[Ri] R. Rivest, "A Fast Stable Minimum Storage Sorting Algorithm," Rep. 43, Institute de Recherche d'Informatique, Rocquencont, France, 1973.

[SS] J. S. Salowe and W. L. Steiger, "Simplified Stable Merging Tasks," *Journal of Algorithms* 8 (1987), 557–571.

[Tr1] L. Trabb Pardo, "Stable Sorting and Merging with Optimal Space and Time Bounds," Computer Science Department Technical Report CS–74–470, Stanford University, 1974.

[Tr2] ———————, "Stable Sorting and Merging with Optimal Space and Time Bounds," *SIAM Journal on Computing* 6 (1977), 351–372.

[Tr3] ———————, private communication.

[Wo] J. K. Wong, "Some Simple In-Place Merging Algorithms," *BIT* 21 (1981), 157–166.

[HL2] ———————, "Stable Duplicate-Key Extraction with Optimal Time and Space Bounds," *Acta Informatica* 26 (1989), 473–484.

[HL3] ———————, "Stable Set and Multiset Operations in Optimal Time and Space," *Proceedings 7th ACM Symposium on Principles of Database Systems* (1988), 288–293.

[Ho] E. C. Horvath, "Stable Sorting in Asymptotically Optimal Time and Extra Space," *Journal of the ACM* 25 (1978), 177–199.

[Jo] D. E. H. Jones, The Inventions of Daedalus, W. H. Freeman and Co., New York, NY, 1982.

[Kn1] D. E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, MA, 1973.

[Kn2] ———————, private communication.

[Kr] M. A. Kronrod, "An Optimal Ordering Algorithm without a Field of Operation," *Dok. Akad. Nauk SSSR* 186 (1969), 1256–1258.

[Mo] D. Motzin,"A Stable Quicksort," *Software – Practice and Experience* 11 (1981), 607–611.

[MU] H. Mannila and E. Ukkonen, "A Simple Linear-time Algorithm for In Situ Merging," *Information Processing Letters* 18 (1984), 203–208.

[Pr] F. P. Preparata, "A Fast Stable Sorting Algorithm with Absolutely Minimum Storage," *Theoretical Computer Science* 1 (1975), 185–190.

References

[Ca] S. Carlsson, "Splitmerge – A Fast Stable Merging Algorithm," *Information Processing Letters* 22 (1986), 189–192.

[CD] E. G. Coffman, Jr. and P. J. Denning, Operating Systems Theory, Prentice-Hall, Englewood Cliffs, NJ, 1973.

[DD1] K. Dudzinski and A. Dydek, "On a Stable Minimum Storage Merging Algorithm," *Information Processing Letters* 12 (1981), 5–8.

[DD2] S. Dvorak and B. Durian, "Towards an Efficient Merging," *Lecture Notes in Computer Science* 233 (1986), 290–298.

[DD3] ————, "Merging by Decomposition Revisited," *The Computer Journal* 31 (1988), 553–556.

[DD4] ————, "Stable Linear Time Sublinear Space Merging," *The Computer Journal* 30 (1987), 372–375.

[De] R. B. K. Dewar, "A Stable Minimum Storage Algorithm," *Information Processing Letters* 2 (1974), 162–164.

[GL] X. Guan and M. A. Langston, "Time-Space Optimal Parallel Merging and Sorting," *Proceedings 1989 International Conference on Parallel Processing*, Vol. III, 1–8.

[HL1] B-C Huang and M. A. Langston, "Practical In-Place Merging," *Communications of the ACM* 31 (1988), 348–352.

including the frequency distribution of keys, the percentage of duplicate keys present, the initial sortedness of files, the break-even point at which less sophisticated schemes for small subfiles are used, record length, computer architecture and so on.

Optimistically, we observe that even simpler, more effective optimal time and space strategies are very possible. Also, the design and analysis of time-space optimal *parallel* algorithms is a subject of obvious importance for future study [GL]. As block rearrangement strategies become more widely known, we hope that their practical potential for merging, sorting, duplicate-key extraction and related file-processing operations will begin to be better understood.

$n + n/2 + O(\sqrt{n})$ comparisons (n to merge, $n/2 + O(\sqrt{n})$ to search for the correct blocks) and n exchanges. In general, pass $2k+1-j$, $1 \le j \le k-1$, needs at most $(n + \frac{n}{2^j}) + O(\sqrt{n})$ comparisons and n exchanges. Therefore, we can balance these leading terms, deriving a cost for the merging passes bounded above by $2kn < n \log_2 n$ comparisons and $n \log_2 n$ exchanges. Linear time suffices for the final merging and sorting steps. We conclude that we are guaranteed a worst-case key-comparison and record-exchange grand total not greater than $2.5\, n \log_2 n$.

