

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

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 maximum value of the elements in the array. This makes BARsort somewhat faster than ordinary 8-bit Radix sort (Radix8). The second and most important improvement is that data is transferred back and forth between the original array and a buffer that can be 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 to acceptable performance. This result is also explained analytically. This flexibility in memory requirement 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 and the ordinary Radix8, which is stable. This is done for 14 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, Flashsort and Radix8 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 candidate for a general replacement for Quicksort because of its faster performance and its stable sorting quality.

**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, 4].

Sorting algorithms are still being developed, like Flashsort [11], “The fastest sorting algorithm” [13], ARL [10] and Permsort [9]. The most influential sorting algorithm introduced since the 60’ies 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

---

<sup>1</sup> This paper was presented at the NIK 200x conference. For more information see <http://www.nik.no/>

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  $O(n \log n)$ . 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, Bucket sort and Counting 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. Counting sort is like Radix but using only one large digit to sort the array in one pass. Because of the effects of caching in modern CPUs, this has been demonstrated to be ineffective for large values of  $n$  [16]. Counting sort also has a space requirement of two additional arrays of length  $n$  – more than almost any other sorting algorithm. Flashsort, used for comparison in this paper, is a modified version of Counting sort, needing two extra arrays that in sum is  $1.1n$ .

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 effect of caching in modern CPUs has a large impact on the performance on various sorting algorithm. In [16] it is demonstrated that if one uses a too large digit in radix based sorting, one might experience a slowdown factor of 10 or more when sorting large arrays. In [17] the effects of the virtual memory system and especially the effects of the caching of the TLB tables are analyzed. The effects of caching are obviously of great importance for the absolute performance of algorithms, but somewhat outside the scope of this paper which is to propose two additions to the traditional Radix sorting algorithm and to investigate how the resulting algorithm compare relative to other well known sorting algorithms.

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 with Radix8, Quicksort and Flashsort for the  $U(n)$  distribution on 4 different CPUs is also presented. To evaluate these relative execution times, the absolute execution times for Quicksort is also given in Figure 4a. Then Radix8, BARsort and Flashsort are tested for 14 different distributions. On the average as well, as in almost all cases, BARsort is shown to be somewhat faster than Radix8 and much faster than Flashsort and Quicksort. Finally, we conclude the paper by proposing BARsort as a candidate for replacing Quicksort.

## **Radix Sorting Algorithms and BARsort**

Basically, there are two Radix sorting algorithms, Left Radix, also called MSD (Most Significant Digit) Radix and Right Radix or LSD (Least Significant Digit) Radix. 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 that sorted order. 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 the 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.

The reason for setting a limit on 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 8 bits might be appropriate for the Pentium 4 while 12 bits might be optimal for Pentium Ms and AMD processors (determined so that the central data structure, an integer array of size  $2^{\text{numBit}}$  fits well into the L1 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 Counting sort or Radix8, your might 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++ usually 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% or more of the size of original array is available on the heap as a buffer.

## 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. A good example is the result from a database query where you want to sort the resulting table on more than one key, say sorting invoices first on date and then on sender. Using a stable sorting algorithm, you could first sort all invoices on date and then on sender. If you used an

unstable algorithm when sorting on sender, the invoices would lose the order they had on date.

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 is presented here in pseudo code. It is an iterative Right-radix algorithm with a dynamically determined number of bits in its sorting digits.

```

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

        if <number of buffer iterations == 1> {
            special case with a 100% buffer
            f) do an ordinary Radix sort but with a digit bits wide
        } else {
            movedSoFar = left;

            while <more buffer iterations> {
                g) 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).
                h) move these element in sorted order to buf starting at movedSoFar in
                   a and replace these moved elements with -1 in a.
                i) 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.
                j) copy buf[0..nMove] to a[movedSoFar..movedSoFar+nMove]

                movedSoFar += nMove;
            }
        }
    }
}

```

Program 1. Pseudo code for BARsort.

BARsort is optimized, as Quicksort is, with Insertsort [15, 4] for sorting very short arrays (not shown in the pseudo code since no test results are presented sorting such short arrays). The actual Java code for BARsort and Radix8 can be found in [18] where this pseudo code for efficiency reasons is split into three methods.

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 space available 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.

