On Optimal and Efficient in Place Merging

Pok-Son Kim[1]* and Arne Kutzner[2]

[1] Kookmin University, Department of Mathematics, Seoul 136-702, Rep. of Korea
pskim@kookmin.ac.kr
[2] Seokyeong University, Department of E-Business, Seoul 136-704, Rep. of Korea
kutzner@skuniv.ac.kr

Abstract. We introduce a new stable in place merging algorithm that needs $O(m \log(\frac{n}{m}+1))$ comparisons and $O(m+n)$ assignments. According to the lower bounds for merging our algorithm is asymptotically optimal regarding the number of comparisons as well as assignments. The stable algorithm is developed in a modular style out of an unstable kernel for which we give a definition in pseudocode.
The literature so far describes several similar algorithms but merely as sophisticated theoretical models without any reasoning about their practical value. We report specific benchmarks and show that our algorithm is for almost all input sequences faster than the efficient minimum storage algorithm by Dudzinski and Dydek. The proposed algorithm can be effectively used in practice.

1 Introduction

Merging denotes the operation of rearranging the elements of two adjacent sorted sequences of size m and n, so that the result forms one sorted sequence of $m+n$ elements. An algorithm merges two sequences *in place* when it needs $O(1)$ bits additional space. It is regarded as *stable*, if it preserves the initial ordering of elements with equal value.
There are two significant lower bounds for merging. The lower bound for the number of assignments is $m+n$ because every element of the input sequences can change its position in the sorted output. As shown by Knuth in [1] the lower bound for the number of comparisons is $\Omega(m \log(\frac{n}{m}+1))$, where $m \leq n$.
So far there are three publications about optimal stable in place merging. The work of Symvonis [2] shows how to get an optimal algorithm by combining several given concepts but contains no information about the involved asymptotic constants or implementation aspects. Geffert et. all present in [3] a rather complex algorithm together with its asymptotic constants, but there are no notes regarding any successful implementation or benchmarking. Chen [4] simplified Geffert's algorithm for the price of slightly worse asymptotic constants but also without any remarks about a concrete implementation. All three publications

* This work was supported by the Kookmin University research grant in 2005.

have some resemblance. They take the algorithm from Mannila and Ukkonen [5] as starting point, rely on the concept of an internal buffer introduced by Kronrod in [6] and develop a stable algorithm out of an unstable one. We will follow this path but with the focus on an improved stable algorithm as well as concrete benchmarking. The proposed stable algorithm can be effectively used in practice as shown by the fact that it can compete with the algorithm of Dudzinski and Dydek [7] that is used as foundation of the `merge_without_buffer` function contained in the C++ Standard Template Libraries (STL) [8].
Significant older works in the area of in place merging are the publications of Pardo [9], Salowe and Steiger [10] and Huang and Langston [11]. All algorithms introduced there are asymptotically optimal regarding the number of assignments, but lack in meeting the lower bound for comparisons. Another class of merging algorithms are the *minimum storage* algorithms presented in [7] and [12] which both rely on $O(\log^2(m+n))$ bits of extra storage. The latter two algorithms are effective in practice and simply structured, but they are not asymptotically optimal regarding the number of assignments.
We will begin with the introduction of our notation and some toolbox algorithms, followed by the presentation of an unstable core algorithm. Afterwards the unstable core algorithm is extended to an unstable in place algorithm which in turn is extended to a stable in place algorithm. We will report some benchmarks and finish with a short conclusion.

2 Notation / Algorithm Toolbox

We now introduce some notations that we will use throughout the paper. Let u and v two ascending sorted sequences. We define $u \leq v$ ($u < v$) iff. $x \leq y$ ($x < y$) for all elements $x \in u$ and for all elements $y \in v$. $|u|$ denotes the size of the sequence u. Unless stated otherwise, m and n ($m \leq n$) are the sizes of two input sequences u and v respectively.

