Performant Stable in-Place Merging[☆]

Arne Kutzner[a], Pok-Son Kim[*,b]

[a]*Hanyang University, Department of Information Systems, Seoul 133-791, Rep. of Korea*
[b]*Kookmin University, Department of Mathematics, Seoul 136-702, Rep. of Korea*

Abstract

We propose two stable in-place merging algorithms that are asymptotically optimal regarding the number of required comparisons as well as assignments. The first algorithm is constructed on the foundation of an unstable core algorithm and is in the tradition of Mannila and Ukkonen's work [1]. The behavior of the second algorithm depends on the ratio $k = \frac{n}{m}$, where m, n are the sizes of both input sequences with $m \leq n$. For each ratio $k \geq \sqrt{m}$ it relies on a simple block-rotation based technique. Otherwise it uses a block redistribution technique in combination with local merges. Both algorithms are modularly designed, solve the problem by performing a series of separated steps and heavily utilize Hwang and Lin's [2] comparison strategy for local merges. They show an excellent runtime behavior on up-to-date hardware as proven by our benchmarking.

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 relies on a fixed amount of extra 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 e.g. in Knuth [3] the lower bound for the number of comparisons is $\Omega(m \log(\frac{n}{m} + 1))$, where $m \leq n$.

[☆]This work represents an extended presentation of the results that appeared in I) Proceedings of 5th International Conference on Theory and Applications of Models of Computation, Lecture Notes in Computer Science 4978, Springer 2008, pp. 246-257 and II) 32nd International Conference on Current Trends in Theory and Practice of Computer Science, Lecture Notes in Computer Science 3831, Springer 2006, pp. 350-359.
[*]Corresponding author
Email addresses: `kutzner@hanyang.ac.kr` (Arne Kutzner), `pskim@kookmin.ac.kr` (Pok-Son Kim)

A merging algorithm is called *asymptotically fully optimal* if it is asymptotically optimal regarding the number of comparisons as well as assignments.

There are already several publications on asymptotically fully optimal in-place merging, notably the work of Symvonis [4], the algorithm proposed by Geffert et al. [5] as well as Chen's [6] simplified form of the latter algorithm. All these algorithms have some similarities. They construct a stable algorithm on the foundation of an unstable core algorithm and they rely on a block management technique proposed by Mannila and Ukkonen in [1]. None of the above publications includes remarks regarding successful implementations or benchmarking.

We will propose two asymptotically fully optimal merging algorithms that rely on different methodologies. The first algorithm follows the path of Geffert et al. but in contrast to their algorithm it works on the foundation of fixed size segments. In the section on benchmarking we will argue why we think this is advantageous on up-to-date hardware, although Geffert et al.'s technique leads to smaller asymptotic constants. The behavior of the second algorithm depends on the ratio $k = \frac{n}{m}$. If we have $k \geq \sqrt{m}$ (this means quite asymmetric inputs), it switches to a simple block-rotation based technique for merging. In the other case it reserves some space of the shorter input sequence as "block distribution buffer", where it stores information about the origin - left input or right input - of fixed size segments. Later this "block distribution buffer" is used for successfully performing local merges. Both algorithms, like all the other algorithms mentioned above, use Kronrod's internal buffer [7] together with Hwang and Lin's comparison strategy [2] for local merges. For both algorithms we will give implementations in pseudocode.

After an introduction of several "toolbox algorithms", which are used throughout this work, we will present our first algorithm in a step-by-step fashion. We start with an unstable core algorithm and refine this algorithm so long, until we get a fully optimal stable in-place algorithm. Afterwards we present our second algorithm, where we distinguish between a section presenting a simple block-rotation based algorithm for ratios $k \geq \sqrt{m}$ and a section where we present a much more sophisticated block redistribution based algorithm. We finish with some notes on benchmarking, where we report that our algorithms are competitive and that they do not have to be viewed as theoretical models merely. One additional finding of the benchmarking will be that their runtimes, despite of different strategies, are quite similar. Therefore there is no clear winner among both algorithms.

2. Toolbox Algorithms

We now introduce some notations and algorithms that will be used throughout this work. Let u and v be 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.

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

Table 1: Complexity of the Toolbox Algorithms

Tab. 1 contains the complexity regarding comparisons and assignments for six elementary algorithms. We now briefly introduce these algorithms and their complexity:

(1) *Block Swapping* denotes the operation of exchanging the contents of two (not necessarily adjacent) blocks u and v with $|u| = |v|$.

(2) Let u and v be two adjacent blocks of not necessarily equal size. The circular rearrangement from $\dots uv \dots$ to $\dots vu \dots$, is called a *Block Rotation*. If we have an intermediate storage of one element only we need $|u|+|v|+\gcd(|u|, |v|)$ 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 [8].

(3) *Hwang and Lin* [2] 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. 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 converted into an in-place algorithm for the price of loosing the asymptotic optimality regarding the number of assignments by relying on repeated block rotations instead of an external buffer (Geffert et al. give a detailed description of that variant in [5]). This variant of Hwang and Lin's algorithm requires $n+m^2+m$ assignments. This can be explained as follows: Let $u = x_1 x_2 \cdots x_m$ and $v = v_0 v_1 \cdots v_m$ with $|u| = m$, $v_0 < x_1 \leq v_1$, $v_1 < x_2 \leq v_1$, $\cdots$ and $v_{m-1} < x_m \leq v_m$. For the rotation from uv to $v_0 u v_1 v_2 \cdots v_m$ the algorithm requires $m + |v_0| + \gcd(m, |v_0|)$ assignments. For the next rotation from $v_0 u v_1 v_2 \cdots v_m$ to $v_0 x_1 v_1 x_2 x_3 \cdots x_m v_2 \cdots v_m$ the algorithm requires $m - 1 + |v_1| + \gcd(m-1, |v_1|)$ assignments, and so on. So, the number of all assignments required by the algorithm is $m + |v_0| + \gcd(m, |v_0|) + m - 1 + |v_1| + \gcd(m-1, |v_1|) + m - 2 + |v_2| + \gcd(m-2, |v_2|) + \cdots + 1 + v_{m-1} + \gcd(1, |v_{m-1}|) \leq$

Pseudo-code Definition	Description of the Arguments
BLOCK-SWAP$(A, pos1, pos2, len)$	u is in $A[pos1 : pos1 + len - 1]$, v is in $A[pos2 : pos2 + len - 1]$
BLOCK-ROTATE$(A, first1, first2, last)$	u is in $A[first1 : first2 - 1]$, v is in $A[first2 : last - 1]$
HWANG-LIN$(A, first1, first2, last)$	u, v as in BLOCK-ROTATE
HWANG-LIN-BUF$(A, first1, first2, last, b)$	u, v as in HWANG-LIN, internal buffer in $A[b, b + (first2 - first1) - 1]$ or $b =$NIL
BSEARCH-LOWER$(A, first, last, x)$	delivers the position of the **first** occurrence of x in $A[first : last - 1]$
BSEARCH-UPPER$(A, first, last, x)$	delivers the first position following the **last** occurrence of x in $A[first : last - 1]$
MINIMUM$(A, pos1, pos2)$	delivers the index of the minimal element in $A[pos1 : pos2 - 1]$
SORT$(A, first, last)$	u is in $A[first : last - 1]$

EXCHANGE$(A, pos1, pos2)$ is equal to BLOCK-SWAP$(A, pos1, pos2, 1)$.

Table 2: Pseudocode Definitions of the Toolbox Algorithms

$2(m + (m - 1) + \cdots + 1) + n = m^2 + m + n.$

(4) *Binary Search* and *Insertion-Sort* are two standard algorithms described in almost all introducing literature about algorithms (e.g. [9]).

(5) In the case of a *Minimum Search* we assume that u is unsorted, therefore a linear search is necessary.

Tab. 2 gives pseudocode definitions for all toolbox algorithms, where the pseudocode conventions are taken from [9]. The procedure HWANG-LIN corresponds to the rotation based variant of Hwang and Lin's algorithm. HWANG-LIN-BUF requires a buffer that has to be delivered as the fifth argument b. If a caller passes the special value NIL for b (this indicates the nonexistence of a buffer), the call is redirected to the rotation based variant of Hwang and Lin's algorithm. If the value x does not occur in the sequence u, the procedures BSEARCH-LOWER and BSEARCH-UPPER deliver a reference to the first element greater than x.

3. An Algorithm in the Tradition of Mannila and Ukkonen's work

Step by step we will develop a stable in-place algorithm out of an unstable core that requires external space of size $\lfloor\sqrt{m}\rfloor$. The central aspect of the unstable core is the integration of Mannila and Ukkonen's [1] efficient block management. In a second step the unstable core becomes an unstable in-place algorithm by the inclusion of a local buffer in the tradition of Kronrod's work [7]. Finally we will get a stable algorithm by detecting and removing all sources of instability. Major characteristics of the stable algorithm are the application of a buffer extraction process and the utilization of a movement imitation buffer.