To find 'bits', the number of bits in the next sorting digit in step c), we first find out how many digits we need if we used the maximum number of bits allowed each time, (here 12 bit). Then the sorting digits are calculated as closely as possible to the average size using that many digits. Say the maximum value to sort is 26 bits wide. This gives 3 sorting digits, and instead of using digits sized 12, 12 and 2 bits wide, this procedure gives sorting digits 9, 9, and 8 bits wide. Because some of the loops in BARsort are of  $O(2^{\text{bits}})$  while other are  $O(n)$ , this choice of sorting digits produces less work than the first alternative and makes BARsort with a 100% buffer faster than Radix8 .

Steps d), e) and 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 with the exception that -1 has been stored in all elements in a.

```
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.

### 3. Testing the effects of buffer size

We first test the effects of the buffer size on the performance of BARSort. In this test as in the rest of the figures, except in figure 4a, 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  $\geq 25\%$  BARSort is faster than Quicksort, and that we get a grateful degradation of the execution time and an acceptable performance for all buffer sizes  $\geq 6\%$ . In Table 2 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. If that is not the case, then these delay factors can be used to multiply the other data presented.

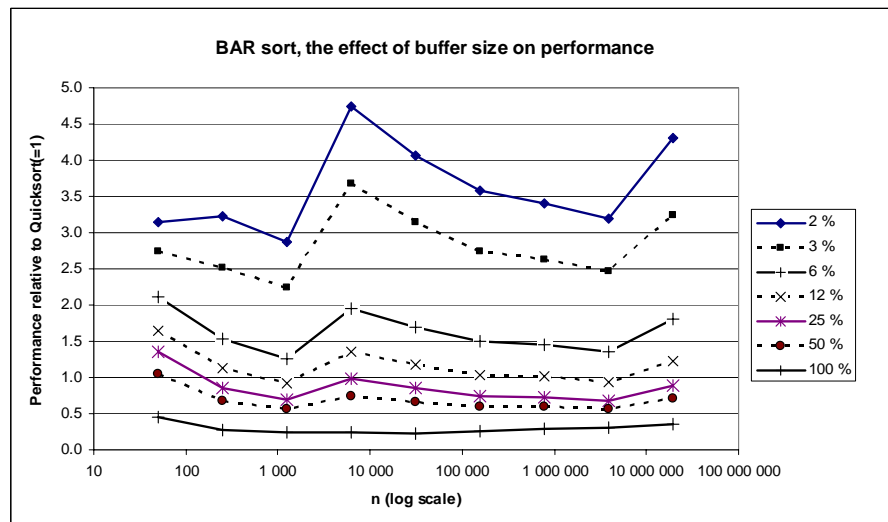


Figure 1. The effect of buffer size as a %-age of the sorted array in BARSort when data are distributed  $U(n)$  and run on an AMD Opteron 254. Results are normalized relative to Quicksort for each length of the array  $n = 50, 250, \dots, 97M$ .

### 4. Comparing BARSort, Radix8, 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 BAR-sort is faster than Radix8 and much faster than the two other algorithms and also note that Flashsort performs badly when the length of the array is larger than the L2 cache and we get cache misses to main memory [16].

The version of Quicksort tested against is the built-in Suns version in the java 1.6 version 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 three to four times 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.

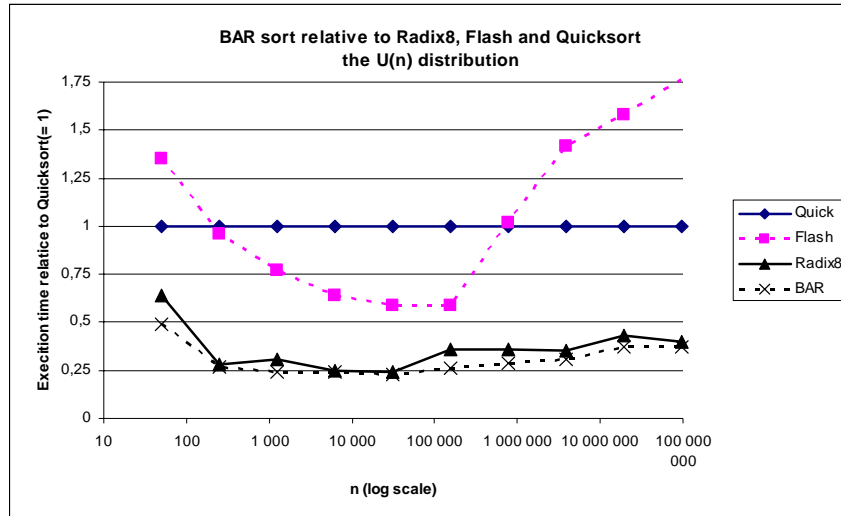


Figure 2. BARSort compared to Radix8, Quicksort and Flashsort on an AMD Opteron 254 with the uniform  $U(n)$  distribution. We see that BARSort is faster than Radix8, three times as fast as Quicksort and much faster than Flashsort. Results are normalized relative to Quicksort for each length of the array.

