Making a fast unstable sorting algorithm stable[1]

Arne Maus, arnem@ifi.uio.no
Department of Informatics
University of Oslo

Abstract
This paper demonstrates how an unstable in place sorting algorithm, the ALR algorithm, can be made stable by temporary changing the sorting keys during the recursion. At 'the bottom of the recursion' all subsequences with equal valued element are then individually sorted with a stable sorting subalgorithm (insertion sort or radix). Later, on backtrack the original keys are restored. This results in a stable sorting of the whole input. Unstable ALR is much faster than Quicksort (which is also unstable). In this paper it is demonstrated that StableALR, which is some 10-30% slower than the original unstable ALR, is still in most cases 20-60% faster than Quicksort. It is also shown to be faster than Flashsort, a new unstable in place, bucket type sorting algorithm. This is demonstrated for five different distributions of integers in a array of length from 50 to 97 million elements. The StableALR sorting algorithm can be extended to sort floating point numbers and strings and make effective use of a multi core CPU.[2]

Keywords: stable sorting, radix, most significant radix, multi core CPU, Quicksort, Flashsort, ALR.

1. Introduction, sorting algorithms

Sorting is maybe the single most important algorithm performed by computers, and certainly one of the most investigated topics in algorithm design. Numerous sorting algorithms have been devised, and the more commonly used are described and analyzed in any standard textbook in algorithms and data structures [Weiss, Aho] or in standard reference works [Knuth, van Leeuwen]. May be the most up to date coverage is presented in [Sedgewick]. The most influential sorting algorithm introduced since the 60'ies is no doubt the distribution based 'Bucket' sort which can be traced back to [Dobosiewicz]. New sorting algorithms are still being developed, like Flashsort [Neubert], an unstable, bucket type sorting algorithm claiming to be faster than Quicksort [Hoare], and 'The fastest sorting algorithm" [Nilsson].

Sorting algorithms can be divided into comparison and distribution based algorithms. Comparison based methods sort by only comparing elements with each other in the array that we want to sort (for simplicity assumed to be an integer array 'a' of length n). It is easily proved [Weiss] that the time complexity of a comparison based algorithm is at best O(*n*logn*). Well known comparison based algorithms are Insertion sort, Heapsort and Quicksort. Distribution based algorithms on the other hand, sort by using directly the values of the elements to be sorted. These algorithms sort in O(*n*log M*) time where M is the maximum value in the array. Well known distribution based

---

[1] *This paper was presented at the NIK-2006 conference. For more information, see* `//www.nik.no/`

[2] *Due to new laws concerning intellectual rights, this algorithm is the property of the University of Oslo, it is: ©Arne Maus 2006, it is free for any academic use, and might be protected by a patent.*

algorithms are Radix sort in its various implementations [Hildebrandt and Isbitz] and Bucket sort.

Quicksort is still in most textbooks regarded as the in practice fastest known sorting algorithm [Weiss] Its good performance is mainly attributed to its very tight and highly optimized inner loop [Weiss]. However, many papers claim that Radix sort algorithms outperform Quicksort [Andersson&Nilsson].

Recent research has been into theoretically proving lower upper bounds for integer sorting [Han] [Anderson et al] assuming a unit cost Random Access Memory – which is far from the actual case with cached memories on 2 or 3 levels on a modern CPU. Now access to data in the L1 cache might be up to 100 times faster than accessing the main memory[Maus00]. On the other hand we find practical algorithms with benchmarks. Improved versions of Radix sorting with insertion sort as a sub algorithm has been introduced by [McIllroy et al] – the American Flag Algorithm and improvements on this in [Andersson&Nilsson]. Both use a MDS radix algorithm – the first uses a fixed radix size while the latter alternates between an 8 bit and a 16 byte sorting digit. They are not in place sorting algorithms, but copy data to a buffer of the same length as the array to be sorted. These algorithms are stable. The ALRsort introduced in [Maus02] can be though of an improvement on these sorting algorithms by using a dynamically determined radix size between 3 and 11 bits and doing in place sorting with permutation cycles, but then ARLsort is no longer stable. This paper introduces an addition to this algorithm that makes it stable.