Algorithm 1 Unstable Core Algorithm

```
UNSTABLE-CORE-MERGE($A$, $first1$, $first2$, $last$, $buf$)
// $u$ is in $A[first1 : first2 - 1]$, $v$ is in $A[first2 : last - 1]$
// required buffer at $A[buf : buf + \lfloor \text{sqrt}(m) \rfloor - 1]$

$m = first2 - first1$; $k = \lfloor \text{sqrt}(m) \rfloor$; $delta = k$
if $m \bmod k == 0$
    $x = first1 + k$
else $x = first1 + (m \bmod k)$

while TRUE
    // Processing of the current minimal block
    $b$ = BSEARCH-LOWER($A$, $first2$, $last$, $A[x - 1]$)
    $sizeL = first2 - x$
    if $b - first2 \geq sizeL$
        BLOCK-ROTATE($A$, $x - 1$, $first2$, $b$)
    else BLOCK-SWAP($A$, $x$, $first2$, $b - first2$);
        BLOCK-ROTATE($A$, $x - 1$, $x$, $x + (b - first2)$)
        $delta = (b - first2 + delta) \bmod k$
        if $delta == 0$
            $delta = k$

    // Local merges
    HWANG-LIN-BUF($A$, $first1$, $x - 1$, $b - sizeL - 1$, $buf$)
    $first1 = b - sizeL$;   $first2 = b$;
    if $first1 \geq first2$
        break // No more blocks to be placed - leave the while-loop

    // Search the next minimal block
    $startU = first1 + k - delta$
    $startOfMinBlock$ = SEARCH-MINIMAL-BLOCK($A$, $startU$, $first2$, $k$, $delta$)

    // Move the minimal block to the front of sequence $q$
    if $startOfMinBlock == first2 - delta$
        BLOCK-SWAP($A$, $startU$, $startOfMinBlock$, $delta$)
        BLOCK-ROTATE($A$, $first1$, $startU$, $first1 + k$)
    else BLOCK-SWAP($A$, $startU$, $startOfMinBlock$, $k$)
        BLOCK-ROTATE($A$, $first1$, $startU$, $startU + k$)
    $x = first1 + k$
```

3.1. The Unstable 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 $u_0u_1\dots u_l$, so that all blocks u_i with $0 < i \leq l$ have size k. The first block u_0 gets the size $m - l\cdot k$. (u_0 is empty in the case $l\cdot k = m$). Let $u_i = b_ix_i$ 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 $v = v_0v_1\dots v_lv_{l+1}$ 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 this granulation of v and u we rearrange our input sequences to $b_0v_0x_0b_1v_1x_1\dots b_lv_lx_lv_{l+1}$. Eventually we get the desired sorted result by local merging of all pairs b_iv_i ($0 \leq i \leq l$). 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 has to be achieved by applying a linear number of assignments only. The following technique can be used to do so. It is similar to the method described by Mannila and Ukkonen in [1]: The rearrangement happens in a sequential style, it starts with block u_0 (u_1 if 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 $u_0\dots u_j$, $0 \leq j < l$. Then we have some sequence $p\ q\ v_{j+1}\dots v_{l+1}$, where $p = b_0v_0x_0b_1v_1x_1\dots b_jv_jx_j$ 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 $u'_1\dots u'_{l-j-1}$ and let u'_i ($1 \leq i < l-j$) 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*) $|v_{j+1}| \geq |q'|$: 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

block swap followed by a rotation to get $v_{j+1}x_{j+1}q'_2q'_1$ out of $x_{j+1}q'_1q'_2v_{j+1}$.

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 $\Sigma_{i=0}^{l}(2q_i+p_i) \leq 2m+n$ assignments, where $p_i = \max\{|u_i|,|v_i|\}$ and $q_i = \min\{|u_i|,|v_i|\}$. Further, since $\lfloor\sqrt{m}\rfloor\,(\lfloor\log n\rfloor+1) \leq \sqrt{m}(\lfloor\log n\rfloor+1) = m\cdot\frac{\lfloor\log n\rfloor}{\sqrt{m}}+\sqrt{m} \leq m\cdot(\log(n+m)-\log m)+\sqrt{m} = O(m\log(\frac{n}{m}+1))$, 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 $\Sigma_{i=1}^{l}2i \leq m+\sqrt{m} \leq 2m$ comparisons. The l extractions of the minimal block need $l(7k) \leq 7m$ assignments. The $l+1$ movements of the minimal block need less than $\Sigma_{i=0}^{l}4|v_i| \leq 4n$ assignments. So, altogether we have $O(m\log(\frac{n}{m}+1))$ comparisons and $O(m+n)$ assignments. □

Alg. 1 gives an implementation for UNSTABLE-CORE-MERGE in pseudocode. The definition of the procedure SEARCH-MINIMAL-BLOCK is left out due to its insignificance in the context of the stable algorithm presented later in this section. The meaning of all variables can be implied from the full definition of the stable algorithm in B.

3.1.1. 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 [7]. 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 $v_1 < x \leq v_2$. 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 call of Hwang and Lin's algorithm in line 22 uses 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.*

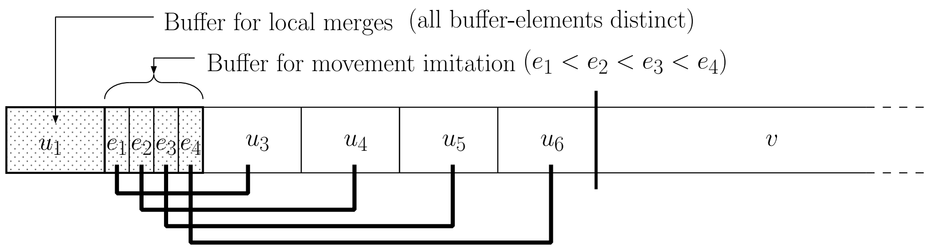

Buffer for local merges (all buffer-elements distinct)
Buffer for movement imitation ($e_1 < e_2 < e_3 < e_4$)
u_1
e_1
e_2
e_3
e_4
u_3
u_4
u_5
u_6
v

Figure 1: Partitioning scheme (here for $|u| = 24$)

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. □

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 3. Unstable-In-Place-Merge *is semi-stable.*

3.2. Deriving a Stable in Place Algorithm

The lack of stability in Unstable-In-Place-Merge is caused by (1) the block extraction in line 29 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. Note that we can disorder these buffers and afterwards sort them without losing stability. The first buffer will be used by the embedded call 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 [4]. 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 [10]. (2) We replace the search for the minimal block (line 29 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 4. *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. □

3.2.1. Undersized Buffer

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 5. *Let p and q be 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 $u_1 u_2$ where u_1 contains the λ buffer elements. We granulate u_2 into λ blocks of size $k = \lfloor\frac{m-\lambda}{\lambda}\rfloor$ 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 6. *In the case of λ distinct elements in u, where $\lambda < 2\left\lfloor\sqrt{m}\right\rfloor$, two adjacent sorted sequences can be merged stable, in place and asymptotically optimal.*

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. $\square$

A complete definition of Stable-In-Place-Merge inclusive documentation is given in A.

4. A Ratio Considering Algorithm

We will now solve the merging problem by inspecting the ratio $k = \frac{n}{m}$. Depending on this ratio the merging problem changes its nature. E.g. for $k = n$ (the shorter input consists here only of a single element) an optimal solution with respect to the number of comparisons represents a simple binary search. On the opposite, for $k = 1$ (both input sequences are now of equal size) such binary search has little significance in the context of an asymptotically optimal solution. The algorithm presented in this section pays attention to this observation and switches to a special subalgorithm for specific ratios. This subalgorithm, called Block-Rotation-Merge, is only optimal for ratios greater than or equal to $\sqrt{m}$ and will be presented in the first subsection. It is structurally quite simple and combines a block redistribution process with local merges on the foundation of Hwang and Lin's technique in combination with rotations. The central algorithm, called Stable-Optimal-Block-Merge, is optimal for arbitrary ratios and will be presented in the second subsection. It is quite complex, requires an internal buffer in the tradition of Kronrod's work and utilizes a block distribution buffer for block redistributions.

4.1. A simple asymptotically optimal algorithm for ratios $k \geq \sqrt{m}$

First we will now show that there is a simple stable merging algorithm called Block-Rotation-Merge that is asymptotically fully optimal for any ratio $k \geq \sqrt{m}$. Afterward we will prove that there is a relation between the number of different elements in the shorter input sequence u and the number of assignments performed by the rotation based variant of Hwang and Lin's algorithm [2]. In the following δ always denotes some block-size with $\delta \leq m$.

Algorithm 1: Block-Rotation-Merge (u, v, δ)

Algorithm 2 Pseudocode of Block-Rotation-Merge

```
Block-Rotation-Merge($A$, $first1$, $first2$, $last$, $delta$)
1  // $u$ is in $A[first1 : first2 - 1]$, $v$ is in $A[first2 : last - 1]$
2  $p = first1 + ((first2 - first1) \bmod delta)$
3  if $p == first1$
4        $p = p + delta - 1$
5  else $p = p - 1$
6  repeat
7        $b =$ BSearch-Lower($first2$, $last$, $A[p]$)
8        Block-Rotate($A$, $p$, $first2$, $b$)
9        Hwang-Lin($A$, $first1$, $p$, $p + (b - first2)$)
10       $first1 = p + 1 + (b - first2)$
11       $first2 = b$
12       $p = first1 + delta - 1$
13 until $first1 \geq first2$
```

1. We split the sequence u into blocks $u_1 u_2 \ldots u_{\lceil \frac{m}{\delta} \rceil}$ so that all sections u_2 to $u_{\lceil \frac{m}{\delta} \rceil}$ are of equal size δ and u_1 is of size $m\,(\mathbf{mod}\,\delta)$. Let x_i be the last element of u_i $(i = 1, \cdots, \lceil \frac{m}{\delta} \rceil)$. Using binary searches we compute a splitting of v into sections $v_1 v_2 \ldots v_{\lceil \frac{m}{\delta} \rceil}$ so that $v_i < x_i \leq v_{i+1} (i = 1, \cdots, \lceil \frac{m}{\delta} \rceil - 1)$.
2. $u_1 u_2 \ldots u_{\lceil \frac{m}{\delta} \rceil} v_1 v_2 \ldots v_{\lceil \frac{m}{\delta} \rceil}$ is reorganized to $u_1 v_1 u_2 v_2 \ldots u_{\lceil \frac{m}{\delta} \rceil} v_{\lceil \frac{m}{\delta} \rceil}$ using $\lceil \frac{m}{\delta} \rceil - 1$ many rotations.
3. We locally merge all pairs $u_i v_i$ using $\lceil \frac{m}{\delta} \rceil$ calls of the rotation based variant of Hwang and Lin's algorithm ([2]).