We will use six other algorithms as subcomponents. We now briefly introduce these algorithms and their complexity (A summary is given in Tab. 1):

(1) *Hwang and Lin* [13] introduced a merging-algorithm that is optimal regarding the number of comparisons as well as assignments. Unfortunately their algorithm is not in-place, it relies on an external buffer of size m when the merging shall be achieved by applying a linear number of assignments only. The algorithm granulates the longer input sequence into segments of size $2^{\lfloor \log(n/m) \rfloor}$ and uses a smart combination of a sequential search together with several binary searches for staying asymptotically optimal regarding the number of comparisons. Hwang and Lin's algorithm can be modified so that it works in place, but for the price of m^2 assignments. The modified form avoids the usage of an external buffer by using repeated rotations instead. Geffert et al. give a detailed description of that variant in [3].
(2) *Block Swapping* denotes the operation of exchanging the contents of two

Algorithm	Arguments	Comparisons	Assignments
Hwang and Lin	u, v with $\lvert u\rvert \leq \lvert v\rvert$ let $m = \lvert u\rvert, n = \lvert v\rvert$	$m(t+1) + n/2^t$ where $t = \lfloor \log(n/m) \rfloor$	
(1) - ext. buffer			$2m + n$
(2) - m rotat.			$n + m^2 + m$
Block Swapping	u, v with $\lvert u\rvert = \lvert v\rvert$	-	$3\lvert u\rvert$
Floating Hole	u, v with $\lvert u\rvert = \lvert v\rvert$ (element x is in front of u)	-	$2(\lvert u\rvert + 1)$
Block Rotation	u, v	-	$\lvert u\rvert + \lvert v\rvert + \gcd(\lvert u\rvert, \lvert v\rvert) \leq 2(\lvert u\rvert + \lvert v\rvert)$
Binary Search	u, x (searched element)	$\lfloor \log \lvert u\rvert \rfloor + 1$	-
Insertion Sort	u, let $m = \lvert u\rvert$	$\frac{m(m-1)}{2} + (m-1)$	$\frac{m(m+1)}{2} - 1$

Table 1. Complexity of the Toolbox-Algorithms

(not necessarily adjacent) blocks u and v with $|u| = |v|$. *Floating Hole* denotes a technique that can sometimes be applied in order to reduce the number of assignments necessary for achieving a block rearrangement. In our algorithms we will have to accomplish a rearrangement from $\ldots xu\,p\,v\ldots$ to $\ldots vx\,p\,u\ldots$, where x is a single element, u and v are blocks of equal size and p is some arbitrary subsequence. [3] gives a detailed description for both operations and their complexity.
(3) Let u and v be two adjacent blocks of not necessarily equal size. The circular rearrangement from $\ldots uv\ldots$ to $\ldots vu\ldots$, is called a *Block Rotation*. If we have an intermediate storage of one element only we need at least $|u|+|v|+\gcd(|u|, |v|)$ assignments for accomplishing a block rotation. Here $\gcd(a, b)$ denotes the greatest common divisor of two positive integers. An algorithm that meets this lower bound is presented in [7].
(4) *Binary Search* and *Insertion-Sort* are two standard algorithms described in almost all introducing literature about algorithms (e.g. [14]).

3 The Core Algorithm

We now give the definition of our unstable core algorithm that relies on extra storage of size $\lfloor \sqrt{m} \rfloor$ for local merges.

Algorithm 1: Unstable-Core-Merge
Let $k = \lfloor \sqrt{m} \rfloor$ and $l = \lfloor m/k \rfloor$. We granulate the sequence u into blocks $u_0 u_1 \ldots u_l$, so that all blocks u_i with $0 < i \leq l$ have size k. The first block u_0 gets the size $m - l * k$. (u_0 is empty in the case $l * k = m$). Let $u_i = b_i x_i$ for all i ($0 \leq i \leq l$), where x_i corresponds to the last element of u_i. If u_0 is empty, then b_0 and x_0 are empty as well. We separate the sequence v into $l + 2$ sections $v = v_0 v_1 \ldots v_l v_{l+1}$ using the x_i ($0 \leq i \leq l$) , so that we get for all i: $v_i < x_i \leq v_{i+1}$. Using these granulations of v and u we rearrange our input