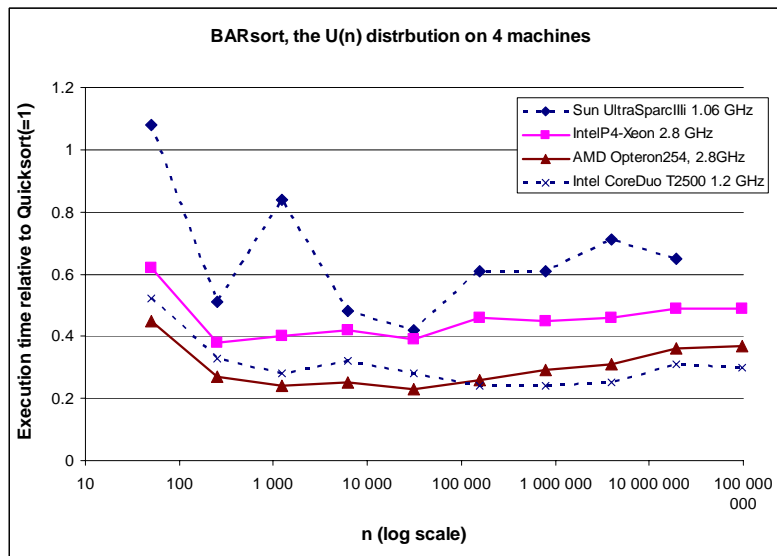


Figure 3. The relative performance of BARSort for the uniform  $U(n)$  distribution on 4 different machines. Results are normalized relative to Quicksort for each length of the array  $n = 50, 250, \dots, 97M$ .

The Radix8 algorithm is an algorithm that first finds the maximum value to sort and then uses as many 8 bit digits it needs to sort the array. It then uses the same method as BARSort in the special case when BARSort has a 100% buffer for sorting each digit (i.e. Counting sort on that digit), but with the exception that Radix8 always uses an 8 bit digit size where as BARSort can vary this digit size. This might not only result in a different digit size for a given digit, but BARSort might also sometimes use fewer digits than Radix8 because it can use a digit size of 12 bits while Radix8 is restricted to 8 bits.

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/Core Duo design. 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 and started swapping when I tried to sort an array of length 97M. Hence, for the UltraSparc, only figures up to 20M is available. In short I chose the Opteron because it has more industrial strength and is clearly the fastest machine on this integer type problem of the CPUs tested.

## 5. Comparing BARSort, Radix8, Flashsort and Quicksort for 14 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 4, the following 14 distributions 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 7<sup>th</sup> 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. The first sequence then starts with 1 and 1, the second with 2 and 2, ... These numbers are then randomly placed in the array. This constructs an exponential distribution.
12. Fixed ( $i\%3$ , random): Fixed distribution,  $i \bmod 3$  ( $i=0,1,\dots,n-1$ ), randomly placed.
13. Fixed ( $i\%29$ , random): Fixed distribution,  $i \bmod 29$  ( $i=0,1,\dots,n-1$ ), randomly placed.
14. Fixed ( $i\%171$ , random): Fixed distribution,  $i \bmod 171$  ( $i=0,1,\dots,n-1$ ), randomly placed.

While the first 10 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.

To first get a grip on the absolute execution times, in Figure 4a 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 8 distributions.

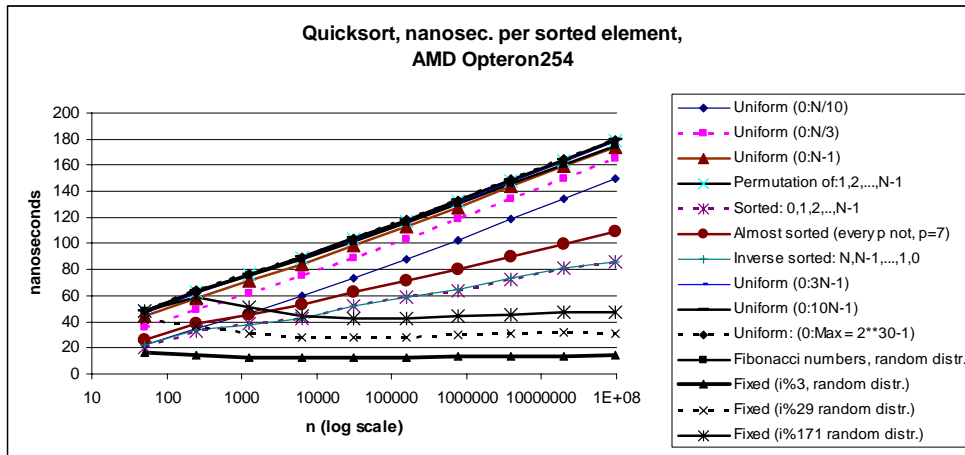


Figure 4a. Absolute performance of Quicksort (= Java Arrays.sort) for 14 different distributions described in the text. We note that Quicksort has a linear performance for the fixed distributions and is fast but  $O(n \log n)$  on the sorted distributions.

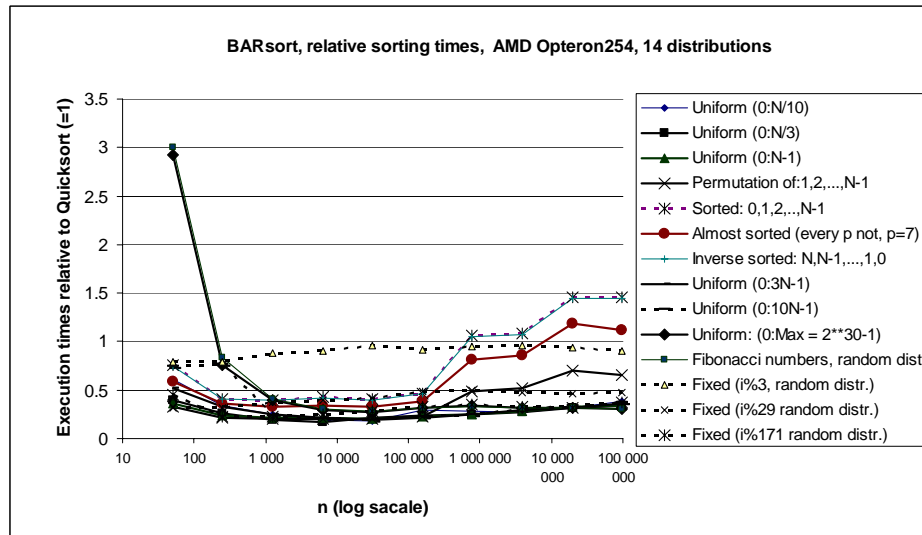