The steps 2 and 3 are interlaced as follows: After creating a new pair $u_i v_i$ $(i = 1, \cdots, \lceil \frac{m}{\delta} \rceil)$ as part of the second step we immediately locally merge this pair as described in step 3. A description of Block-Rotation-Merge in pseudocode is given in Alg. 2.

Lemma 7. Block-Rotation-Merge *performs less than* $\frac{m^2}{\delta} + 2m + m \cdot \delta + 2n$ *assignments if we use the optimal algorithm from Dudzinski and Dydek [8] for all block-rotations .*

Proof. For the first rotation from $u_1 u_2 \cdots u_{\lceil \frac{m}{\delta} \rceil} v_1$ to $u_1 v_1 u_2 \cdots u_{\lceil \frac{m}{\delta} \rceil}$ the algorithm performs $|u_2| + \cdots + |u_{\lceil \frac{m}{\delta} \rceil}| + |v_1| + \gcd(|u_2| + \cdots + |u_{\lceil \frac{m}{\delta} \rceil}|, |v_1|)$ assignments. The second rotation from $u_2 u_3 \cdots u_{\lceil \frac{m}{\delta} \rceil} v_2$ to $u_2 v_2 u_3 \cdots u_{\lceil \frac{m}{\delta} \rceil}$ requires $|u_3| + \cdots + |u_{\lceil \frac{m}{\delta} \rceil}| + |v_2| + \gcd(|u_3| + \cdots + |u_{\lceil \frac{m}{\delta} \rceil}|, |v_2|)$ assignments, and so on. For the last rotation from $u_{\lceil \frac{m}{\delta} \rceil - 1} u_{\lceil \frac{m}{\delta} \rceil} v_{\lceil \frac{m}{\delta} \rceil - 1} v_{\lceil \frac{m}{\delta} \rceil}$ to $u_{\lceil \frac{m}{\delta} \rceil - 1} v_{\lceil \frac{m}{\delta} \rceil - 1} u_{\lceil \frac{m}{\delta} \rceil} v_{\lceil \frac{m}{\delta} \rceil}$

the algorithm requires $|u_{\lceil \frac{m}{\delta} \rceil}| + |v_{\lceil \frac{m}{\delta} \rceil -1}| + \gcd(|u_{\lceil \frac{m}{\delta} \rceil}|, |v_{\lceil \frac{m}{\delta} \rceil -1}|)$ assignments. So, for all rotations the algorithm requires less than $2\delta(\frac{m}{\delta} + (\frac{m}{\delta} - 1) + \cdots + 1) + n = \frac{m^2}{\delta} + m + n$ assignments. Additionally the number of required assignments for the local merges is smaller than $\frac{m}{\delta}(\delta^2 + \delta) + n = m \cdot \delta + m + n$. Altogether the algorithm performs less than $\frac{m^2}{\delta} + m + n + m \cdot \delta + m + n = \frac{m^2}{\delta} + 2m + m \cdot \delta + 2n$ assignments. □

Lemma 8. *If $k = \sum_{i=1}^{n} k_i$ for any $k_i > 0$ and integer $n > 0$, then $\sum_{i=1}^{n} \log k_i \leq n \log(k/n)$.*

Proof. It holds because the function $\log x$ is concave. □

Lemma 9. *If we assume a block-size of $\lfloor \sqrt{m} \rfloor$, then* Block-Rotation-Merge *is asymptotically optimal regarding the number of comparisons.*

Proof. The binary searches for splitting v into sections $v_1 v_2 \ldots v_{\lceil \frac{m}{\delta} \rceil}$ require altogether less than $\sqrt{m}(\log n + 1) = \sqrt{m}(\log \frac{n}{m} + \log m + 1)$ comparisons. Because of the assumption $k = \frac{n}{m} \geq \sqrt{m}$ it holds $2 \log \frac{n}{m} \geq \log m$. Hence we have $\sqrt{m}(\log n + 1) \leq \sqrt{m}(3 \log \frac{n}{m} + 1) = O(\sqrt{m} \log \frac{n}{m})$ comparisons. For step 3 (local merges) we need $\sum_{i=1}^{\sqrt{m}} (\sqrt{m}(\log \frac{|v_i|}{\sqrt{m}} + 1) + \frac{|v_i|}{|v_i|/\sqrt{m}}) = \sqrt{m}(\sum_{i=1}^{\sqrt{m}} \log |v_i|) - m \log \sqrt{m} + 2m$ comparisons at most (Here we simply assume $\sqrt{m}$ is a positive integer number and it holds $v_i > 0$ for all $i = 1, 2, \cdots, \sqrt{m}$.). According to Lemma 8, the algorithm performs $\sum_{i=1}^{\sqrt{m}} (\sqrt{m}(\log \frac{|v_i|}{\sqrt{m}} + 1) + \frac{|v_i|}{|v_i|/\sqrt{m}}) \leq \sqrt{m} \cdot \sqrt{m} \cdot \log \frac{n}{\sqrt{m}} - m \log \sqrt{m} + 2m = O(m \log \frac{n}{m})$ comparisons for step 3. □

From the above Lemmas (7 and 9) we get the following result:

Corollary 10. *If we assume a block-size of $\lfloor \sqrt{m} \rfloor$, then* Block-Rotation-Merge *is asymptotically fully optimal for all $k \geq \sqrt{m}$.*

So, for $k \geq \sqrt{m}$ there is a quite primitive asymptotically fully optimal stable in-place merging algorithm. In the context of complexity deliberations in the next section we will rely on the following Lemma.

Lemma 11. *Let λ be the number of different elements in u. Then the number of assignments performed by the rotation based variant of Hwang and Lin's algorithm is $O(\lambda \cdot m + n) = O((\lambda + k) \cdot m)$.*

Proof. Let $u = u_1 u_2 \ldots u_\lambda$, where every $u_i (i = 1, \cdots, \lambda)$ is a maximally sized section of equal elements. We split v into sections $v_1 v_2 \ldots v_\lambda v_{\lambda+1}$ so that we get $v_i < u_i \leq v_{i+1}$ $(i = 1, \cdots, \lambda)$. (Some v_i may be empty.) We assume that Hwang and Lin's algorithm already merged a couple of section and comes to the first elements of the section $u_i (i = 1, \cdots, \lambda)$. The algorithm now computes the section v_i and moves it in front of u_i using one rotation of the form $\cdots u_i \ldots u_\lambda v_i \cdots$ to $\cdots v_i u_i \ldots u_\lambda \cdots$. This requires $|u_i| + \cdots + |u_\lambda| + |v_i| + \gcd(|u_i| + \cdots + |u_\lambda|, |v_i|) \leq$

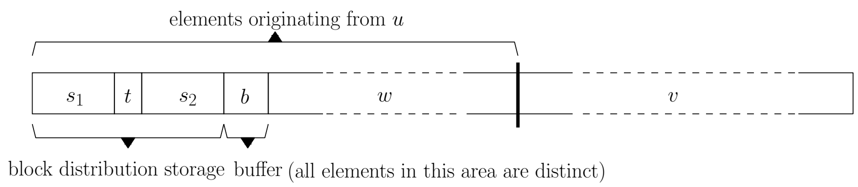

elements originating from u
s_1
t
s_2
b
w
v
block distribution storage
buffer
(all elements in this area are distinct)

Figure 2: Segmentation after the buffer extraction

$2(m + |v_i|)$ many assignments. Afterward the algorithm continues with the second element in u_i. Obviously there is nothing to move at this stage because all elements in u_i are equal and the smaller elements from v were already moved in the step before. Because we have only λ different sections, the lemma is true. □

Corollary 12. *Hwang and Lin's algorithm is fully asymptotically optimal if we have either $k \geq m$ or $k \geq \lambda$ where λ is the number of different elements in the shorter input sequence u .*

4.2. An asymptotically optimal algorithm for arbitrary ratios

We will now propose a stable in-place merging algorithm called Stable-Optimal-Block-Merge that is fully asymptotically optimal for arbitrary ratios. Notable properties of our algorithm are: It does not rely on the block management techniques described in Mannila and Ukonnen's work [1] in contrast to all other such algorithms proposed so far. It degenerates to the simple Block-Rotation-Merge algorithm for roughly $k \geq \sqrt{m}/2$. The internal buffer for local merges and the movement imitation buffer share a common buffer area. The two operations "block rearrangement" and "local merges" stay separated and communicate with each other using a common block distribution storage. There is no lower bound regarding the size of the shorter input sequence.