This worst-case total compares favorably with *average-case* key-comparison and record-exchange totals for popular *unstable* methods: quick-sort's average-case figure is a little more than $1.4\, n \log_2 n$; heap-sort's is about $2.3\, n \log_2 n$. (These values are derived from the analysis in [Kn1], where we count a single record movement at one third the cost of a two-record exchange.)

6. Directions for Continued Research

We have presented relatively straightforward and efficient stable merging and sorting strategies that simultaneously optimize both time and space (to within a constant factor). The upper bounds we have derived on constants of proportionality are probably overly pessimistic, representing extreme and possibly unrealizable cases hardly representative of expected behavior. On the other hand, we again remind the reader that we have brushed aside lower-order terms that can be significant in practice, especially for small files.

Given the obvious importance of merging and sorting, a next logical step along this general line of investigation might be a thorough testing of careful implementations of these algorithms. A great number of factors would likely be relevant to such an empirical study, INSERT FIGURE 3

After pass $2k$, BC has been replaced by BD, where D (like C) contains 2^k sorted blocks, each of size 2^k. However, we can now complete our stable sort of L by stably sorting B, sorting D with a stable $BLOCKSORT$, and stably merging B with D. See Figure 4. Thus the entire strategy runs in $O(n \log n)$ time and $O(1)$ extra space.

INSERT FIGURE 4

We observe that major factors that make this scheme so much faster than a direct merge-sort implementation are these: 1) we extract the internal buffer only once, not at every merge operation, 2) we use the buffer in a novel and very efficient fashion for passes 1 through $k+1$ as we break it into advantageously-sized pieces and pass them across the file, and 3) we avoid unnecessary record movement by delaying the use of $BLOCKSORT$ until the final step.

5.3 Constant of Proportionality Bounds

As in Section 4.5, we focus on key comparisons and record exchanges, concentrating on the constant of proportionality for the leading (this time, $O(n \log n)$) time complexity term.

Filling the buffer requires no more than $n \log_2 \sqrt{n} = .5n \log_2 n$ comparisons and only a linear number of exchanges. (When n is not of the special form, this is the only step whose time complexity may increase, since a bigger buffer is used. Even so, the buffer's size is no more than doubled, thereby affecting at most the linear term.) The first merging pass needs at most $n/2$ comparisons and n exchanges. In general, pass i, $1 \leq i \leq k+1$, needs at most $(n - \frac{n}{2^i})$ comparisons and n exchanges. The last merging pass, pass $2k$, needs at most blocks, each of size 2^k. In pass $k+2$, we use B to obtain 2^{k-2} sublists, each of size 2^{k+2} as follows. Let X and Y denote a pair of sublists in C to be merged. We first locate the block of X whose head contains the smallest key in X. Let X_1 denote this block. Let Y_1 denote the corresponding block of Y. (Note: We will search X and Y for these and all remaining merging blocks as they are needed. Although blocks will always be sorted internally, they will in general become unordered within a sublist with respect to each other. This turns out to be advantageous in the long run, requiring but a single $BLOCKSORT$ after the final pass rather than a series of $BLOCKSORT$'s at each pass that would result in a great deal of unnecessary record movement.)

X_1 and Y_1 are now merged into the buffer's block until it is filled. Whenever a block is filled, we must determine the next block to fill, as follows. If the buffer is now contained within one block, then that block is filled next. Otherwise, the buffer must be split into two pieces, one in a block of X and the other in a block of Y. If one piece is a suffix for its block (both cannot be), then we resume the merge at that block. If not, then each piece must be a prefix for its block, and we resume the merge at the block with the smaller tail, ties broken in favor of the X block. After all blocks of X and Y are merged in this manner, the buffer (now in one block) is moved back to its original position and we begin to merge the next pair of sublists in the same fashion.

This procedure is repeated in every subsequent pass, each time with half as many sublists, each sublist with twice as many blocks, until pass $2k$, which is the last. See Figure 3. In this step, we have used at most constant extra space and each of the $k-1$ passes needs only linear time.

on B to see if its key is already present. If so, we go on to scan the next record. If not, we $ROTATE$ the appropriate segment of L so that B's rightmost record occupies position $i-1$. We then insert the new record into B. As soon as B is filled, we invoke $ROTATE$ to make B a prefix of L. Figure 1 illustrates how this process modifies our example list of 20 elements. We have used only $O(1)$ extra space, $O(n)$ exchanges and $O(n \log 2^k = nk)$ comparisons.

