

Buffered Adaptive Radix – a fast, stable sorting algorithm that trades speed for space at runtime when needed.¹

Arne Maus, arnem@ifi.uio.no

Department of Informatics,
University of Oslo

Abstract

This paper introduces Buffered Adaptive Radix (BARsort) that adds two improvements to the well known right-to-left Radix sorting algorithm (Right Radix or just Radix). The first improvement, the adaptive part, is that the size of the sorting digit is adjusted according to the distribution and the maximum value of the elements in the array. The second and most important improvement is that data is transferred back and forth between the original array and a buffer that is only a percentage of the size of the original array, as opposed to traditional Radix where that second array is the same length as the original array. Even though a buffer size of 100% of the original array is the fastest choice, any percentage larger than 6% gives a good performance. This result is also explained analytically. The main motivation for introducing BARsort is to construct a fast and stable sorting algorithm with minimal space overhead and where the amount of extra space needed can be determined at before the sorting starts. This is important in programming languages such as Java where the heap size is fixed at load time and the `new` operator for constructing the extra array can terminate the program if the requested space is not available. BARsort avoids this problem. In this paper the stable BARsort algorithm is compared to the two non-stable algorithms Quicksort and Flashsort for 15 different distributions of data and n , the length of the array, varying from 50 to 97M. For the standard distribution, uniform $0:n-1$, the relative performance of BARsort versus Quicksort and Flashsort is also shown for Sun UltraSparcIIIi, Intel Core Duo 2500T, Intel Pentium IV and AMD Opteron 254 based machines. A model for the observed execution time is also presented, and, finally, BARsort is proposed as a general replacement for Quicksort because of its faster performance, its stable sorting quality and that it will not abort the application if its primary claim for a 100% buffer is rejected.

Index terms: Sorting, Radix, buffered algorithm, stable sorting, graceful degradation, Quicksort, Flashsort, Java.

1. Introduction

Sorting is perhaps the single most important algorithm performed by computers, and certainly one of the most investigated topics in algorithmic design[1]. Numerous sorting algorithms have been devised, and the more commonly used are described and analyzed in any standard textbook in algorithms and data structures[14,15] or in standard reference works [6,8]. Maybe the most up to date coverage is presented in [14]. New sorting algorithms are still being developed, like Flashsort [11], “The fastest sorting algorithm”[13] , ARL[10] and Permsort[9] . The most influential sorting algorithm

¹ This paper was presented at the NIK-2007 conference. For more information, see [//www.nik.no/](http://www.nik.no/)

introduced since the 60's is no doubt the distribution based 'Bucket' sort which can be traced back to [3].

Sorting algorithms can be divided into comparison and distribution based algorithms. Comparison based methods sort by repeatedly comparing two elements in the array that we want to sort (for simplicity assumed to be an integer array a of length n). It is easily proved [15] that the time complexity of a comparison based algorithm is at best $O(n \log n)$. Well known comparison based algorithms are Insertsort, Heapsort and Quicksort [15,5,2]. Distribution based algorithms, on the other hand, sort by using directly the values of the elements to be sorted. Under the assumption that the numbers are (almost) uniformly distributed, these algorithms can sort in $O(n \log m)$ time where m is the maximum value in the array. Well known distribution based algorithms are Radix sort in its various implementations and Bucket sort. The difference between Radix sort and Bucket sort is that the Radix algorithms sort on the value of successive parts of the elements (called digits) while in Bucket sort, the algorithm sorts on the whole value of the element each time.

Quicksort is still in textbooks regarded as the fastest known sorting algorithm in practice, and its good performance is mainly attributed to its very tight and highly optimized inner loop [15]. It is used here as a reference for the other algorithms. The reason for also including Flashsort in the comparison are twofold – it is one of the newer sorting algorithms of the Bucket type and its inventor claims that it is twice as fast as Quicksort [11,2]. The code for Flashsort has been obtained from its homepage[12] and is one of the few recent algorithms where the actual code is available.

The rest of this paper is organized as follows: First, BARsort is presented using pseudo code. Then two features of BARsort are presented, the ability to use a 'small' buffer and the tuning of the digit size. In the testing section, the effect of 7 different buffer sizes on performance of BARsort is presented and a comparison to Quicksort and Flashsort for the Uniform(n) ($U(n)$) distribution on 4 different CPUs are made. To evaluate these relative execution times, the absolute execution times for Quicksort is also presented in Figure 5a. Then BARsort and Flashsort are tested for 15 different distributions. On the average as well, as in almost all cases, BARsort is shown to be much faster than Flashsort and Quicksort are. Finally, we conclude the paper by proposing BARsort as a general replacement for Quicksort.