Algorithm 1 Unstable Core Algorithm

```
UNSTABLE-CORE-MERGE($A, first1, first2, last$)
▷ $u$ is in $A[first1 : first2 - 1]$, $v$ is in $A[first2 : last - 1]$
$m \leftarrow first2 - first1$; $k \leftarrow \lfloor \text{sqrt}(m) \rfloor$; $delta \leftarrow 0$;
if $m$ mod $k = 0$
   then $blockEnd \leftarrow first1 + k$
   else  $blockEnd \leftarrow first1 + (m \bmod k)$

while true
      do ▷ Processing of the current minimal block
         $b \leftarrow$ BINARY-SEARCH($A$, $first2$, $last$, $A[blockEnd - 1]$)
         $to \leftarrow b - (first2 - blockEnd)$
         if $to > first2$
            then BLOCK-ROTATION($A$, $blockEnd - 1$, $first2$, $b$)
            else  FLOATING-HOLE($A$, $blockEnd$, $first2$, $b - first2$)
                  $delta \leftarrow (b - first2 + delta) \bmod k$
         HWANG-AND-LIN($A$, $first1$, $blockEnd - 1$, $to - 1$)
         $first2 \leftarrow b$; $first1 \leftarrow to$
         if $first1 \geq first2$
            then break ▷ No more blocks to be placed - leave the while-loop

         ▷ Search the next minimal block
         $t \leftarrow first1 + k - delta$; $e \leftarrow first2 - delta$
         if $delta > 0$
            then $startMin \leftarrow$ SEARCH-MINIMAL-BLOCK($A$, $k$, $t$, $e$, $e$)
            else  $startMin \leftarrow$ SEARCH-MINIMAL-BLOCK($A$, $k$, $t$, $e$, $first1$)
                  $t \leftarrow first1$

         ▷ Move the minimal block to the front of sequence $q$
         if $startMin = e$
            then BLOCK-SWAP($A$, $t$, $e$, $delta$)
                 BLOCK-ROTATION($A$, $first1$, $t$, $first1 + k$)
            else BLOCK-SWAP($A$, $t$, $startMin$, $k$)
                 BLOCK-ROTATION($A$, $first1$, $t$, $t + k$)
         $blockEnd \leftarrow first1 + k$
```

sequences to $b_0 v_0 x_0 b_1 v_1 x_1 \dots b_l v_l x_l v_{l+1}$. Eventually we get the desired sorted result by local merging of all pairs $b_i v_i$ $(0 \leq i \leq l)$. Please note that all x_i are at their final position after the rearrangement-step and do not need to be part of the local merges.
In order to keep the optimality the rearrangement must be achieved by applying a linear number of assignments only. The following technique can be used to do so. It is similar to the following method described by Mannila and Ukkonen in [5]:
The rearrangement happens in a sequential style, it starts with block u_0 (u_1 if

Pseudocode Definition	Description of the Arguments
HWANG-AND-LIN$(A, first1, first2, last)$	u is in $A[first1 : first2 - 1]$, v is in $A[first2 : last - 1]$
BSEARCH$(A, first, last, x)$	delivers the position of the **first** occurrence of x in $A[first : last-1]$
BLOCK-SWAP$(A, pos1, pos2, len)$	u is in $A[pos1 : pos1 + len]$, v is in $A[pos2 : pos2 + len]$
FLOATING-HOLE$(A, pos1, pos2, len)$	u, v as in BLOCK-SWAP, element x in $A[pos - 1]$
BLOCK-ROTATE$(A, first1, first2, last)$	u, v as in HWANG-AND-LIN