INSERT FIGURE 1

In the second step of the algorithm, we use B to conduct the first $k+1$ passes of a merging sort. We first employ the rightmost buffer element to conduct a left-to-right pass of A, producing a sequence of sorted two-record sublists as we go. The second merging pass is done with the rightmost two remaining buffer elements, this time producing sorted four-element sublists, and so on. The size of B ensures that this simple strategy suffices for k passes, each doubling the length of the sorted sublists. (Since $2^k = \sum_{i=1}^{k} 2^{i-1} + 1$, pass i is performed with 2^{i-1} buffer elements for $1 \leq i < k$, but in pass k we use $2^{k-1}+1$ elements, one more than we really need.) Now that B is reassembled as a suffix of L, we proceed to use it in a right-to-left fashion to perform merging pass $k+1$. Therefore BA has been transformed into BC, where C contains 2^{k-1} sorted sublists, each of size 2^{k+1}. See Figure 2. No more than $O(1)$ space and $O(nk)$ time has been used.

INSERT FIGURE 2

For the third step of the algorithm, it is helpful to think of C as a collection of 2^k sorted

5.2 A Nontrivial Sort-by-Merging Strategy

Consider an environment in which no key is duplicated more than about $\sqrt{n}$ times, which may be plausible in many settings. With this restriction, we now outline a method to sort stably by merging so as to bypass much of the overhead involved in a direct merge-sorting scheme. (Nevertheless, without this restriction, we must endorse instead the direct merge-sort approach. That is, we have found no general mechanism by which our nontrivial merge-sort described below can stably handle large numbers of homogeneous blocks in its later passes without overstepping our professed goal of presenting relatively simple and practical algorithms.)

To facilitate discussion, suppose that n is of the form $2^{2k} + 2^k$ for some positive integer k. We assume that no single key is represented more than 2^k times. Consequently, we can use blocks of size 2^k since there are more than 2^k distinct keys available for the buffer. (No laborious discussion of implementation details is necessary. If n is not of the proper form, we merely determine the value of k for which $2^{2k} + 2^k < n < 2^{2(k+1)} + 2^{k+1}$. Our restriction becomes that no single key is represented more than 2^{k-1} times, insuring blocks of size of 2^{k+1}, which will do.) Figure 1a) depicts such a list with $k = 2$ and $n = 20$. Only record keys are listed, denoted by capital letters. Subscripts are included to keep track of duplicate keys as the algorithm progresses.

The first step of the algorithm is to fill an internal buffer of size 2^k with records having distinct keys. Thus we seek to convert L into the form BA, where B is the buffer and $A = L - B$ such that $STABLESORT(L) = STABLEMERGE(B, STABLESORT(A))$. To do this, we perform a left-to-right scan of L, "growing" B as a sorted sublist. The first record of L is placed in B. As we scan the ith record, $i > 1$, we conduct a binary search

Curiously, these works focus only on establishing the existence of algorithms, and include no constant of proportionality analysis. However, we have studied the intricate details of the general method they use, as described in full in [Tr1], and have found that they yield a worst-case key-comparison and record-exchange grand total in excess of $15n$. Perhaps more importantly, we observe that our approach is dramatically simpler. As one rough estimate of the cost of stability, we remark that the key-comparison and record-exchange total of our underlying, unstable merge was bounded above by $3.5n$ in [HL1].

5. Stable In-Place Sorting

5.1 The Direct Merge-Sort Approach

We can, naturally, now take the simple course suggested in [Tr2] and directly use our stable, in-place linear-time merge as a subroutine for merge-sorting. We observe that this gives rise to a key-comparison and record-exchange total bounded above by $7\ n \log_2 n$, plus lower order terms (elementary combinatorics guarantees that our merge's sublinear terms, the largest of which is $O(\sqrt{n} \log n)$, give rise to terms of at most $O(n)$ in the resulting "divide and conquer" merge-sort scheme).

As with the traditional, memory-dependent merge-sort, this scheme will be more effective in practice if we use a less-complicated, quadratic-time sort when subfile sizes fall below some established "break-even" point that depends on a number of factors local to a given sorting environment. Even so, a lot of time will be spent in extracting an internal buffer at each call of the merge subroutine. We shall demonstrate in the next subsection that this effort can be avoided as long as no single key is permitted to dominate the file.