1.1 Radix Sorting Algorithms and BARsort

Basically, there are two Radix sorting algorithms, Left Radix, also called MSD (Most Significant Digit) and Right Radix or LSD (Least Significant Digit). Right Radix, or just Radix, is by far the most popular variant of the radix sorting algorithms. This paper deals only with the Right Radix algorithm.

In Radix, one starts with the least significant digit and sort the elements on that digit in the first phase by copying all data into another array in sorted order on that digit. Then Radix sorts on the next digit, right-to-left. Each time, Radix copies the data between two arrays of length n , and runs through the whole data set twice in each pass. Radix is a very fast and stable sorting algorithm, but has in its usual implementation a space requirement of $2n$ – twice that of in-place sorting algorithms. This paper improves on Radix in two ways –how the size of the digit is selected for each iteration of the main

loop and how the size of this extra array, hereafter called the buffer, can, when needed, be made smaller than the original array.

A digit is a number of consecutive bits, and can in both radix algorithms be allowed to be set individually for each pass. It is any value from 1 and upwards and is not confined to an 8-bit byte – although all reported algorithms so far use a fixed 8-bit byte as the digit size. BARSORT, presented in this paper, owes some its good performance to varying numBit, the size of the sorting digit, between 1 and some maximum value: maxNumBits for each sorting digit.

Basically, the number of bits in the sorting digit, numBit = min(log n, maxNumBits). The reason for reducing the digit size is twofold. First, some of the work done in a pass in any Radix algorithm is proportional with the maximum value of the sorting digit. Secondly, the reason for having an upper limit on the digit is that all frequently and randomly accessed data structures should fit into the L1 cache of the CPU. On the Pentium 4, UltraSparcIIIi, Intel CoreDuo/M T2500 and ADM Opteron 254, the sizes of the L1 data caches are 8 Kb, 64Kb, 64Kb and 64Kb respectively. The upper limit for the sorting digit size should be tuned to the size of the L1 cache in the processor used. A maximum size of 6 to 8 bits might be appropriate for the Pentium 4 while 9 to 11 bits might be optimal for Pentium Ms and AMD processors. (determined so that the central data structure, one array of size 2^{numBit} fits well into the level 1 cache).

The main advantage of using as little memory as possible is not that modern computers usually don't have enough of it, but that your extra data is moved down in the memory hierarchy from the caches to main memory with the associated speed penalty of factor 40-70[7, 16]. And if you sort really large arrays and have a sorting algorithm that needs large extra data structures like Flashsort or especially Radix, you might also trigger the disk-paging mechanism earlier than necessary.

A second and just as important reason for not having an algorithm that demands a fixed size for its extra data structure, is that it is allocated at runtime. In many programming languages such as Java, the size of the heap is statically determined at load-time, and hence there might not be enough free space when the method call to sort is issued. The program then terminates with an error, which can hardly be called acceptable. Other languages like C++ allocates its arrays statically and hence has to allocate the largest possible buffer which also is undesirable. In the next section I describe how this problem can be avoided in BARSORT and that acceptable sorting times will be achieved if only 6% of the size of original array is available on the heap as a buffer.

1.2 Stable sorting

Stable sorting is defined as having equal valued input elements presented in the same order in the output – i.e. if you sort 2,1,1, and obtain 1,1,2, – the two 1's in the input must not be interchanged on the output. This is an important feature. For example, in my e-mail reader, if I first sort e-mails on arrival date and then on sender, I get my e-mail sorted on sender, but the emails from each sender are then no longer sorted on arrival date, which I also want. The same valued elements (on sender) that were once in correct order on date, are now rearranged because the sorting algorithm used in the e-mail reader is not stable. A more substantial example is the result from a database query where you want to sort the resulting table on more than one key.

Generally, stable sorting is a harder problem than un-stable sorting, because a stable sorting algorithm is rearranging input according to the unique permutation of input that will produce the stably sorted output; whereas a non-stable sorting algorithm only has to find one among many permutations (if there are equal valued elements) that produces a non-stable sorted permutation of the input. It is then expected that stable sorting algorithms should use more time and/or space than unstable sorting ones.