Table 2. Pseudocode Definitions of the Toolbox Algorithms

u_0 is empty) and continues by placing the blocks in increasing order one by one. During the rearrangement all unprocessed blocks, this means blocks that are not moved to their final position, stay together as a group, but we allow that these blocks become interleaved and rotated as a complete segment.
Let us now assume that we have already successfully processed all blocks $u_0 \dots u_j$ with $(0 \le j < l)$. Then we have some sequence $p\ q\ v_{j+1} \dots v_{l+1}$, where $p = b_0 v_0 x_0 b_1 v_1 x_1 \dots b_j v_j x_j$ contains all blocks already processed and $q = c'' u'_1 \dots u'_{l-j-1} c'$ comprises the unprocessed blocks $u_{j+1} \dots u_l$ in some interleaved form. Additionally, due to the rotation, one unprocessed block can be split into two parts, this is $c'c''$. To place the next block u_{j+1}, we have first to find the position of that block in q. Due to the increasing order of the elements in u, we have to find the block with the smallest elements in order to find b_{j+1}. We can do so by looking for the block with the smallest first and last element. Depending on the result of this search, we have to distinguish two different cases:
(*Case 1*) The minimal block is $c'c''$: We split u'_1 into $d'd''$, so that $|d'| = |c'|$ and $|d''| = |c''|$. Then we exchange c' and d' in order to get $q = c''c'd''u'_2 \dots u'_{l-j-1}d'$. Afterwards we rotate $c''c'$ to $c'c''$ and get $q = u_{j+1}d''u'_2 \dots u'_{l-j-1}d'$.
(*Case 2*) The minimal block is in $u'_1 \dots u'_{l-j-1}$, let u'_i $(1 \le i < l - j)$ be the minimal block. Then we exchange u'_1 and u'_i in order to get $q = c''u'_iu'_2 \dots u'_1 \dots u'_{l-j-1}c'$. Afterwards we rotate $c''u'_i$ to u'_ic'' and get $q = u_{j+1}c''u'_2 \dots u'_1 \dots u'_{l-j-1}c'$.
Hence, after moving u_{j+1} to the front position we have some sequence $p\ b_{j+1}x_{j+1}q'v_{j+1} \dots v_{l+1}$ $(q = b_{j+1}x_{j+1}q')$. Now we will move v_{j+1} to its final position just in front of x_{j+1}. Once more we have to distinguish two cases:
(*Case 1*) $|v_{j+1}| \ge |q'|$: We use a rotation in order to get $v_{j+1}x_{j+1}q'$ out of $x_{j+1}q'v_{j+1}$.
(*Case 2*) $|v_{j+1}| < |q'|$: We split q' into $q'_1q'_2$ so that $|q'_1| = |v_{j+1}|$ and use a floating-hole operation to get $v_{j+1}x_{j+1}q'_2q'_1$ out of $x_{j+1}q'_1q'_2v_{j+1}$. Please note that $q'_1q'_2$ is a rotated form of q' merely.

Alg. 1 gives an implementation for the UNSTABLE-CORE-MERGE algorithm in pseudocode. Table 2 comprises the pseudocode definitions for all toolbox algorithms. The pseudocode conventions are taken from [14].

Theorem 1. *The* Unstable-Core-Merge *algorithm needs* $O(m\log(\frac{n}{m}+1))$ *comparisons and* $O(m+n)$ *assignments.*

Proof. The $l+1$ calls of Hwang and Lin's algorithm need less than $\Sigma_{i=0}^{l}(q_i\log(\frac{p_i}{q_i}+1))+q_i = O(m\log(\frac{n}{m}+1))$ comparisons and $\Sigma_{i=0}^{l}(2q_i+p_i) \le 2m+n$ assignments, where $p_i = \max\{|u_i|,|v_i|\}$ and $q_i = \min\{|u_i|,|v_i|\}$. Further, since $\lfloor\sqrt{m}\rfloor\,(\lfloor\log n\rfloor+1) \le \sqrt{m}(\log n+1) = m\cdot\frac{\log n}{\sqrt{m}}+\sqrt{m} \le m\cdot(\log n-\log m)+\sqrt{m} = O(m\log\frac{n}{m})$, the $l+1$ calls of the binary search need $O(m\log(\frac{n}{m}+1))$ comparisons. The l searches of the minimal block consume $\Sigma_{i=1}^{l}2i \le m+\sqrt{m} \le 2m$ comparisons. The l extractions of the minimal block need $l(7k) \le 7m$ assignments. The $l+1$ movements of the minimal block need less than $\Sigma_{i=0}^{l}4|v_i| \le 4n$ assignments. So, altogether we have $O(m\log(\frac{n}{m}+1))$ comparisons and $O(m+n)$ assignments. $\square$

3.1 Extending the Core Algorithm to an Unstable in Place Algorithm

The Unstable-Core-Merge algorithm is asymptotically optimal, but it demands an extra storage of size $O(\lfloor\sqrt{m}\rfloor)$. We will now apply a technique called *internal buffer* for reducing the necessary extra storage to $O(1)$. The notion internal buffer is due to Kronrod and was first proposed in [6]. The basic idea is to use some particular area of the input sequences repeatedly as buffer and to accept that the area elements are disordered by this usage. At the end the internal buffer is sorted by applying some sorting algorithm and afterwards the buffer elements are merged by some way. Using this approach we now derive an unstable in-place algorithm from our core algorithm:

Algorithm 2: Unstable-In-Place-Merge (u, v)
We split the input sequence u into u_1u_2 so that $|u_1| = \lfloor\sqrt{m}\rfloor$. Let x be the last element of u_1. By applying a binary search we separate v into v_1v_2, so that $v_1 < x \le v_2$. We rearrange $u_1u_2v_1v_2$ to $u_1v_1u_2v_2$ using a block rotation. Then we merge u_2 and v_2 using the Unstable-Core-Merge algorithm (Alg. 1), where the embedded calls of Hwang and Lin's algorithm use the segment u_1 as buffer area. Because the elements of u_1 can be disordered during the last step, we afterwards sort them using Insertion-Sort. Finally we use the rotation based variant of Hwang and Lin's algorithm for merging the two segments u_1 and v_1.

Theorem 2. *The* Unstable-In-Place-Merge *algorithm needs* $O(m\log(\frac{n}{m}+1))$ *comparisons and* $O(m+n)$ *assignments.*

Proof. We have simply to count the additional operations. The unique additional binary search and call of Hwang and Lin's algorithm trivially doesn't change the asymptotic number of comparisons. Hwang and Lin's call poses $|v_1|+|u_1|^2+|u_1| = O(m+n)$ additional assignments. The final insertion sort needs $O(m)$ comparisons as well as assignments (see Table 1). So altogether the algorithm performs $O(m\log(\frac{n}{m}+1))$ comparisons and $O(m+n)$ assignments. $\square$

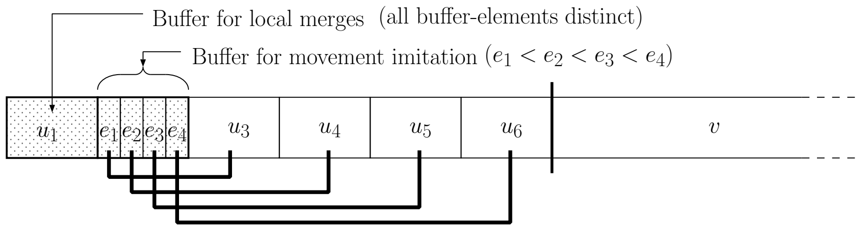
Buffer for local merges (all buffer-elements distinct)
Buffer for movement imitation ($e_1 < e_2 < e_3 < e_4$)
u_1
e_1
e_2
e_3
e_4
u_3
u_4
u_5
u_6
v

Fig. 1. Partitioning scheme (here for $|u| = 24$)

A merging algorithm is called *semi-stable* when it preserves the initial ordering of equal elements of at least one of either input-sequences. It is easy to check that none of the applications of toolbox algorithms in Unstable-In-Place-Merge changes the initial ordering of equal elements in v.

Corollary 1. Unstable-In-Place-Merge *is semi-stable.*

4 Deriving a Stable in Place Algorithm

The lack of stability in Unstable-In-Place-Merge is caused (1) by the block extraction in the lines 27-32 of Alg. 1 and (2) the usage of the first elements $\lfloor\sqrt{m}\rfloor$ of u as internal buffer. The block extraction raises stability-problems because there might be two blocks containing equal elements. Such two blocks can't be distinguished during the search of the minimal block and so we can't reconstruct their initial order. We will fix these problems as follows:
We extract $2\lfloor\sqrt{m}\rfloor$ *distinct* elements out of u and create 2 buffers of size $\lfloor\sqrt{m}\rfloor$ by moving these elements to the front of u. Please note that we can disorder and afterwards sort these buffers without losing stability. The first buffer will be used by the embedded calls of Hwang and Lin's algorithm, the second buffer will be used to keep track of the order of unprocessed blocks in u. To keep track we will apply a technique called *movement imitation* that is described by Symvonis in [2]. Movement imitation means that we establish a 1-to-1 correspondence between elements of the movement imitation buffer (mi-buffer) and u-blocks as shown in Fig. 1. Each time when we change the order of the u-blocks during the processing or extraction of a minimal block, we imitate this reordering in the mi-buffer. Hence, we can find the minimal block by searching for the minimal element in the mi-buffer.