Algorithm 2: Stable-Optimal-Block-Merge

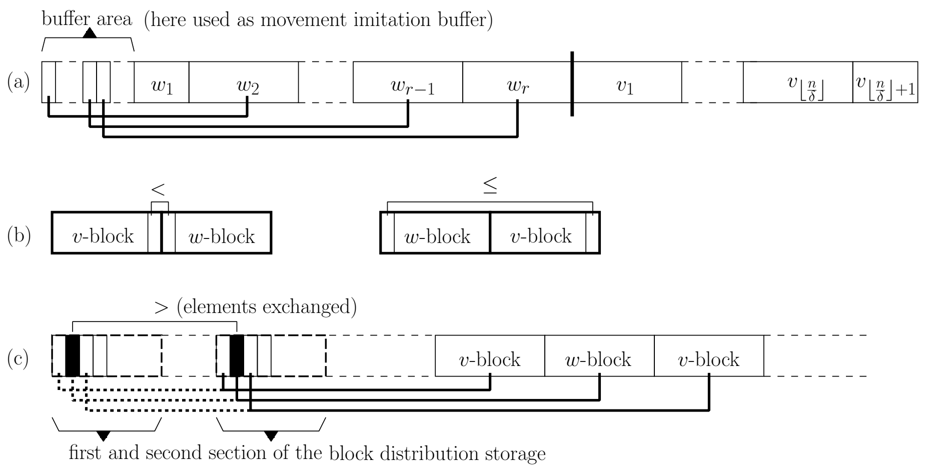
Step 1: Block distribution storage assignment

Let $\delta = \lfloor\sqrt{m}\rfloor$ be our block-size. We split the input sequence u into $u = s_1 t s_2 u'$ so that s_1 and s_2 are two sequences of size $\lfloor m/\delta\rfloor + \lfloor n/\delta\rfloor$ each and t is a sequence of maximal size with elements equal to the last element of s_1. We assume that there are enough elements to get a nonempty u'. We call s_1 together with s_2 our *block distribution storage* (in the following shortened to bd-storage).

Step 2: Buffer extraction

In front of the remaining sequence u' we extract an ascending sorted buffer b of size δ so that all pairs of elements inside b are distinct (as described by Pardo in [10]). Once more we assume that there are enough elements to do so. Now let w be the remaining right part of u' after the buffer extraction.

The segmentation of our input sequences after the buffer extraction is shown in Fig. 2.



buffer area (here used as movement imitation buffer)
(a)
w_1
w_2
w_{r-1}
w_r
v_1
$v_{\lfloor \frac{n}{\delta} \rfloor}$
$v_{\lfloor \frac{n}{\delta} \rfloor+1}$
(b)
<
v-block
w-block
≤
w-block
v-block
(c)
> (elements exchanged)
v-block
w-block
v-block
first and second section of the block distribution storage

Figure 3: Graphical remarks to the block rearrangement process

Step 3: Block rearrangement
We logically split the sequence wv into blocks of equal size δ as shown in Fig. 3 (a). The two blocks w_1 and $v_{\lfloor \frac{n}{\delta} \rfloor+1}$ are undersized and can even be empty. In the following we call every block originating from w a w-block and every block originating from v a v-block. The minimal w-block of a sequence of w-blocks is always the w-block with the lowest order (smallest elements) regarding the original order of these blocks.

We rearrange all blocks except of the two undersized blocks w_1 and $v_{\lfloor \frac{n}{\delta} \rfloor+1}$, so that the following 3 properties hold:
(1) If a v-block is followed by a w-block, then the last element of the v-block must be smaller than the first element of the w block (Fig. 3(b)).
(2) If a w-block is followed by a v-block, then the first element of the w-block must be smaller or equal to the last element of the v-block (Fig. 3(b)).
(3) The relative order of the v-blocks as well as w-blocks stays unchanged.

This rearrangement can be easily realized by "rolling" the w-blocks through the v-blocks and by "dropping" minimal w-blocks so that the above properties are fulfilled. The method of "rolling" is illustrated in Fig. 4 and can be explained as follows:
Let $w_{i,f}$ and $v_{i,e}$ be the first element of w_i and the last element of v_i for each i, respectively. First we compare $w_{2,f}$ with $v_{1,l}$. If $w_{2,f} > v_{1,l}$, then we swap w_2 and v_1 and afterwards we compare $w_{2,f}$ with $v_{2,l}$. In the other case, we compare $w_{3,f}$ with $v_{1,l}$. If $w_{3,f} > v_{1,l}$, then we swap w_3 and v_1 and afterwards we compare $w_{3,f}$ with $v_{2,l}$. We continue doing so until all blocks have been moved to their correct positions.
During this rolling the unplaced w-blocks stay together as group but they can be interlaced. So, due to the need for stability, we have to track their positions. For

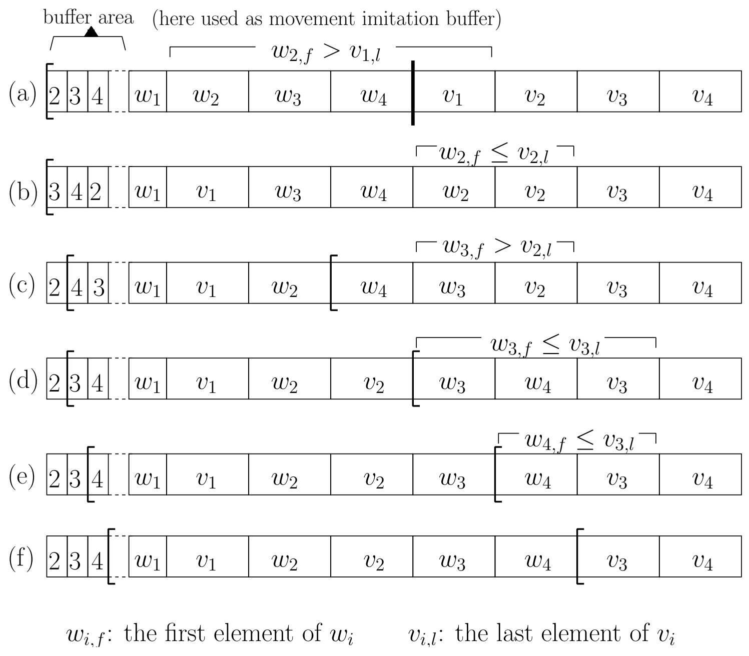


Figure 4: Block rearrangement - Rolling of w-blocks - An example

this reason we mirror all block replacements in the buffer area using a technique called movement imitation (The technique of movement imitation is described e.g. in [4]). Each time when a minimal w-block was dropped, we can find the position of the next minimal block using this buffer area.

Later we will have to find the positions of w-blocks in the block-sequence created as output of the rearrangement process. For this purpose we store the positions of w-blocks in the block distribution storage as follows:

The block distribution storage consists of two sections of size $\lfloor m/\delta \rfloor + \lfloor n/\delta \rfloor$ each and the i-th element of the first section together with the i-th element of the second section belong to the i-th block in the result of the rearrangement process. Note that, due to the technique used for constructing the bd-storage, such pairs of elements are always different with the first one smaller than the second one. If the i-th block originates from w, we exchange the corresponding elements in the bd-storage. Otherwise we leave them untouched. Fig. 3(c) shows this graphically.

Step 4: Local merges

We visit every w-block and proceed as follows:

Let p be the w-block to be merged and let q be the sequence of all v-originating elements immediately to the right of p that are still unmerged. Further let x be the first element of p.

(1) Using a binary search we split q into $q = q_1 q_2$ so that we get $q_1 < x \leq q_2$.

It holds $|q_1| < \delta$ due to the block rearrangement applied before. (2) We rotate pq_1q_2 to q_1pq_2. (3) We locally merge p and q_2 by Hwang and Lin's algorithm, where we use the buffer area as internal buffer.

This visiting process starts with the rightmost w-block and moves sequentially w-block by w-block to the left. The positions of the w-blocks are detected using the information hold in the bd-storage. Every time when we locate the position of a w-block in the bd-storage we bring the corresponding bd-storage elements back to their original order. So, after finishing all local merges both sections of the bd-storage are restored to their original form.

Step 5: Final sweeping up
On the left there is a still unmerged subsequence $s_1ts_2bw_1v'$ where v' is the subsection of v that consists of the remaining unmerged elements. We proceed as follows: (1) We split v' into $v' = v'_1v'_2$ so that $v'_1 < x \leq v'_2$ where x is the last element of s_2. Afterward we rotate $bw_1v'_1v'_2$ to $v'_1bw_1v'_2$ and locally merge w_1 and v'_2 using Hwang and Lin's algorithm with the internal buffer. (2) In the same way we split v'_1 into $v'_1 = v'_{1,1}v'_{1,2}$ so that we get $v'_{1,1} < y \leq v'_{1,2}$ where y is the last element of s_1. We rotate $s_1ts_2v'_{1,1}v'_{1,2}$ to $s_1v'_{1,1}ts_2v'_{1,2}$ and locally merge s_1 with $v'_{1,1}$ and s_2 with $v_{1,2}$' using the Block-Rotation-Merge algorithm with a block-size of $\lfloor\sqrt{m}\rfloor$. (3) We sort the buffer area using Insertion-Sort and merge it with all elements right of it using the rotation based variant of Hwang and Lin's algorithm.

Lack of Space in Step 1:
The inputs are so asymmetric that u' becomes empty. Using a binary search we split v into $v = v_1v_2$ so that we get $v_1 < t \leq v_2$ and rotate $s_1ts_2v_1v_2$ to $s_1v_1ts_2v_2$. Using the Block-Rotation-Merge algorithm with a block-size $\lfloor\sqrt{m}\rfloor$ we locally merge s_1 with v_1 and s_2 with v_2. If s_2 is empty we ignore it and directly merge s_1 with v in the same style.

Extracted buffer smaller than $\lfloor\sqrt{m}\rfloor$ in Step 2:
We assume that we could extract a buffer of size λ with $\lambda < \lfloor\sqrt{m}\rfloor$. We change our block-size δ (the size of w-blocks as well as v-blocks) to $\lfloor|u|/\lambda\rfloor$ and apply the algorithm as described but with the modification that we use the rotation based variant of Hwang and Lin's algorithm for all local merges.