(the first sublist does not become disordered) and fewer than n exchanges (each of the $\sqrt{n}-1$ $BLOCKSWAP$ invocations puts $\sqrt{n}$ records in position). For arbitrary list and sublist sizes, the $ROTATE$ used to move the left small block needs at most n_2 exchanges. For the series-merging phase, fewer than n comparisons and n exchanges suffice. Finally, since we use a binary search to locate the insertion points for merging back the buffer, this operation needs only $O(\sqrt{n}\log n)$ comparisons and no more than $(n-\sqrt{n})+\sum_{i=1}^{\sqrt{n}} i < 1.5n$ exchanges. Therefore, we are guaranteed a worst-case key-comparison and record-exchange grand total of something less than $6.5n$.

For the special case in which we exhaust the first sublist before filling the buffer, the only operation whose constant is affected is the series-merge, which is implemented with a series of $BLOCKMERGE$ invocations. In this simple scheme, we work from left to right, always merging a series of one or more blocks with the single block to its immediate right. Since there are only $s < \sqrt{n}$ distinct keys in the first sublist, we shall in this case employ two binary searches to locate insertion points for merging (the first search on the series or block from the first sublist, the second search on the other) and require only $O(\sqrt{n}\log n)$ comparisons. As for exchanges, we observe that no record is moved to the right more than once. Each distinct key in the first sublist gives rise to at most one invocation of $ROTATE$, except when such a key is represented in two distinct first-sublist series (it cannot be in three or more), which can happen at most $s/2$ times, each time giving rise to at most one more $ROTATE$ operation. Since each $ROTATE$ moves at most n/s records to the left, a total of at most $n+1.5s(n/s)=2.5n$ exchanges are required. Hence, we are assured a worst-case key-comparison and record-exchange grand total bounded above by $7n$.

For comparison, consider previously-published methods to solve this problem [SS, Tr2].

after $BLOCKSORT$ is finished, when a second-sublist series should precede it.

4.5 Constant of Proportionality Bounds

In an effort to measure the practical potential of this stable, optimal time and space merge, we shall study the number of key comparisons and record exchanges it demands. These two primitives are generally regarded as by far the most time consuming operations for internal file processing, requiring storage-to-storage instructions for many architectures. Since it is possible to count them independently from the code of any particular implementation, their total gives a meaningful estimate of the size of the linear-time constant of proportionality for the algorithm we have devised. (As for the issue of constant extra space, a careful review of our method reveals that a couple of dozen additional storage cells is all we need for use as pointers and counters.)

We now proceed to derive a worst-case upper bound on the key-comparison and record-exchange sum. For simplicity, we allow for a (possibly unrealizable) worst-case scenario, implying that the figures we produce may be rather conservative upper bounds. (This is offset to some extent, especially for small inputs, by the fact that we are ignoring operations bounded above by lower-order terms. For example, sorting the buffer can be accomplished in-place with heap-sort in $O(\sqrt{n}\log n)$ time. In fact, our main idea for achieving stability needs only $O(\sqrt{n})$ time.) Let $n_1(n_2)$ denote the size of the first (second) sublist, and thus $n_1 + n_2 = n$.

Consider the general case, in which there are plenty of distinct keys to fill the buffer. Extracting the buffer uses at most n_1 comparisons and n_1 exchanges. By selecting blocks from right to left, our stable $BLOCKSORT$ requires at most $\sum_{i=1}^{n_2/\sqrt{n}} i < n_2/2$ comparisons stable $BLOCKSORT$ described below, it is of no help in the merging phase. Fortunately, however, such a small number of distinct keys in the first sublist ensures that we can, in $O(n)$ time, stably merge the sorted series of blocks with a left-to-right series of backward $BLOCKMERGE$ operations, each using the proper insertion point, which is the leftmost if the left series is from the second sublist, and the rightmost otherwise. Since the buffer does not in this scheme end up adjacent to the unmerged suffix of the right series when the left is exhausted, we use a pointer to indicate this boundary, namely, the location of the leftmost record in the right series not moved by $ROTATE$. (Although this method is easy to implement, it is not perhaps obvious that it takes only linear time. See Section 4.5.) We then stably merge the buffer with the remainder of the list with a forward $BLOCKMERGE$ using leftmost insertion points.