Figure 4b. The relative performance of BARsort versus Quicksort (= Java Arrays.sort) for 14 different distributions described in the text. Not for any distribution is BARsort consistently slower than Quicksort. but BARsort is somewhat slower for the sorted, the almost sorted and the inverse sorted distributions for large values of  $n$ , and for  $n < 250$ , BARsort is slower for the Fibonacci distribution. For the remaining 10 distributions, BARsort is always faster.

From figures 2 and 3, we can conclude that for the standard test case  $U(n)$ , BARsort is much faster than Quicksort – on the average more than three times as fast for  $n > 250$ . For 10 distributions out of a total of 14 tested distributions, BARsort is always faster than Quicksort [Fig. 4b] and on the average faster for all tested distributions [Table 1]. We conclude that in general BARsort is faster than Radix8 and much faster than Quicksort, which again is faster than Flashsort.

	<b>Quick</b>	<b>Flash</b>	<b>Radix 8</b>	<b>BAR</b>
Uniform(n/10)	1	1.16	0.40	0.31
Uniform(n/3)	1	1.07	0.33	0.27
Uniform(n)	1	0.99	0.32	0.27
Permutation of(1..n)	1	0.93	0.42	0.37
Sorted	1	1.12	0.92	0.79
Almost sorted	1	0.98	0.73	0.63
Inverse sorted	1	1.20	0.92	0.78
Uniform (3n)	1	0.95	0.32	0.29
Uniform (10n)	1	0.94	0.36	0.37
Uniform(2 <sup>30</sup> )	1	0.93	0.52	0.62
Fibonacci	1	16.12	0.53	0.64
Fixed random(i%3)	1	2.75	1.03	0.90
Fixed random(i%29)	1	1.31	0.46	0.42
Fixed random(i%171)	1	0.97	0.33	0.31
<b>Grand total average</b>	<b>1</b>	<b>2.245</b>	<b>0.543</b>	<b>0.499</b>

Table 1. The relative performance of Flashsort, Radix8 and BAR sort versus Quicksort presented as average values over all lengths of arrays sorted (n= 50, 250,...,97M) and as grand total averages for the 14 different distributions described in the text.

## 6. An analytical model for BARsort

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

$$(1) M_1 = n[R + d(5R + 4W)]$$

When using a buffer size than require 2 or more passes through the data, we have by careful counting in the Java implementation [18]:

$$(2) M_b = n[R + d(5R + 5W + b(2R + W)/2)], \quad b > 1$$

The difference between the formulas is that  $M_b$  contains an extra  $W$  for marking moved elements with -1, and the last term is one  $R$  for finding elements to move to the buffer and one  $R$  and one  $W$  for compacting the array.

It is important to note that all data structures that are accessed randomly, are done so at a limited number of places (because of the maximum size of the sorting digit =12 bits) so the L1 and L2 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  $M_b/M_1$  in order to find out how much performance is degraded by using smaller buffers, we have to determine the relative weight of  $W$  and  $R$