Corollary 13. Stable-Optimal-Block-Merge *is stable.*

Theorem 14. *The* Stable-Optimal-Block-Merge *algorithm requires* $O(m+n) = O(m \cdot (k+1))$ *assignments.*

Proof. It is enough to prove that every step requires $O(m+n)$ assignments. We inspect all steps:
Step 1: This step comprises no assignments at all.
Step 2: The buffer extraction requires $O(m)$ assignments.
Step 3: The "rolling" of the w-blocks through the v-blocks together with the "dropping" of the minimal w-blocks requires less than $3\sqrt{m} \cdot (\sqrt{m} + \frac{n}{\sqrt{m}}) = O(m+n)$ assignments. The rotations for the integrated "movement imitation" contribute $O(\sqrt{m} \cdot (\sqrt{m} + \frac{n}{\sqrt{m}})) = O(m+n)$ assignments. The marking of

the positions of the w-blocks in the bd-storage needs $O(\sqrt{m})$ assignments. So, altogether step 3 requires $O(m+n)$ assignments.
Step 4: Each w-block rotation (e. g. rotation from pq_1q_2 to q_1pq_2) requires $\sqrt{m}+\sqrt{m}+\text{gcd}(\sqrt{m},\sqrt{m}) = 3\sqrt{m}$ assignments at most. So all w-block rotations need $3\sqrt{m}\cdot(\sqrt{m}) = O(m)$ assignments at most. The local mergings using Hwang and Lin's algorithm require less than $2m+n$ assignments, altogether. The reconstruction of the original order of the swapped elements in the bd-storage contributes $3\sqrt{m} = O(\sqrt{m})$ assignments.
Step 5: The first rotation requires $|v_1^{'}| + |bw_1| + \text{gcd}(|v_1^{'}|, |bw_1|)$ assignments. Here the length of v' in the unmerged sub-sequence $s_1ts_2bw_1v'$ may become as long as n. The local merging of w_1 and v_2' needs $2|w_1| + |v_2^{'}|$ assignments. Therefore the required number of assignments for completing both operations is less than $6\sqrt{m}+n$. The success in step 1 implies that $k \leq \sqrt{m}/2$, so we get $k\cdot\sqrt{m} \leq \frac{m}{2}$. Further $\lfloor m/\delta \rfloor + \lfloor n/\delta \rfloor$ is roughly equal to $(k+1)\cdot\sqrt{m} = \frac{m+n}{\sqrt{m}}$. So, the second rotation requires $|v_{1,1}'| + |ts_2| + \text{gcd}(|v_{1,1}'|, |ts_2|) \leq 2(\frac{m+n}{\sqrt{m}}) + n$ assignments. According to Lemma 7 the local merging of $s_1$with $v_{1,1}'$ ($s_2$with $v_{1,2}'$) by Block-Rotation-Merge needs $\frac{(k+1)^2\cdot m}{\sqrt{m}} + 2\cdot(k+1)\cdot\sqrt{m} + m\sqrt{m} + 2|v_{1,1}'|$ $(\frac{(k+1)^2\cdot m}{\sqrt{m}} + 2\cdot(k+1)\cdot\sqrt{m} + m\sqrt{m} + 2|v_{1,2}'|)$ assignments at most. Further it holds $\frac{(k+1)^2\cdot m}{\sqrt{m}} + 2\cdot(k+1)\cdot\sqrt{m} + m\sqrt{m} \leq 3\sqrt{m} + 3m + \frac{3}{2}n$. The buffer sorting using insertion sort contributes $O(m)$ assignments and the final call of Hwang and Lin's algorithm requires $n+m+\sqrt{m}$ assignments. So, step 5 needs altogether $O(m+n)$ assignments at all.
In the first exceptional case "Lack of Space in Step 1" we have $k \geq \sqrt{m}/2$ and directly switch to Block-Rotation-Merge. According to Lemma 7 the local merging s_1 with v_1 (the local merging s_2 with v_2) requires less than $(\frac{m}{2})^2/\sqrt{m} + 2\cdot(\frac{m}{2}) + m\sqrt{m} + 2\cdot|v_1|$ $((\frac{m}{2})^2/\sqrt{m} + 2\cdot(\frac{m}{2}) + m\sqrt{m} + 2\cdot|v_2|\ )$ assignments. Thus the required number of assignments is smaller than $5m\sqrt{m}/2 + 2m + 2n \leq 2m + 7n$ altogether.
In the second exceptional case "Extracted buffer smaller than $\lfloor\sqrt{m}\rfloor$" we change the block-size to $\lfloor |u|/\lambda \rfloor$ with $\lambda < \sqrt{m}$ and use the rotation based variant of Hwang and Lin's algorithm for local merges. A recalculation of the steps 3 to 5, where we use Lemma 11 in the context of all local merges, proves that the number of assignments is still $O(m+n)$. □

Theorem 15. *The* Stable-Optimal-Block-Merge *algorithm requires* $O(m\log(\frac{n}{m}+1)) = O(m\log(k+1))$ *comparisons.*

Proof. As in the case of the assignments it is enough to show that every step keeps the asymptotic optimality. In step 1 t is a sequence of maximal size with elements equal to the last element of s_1. Using binary search we can find the position of the last element of t. Therefore step 1 contains one binary search over m merely . The buffer extraction in step 2 requires m comparisons at most. The rearrangement of all blocks except of the two undersized blocks w_1 and $v_{\lfloor \frac{n}{\delta} \rfloor +1}$ in step 3 requires less than $\sqrt{m} + \frac{n}{\sqrt{m}}$ comparisons. As already mentioned in

Th. 14, the success in step 1 implies that $k \leq \sqrt{m}/2$, i.e. $n \leq m\sqrt{m}/2$. Thus $\sqrt{m}+\frac{n}{\sqrt{m}} \leq \sqrt{m}+m/2$. The detection of the minimal element in the movement imitation buffer demands $\sqrt{m} \cdot \sqrt{m}$ many comparisons at most. In step 4 the binary searches for splitting the q-sequences cost less than $\sqrt{m} \cdot (\log \sqrt{m} + 1)$ comparisons. Now let $(m_1, n_1), (m_2, n_2), \cdots, (m_r, n_r)$ be the sizes of all r-groups that are locally merged by Hwang and Lin's algorithm. According to Lemma 8, Table 1 and since $r < \sqrt{m}$ this task requires $\sum_{i=1}^{r}(m_i(\log(\frac{n_i}{m_i})+1)+m_i) = \sum_{i=1}^{r}(m_i \log(\frac{n_i}{m_i}) + 2m_i) \leq \sum_{i=1}^{r} m_i \log(\frac{n_i}{m_i}) + 2m = \sum_{i=1}^{r}(m_i \log n_i - m_i \log m_i) + 2m \leq \sqrt{m}(\sqrt{m}\log\frac{n}{r} - \sqrt{m}\log\frac{m}{r}) + 2m \leq m(\log(\frac{n}{m}+1)) + 2m = O(m\log(\frac{n}{m}+1))$ comparisons. The asymptotic optimality in step 5 as well as in the exceptional case "Lack of Space in Step 1" is obvious due to Lemma 9. The change of the block-size in the second exceptional case "Extracted buffer smaller than $\lfloor\sqrt{m}\rfloor$" triggers a simple recalculation of step 3 and step 4, where we leave the details to the reader. $\square$

Corollary 16. Stable-Optimal-Block-Merge *is an asymptotically fully optimal stable in-place merging algorithm.*

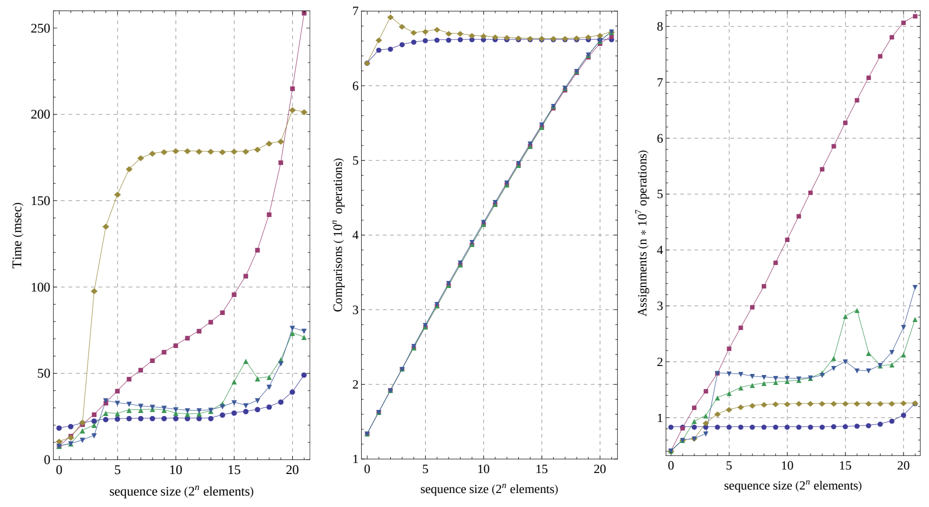
A full implementation inclusive documentation of Stable-Optimal-Block-Merge in pseudocode is given in B.

4.3. Optimizations

We now report about several optimizations that help improving the performance of Stable-Optimal-Block-Merge without any impact to its asymptotic properties. The immediate mirroring of all w-block movements in the movement imitation buffer (occurs in Step 3) triggers a rotation (line 10 in Alg. 6) every time when a v-block is moved into front of the group of w-blocks. The number of necessary rotations can be reduced by first counting the number of v-blocks moved into front of the w-blocks. This counting follows a single update of the movement imitation buffer if the placement of a minimal w-block happens. In the context of the movement of v-blocks into front of w-blocks (Step 3) a floating hole technique as described by Geffert et al. [5] can be applied for reducing the number of assignments. Similarly such a floating hole technique can also be applied during the local merges (Step 4) by combining the block swap to the internal buffer with the rotation that moves all smaller v-originating elements to the front of the w-block. In the special case "Extracted buffer smaller then $\lfloor\sqrt{m}\rfloor$" the sorting of the buffer b in Step 5 is unnecessary because the buffer is already sorted after Step 3 and stays unchanged during Step 4. Insertion-Sort can be replaced by some more efficient sorting algorithm. Note that there is no need for stability in the context of the buffer sorting because all buffer elements are distinct.