2. The BAR sort algorithm

The BARsort algorithm presented here in pseudo code, is an iterative Right-radix algorithm with a dynamically determined number of bits in its digits. It is optimized with Insertsort [15] for sorting arrays of length less than 32 (not shown in the pseudo code). The actual Java code for BARsort can be found in[17].

```

static void BARsort (int[] a, int left, int right) {
    int len =right -left+1;
    int [] buf;
    int movedSoFar;

    a) allocate a buffer 'buf', if possible of length = 'len' (but any length >0 will do)
    b) find the max element in 'a' and 'leftbitno' = (the highest numbered bit = 1 in max).

    while <more digits>
        c) determine 'bits', the number of bits for this sorting step;
        d) count how many elements of each digit value there are in an array 'count'. Then
            cumulatively add these values such that count[i] is the index in buf where the first
            element with digit value= i should be placed when sorting.
        e) calculate how many buffer iterations we need to move 'a' to 'buf' in sorted order
            (= 1 if we get a 100% buffer of length 'len')

        movedSoFar = left;

        while <more buffer iterations> {
            f) find 'nMove'= how many (and in some other variable which) digit values
                to move in this pass to 'buf' (all of some values and some of the last
                moved digit value).
            g) move these element in sorted order to buf, starting at 'movedSoFar' in
                'a' and replace these moved elements with -1 in 'a'.
            h) move the positive elements down in 'a' to places containing -1, starting
                with the last element in 'a' that was moved to 'buf' and scan upwards.
                Stop the search when we pass the limit:'movedSoFar + nMove'.
            e) copy buf[0..nMove] to a[movedSoFar..movedSoFar+nMove]

            movedSoFar += nMove;

        }
    }
}

```

Program 1. Pseudo code for BARsort.

Some comments to the code. In a) we try to get a maximum sized buffer, but back off and settle for less if there is not enough room on the heap. The actual Java code for this is given in Program 2. The factor 0.7 is used to halve the buffer claim in two iterations.

As demonstrated (fig. 2), BARsort performs well if it gets buffer $\geq 12\%$ of the length of *a*, and acceptable performance if it gets at least a 3% buffer.

To find 'bits', the number of bits in the next sorting digit in step b), 'bits' is first set to $\min(\text{bits left to sort on}, \text{MAX_NUM_BIT})$. Then bits is, if necessary, reduced to the largest integer such that $2^{\text{bits}} > n$. The effect of this is twofold: The last sorting digit (which might also be the first) is set no larger than the remaining relevant bits. The reason is that some of the loops in BARsort is of $O(2^{\text{bits}})$ while other are $O(n)$. For all the data points in Fig. 3a, the 15 distributions, the this adaptive part reduces the execution time relative to Quicksort on the average by 10% compared to use a fixed digit size = 8 bits for all the sorting digits.

Steps d) to g) are easy to implement. The point with the two limits 'movedSoFar' and 'movedSoFar'+nMove' is that above movedSoFar all elements in *a* have been transferred to the buffer and back, and are sorted. When we in step h) move positive elements downwards, starting at the last element in this loop iteration moved to 'buf', we have two pointers, one 'neg' pointing at the next free place (containing -1) to move a positive number, and one 'pos' looking for the next positive number to move. If we find a positive number, then that element *a*[pos] is moved to *a*[neg] and 'neg' and 'pos' are decremented by one. If we find a negative number then only the 'pos' is decremented by one. This way of moving elements is order preserving in accordance with stable sorting. We stop the scan for the next positive number when 'neg' goes above the limit 'movedSoFar'+nMove', because we then have moved all elements that must be moved down. Also, if we get a 100% buffer, then step f) reduces to all elements, step h) becomes a no-operation, and the inner loop will only have one iteration. BARsort with a 100% buffer is then almost as fast as the original Radix algorithm.

```
int [] getBuffer(int len) {
    // get largest possible buffer <= len
    int [] buf = new int[1];
    while (buf.length < len) {
        try{
            buf = new int[len];
        } catch (Error o){
            len = len *7/10;
        }
    }
    return buf;
} // end getBuffer
```

Program 2. Java method for adjusting the buffer size, thus avoiding heap overflow and abnormal termination.

The reason for using a maximum value for the digit size is not primarily to save space for the supporting array count in BARsort, but to effectively utilize the first level data cache (L1), which is assumed to be 64 KB (Intel Pentium CoreDuo/M, UltraSparc or AMD Opteron) or 8-16KB (Pentium 4). With four byte integers and a maximum digit size of 9 bits, the array count, which is frequently accessed in the innermost loop of BARsort, fills up one quarter of the L1 data cache in a Pentium 4.