Algorithm 3: Stable-In-Place-Merge (u, v)
We take the Unstable-In-Place-Merge algorithm as basis and apply the following modifications:
(1) We start by extracting two buffers of size $\lfloor\sqrt{m}\rfloor$ (mi-buffer and buffer for local merges) at the beginning of u, where all buffer-elements are distinct. Such buffer

extraction can happen by performing $O(m)$ comparisons and $O(m)$ assignments as described by Pardo in [9]. (2) We replace the search for the minimal block (lines 23-24 in Alg. 1) by a procedure using the mi-buffer. (3) Any u-block reordering must be imitated in the mi-Buffer. (4) We need a counter variable that counts the number of unprocessed blocks for maintaining the size of the mi-Buffer. (5) At the end we must sort and merge the two buffers extracted in the beginning, this replaces 2 corresponding tasks in the unstable algorithm.

Theorem 3. *The* STABLE-IN-PLACE-MERGE *algorithm needs* $O(m \log(\frac{n}{m}+1))$ *comparisons and* $O(m+n)$ *assignments.*

Proof. We have to check the effect of all modifications applied to the unstable in place algorithm. The extraction of a buffer of size $2\lfloor\sqrt{m}\rfloor$ in u needs $O(m)$ additional comparisons and $O(m)$ additional movements. The repeated search of the minimal block needs $\Sigma_{i=1}^{l} i \leq m$ comparisons. The management of the mi-buffer causes less than $l \cdot 2\lfloor\sqrt{m}\rfloor \leq 2m$ assignments. For the final sorting and merging the same argumentation can be applied as in Theorem 2. □

There might be the case that there are less than $2\lfloor\sqrt{m}\rfloor$ distinct elements in u and so, due to the lack of a buffer of appropriate size, the above algorithm fails. In order to give a solution for this case we first slightly extend the rotation-based variant of Hwang and Lin's algorithm as follows:
Instead of directly inserting an element x as in the original algorithm, we first extract a maximal segment of elements equal to x by a simple linear search. Afterwards we treat this segment as one element. This extension causes m additional comparisons at most but allows us to express the number of necessary assignments depending on the number of different elements in u.

Lemma 1. *Let p and q two ascending sorted sequences with $p \leq q$. The rotation-based variant of Hwang-and-Lin's algorithm extended by the extraction of maximal segments of equal elements needs $2(\lambda|p|+|q|)$ many assignments at most, where λ is the number of distinct elements in p.*

Based on the above extension we handle the case of too few distinct elements as follows:
Let us assume that we could extract a buffer of λ distinct elements, where $\lambda < 2\lfloor\sqrt{m}\rfloor$ and that this buffer extraction divides u into u_1u_2 where u_1 contains the λ buffer elements. We granulate u_2 into λ blocks of size $k = \left\lfloor\frac{m-\lambda}{\lambda}\right\rfloor$ and one segment containing λ elements at most. We apply the stable merging algorithm using this modified block size and for the local merges we use the variant of Hwang and Lin's algorithm introduced above that doesn't rely on any internal buffer.

Theorem 4. *In the case of λ distinct elements in u, where $\lambda < 2\lfloor\sqrt{m}\rfloor$, two adjacent sorted sequences can be merged stable, in place and asymptotically optimal.*

n	m	UNSTAB.-IN-PL.-MERGE		STAB.-IN-PL.-MERGE		RECMERGE		Standard alg.	
		$\#comp$	t_e	$\#comp$	t_e	$\#comp$	t_e	$\#comp$	t_e
2^{21}	2^{21}	7373277	721	6359488	891	4631976	1172	4194166	180
2^{21}	2^{18}	1572372	185	1448275	210	1268154	550	2359280	95
2^{21}	2^{15}	290180	70	277387	90	255641	240	2129916	80
2^{21}	2^{12}	48638	60	47501	80	44238	200	2100313	80
2^{23}	2^{9}	8588	260	8538	330	8064	721	8383203	340
2^{23}	2^{6}	1257	301	1271	320	1195	611	8287178	330
2^{23}	2^{3}	176	411	180	250	172	421	7381470	330
2^{23}	2^{0}	24	70	24	110	23	101	6537757	327

t_e : Execution time in ms, $\#comp$: Number of comp., m, n : Lengths of inp. seq.