5. Experimental work

We did some experimental work with our algorithms in order to get an impression of their performance. We compared them with the following 3 competing



Standard Alg. ● RecMerge ■ Chen's Alg. ◆
(Sec. 3) Stable-In-Place-Merge ▼ (Sec. 4) Stable-Optimal-Block-Merge ▲
Asymmetric inputs of sizes $(2^n, 2^{22} - 2^n)$ for $0 \leq n \leq 21$ (x-axis: size of left input)

Figure 5: Benchmarking for asymmetrically sized inputs

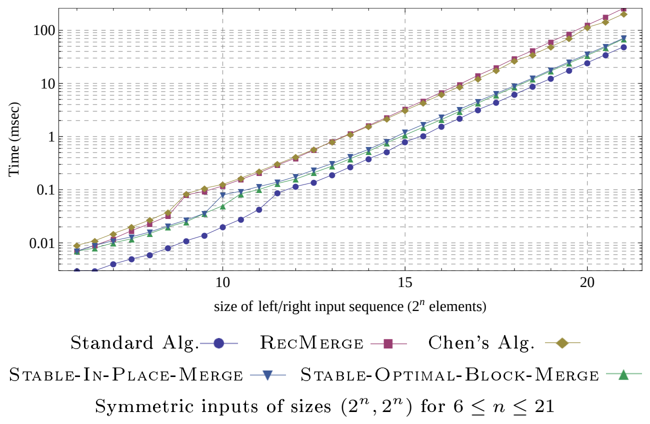
merging algorithms:

1. The standard textbook algorithm: Starting at the leftmost positions of either input sequences we compare pairs of elements, called head elements, such that the smaller element is written to the output and its successor replaces it as head element. The standard algorithm requires external memory of at least the size of the shorter input sequence.

2. The RecMerge algorithm proposed in [8]: RecMerge represents a minimum storage algorithm that relies on a divide and conquer strategy for merging. It is asymptotically optimal regarding the number of comparisons but not linear with respect to the number of necessary assignments. This algorithm gained attention in praxis because it is the foundation of the `merge_without_buffer` function in the C++ Standard Template Library (STL) [11].

3. The simply structured in-place Alg. proposed by Chen in [12]: Chen's algorithm is an unstable in-place merging algorithm that is asymptotically optimal regarding the number of assignments but not asymptotically optimal regarding the number of required comparisons. Chen gives a pseudocode description of this algorithm that we could successfully map to C++ code. In the context of our benchmarking we chose as block-size $\lfloor \sqrt{m} \rfloor$, where m is the size of the left input-sequence.

All coding and benchmarking were done by using the Visual C++ 2008 compiler with O2-optimization (maximal speed) switched on. On hardware side we



detailed times for $(2^{21}, 2^{21})$ elements

Algorithm	time (msec)
RecMerge	260.3
Chen's Alg.	204.3
Alg. of Sec. 3	71.3
Alg. of Sec. 4	69.4
Standard	49.0

Figure 6: Benchmarking for symmetrically sized inputs

used an up-to-date standard configuration with an AMD Athlon^{TM}Dual Core Processor 4850e operating at 2.5 Ghz and 2 GB RAM. The outcome of our benchmarking is shown in Fig. 5 and Fig. 6, where each marker represents an average value of 50 runs with sequences of randomly chosen 32-bit integer values. Fig. 5 investigates the behavior of all five algorithms if we successively increase the size of the left input sequence, where the overall size stays equal with 2^{22} elements. It allows a practical verification of the asymptotically optimal behavior regarding the number of comparisons as well as assignments. Fig. 6 inspects the algorithms for symmetrically sized inputs and aims to compare the performance of all 5 algorithms for a broader range of input sizes.

There are several alternatives regarding the buffer extraction that occurs as sub-operation in both algorithms. The extraction process can be started from the left end as well as from the right end of the input and we can choose between a binary search and linear search for the determination of the next element. All 4 possible combinations keep the asymptotic optimality. However, there is no clear "best choice" among them because the most advantageous combination can vary depending on the structure of the input. In the context of our benchmarking we decided for the variant "starting from the left combined with linear search".
The runtimes of our two algorithms are quite close to each other. This can be explained by the fact, that both algorithms rely on Hwang and Lin's strategy for local merges and that these local merges contribute heavily to the overall runtime. Fig. 6 shows that the runtime overhead in comparison to the STL-implementation of the standard textbook algorithm is roughly 50% starting with 2^{11} elements. So, the algorithms are practically usable for inputs of reasonable size, for example in the context of some Merge-Sort. The runtimes of our algorithms are always better than the times for RecMerge or Chen's algorithm. The superiority regarding RecMerge can be easily explained by RecMerge's lack of asymptotic optimality regarding the number of assignments. The superi-

ority regarding Chen's algorithm is rather surprising, given the fact that Chen's algorithm requires only half of the assignments of our algorithms. However, it can be explained by the pattern used by Chen's Alg. for accessing memory elements. This pattern does not fit well with the cache architectures of modern CPUs because it tends to create costly page conflicts. A discussion of these aspects of algorithms on the background of block-rotations can be found in [13].

6. Conclusion

We proposed two stable in-place merging algorithms that are asymptotically optimal regarding the number of comparisons as well as assignments. Using benchmarking we could show that our algorithms behave performantly on up-to-date hardware for inputs of reasonable size. So, they are not only of theoretical interest but also of practical value.
Our first algorithm was in the tradition of Mannila and Ukkonen's work [1]; it can be seen as a specific instantiation of the ideas expressed by Symvonis in [4]. This algorithm did not comprise techniques for coping with expandable local buffers as proposed by Geffert et al. in [5], instead it followed a simpler fixed size approach. The more sophisticated concept of expandable buffers delivers improved asymptotic constants in the context of some complexity analysis regarding the number of assignments, but it exposes the algorithm to a higher risk of page conflicts on up-to-date hardware, due to the resulting pattern for memory access. This allows the assumption, that the inclusion would lead to rather deteriorated runtimes, as observed in the context of the comparison with Chen's algorithm in section 5. Due to the lack of the availability of a working implementation for Geffert et al.'s algorithm a direct comparison could not be included.
The behavior of our second algorithm was driven by the ratio of the sizes of both input sequences. For severely asymmetric inputs (inputs with some ratio $k \geq \sqrt{m}$, where $k = \frac{n}{m}$ and m, n are the sizes of both input sequences with $m \leq n$) it simply performed several block rotations interchanged with local merges. Otherwise it applied a more complex technique that comprised the redistribution of fixed sized blocks on the foundation of a block distribution storage as central component.
Although they are methodologically different, both algorithms share a central characteristic. This is the utilization of Hwang and Lin's strategy for performing a series of local merges. These local merges in turn contribute heavily to the overall runtime. Altogether this explains the observed vicinity of their runtimes. Further it allows the formulation of the hypothesis that no asymptotically optimal in-place merging algorithm, which is constructed on the top of local merges using Hwang and Lin's strategy, will deliver significant better runtimes.

References

[1] H. Mannila, E. Ukkonen, A simple linear-time algorithm for in situ merging, Information Processing Letters 18 (1984) 203–208.

[2] F. Hwang, S.Lin, A simple algorithm for merging two disjoint linearly ordered sets, SIAM J. Comput. 1 (1) (1972) 31–39.

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

[4] A. Symvonis, Optimal stable merging, Computer Journal 38 (1995) 681–690.

[5] V. Geffert, J. Katajainen, T. Pasanen, Asymptotically efficient in-place merging, Theoretical Computer Science 237 (1/2) (2000) 159–181.

[6] J. Chen, Optimizing stable in-place merging, Theoretical Computer Science 302 (1/3) (2003) 191–210.

[7] M. A. Kronrod, An optimal ordering algorithm without a field operation, Dokladi Akad. Nauk SSSR 186 (1969) 1256–1258.

[8] K. Dudzinski, A. Dydek, On a stable storage merging algorithm, Information Processing Letters 12 (1) (1981) 5–8.

[9] T. Cormen, C. Leiserson, R. Rivest, C. Stein, Introduction to Algorithms, 2nd Edition, MIT Press, 2001.

[10] L. T. Pardo, Stable sorting and merging with optimal space and time bounds, SIAM Journal on Computing 6 (2) (1977) 351–372.

[11] C++ Standard Template Library, http://www.sgi.com/tech/stl.

[12] J.-C. Chen, A simple algorithm for in-place merging, Information Processing Letters 98 (2006) 34–40.

[13] J. Bojesen, J. Katajainen, Interchanging two segments of an array in a hierarchical memory system, in: S. Näher, D. Wagner (Eds.), Algorithm Engineering, Vol. 1982 of Lecture Notes in Computer Science, Springer, 2000, pp. 159–170.

A. Pseudocode and Documentation of Stable-In-Place-Merge

For getting the Definition of Stable-In-Place-Merge the three code segments given in Alg. 3, Alg. 4 and Alg. 5 have to be combined sequentially, so that the outcome forms one single code segment.
Alg. 3 comprises the initialization process. One major subsection of the initialization represents the buffer extraction, where we handle in the lines 14 to 17 the special case of too few distinct elements as explained in Sec. 3.2.1.