3. Testing the effects of buffer size

The first test of BARSort was to test the effects of the buffer size on performance. In this test we compare BARSort with Quicksort such that all numbers are normalized with the execution time for Quicksort = 1. We see that for all buffer sizes > 25 % is BARSort faster than Quicksort, and that we get a grateful degradation of the execution time and an acceptable performance if the buffer size $\geq 6\%$. In section 6, the average degradation for these buffer sizes are given. It is assumed that in most circumstances we will get a 100% buffer, so in all the rest of the figures in this paper, a full sized buffer is used.

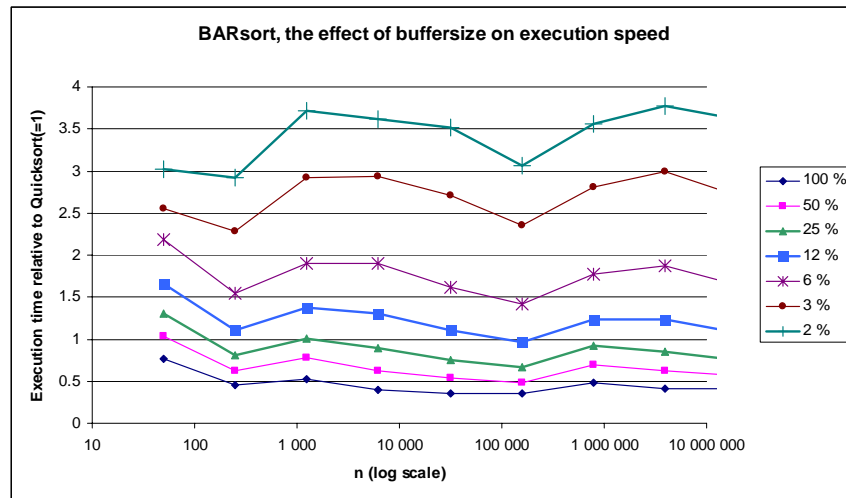


Figure 1. The effect of buffer size as a %-age of the sorted array in BARSort when data are distributed $U(n)$ and run on an AMD Opteron 254. We see that for any buffer size of 25 % or greater than the original array, BARSort is faster than Quicksort. Results are normalized relative to Quicksort for each length of the array $n = 50, 250, \dots, 97M$.

4. Comparing BARSort, Flashsort and Quicksort

By far the most popular distribution used for comparing sorting algorithms, is $U(n)$, the random uniform distribution $0..n-1$, where n is the length of the integer array. In figure 2 we use this distribution and test the relative performance of Flashsort and BARSort versus Quicksort for $n = 50, 250, \dots, 97\,656\,250$. We note that BARSort is much faster than the two other algorithms and that Flashsort performs badly when the length of the array is larger than the L2 cache and we get cache misses from L2 to main memory [16].

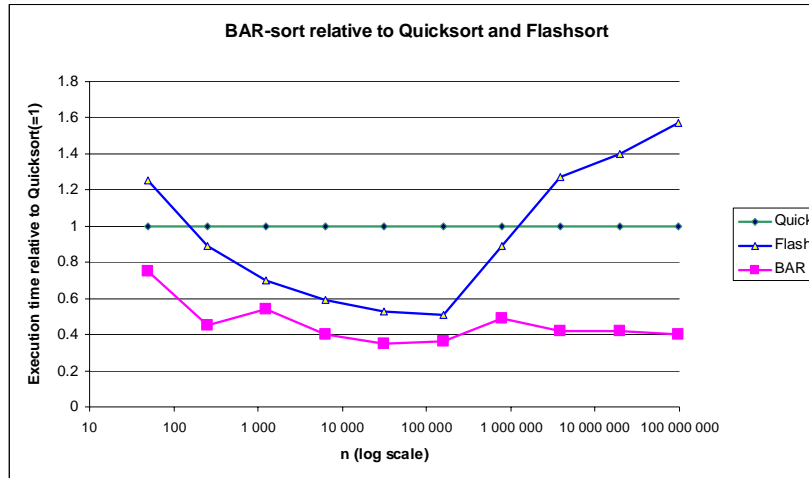


Figure 2. BARsort compared to Quicksort and Flashsort on an AMD Opteron 254 with the uniform $U(n)$ distribution. We see that BARsort is at least twice as fast as Quicksort and faster than Flashsort. Results are normalized relative to Quicksort for each length of the array.

