765

Table 12: *Memory usage (in megabytes) for the largest set size.*

bytes. Third, buckets that burst at a fixed pointer count are not efficient when average key length varies significantly from one bucket to another; if the threshold is set low enough that buckets of long keys fit into cache for sorting, buckets of short keys will waste a substantial amount of cache space.

We expect the fastest sorting when buckets are designed to assemble exactly the information needed for final sorting, when they use cache efficiently without wastage, and when they are as few and as large as possible (since there is an overhead cost for each bucket). C-burstsort was designed to address these goals as well as the specific limitations of pointer-based burstsorts. Additional goals were to decrease memory requirements, to increase the locality of data processing, and for CP-burstsort, to achieve the speed of burstsort in a stable sort suitable for processing multiple field records.

In large part, the goals have been met. Since C-burstsort loads buckets with unexamined key tails instead of pointers, the exact data needed for final sorting is already present, and there is no need to re-access the original memory locations of the keys at bucket sorting time. Since key suffixes sharing a common prefix are copied into buckets contiguously, they are adjacent to related suffixes except at bucket boundaries, and cache lines loaded for bucket sorting will contain very few irrelevant bytes. Finally, buckets are grown or burst based on the cumulative length of the key tails copied to them instead of a fixed count; if keys are short, buckets automatically hold more key tails and cache space is not wasted. Thus, the specific problems of pointer-based burstsorts are all addressed by inserting string tails instead of pointers.

In more general terms, C-burstsort comes close to assembling exactly the information needed for final sorting and using cache efficiently without wastage. The exception is the terminal bytes of suffixes that are sorted before all their bytes are read; these are not needed, but there is no time-saving way to identify such bytes in advance. The goal of making buckets as few and large as possible is less clear cut; certainly,

	Collections				
	Duplicates	No duplicates	Genome	URL	Random
Pointer-based	8,570	10,200	10,220	4,420	10,470
Copy-based	3,750	4,570	3,350	2,080	4,240

Table 13: *Time (milliseconds) to copy strings in sort order to destination array for the largest set size for each collection in Pentium IV.*

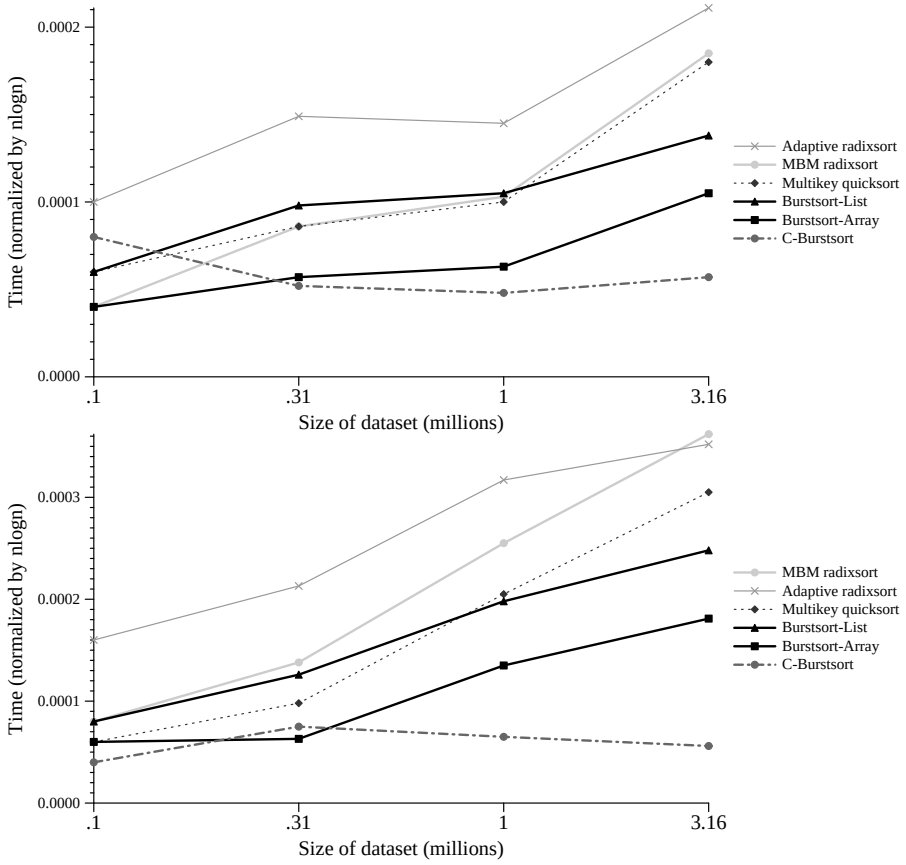

Figure 10: *PowerPC 970, relative sorting time for each method. The vertical scale is time in milliseconds divided by* $n \log n$*. Top: duplicates. Bottom: genome.*

C-burstsort enables a given amount of cache memory to hold and process more short keys, but it is still the case that most buckets will neither be full nor as large as cache. It seems likely that control of bucket count and size can be further improved. The basic change in C-burstsort—filling buckets with suffixes rather than pointers—also greatly increases the spatial locality of data. Finally, CP-burstsort provides stable sorting with much of the speed of C-burstsort.