Our $BLOCKSORT$ implementation must be stable. This is easily achieved by first invoking $SORT$ on the buffer and then using it to "remember" the original block sequence. That is, we exchange each buffer element with the proper block's tail before blocks are rearranged, and then undo each exchange as the corresponding block is selected by $BLOCKSORT$. This simple scheme, a variation of the "segment insertion process" used in [Tr2], thus restores the tails in time to perform the series-encoding task (as described in Section 4.3) as the sort progresses.

Finally, for lists and sublists of arbitrary sizes, we employ a method analogous to the one we used for unstable merging [HL1]. This gives potential rise to one small block (of size less than $\lfloor\sqrt{n}\rfloor$) at the extreme right end of the list, and one at the left end to the immediate right of the buffer. For the right block, no modification is necessary. For the left one, we observe that in general a $ROTATE$ may be necessary to insert the block in its proper place, We thus make use of the fact that one or more blocks from the first sublist must lie between the blocks we have modified, insuring that we can correctly decode the series delimiters during the series-merging phase. That is, it directly follows that the head of the rightmost block of a second-sublist series will temporarily have a key strictly greater than that of the record to its immediate right, while the tail of the leftmost block of a second-sublist series will temporarily have a key strictly less than that of the tail of the block to its immediate left. Of course, with this stringent mechanism for defining each distinct series to be merged, we do not employ the simpler criteria used in the unstable merge to locate series. Now, as we merge, we undo the exchanges that delimit the series and always break ties in favor of the series from the first sublist. $O(\sqrt{n})$ time and $O(1)$ space are sufficient for this scheme.

4.4 Other Relevant Details

We attempt to load the internal buffer with distinct-keyed records as follows. We begin at the right end of the first sublist and scan to the left. When a comparison of adjacent keys reveals that the leftmost copy of a key has been found, that record is coalesced into the buffer. Other records are exchanged with the rightmost current buffer element. Therefore, the buffer begins with size zero and grows as we "roll" it to the left. When it has attained size $\sqrt{n}$, we invoke $ROTATE$ to left-justify it. At the end of the series-merging phase (the buffer is now right-justified), we stably merge the buffer with the remainder of the list with a backward $BLOCKMERGE$ using leftmost insertion points.

In the event that we exhaust the first sublist without filling the internal buffer, we must employ fewer but larger blocks. Specifically, if we obtain only $s < \sqrt{n}$ buffer elements, then we use s blocks, each of size at most $\lceil n/s \rceil$. Although this permits the use of the

the right otherwise.)

Additionally, we must be wary of a few other details that, if neglected, can compromise stability. For example, we need to load the buffer with records having distinct keys, if that is possible, since the buffer's contents are arbitrarily permuted during the series-merging phase. Correspondingly, we must provide for the special case in which there are not enough distinct keys to fill the buffer. We also want to make the $BLOCKSORT$ subprogram of the block-sorting phase stable, because otherwise a large collection of homogeneous blocks may be unpredictably rearranged. Finally, as with the fundamental, unstable merge [HL1], we need to specify implementation details for handling lists and sublists of arbitrary sizes.

4.3 The Main Idea

Since the possibility of troublesome homogeneous blocks prevents the use of any simple scheme for identifying individual blocks as to their origin, we shall seek instead to devise a strategy by which we can distinguish a *series* of consecutive blocks from the first sublist from a *series* of consecutive blocks from the second. To this end, it is enough if we can be sure of the first and last block in every series from the second sublist only.

We shall encode this information in L during the (stable) block-sorting phase and decode it during the series-merging phase. To encode, we use two memory cells, one to point to the (post-sorting) position of the leftmost block of the leftmost second-sublist series and one to point to the (post-sorting) position of the rightmost block of the rightmost second-sublist series. As the sort phase progresses, we mark each series by exchanging the head of the rightmost block of one second-sublist series with the tail of the leftmost block of the next second-sublist series.

We continue this process of locating series of records and merging them until we reach a point were only one such series exists, which we merely shift left, leaving the buffer in the last block. A sort of the buffer completes the merge of L.

$O(1)$ space suffices for this procedure, since the buffer was internal to the list, and since only a handful of additional pointers and counters are necessary. $O(n)$ time suffices as well, since the block sorting, the series merging and the buffer sorting each require at most linear time.