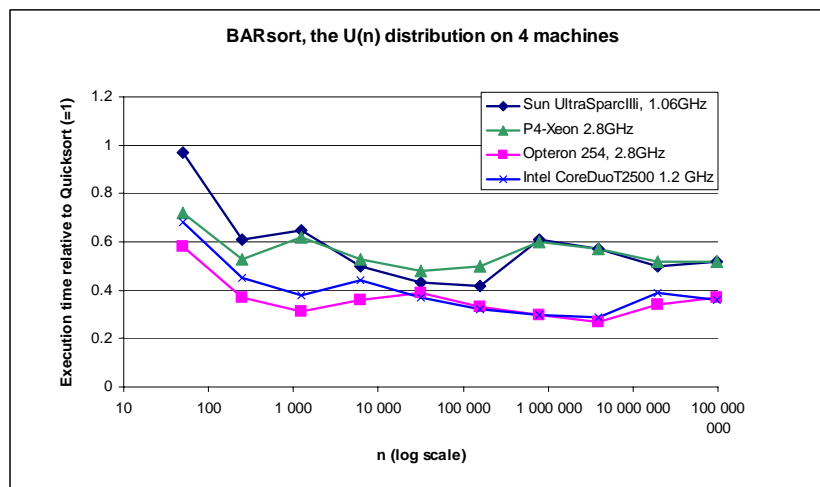


Figure 3. The relative performance of BARsort for the uniform $U(n)$ distribution on 4 different machines. Results are normalized relative to Quicksort for each length of the array $n = 50, 250, \dots, 97M$. Each data point represents the sorting of approx 97 million numbers.

The version of Quicksort tested against is the built-in version in the java API: `Arrays.sort(int[], int, int)` which also uses Insertsort as a sub-algorithm for sorting short subsections of the array. Flashsort is obtained from the Flashsort homepage [12]. Flashsort comes in two varieties, a one-pass version and a recursive descent version. The one-pass version is by far the fastest, and only results for that version are given.

The execution time for BARsort for each value of n is divided by the execution time for Quicksort to get relative performance figures. Each data point for sorting an array of length n is the average of sorting such arrays $j = M/n$ times where M = the length of the longest array in this graph. If $j > 1000$, j is set to 1000. We see that BARsort approaches twice the speed of Quicksort for $n > 1000$. The best performance for

BARsort is on a IntelCoreDuo with a large L1 cache and on the 2.8 GHz AMD Opteron 254; while the 2.8 GHz Pentium4 and the 1.06 GHz UltraSparcIIIi have the relative smallest differences between Quicksort and BARsort. In most cases, and on the average, BARsort is a clear improvement over Quicksort.

The rest of the figures in this paper are from the AMD Opteron 254. The reason for this is twofold. First, Intel has announced that it will abandon the Pentium 4 in favor of the Pentium M. UltraSparcIIIi is also an old design and the results from this Unix server are also not as reliable as the other machines because it is heavily timeshared. In short I chose the Opteron because it has a more industrial strength and is clearly the fastest machine on this integer type problem of the CPUs tested.

5. Comparing BARsort, Flashsort and Quicksort for 15 different distributions

Since not all numbers we sort are taken from a uniform distribution, it would be wise to test other distributions, of which there are as many as our imagination can create. In figures 5, the following 15 distribution are used:

1. $U(n/10)$: The uniform distribution $0:n/10-1$.
2. $U(n/3)$: The uniform distribution $0:n/3-1$.
3. $U(n)$: The uniform distribution $0:n-1$.
4. $Perm(n)$: A random permutation of the numbers $0:n-1$;
5. Sorted: The sequence $0,1,\dots,n-1$;
6. Almost sorted: A sequence of almost sorted numbers, with every 7th element randomly permuted;
7. Inverse sorted: The sequence: $n-1,n-2,\dots,1,0$
8. $U(3n)$: The uniform distribution $0:3n-1$
9. $U(10n)$: The uniform distribution $0:10n-1$
10. $U(2^{30})$: Uniform distribution $0:2^{30}-1$
11. Exponential Fib: The Fibonacci sequences starting with $\{1,1\},\{2,2\},\{3,3\},\dots$
A Fibonacci sequence is used until it reaches maximum allowable value for a positive integer or we reach n numbers, then the next sequence is used. These numbers are then randomly placed in the array. This constructs an exponential distribution.
12. Uniform $(-N:N-1)$
13. Fixed $(i\%3, \text{random})$: Fixed distribution, $i \bmod 3$ ($i=0,1,\dots,n-1$), randomly placed .
14. Fixed $(i\%29, \text{random})$: Fixed distribution, $i \bmod 29$ ($i=0,1,\dots,n-1$), randomly placed.
15. Fixed $(i\%171, \text{random})$: Fixed distribution, $i \bmod 171$ ($i=0,1,\dots,n-1$), randomly placed.