Alg. 4 contains the central while-loop. Each iteration of this while-loop corresponds to the processing of a single u-block. The first u-block processed by the while loop can be undersized; all following blocks have the fix size $\lfloor\sqrt{m}\rfloor$. The while loop consists of four major sections that correspond to the four major tasks associated with the processing of each block.

The variable k keeps the size of the u-blocks and the variable nub the number of unprocessed u-blocks. $delta$ expresses the shifting of the broken u-block and is equal to k in the case that there is no broken block. $bstart$ and $bend$ indicate the bounds of the extracted internal buffer. The first k elements of the buffer are used as Kronrod's internal buffer for local merges and the last n elements of the buffer are used as movement imitation buffer (mi-buffer). In the special case of an undersized buffer all buffer elements are used as mi-buffer. The variable x is a reference to the first element of the second u-block. $sizeL$ indicates the number of u-elements following the first u-block. b keeps the split position that we get as outcome of the binary search. In the context of the binary search we distinguish among the case that v_i is larger or equally sized to that remaining part of u and the case that it is smaller. The first case is handled by the simple rotation in line 6. The second case is more complex because the block-swap can change the position of the broken block - this has to be mirrored in the mi-buffer as done by the lines 10 to 12 - as well as the shifting of the broken block, as reflected by the recalculation of $delta$ in the lines 14 to 17. As additional outcome of the binary search we get the final position of the element $A[x-1]$, so it has not to participate in any local merge. The rotation in line 8 performs the final placement of $A[x-1]$ in the second case. The code segment from line 29 to line 40 handles the detection of the minimal block using the mi-buffer as well as the front-placement of the minimal block using the technique described by Mannila and Ukkonen. Line 31 identifies the position of the minimal block using the mi-buffer. Line 32 translates the index to an actual sequence position. In line 33 we mirror the block swapping of line 35 to line 39 in the mi-buffer. If the condition in line 35 is true, then we have the situation that the minimal block is equal to the broken block and we move the broken block to front position in the lines 36 to 37. In the other case some complete block is the minimal block and we move it to the front position in the lines 38 to 39. After line 40 we have the situation that a complete minimal block of k elements is on position $first1$. If there is a new broken block, then the complete block on position $first1$ is followed by the front section of this broken block. In the other case it follows a complete block.

The final sweeping up (Alg. 5) consists of the buffer sorting and merging of the buffer elements with the already merged outcome of the while-loop. The sorting of the buffer is restricted to the first k elements (used as Kronrod's internal buffer), because the mi-buffer is already sorted after the termination of the while loop (Line 32 reconstructs the original position of the elements in the mi-buffer area.)

Algorithm 3 Pseudocode of STABLE-IN-PLACE-MERGE - Part 1

```
STABLE-IN-PLACE-MERGE($A, first1, first2, last$)
 1  // $u$ is in $A[first1 : first2 - 1]$, $v$ is in $A[first2 : last - 1]$
 2  $m = first2 - first1$
 3  $k = \lfloor \text{sqrt}(m) \rfloor$
 4
 5  // Buffer Extraction
 6  $bSize$ = EXTRACT-BUFFER($first1, first2, 2 * k$)
 7  $bStart = first1$
 8  $bEnd = first1 + bSize$
 9  // internal buffer is in $A[bStart : bEnd - 1]$
10  if $bEnd - bStart == 2 * k$
11      // Fullsize Buffer
12      $nub = m/k - 2$
13      $fullSizeBuffer$ = TRUE
14  else // Undersized Buffer
15      $nub = bEnd - bStart$
16      $k = (first2 - bEnd)/nub$ // Adaption of Blocksize
17      $fullSizeBuffer$ = FALSE
18
19  // Final Variable Initializations
20  $first1 = bEnd$
21  $x = first2 - (nub * k)$
22  if $x == first1$
23      $x = first1 + k$
24      $nub = nub - 1$
25  $delta = k$
```

Algorithm 4 Pseudocode of STABLE-IN-PLACE-MERGE - Part 2

```
while TRUE
    // Binary Search and Movement of the Segment $v_i$
    $sizeL = first2 - x$
    $b =$ BSEARCH-LOWER($A$, $first2$, $last$, $A[x-1]$)
    if $b - first2 \geq sizeL$
        BLOCK-ROTATE($A$, $x-1$, $first2$, $b$)
    else BLOCK-SWAP($A$, $x$, $first2$, $b - first2$);
        BLOCK-ROTATE($A$, $x-1$, $x$, $x + (b - first2)$)

        // Mirroring of the Rearragement in the MI-Buffer
        $shift = ((delta + b - first2 - 1)/k) \bmod nub$
        BLOCK-ROTATE($A$, $bEnd - nub$, $shift + bEnd - nub$, $bEnd$)

        // Recalculation of the Shifting of the Broken Block
        $delta = (b - first2 + delta) \bmod k$
        if $delta == 0$
            $delta = k$

    // Local Merges
    if $fullSizeBuffer ==$ TRUE
        HWANG-LIN-BUF($A$, $first1$, $x-1$, $b - sizeL - 1$, $bStart$)
    else HWANG-LIN($A$, $first1$, $x-1$, $b - sizeL - 1$)

    // Recalculation of $first1$ and $first2$, Check for Termination
    $first1 = b - sizeL$; $first2 = b$
    if $first1 \geq first2$
        break

    // Detection of the New Minimal Block and its Placement
    $startU = first1 + k - delta$  // Position of the First Unbroken Block
    $indexOfMinBlock =$ MINIMUM($A$, $bEnd - nub$, $bEnd$) $- (bEnd - nub)$
    $startOfMinBlock = startU + indexOfMinBlock * k$
    EXCHANGE($A$, $bEnd - nub$, $bEnd - nub + indexOfMinBlock$)
    $nub = nub - 1$  // Decrease the Number of Unprocessed $u$-Blocks
    if $startOfMinBlock == first2 - delta$
        BLOCK-SWAP($A$, $startU$, $startOfMinBlock$, $delta$)
        BLOCK-ROTATE($A$, $first1$, $startU$, $first1 + k$)
    else BLOCK-SWAP($A$, $startU$, $startOfMinBlock$, $k$)
        BLOCK-ROTATE($A$, $first1$, $startU$, $startU + k$)
    $x = first1 + k$
```

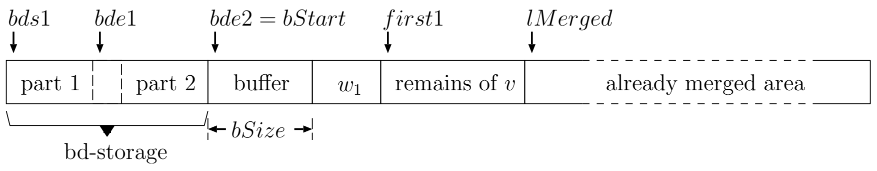


Figure 7: Situation when the execution of Alg. 8 reaches line 28

Algorithm 5 Pseudocode of STABLE-IN-PLACE-MERGE - Part 3

```
1  // Final Sweeping Up
2  if $fullSizeBuffer$ == TRUE
3        SORT($A$, $bStart$, $bStart + k$)
4  HWANG-LIN($A$, $bStart$, $bEnd$, $last$)
```

B. Pseudocode and Documentation of STABLE-OPTIMAL-BLOCK-MERGE

The definition of STABLE-OPTIMAL-BLOCK-MERGE comprises Alg. 6, Alg. 7 and Alg. 8. The definition starts with Alg . 8. The variable k specifies the size of all w-blocks as well as v-blocks (see Sec. 4.2), nb keeps the overall number of these blocks. The code section from line 4 to 14 performs the bd-storage allocation, as described in Step 1 of Sec. 4.2. At the end of this section we have a bd-storage, where the first part is in $A[bds1 : bde1 - 1]$ and the second part is in $A[bds2 : bde2 - 1]$. The allocation of the bd-storage can fail due to a lack of distinct elements or a ratio greater than $\sqrt{m}$. In this case the algorithm switches in line 12 immediately to BLOCK-ROTATION-MERGE and terminates. The extraction of an internal buffer consisting of distinct elements happens in the lines 16 to 22. At the end of the extraction process the buffer occupies the section $A[bStart : bStart + bSize\text{-}1]$. $bStart'$ represents an extended form of $bStart$ and indicates Hwang and Lin's algorithm by the special value NIL to rely on rotations instead of Kronrod's internal buffer. k' keeps an adapted block size. In the case of an undersized buffer the variable k' gets an enlarged block size computed on the foundation of $bSize$. Otherwise it is equal to k. In line 22 $first1$ is redirected to the first element of the first fully sized w-block. Left of this fully sized block can be a single undersized block w_1 (see Sec. 4.2) that is merged as suboperation of the final sweeping up in the lines 32 to 34. Line 25 calls the procedure for the block rearrangements, where the internal buffer is used as movement imitation buffer. Line 26 calls the function for the local merges. This function determines the value of $lMerged$ that indicates the left end of the area merged so far.