4.2 Obstacles to Stability

The primary problem to be addressed in order to achieve stability is the need to be able to distinguish blocks as to whether each originated in the first or the second sublist. This is a more difficult task than it may seem at first blush. A number of schemes will do if each block has different keys at its head and its tail. For example, we could simply make a temporary swap of the records at the head and tail of a block if and only if it originated in, say, the second sublist. (Such a swap would be made during the block-sorting phase and undone during the series-merging phase.) The real problem lies with *homogeneous* blocks, those in which every record in the block has the same key as every other record in the block. To illustrate this conundrum, suppose we know by some artifice that block $i > 2$ originated in the first sublist, but only that block $i-1$ is homogeneous. Also, suppose the key of the tail of block $i-1$ equals the key of the head of the block i, but is strictly less than the key of the tail of block i. In this circumstance, stability is jeopardized since we cannot determine whether the head of block i should be merged to the left or to the right of the records of block $i-1$. (It should go to the left if block $i-1$ originated in the second sublist, but to time and $O(1)$ space implementation details necessary for handling arbitrary inputs.

Let us suppose that n is a perfect square, and that we have already permuted the records of L so that $\sqrt{n}$ largest-keyed records are at the front of the list (their relative order there is immaterial), followed by the remainders of the two sublists, each of which we now assume contains an integral multiple of $\sqrt{n}$ records in nondecreasing order.

Therefore, we view L as a series of $\sqrt{n}$ blocks, each of size $\sqrt{n}$. We will use the leading block as an internal buffer to aid in the merge. Our first step is to invoke $BLOCKSORT$ on the $\sqrt{n}-1$ rightmost blocks, after which their tails form a nondecreasing key sequence. (In this setting, selection sort requires only $O(n)$ key comparisons and record exchanges.) Records within a block retain their original relative order.

Next, we locate two series of records to be merged. The first series begins with the head of block 2 and terminates with the tail of block i, $i \geq 2$, where block i is the first block such that the key of the tail of block i exceeds the key of the head of block $i+1$. The second series consists solely of the records of block $i+1$. We now use the buffer to merge these two series. That is, we repeatedly compare the leftmost unmerged record in the first series to the leftmost unmerged record in the second, swapping the smaller-keyed record with the leftmost buffer element. Ties are broken in favor of the leftmost series. (In general, the buffer may be broken into two pieces as we merge.) We halt this process when the tail of block i has been moved to its final position.

We now locate the next two series of records to be merged. This time, the first begins with the leftmost unmerged record of block $i+1$ and terminates as before for some $j \geq i$. The second consists solely of the records of block $j+1$. We resume the merge until the tail of block j has been moved.

is used to find the leftmost *insertion point* for the leftmost record of the first block. That is, assuming $KEY(i+h) < KEY(i) \leq KEY(i+h+k-1)$, the displacement p is computed for which $KEY(i+h+p) < KEY(i) \leq KEY(i+h+p+1)$, followed by an invocation of $ROTATE(i, h+p, h)$. The first record of the shorter block and all records to its left are now merged. The merge is completed by iterating this operation until one of the blocks is exhausted, resulting in a time complexity of $O(h^2+k)$. (There are at most $O(h\,log\,k)$ comparisons. Records from the shorter block are moved no more than h times, while records from the longer block are moved only once.) Of course, if $h > k$, then $BLOCKMERGE$ is better off to merge the second block *backward* into the first in $O(h+k^2)$ time.

4. Stable In-Place Merging

4.1 A Review of the Fundamental, Unstable Merge

Suppose L contains two sublists to be merged, each with its keys in nondecreasing order. In [HL1] we presented a fast and surprisingly simple algorithm for (unstably) merging in linear time and constant extra space. Even without "tinkering" with it to achieve an especially efficient implementation, its average run time on large lists exceeds that of the standard, widely-used merge (which is free to exploit $O(n)$ temporary extra memory cells) by less than a factor of two. Aspects that contribute to its straightforwardness include a rearrangement of blocks before a merging phase is initiated and an efficient pass of the internal buffer across the list to reduce unnecessary record movement.

We now briefly review the central features of this $O(n)$ time and $O(1)$ extra space method, with a number of simplifying assumptions made about L to facilitate discussion. We refer the reader to [HL1] for a complete exposition of the algorithm, an example, and the $O(\sqrt{n})$

being compared.