While the first 12 distributions represent values that are some function of the number of elements sorted, the last three represent data collections where there are only a fixed, small set of possible values. Such distributions are not uncommon. Say you want to sort the entire population on their gender, then you only have two or three possible values (male/female/unknown). Other such distributions are sorting people by height, home state, country. A note is appropriate on distribution 12 where we sort a mixture of positive and negative numbers. In the BARsort version tested, the array is first tested for any negative number and they are moved to the top of the array and negated to positive numbers. Then the positive and negative part of the array are sorted separately. Finally the now sorted originally negative numbers are again negated and reversed. All execution times reported includes this preprocessing.

To first get a grip on the absolute execution times, in Figure 5a the nanoseconds per sorted element for Quicksort are presented. Since there is a log scale on the x-axis, we see that for all but the three fixed distributions, Quicksort has a $O(n \log n)$ execution time. We also note that for three of the other distributions, the sorted, the inverse sorted and the almost sorted distributions, Quicksort is roughly twice as fast as for the remaining 9 distributions.

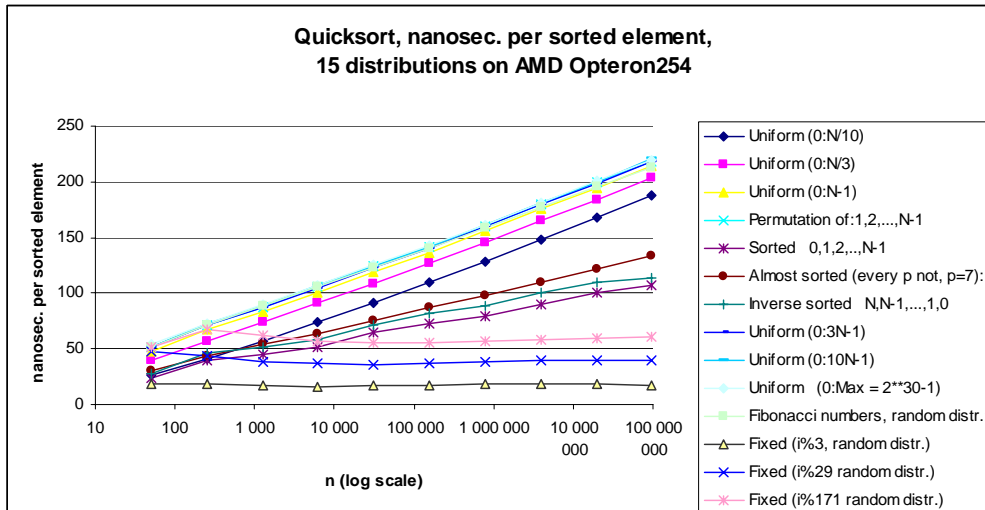


Figure 5a. Absolute performance of Quicksort (= Java Arrays.sort) for 15 different distributions described in the text. We note that Quicksort has a linear performance for the fixed distributions and is fast but $O(n \log n)$ on the sorted distributions.

We then show the results in figure 5b as BARSort versus Quicksort where all execution times are normalized with Quicksort = 1 to get relative performance figures and in figure 5c as Flashsort versus Quicksort where also all execution times are normalized with Quicksort = 1.