in an execution with almost no cache misses. Since this ratio is almost insensitive to the relative weight of these two operations,  $W=R$  was chosen for simplicity.

p, % buffer size	b, number of buffer iterations	Avg. measured performance relative to a 100% buffer (from fig. 1)	model prediction $M_b/M_1$	$1.6 \cdot (\text{model prediction})$ $M_b/M_1$ for $b > 1$
100	1	1.00	1.00	1.00
50	2	2.26	1,42	2,27
25	4	2.85	1,74	2,78
12	8	3.82	2,37	3,79
6	17	5.38	3,26	5,22
3	33	9.35	6,32	10,11
2	50	12.02	8,95	14,32

Table 2. 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 underestimates the observed effects by a constant factor 1.6 for buffer sizes between 50% and 6% of the array.

Explaining the factor 1.6, we note that model is rather crude by ignoring the access times to simple variables in the loops and the loop overhead and that the code is much shorter, 140 vs. 30 lines of code, and more importantly, the central loops have fewer statements when doing a full sized buffer than the code we execute when we have a smaller buffer. Hence the 100% buffer code fits much better into the L1 program cache which again makes it faster. This said, it is still reasonable to claim that the above analytical model is by a small constant factor a fairly good description of the performance of BARsort.

## 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 somewhat faster than the ordinary 8bit Radix algorithm when it has a 100% and superior to Quicksort and Flashsort, and in almost all cases by a factor of three or more against Quicksort. 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 could 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) has also be constructed along the lines outlined here, and with most of the same good performance as BARsort.

## Acknowledgement

I would like to thank Dag Langmyhr, Stein Krogdahl and three anonymous referees who made many suggestions for improvements to 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.
- [4] [http://en.wikipedia.org/wiki/Sorting\\_algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm)
- [5] C.A.R Hoare : *Quicksort*, *Computer Journal* vol 5(1962), 10-15
- [6] Donald Knuth :*The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Second Edition. Addison-Wesley, 1998.
- [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*, The Second International Conference on Software Engineering Advances, ICSEA 25.Aug. 2007, France
- [17] Naila Rahman and Rajeev Rahman, *Adapting Radix Sort to the Memory Hierarchy*, *Journal of Experimental Algorithmics (JEA)* ,Vol. 6 , 7 (2001) , ACM Press New York, NY, USA
- [18] BARsort and Radix8 code: <http://www.ifi.uio.no/~arnem/BARsort>