Table 3. Practical comparison of various merge algorithms

Proof. The only significant modification compared to the STABLE-IN-PLACE-MERGE algorithm concerns the size of the u-blocks and the number of different elements in all u-blocks. It is easy to verify that this keeps the algorithm asymptotically optimal. □

5 Experimental Results

We did some benchmarking with the algorithms developed here, in order to get an impression of their practical value. We compared the stable and unstable variant with the RECMERGE algorithm proposed by Dudzinsky and Dydek in [7] as well as the well known standard algorithm that needs linear extra storage. Table 3 contains a summary of our results. Each entry shows a mean value of 30 runs with different data. We took a state of the art hardware platform with 2.4 Ghz processor speed and 512MB main memory, all coding was done in the C programming language.

Despite their rather complex inner structure, the algorithms proposed here are surprisingly fast. The stable variant is almost always a bit slower than the unstable one, so stability seems to have a price. Additionally we observed that our algorithm is almost always a bit faster than RECMERGE. The latter algorithm is not optimal regarding the number of assignments. Hence, our algorithm would be the best selection in practice, particularly if you have to cope with input sequences of big size.

6 Conclusion

We could show that optimal stable in place merging is not merely a theoretical model but effectively usable in practice. Although our stable algorithm is fairly complex, it is fast, for almost all inputs even faster than the algorithm of Dudzinski and Dydek that is used as foundation of the `merge_without_buffer`

function contained in the C++ Standard Template Libraries. The reason for this performance can be seen in the algorithm's structure. The kernel provides only a mechanism for $\sqrt{m}$ calls of Hwang-and-Lin's algorithm. So most of the work is done by Hwang-and-Lin's algorithm that is well known for its efficiency.
A serious question that still remains is the role of the subalgorithms as driving factor of the overall running time. E.g. there are several rotation algorithms and Bentley shows in [15] that the best one from the theoretical point of view is not always the best one in practice. Another question is whether there exists a structurally more homogeneous and less complex algorithm with the same characteristics regarding optimality. Even an easy structured minimum storage algorithm that is optimal regarding both aspects (comparisons and assignments) would be interesting, but is not known so far. We lead these questions to further research in this area.

References

1. Knuth, D.E.: The Art of Computer Programming. Volume Vol. 3: Sorting and Searching. Addison-Wesley (1973)
2. Symvonis, A.: Optimal stable merging. Computer Journal **38** (1995) 681–690
3. Geffert, V., Katajainen, J., Pasanen, T.: Asymptotically efficient in-place merging. Theoretical Computer Science **237** (2000) 159–181
4. Chen, J.: Optimizing stable in-place merging. Theoretical Computer Science **302** (2003) 191–210
5. Mannila, H., Ukkonen, E.: A simple linear-time algorithm for in situ merging. Information Processing Letters **18** (1984) 203–208
6. Kronrod, M.A.: An optimal ordering algorithm without a field operation. Dokladi Akad. Nauk SSSR **186** (1969) 1256–1258
7. Dudzinski, K., Dydek, A.: On a stable storage merging algorithm. Information Processing Letters **12** (1981) 5–8
8. C++ Standard Template Library: (http://www.sgi.com/tech/stl)
9. Pardo, L.T.: Stable sorting and merging with optimal space and time bounds. SIAM Journal on Computing **6** (1977) 351–372
10. Salowe, J., Steiger, W.: Simplified stable merging tasks. Journal of Algorithms **8** (1987) 557–571
11. Huang, B.C., Langston, M.: Practical in-place merging. Communications of the ACM **31** (1988) 348–352
12. Kim, P.S., Kutzner, A.: Stable minimum storage merging by symmetric comparisons. In Albers, S., Radzik, T., eds.: Algorithms - ESA 2004. Volume 3221 of Lecture Notes in Computer Science., Springer (2004) 714–723
13. Hwang, F., S.Lin: A simple algorithm for merging two disjoint linearly ordered sets. SIAM J. Comput. **1** (1972) 31–39
14. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms. 2nd edn. MIT Press (2001)
15. Bentley, J.: Programming Pearls. 2nd edn. Addison-Wesley, Inc (2000)