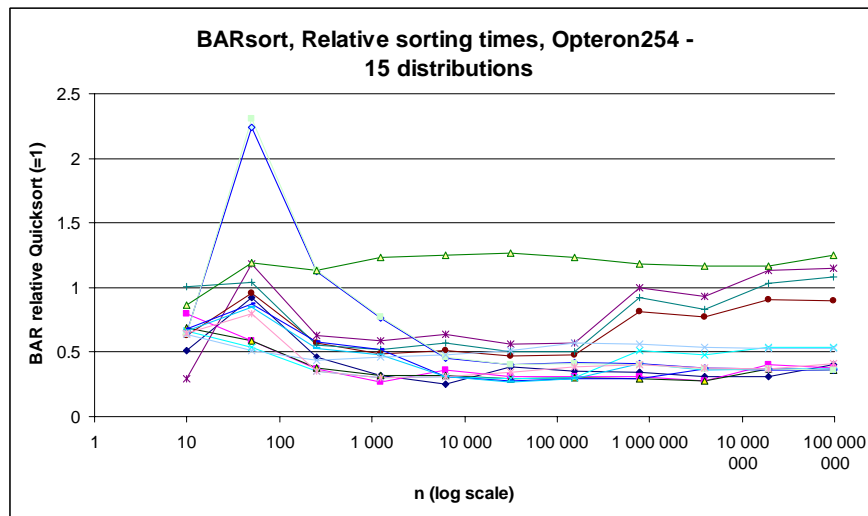


Figure 5b. The relative performance of BARSort versus Quicksort (= Java Arrays.sort) for 15 different distributions described in the text. The only distribution where BARSort is consistently slower than Quicksort, is the fixed(i%3) distribution. BARSort is also

somewhat slower for the sorted, the almost sorted and the inverse sorted distributions for large values of n . For the remaining 11 distributions, BARsort is much faster.

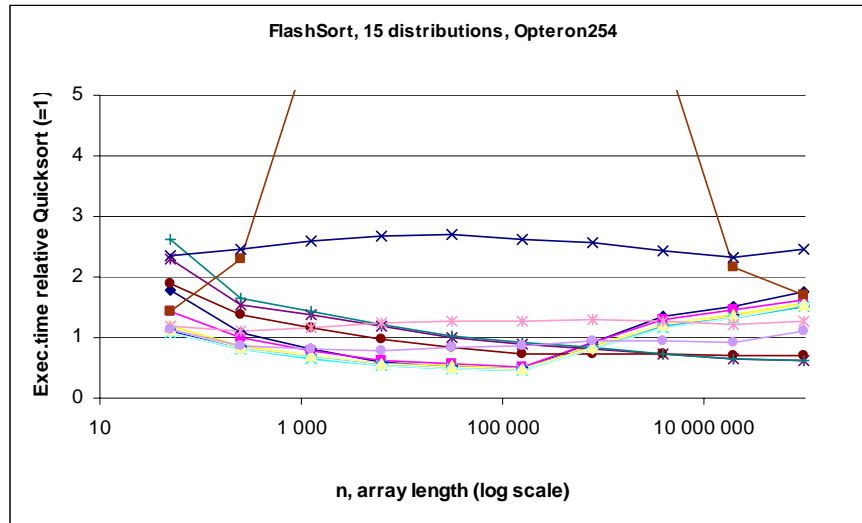


Figure 5c. The relative performance of Flashsort versus Quicksort (= Java Arrays.sort) for 15 different distributions described in the text. The one distribution where the curve goes out of the graph and back again (to a max value of 52) is the exponential Fibonacci distribution which Flashsort seems unable to handle well. Flashsort is also slower than Quicksort for 11 of the other distributions and only faster for the sorted, the almost sorted and the inverse sorted distribution.

We can conclude that for the standard test case $U(n)$, BARsort is much faster than Quicksort – on the average three times as fast for $n > 250$. For 11 other distributions out of a total of 15 BARsort is faster than Quicksort (Fig. 5b). We conclude that in general BARsort is faster than Quicksort which again is much faster than Flashsort.

6. An analytical model for BARsort

Let r denote the number of digits used for sorting the array of length n ; W and R denote a write and read of an array element; and w and r the reading and writing of a simple variable. Let $k = 100/p$ where p is the percentage of buffer used, then M_k is our model for the execution time for a buffered radix algorithm that uses k passes for each digit to sort the array. Then an implementation of Radix with always a 100% buffer in [16] can be modeled with M_1 by counting the accesses made (ignoring loop variables):

$$(1) M_1 = rn(6R + 5W)$$

When using a buffer than requires 2 or more passes through the data, a number of simple variables have to be introduced to guard the new inner loop over the buffer iterations, and in the Java implementation of BARsort used for this paper, we had:

$$(2) M_k = rn(5R + 5W + k(2R + 5r + 2w) / 2), \quad k > 1$$

It is important to note that all data structures that are accessed randomly are accessed randomly at a limited number of places (because of the maximum size of the sorting digit = 11 bits) so the caches should be able to hold all active cache lines during sorting.

If this assumption is broken, and a large sorting digit is used, the sorting may slow down by a factor of 10 or more because of cache misses from the L2 cache to main memory [16].