An instance in which copy-based burstsorts are slower than pointer-based burstsorts is the case of long random strings, where copy-based burstsorts are weighed down by the cost of moving long terminal sequences that do not affect sort order. An alternative is to copy a record pointer and a short fixed-length tail segment to buckets. We found that this hybrid paging scheme, CPL-burstsort, was up to three times

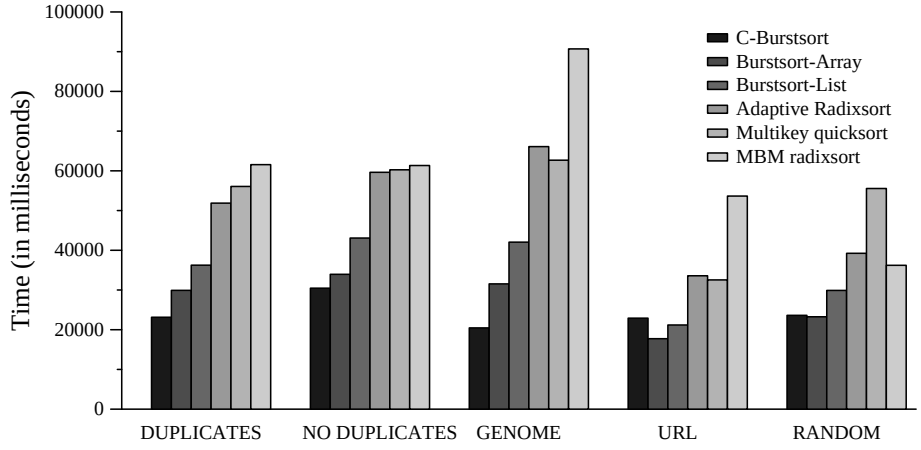

Figure 11: *Pentium III, sorting times (milliseconds) for the largest set size for each collection.*

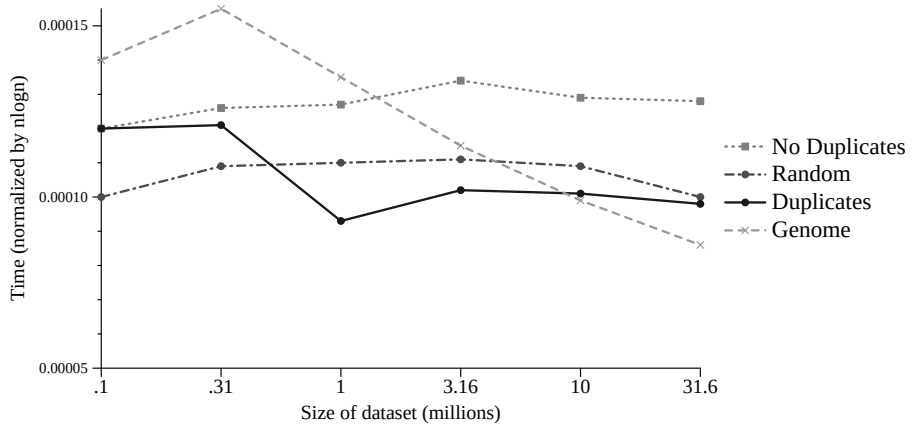

Figure 12: *Pentium III, time for C-burstsort for all collections. The vertical scale is time in milliseconds divided by* $n \log n$.

faster than CP-burstsort on 100-byte random strings while using lesser memory.

Aside from long strings of random data, we do not know if there are pathological forms or orders of string data that severely degrade the performance of copy-based burstsorts. We know that free bursts can improve performance, and that random sampling can do better than free bursts in some cases, but we do not know the best scheme for bucket growth and bursting that will produce the fewest buckets and the most efficient use of cache. The burst trie structure results in many buckets that are not full and thus much smaller than cache; development of a modified dynamic structure that efficiently partitions large buckets while not splitting into as many small ones is being investigated.

9 Conclusions

We have proposed variants of burstsort that are based on copying the strings into buckets. This improves the spatial locality of the strings compared to the pointer-based burstsort. Several experiments on different architectures with both real and artificial collections of several data sizes were used to measure efficiency.