Fig. 7 shows the structure of the input when we reach line 28. The binary search in line 29 computes the subsection of the remaining v that contains elements that finally occur left of the internal buffer. Line 30 rotates this section to the

Algorithm 6 Pseudocode of procedure for block rearrangements

```
REARRANGE-BLOCKS($A$, $first1$, $first2$, $last$, $bStart$, $bds1$, $bds2$, $k$)
 1  // $w_2 \dots w_x$ is in $A[first1 : first2 - 1]$, $v_1 \dots v_{y-1}$ is in $A[first2 : last - 1]$
 2  // buffer $b$ is in $A[bStart : bStart + \lfloor\sqrt{m}\rfloor - 1]$
 3  // bd-storage $s_{\{1|2\}}$ is in $A[bds\{1|2\} : bds\{1|2\} + \lfloor\sqrt{m}\rfloor + \lfloor n/\sqrt{m}\rfloor - 1]$
 4
 5  $bEnd = bStart + (first2 - first1)\ /\ k$
 6  $wBlock = first1$
 7  while $first1 < first2$
 8      if $first2 + k < last$ and $A[first2 + k - 1] < A[wBlock]$
 9          BLOCK-SWAP($A$, $first1$, $first2$, $k$)
10          BLOCK-ROTATION($A$, $bStart$, $bStart + 1$, $bEnd$)
11          if $wBlock == first1$
12              $wBlock = first2$
13          $first2 = first2 + k$
14      else BLOCK-SWAP($A$, $wBlock$, $first1$, $k$)
15          EXCHANGE($A$, $bds1$, $bds2$)
16          EXCHANGE($A$, $bStart$, $bStart + (wBlock - first1)\ /\ k$)
17          $bStart = bStart + 1$
18          if $bStart < bEnd$
19              $minIndex$ = MINIMUM($A, bStart, bEnd$)
20              $wBlock = first1 + (minIndex - bStart) * k$
21      $bds1 = bds1 + 1$; $bds2 = bds2 + 1$
22      $first1 = first1 + k$
```

front of the buffer. Due to this rotation the location of the buffer may change and has to be recomputed in line 31. The section from line 32 to line 34 performs the merging of the single undersized block w_1 with its corresponding subsection of v. The code block from line 35 to 37 performs the sorting of the internal buffer as well as its merging with the already merged section right of it. Finally we assign each bd-storage section its corresponding v-section using a binary search together with a rotation and merge these v-sections with either parts of the bd-storage in line 41 and line 42 using the procedure BLOCK-ROTATION-MERGE.

The procedure REARRANGE-BLOCKS (Alg. 6) implements the block rearrangement as described in Step 3 in Sec. 4.2. In line 5 the difference $first2 - first1$ has to be a multiple of the block-size k. The size of the mi-buffer, which is located in $A[bStart : bEnd - 1]$, is equal to the number of k-sized blocks in $A[first1 : first2 - 1]$. $bds1$ and $bds2$ will move throughout the bd-storage in the context of the while-loop. By exchanging $A[bds1]$ and $A[bds2]$ we will indicate the position of a w-block in the bd-storage. Each iteration of the central while-loop corresponds to the placement of a single minimal block. During the execution of the while-loop $first1$ indicates always the position for the placement of next minimal block. $wBlock$ is a reference to the current w-block and $first2$ is a reference to the current v-block that participate in the placement decision, represented by the condition in line 8. If the minimal block originates from v (true-branch), then we swap the first k elements originating from v to the front position (indicated by $first1$) and mirror the resulting new order of the unprocessed w-blocks in the mi-buffer in line 10. Line 13 changes the reference $first2$ to the position of the next current v-block. If the minimal block originates from w (false-branch), then we swap the current w-block to the front in line 14 and memorize the w-origin of this block in the bd-storage in line 15. Further we adapt the size of the mi-buffer in line 17 and determine the location of the next current w-block using the mi-buffer in the code block from line 18 to 20. Line 21 and 22 increment $bds1$, $bds2$ and $first1$ as required.

Alg. 7 contains the code for local merges as described in Step 4 of Sec. 4.2. The argument $numWBlocks$ receives the overall number of w-blocks. The outer while-loop counts $numWBlocks$ down, where each iteration of the while loop represents the local merging of a w-block with its corresponding section from v, which may consist of several v-blocks. $index$ is a counter for the bd-storage and is used for recognizing and memorizing the positions of w-blocks. For this purpose the inner while loop from line 5 to line 6 counts $index$ repeatedly down to the position of the next w-block, where the position of a w-block is indicated by a pair of bd-elements in wrong order. The processing of the w-blocks start with the rightmost one and moves forward to the left. $vBlock$ in line 7 becomes a reference to the first v-block following the currently processed w-block. Because the processing happens block oriented, a sequence of v-blocks may comprise elements that shall finally appear in the front of the currently processed w-block (the maximal number of these elements is always smaller than the block-size k). The binary search in line 9 identifies these elements and the rotation in line 10 moves them to the front of the currently processed w-block. Line 11 calls Hwang and Lin's algorithm in order to merge the current w-block with its counterpart from v using the internal buffer starting at $bStart$. $bStart$ may have the special value NIL in order to indicate that Hwang and Lin's Alg. shall be performed on the foundation of rotations instead of an internal buffer. Line 12 adapts $last$ so that it refers to the leftmost position of the fully merged outcome on the right. Because the information kept by the bd-storage is not required any longer we reconstruct the original order of the w-block indicating element-pair in line 13.

Optimization of Final Sweeping Up

The final sweeping up (Step 5 in Alg. 8) comprises two calls of BLOCK-ROTATION-MERGE, where BLOCK-ROTATION-MERGE, according to its formal definition in Sec. 4.1 and its implementation in Alg. 2, relies on rotations for achieving its local merges. The definition of BLOCK-ROTATION-MERGE can be extended, so that the procedure receives an additional argument that informs (like $bStart$ in Alg. 6) about an optional internal buffer. Such an internal buffer, assumed it is of sufficient size, can be used for optimizing the local merges using Hwang and Lin's Alg. in line 9 of Alg. 2. Altogether this allows

Algorithm 7 Pseudocode of function for local merges

```
LOCAL-MERGES($A$, $first$, $last$, $bStart$, $bds1$, $bds2$, $k$, $numWBlocks$)
// $A[first : last - 1]$ contains all blocks in distributed form

$index = ((last - first) / k) - 1$
while $numWBlocks > 0$
    while $A[bsd1 + index] < A[bsd2 + index]$
        $index = index - 1$
    $vBlock = first + ((index + 1) * k)$
    if $vBlock < last$
        $b$ = BINARY-SEARCH($vBlock, last, A[vBlock - k]$)
        BLOCK-ROTATION($A$, $vBlock - k$, $vBlock$, $b$)
        HWANG-LIN($A$, $b - k$, $b$, $last, bStart$)
        $last = b - k$
    EXCHANGE($A$, $bds1 + index$, $bds2 + index$)
    $numWBlocks = numWBlocks - 1; index = index - 1$
return last
```

the following optimization of the code segment spanning from line 35 to 42 in Alg. 8: First the merging of the bd-storage happens using the extended form of BLOCK-ROTATION-MERGE under the aid of the still available internal buffer in $A[bStart : bStart + bSize - 1]$ and then the sorting of the internal buffer happens, followed by the merging with the already merged outcome right of it. In the context of the benchmarking we did implement this optimization and recognized roughly 10% better runtimes for symmetrically sized inputs.

Algorithm 8 Pseudocode of STABLE-OPTIMAL-BLOCK-MERGE

```
STABLE-OPTIMAL-BLOCK-MERGE($A$, $first1$, $first2$, $last$)
// $u$ is in $A[first1 : first2 - 1]$, $v$ is in $A[first2 : last - 1]$
$k = \lfloor \mathrm{sqrt}(first2 - first1) \rfloor$

// Step 1 : Block Distribution Storage Assignment
$nb = \lfloor (last - first1)/k \rfloor$
$bds1 = first1$; $bde1 = first1 + nb$
if $bde1 \geq first2$
    $bds2 = first2$
else $bds2 =$ BSEARCH-UPPER($bde1$, $first2$, $A[bde1 - 1]$)
$bde2 = bds2 + nb$
if $bde2 \geq first2$
    BLOCK-ROTATION-MERGE($A, first1$, $first2$, $last$, $k$)
    return
$first1 = bde2$

// Step 2 : Buffer Extraction
$bSize =$ EXTRACT-BUFFER($first1$, $first2$, $k$)
$bStart = first1$
if $bSize < k$
    $bStart' =$ NIL;   $k' = \lfloor (first2 - bds1)/bSize \rfloor$
else $bStart' = bStart$;   $k' = k$
$first1 = (first1 + bSize) + (first2 - (first1 + bSize)) \bmod k'$

// Step 3 and 4 : Block Rearrangement and Local Merges
REARRANGE-BLOCKS($first1$, $first2$, $last$, $bStart$, $bds1$, $bds2$, $k'$)
$lMerged =$ LOCAL-MERGES($first1$, $last$, $bStart'$, $bds1$, $bds2$, $k'$, $\lfloor (first2 - first1)/k' \rfloor$)

// Step 5 : Final Sweeping Up
$b =$ BSEARCH-LOWER($first1$, $lMerged$, $A[bStart - 1]$)
BLOCK-ROTATE($A$, $bStart - 1$, $first1$, $b$)
$bStart = bStart + (b - first1)$
if $bStart' \neq$ NIL
    HWANG-LIN-BUF($A$, $bStart + bSize$, $b$, $lMerged$, $bStart$)
else HWANG-LIN-BUF($A$, $bStart + bSize$, $b$, $lMerged$, NIL)
SORT($A$, $bStart$, $bStart + bSize$)
HWANG-LIN($A$, $bStart$, $bStart + bSize$, $last$)
$lMerged = bStart - 1$

$b =$ BSEARCH-LOWER($lMerged - (b - first1)$, $lMerged$, $A[bde1 - 1]$)
BLOCK-ROTATE($A$, $bde1 - 1$, $bde2 - 1, b$)
BLOCK-ROTATION-MERGE($A$, $bds1$, $bde1 - 1$, $bde1 - bde2 + b$, $k$)
BLOCK-ROTATION-MERGE($A$, $b - nb + 1$, $b$, $lMerged$, $k$)
```