To evaluate the ratio M_k/M_1 , we have to determine the relative weight of W, R, w and r in an execution with almost no cache misses. We choose $W=R$ and $R=5r$ and $r=w$. The reason for weighting an array access basically 5 times as much as a simple variable, is that in Java every array access is checked to be within the array limits. It is also customary to assume that a write is more time consuming than a read but we see no reason to do this here when we assume no cache misses.

p, % buffer size	k, number of buffer iterations	model prediction M_k/M_1	Avg. measured performance relative to a 100% buffer (from fig. 1)
100	1	1.00	1.00
50	2	1.22	1.44
25	4	1.53	1.93
12	8	2.20	2.69
6	17	3.48	3.91
3	33	6.06	6.12
2	50	8.64	7.85
1	100	16.36	13.31

Table 1. The model versus measured effects of using a buffer of size equal to p% of the array to be sorted. We note that the model somewhat underestimates the effects for large values of p, but by and large the model is in good accordance with the measurements in Fig.1.

7. Conclusions

A new algorithm BARsort is presented that introduces two new features to the right radix sorting algorithm. These additions make BARsort a fast stable sorting algorithm that adapts well to any distribution of data and all machines tested and will not terminate abnormally if not enough space is found for a full sized buffer. It has been demonstrated that it is superior to Quicksort, and in almost all cases by a factor of two or more. An advantage with BARsort is that it is stable, meaning in practice that one can sort data on more than one category (say people by country, city and name) without losing the effect of one sorting in the next sorting phase. It is thus reasonable to propose that BARsort should replace Quicksort as a preferred, general purpose sorting algorithm.

It must finally be mentioned that a buffered version of the recursive decent Most Significant Radix (Left Radix) can also be constructed along the lines outlined here, and with the same good performance as BARsort.

Acknowledgement

I would like to thank Dag Langmyhr and Stein Krogdahl for many suggestions and improvements to earlier drafts of this paper.

Bibliography

- [1] V.A. Aho , J.E. Hopcroft., J.D. Ullman, *The Design and Analysis of Computer Algorithms*, second ed, .Reading, MA:Addison-Wesley, 1974
- [2] Jon L. Bentley and M. Douglas McIlroy: *Engineering a Sort Function*, Software-Practice and Experience, Vol. 23(11) P. 1249-1265 (November 1993)
- [3] Wlodzimierz Dobosiewicz: *Sorting by Distributive Partition*, Information Processing Letters, vol. 7 no. 1, Jan 1978.
- [5] C.A.R Hoare : *Quicksort*, *Computer Journal* vol 5(1962), 10-15
- [6] Donald E. Knuth: *The art of computer programming - vol.3 Sorting and Searching*, Addison-Wesley, Reading Mass. 1973
- [7] Anthony LaMarcha & Richard E. Ladner: *The influence of Caches on the Performance of Sorting*, *Journal of Algorithms* Vol. 31, 1999, 66-104.
- [8] Jan van Leeuwen (ed.), *Handbook of Theoretical Computer Science - Vol A, Algorithms and Complexity*, Elsevier, Amsterdam, 1992
- [9] Arne Maus, *Sorting by generating the sorting permutation, and the effect of caching on sorting*, NIK'2000, Norwegian Informatics Conf. Bodø, Norway, 2000 (ISBN 82-7314-308-2)
- [10] Arne Maus. *ARL, a faster in-place, cache friendly sorting algorithm*. in NIK'2002, Norwegian Informatics Conf, Kongsberg, Norway, 2002 (ISBN 82-91116-45-8)
- [11] Dietrich Neubert, *Flashsort*, in Dr. Dobbs Journal, Feb. 1998
- [12] Flashsort, <http://www.neubert.net/Flacodes/FLACodes.html>
- [13] Stefan Nilsson, *The fastest sorting algorithm*, in Dr. Dobbs Journal, pp. 38-45, Vol. 311, April 2000
- [14] Robert Sedgewick, *Algorithms in Java*, Third ed. Parts 1-4, Addison Wesley, 2003
- [15] Mark Allen Weiss: *Datastructures & Algorithm analysis in Java*, Addison Wesley, Reading Mass., 1999
- [16] Arne Maus and Stein Gjessing: *A Model for the Effect of Caching on Algorithmic Efficiency in Radix based Sorting*, to be presented at The Second International Conference on Software Engineering Advances, ICSEA 25.Aug. 2007, France
- [17] BARsort code: <http://www.ifi.uio.no/~arnem/BARsort>