From these primitive operations, we construct a few $O(1)$ space useful subprograms for dealing with *blocks*. Let us define a block to be a set of records from L with consecutive indices. The *head* of a block is the record with the lowest index (or, informally, the "leftmost" record in the block); the *tail* of a block is the record with the highest index (the "rightmost" record in the block). The procedure $BLOCKSWAP(i, j, h)$ exchanges a block of h records beginning at index i with a block of h records beginning at index j in $O(h)$ time. We specify that blocks do not partially overlap (i.e., if $i \neq j$ then $h \leq |i - j|$) and that, when $BLOCKSWAP$ is finished, records within a moved block retain the order they possessed before $BLOCKSWAP$ was invoked. A block of h records beginning at index i is sorted in nondecreasing order by the procedure $SORT(i, h)$. The procedure $BLOCKSORT(i, h, p)$ uses $BLOCKSWAP$ to rearrange the p consecutive blocks, each with h records, beginning at index i so that their tails are sorted in nondecreasing order. To reduce unnecessary record movement, an important consideration when records are relatively long, we insist that $BLOCKSORT$ use the $O(p^2 + ph)$ time straight selection sort [Kn1].

The procedure $ROTATE(i, h, \ell)$ rotates (circularly shifts) a block of h records, beginning at index i, ℓ places to the left. We assume that $ROTATE$ is implemented in the common fashion with three sublist reversals, thereby requiring no more than h invocations of $SWAP$.

Finally, a pair of consecutive blocks, each sorted in nondecreasing order, is merged with $BLOCKMERGE(i, h, k)$, where the first block contains h records beginning at index i and the second contains k records beginning at index $i+h$. $BLOCKMERGE$ uses $ROTATE$ to merge the shorter block into the longer one. For example, if $h \leq k$, then $BLOCKMERGE$ merges the first block *forward* into the second as follows. A binary search of the second block selection sorting both the buffer and the blocks (each sort involves $O(\sqrt{n})$ keys).

After the unstable method in [Kr] appeared, a stable procedure was proposed in [Ho] that, unfortunately, had the rather undesirable side-effect that records had to be alterable during its execution. Subsequently, a general algorithm for optimal time and space, stable merging and sorting was published [Tr1, Tr2], as was a technique for simplifying parts of its control structure [SS]. For the most part, however, these results have been of academic interest only, due primarily to their discouraging complexity and their prohibitively large time-complexity constants of proportionality.

More recent research efforts have begun to focus on simpler, more practical optimal time and space internal buffering and block rearranging strategies for unstable merging [DD2, HL1, MU] as well as for extracting duplicates from a sorted list [HL2] and for all of the binary set and multiset operations on sorted lists [HL3], with potential application to a number of file processing problems.

3. Notation, Definitions and Useful Subprograms

Let L denote a list (internal file) of n records, indexed from 1 to n. An algorithm for rearranging the order of the records of L is said to be *stable* if it ensures that, when it is done, records with identical keys retain the relative order they had before the algorithm began. We use $KEY(i)$ as a shorthand to denote the key of the record with index i. Only the two common $O(1)$ time and space primitive operations are assumed, namely, record exchanges and key comparisons. The exchange procedure, $SWAP(i,j)$, directs that the ith and jth records are to be exchanged. The comparison functions, for example $KEY(i) < KEY(j)$, return the expected Boolean values dependent on the relative values of the keys such as tape: larger initial runs mean fewer passes of the file (the common replacement-selection method is unstable), and more available memory for buffer space can mean less time consumed in each pass.

In the next section, we discuss pertinent background information and related work. Section 3 comprises the definitions and notational conventions we shall need to present and analyze our algorithms. In Section 4, we review the fundamental, optimal time and space unstable merge of [HL1] and define our modifications that ensure stability. Also, to provide an upper bound on the resultant procedure's worst-cast constant of proportionality, we prove that the total number of key comparisons and record exchanges required never exceeds $7n$ (plus lower-order terms). Section 5 extends our work to the problem of optimal time and space stable sorting in a nontrivial way. We devise an alternative to the obvious merge-sort strategy, and show that it never needs more than $2.5\, n \log_2 n$ (plus lower-order terms) key comparisons and record exchanges. In the final section, we draw a few conclusions from this effort and pose questions that we believe merit further investigation.