These results have shown substantial reductions in sorting time compared to pointer-based sorting algorithms. These copy-based versions are efficient on modern processors, where locality is an increasingly important issue. The time to sort and the number of L2 cache misses incurred are half those of pointer-based burstsort. Random accesses to strings are completely eliminated, as all string accesses in the copy-based versions involve a scan traversal of an array.

Reducing unnecessary random accesses to strings improves the page locality of the copy-based variants, resulting in far fewer TLB misses. The scan traversal of the buckets implies that the copy-based algorithms should cause fewer misses in the lower levels of the cache hierarchy, while the use of free bursts to rapidly build a skeleton tree structure minimizes copying costs when buckets are burst at large sizes. This work has shown that algorithms that improve the spatial locality of strings can yield substantial improvements in performance. While the absolute and relative sorting speeds observed will depend on the cache architecture of a particular system, we expect these algorithms to be the fastest available for sorting large collections of string keys when memory latency is an important factor.

There is still significant scope to improve copy-based burstsorts. More sophisticated control of bucket growth may yield better results than our simplistic doubling scheme. The traversal and output in C-burstsort can be integrated into a single pass; the strings in a bucket can be output after the bucket is sorted. Memory requirements can be substantially reduced by reading keys directly into the trie structure, and outputting directly from each bucket after it is sorted. This lowers C-burstsort's average memory usage from 1.3 bytes to about 0.84 bytes allocated per byte sorted. Further investigation is needed to determine the most compact way to implement trie nodes for the largest collection sizes. Finally, use of special-purpose memory management code and tuning algorithm parameters to the TLB sizes of particular systems may further increase efficiency.

However, our new copy-based burstsorts are already by far the most efficient algorithms for sorting large string collections on cache architectures. They demonstrate that cache-awareness can provide large improvements over previous methods. In our experiments, the gain in sorting speed between quicksort (1961) and adaptive radixsort (1998) is less than the additional improvement by the copying burstsorts in 2005. These are dramatic results.

Acknowledgments

David Ring would like to acknowledge James Barbetti for insightful remarks regarding the performance of pointerized sorts on cache versus non-cache architectures. This research was supported by the Australian Research Council.

References

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

BENSON, D., LIPMAN, D. J., AND OSTELL, J. 1993. Genbank. *Nucleic Acids Research 21,* 13, 2963–2965.

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

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

DONGARRA, J., LONDON, K., MOORE, S., MUCCI, S., AND TERPSTRA, D. 2001. Using PAPI for hardware performance monitoring on linux systems. In *Proc. Conf. on Linux Clusters: The HPC Revolution*. Urbana, Illinois, USA.

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

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

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

HENNESSY, J. L. AND PATTERSON, D. A. 2002. *Computer Architecture: A Quantitative Approach*, Third ed. Morgan Kaufmann Publishers, 929 Campus Drive, Suite 260, San Mateo, CA 94403, USA.

HOARE, C. A. R. 1961. Algorithm 64: Quicksort. *Communications of the ACM 4,* 7, 321.

HOARE, C. A. R. 1962. Quicksort. *Computer Jour. 5,* 1, 10–15.

JIMENEZ-GONZALEZ, D., NAVARRO, J., AND LARRIBA-PEY, J. L. 2003. CC-radix: a cache conscious sorting based on radix sort. In *Proc. Euromicro Workshop on Parallel, Distributed and Network-based Processing*, A. Clematis, Ed. IEEE Computer Society Press, 101–108.

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

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

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

RAHMAN, N. AND RAMAN, R. 2000. Analysing cache effects in distribution sorting. *ACM Jour. of Experimental Algorithmics 5,* 14.

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

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

SINHA, R. 2004. Using compact tries for cache-efficient sorting of integers. In *Proc. International Workshop on Effi cient and Experimental Algorithms*, C. C. Ribeiro, Ed. Lecture Notes in Computer Science, vol. 3059. Springer Verlag, Angra dos Reis, Rio de Janeiro, Brazil, 513–528.

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

SINHA, R. AND ZOBEL, J. 2004b. Using random sampling to build approximate tries for efficient string sorting. In *Proc. International Workshop on Effi cient and Experimental Algorithms*, C. C. Ribeiro, Ed. Lecture Notes in Computer Science, vol. 3059. Springer Verlag, Angra dos Reis, Rio de Janeiro, Brazil, 529–544.

WICKREMESINGHE, R., ARGE, L., CHASE, J., AND VITTER, J. S. 2002. Efficient sorting using registers and caches. *ACM Jour. of Experimental Algorithmics 7,* 9.

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