2. Related Work

The general approach that we shall employ inherently relies on the notions of *internal buffering* and *block rearranging*, and can be traced back to the seminal work on unstable merging described in [Kr]. Simply stated, with this approach we attempt to view a list of n records as a sequence of $O(\sqrt{n})$ blocks, each of size $O(\sqrt{n})$. This allows us to employ one block as an internal buffer to aid in rearranging or otherwise manipulating the other blocks in constant extra space. Since only the contents of the buffer and the relative order of the blocks need be out of sequence, linear time is sufficient to perform a merge with the aid of in $O(1)$ extra space are either unstable or require $\Omega(n^2)$ time. A number of stable merging schemes that use more than linear time or more than constant extra space have been suggested [Ca, DD1, DD3, DD4, Wo], as have several stable sorting strategies that use more than $O(n \log n)$ time or more than $O(1)$ extra space [De, Mo, Pr, Ri]. Also, a routine that dynamically alters keys has been defined [Ho], but is thus applicable only to files in which keys are explicitly stored within records. The only previously-known general method for stably merging and sorting in *both* optimal time and optimal extra space [Tr1, Tr2] is widely regarded as a result of purely theoretical interest [Kn2, Tr3], since it is exceedingly complex and its time-complexity constant of proportionality is so huge that it hasn't even been derived. (Recent modifications have been suggested that simplify parts of this method, but its overall constant of proportionality remains prohibitively large and unbounded [SS].) This contrasts poorly with unstable merging, where much progress has been made in achieving practical and straightforward optimal time and space methods and with unstable sorting, where the simple heap-sort algorithm suffices.

The main result of this paper is a relatively simple, efficient and general scheme for stable merging (and thus stable merge-sorting) in optimal time and space. Our method is based on our recently-reported algorithm for fast, in-place unstable merging [HL1]. We also present an even more streamlined $O(n \log n)$ time and $O(1)$ extra space stable sorting procedure for files for which there is a reasonable limit on the number of times each key can appear. Significantly, and unlike previously-reported schemes to solve these problems, we derive explicit upper bounds on the number of key comparisons and record exchanges our methods require. Note that these strategies may be especially useful for operations such as stable polyphase, balanced or cascade merge-sorting with external storage media

1. Introduction

It is a well-recognized phenomenon that no matter how much main memory (also known as core memory, directly-addressable memory, non-virtual memory or real memory) is made available to a collection of users and systems, it seems never to be enough to satisfy everyone completely. Although main memory is often rather cavalierly regarded as an inexpensive resource, its availability is in fact critical in many applications. We are reminded of the following passage by the witty and imaginative science writer D.E.H. Jones [Jo]:

> *Let's assume that the brain, like most computers, stores intelligence (programs) and memory (data) in the same form and distributed throughout the same volume. Then the more space is taken up by data the less is available for programs and working space. Clearly as life progresses and memories multiply, there must come a time when programs and working space get squeezed. This must be senility.*

Naturally, main memory should be allocated and managed carefully to avoid thrashing [CD] and other forms of *computer senility.* This is particularly true for heavily-used operations like merging and sorting, known to dominate a large portion of all available execution time over broad classes of computer systems [Kn1]. This is even more evident when performing these operations over enormous external files. In such an environment, the overall processing time is frequently determined not by the speed of the algorithm used to process file segments internally, but rather by the ways in which available main memory can be used to accommodate more and larger buffers, thereby increasing device and channel parallelism while decreasing the number of I/O transfers required.

Unfortunately, for stable merging and sorting, the obvious algorithms that work in asymptotically optimal time ($O(n)$ and $O(n \log n)$, respectively) waste a whopping $\Omega(n)$ extra memory cells for temporary storage. Conversely, the conspicuous ways to merge and sort

Accepted for Publication in *The Computer Journal*

Fast Stable Merging and Sorting in Constant Extra Space[*]

Bing-Chao Huang[†] and Michael A. Langston[‡]

Abstract. In an earlier research paper [HL1], we presented a novel, yet straightforward linear-time algorithm for merging two sorted lists in a fixed amount of additional space. Constant of proportionality estimates and empirical testing reveal that this procedure is reasonably competitive with merge routines free to squander unbounded additional memory, making it particularly attractive whenever space is a critical resource. In this paper, we devise a relatively simple strategy by which this efficient merge can be made stable, and extend our results in a nontrivial way to the problem of stable sorting by merging. We also derive upper bounds on our algorithms' constants of proportionality, suggesting that in some environments (most notably external file processing) their modest run-time premiums may be more than offset by the dramatic space savings achieved.

[*] A preliminary version of a portion of this paper was presented at the International Conference on Computing and Information, held in Toronto, Ontario, Canada, in May 1989.

[†] Department of Computer Science, University of South Carolina, Columbia, SC 29208. This author's research has been supported in part by the Washington State University Graduate Research Assistantship Program.

[‡] Department of Computer Science, University of Tennessee, Knoxville, TN 37996–1301. This author's research has been supported in part by the National Science Foundation under grants ECS–8403859 and MIP–8603879, and by the Office of Naval Research under contract N00014–88–K–0343.