Even though the investigation is based on sorting non-negative integers, it is well known that any sorting method well suited for integers is also well suited for sorting floating point numbers, text and more general data structures [Knuth]. More on this in section 8. Most recent research in sorting is on using merge sort or radix sort for sorting strings [Sins & Zobel]. A comparison based, in place, not stable sorting algorithm is presented in [Franceschini & Geffert].

2. Stable sorting

Stable sorting is defined as equal valued elements on input shall be presented in the same order on output – i.e. if we sort 2,1,1 and obtain 1,1,2 – the two 1's on input must not be interchanged on output. This is an important feature. 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 is then no longer sorted on arrival date, which I also want. The same valued elements (on sender) that was once in correct order on date, is now rearranged because the sorting algorithm used in the e-mail reader was not stable. A more significant example is data base sorting where one wants the output in succession ordered by more than one key.

Generally, stable sorting is a harder problem, because a stable sorting algorithm is rearranging input according to the unique permutation of input that will produce the stable sorted output; where as a non stable sorting algorithm only has to find one among many permutations (if there are equal valued elements) that produces a non stable permutation of input. It is then expected that stable sorting algorithms should use more time and/or space than unstable sorting algorithms. In general, stable sorting is never a disadvantage, and often a desirable property of any sorting algorithm.

Few of the well known sorting algorithms are stable. Unstable sorting algorithms include Shell sort, Heap sort, Quicksort and Bucket sort. Stable sorting algorithms include Insertion sort, Merge sort and ordinary Radix (from right to left) . The algorithm presented in this paper, the ALR algorithm, is an in place Most Significant Digit (MSD) Radix-sorting algorithm (i.e. radix sorting recursively from left to right) and is not stable as originally proposed [Maus02].

3. The unstable ALR sorting algorithm

The ALR (Adaptive Left Radix) algorithm was introduced with source code in [Maus02]. The pseudo code in program 1 describes the algorithm.

1. First find the maximum value in a, and hence the leftmost bit set in any number in the array a.

2. For the segment of the array **a** we sort on, decide a digit size: numBits less or equal r bits for the next sorting digit, and count how many there are of each value for this digit (all digits to the left of the segment of 'a' you sort, now have the same value)

3. We have now found the $2^{numBits}$ '**boxes**' and their sizes for each value of current digit: Move elements in permutation cycles until all have been moved, repeat:
   Select the first element **e1**(from top) that hasn't been moved, from the position **p1** in the array (in the box **b1,** the first box from top that hasn't had all its elements moved). The value of the sorting digit in **e1**, we call **v1.**
   a) Find out where (in which box **b2**) it should be moved
   b) Find the first not moved position **p2** in **b2**
   c) Place **e1** in **p2** and repeat a-c, now with the element **e2** (with digit value **v2**) from **p2** (later **e3** with digit value **v3** from **p3**,...) until we end up in some **bk** = **b1** (where the cycle started), and we place **ek** there in position **p1**.

4. For each box of length > 1of the current digit, call ALR recursively (to sort on the next digit) – i.e. perform 1,2 and 3 until there are no more digits to sort on
   (an obvious speed optimization is here to use Insertion sort if the length of the box is < some constant – say 30 - and terminate this recursive branch).

Program 1. *The pseudo code for the Unstable ALR sorting algorithm. The step 3 with the in-place permutation cycle, makes this an unstable algorithm.*

Before explaining the pseudo-code we define two terms: A sorting ***digit*** in ALR is a field in the sorting key consisting of a number of bits, and is not confined to say an 8-bit byte. The A in ALR stands for adaptive, and ALR adapts numBits = the width of the current sorting digit up to a maximum of r bits (this upper limit of r bits is justified by our wish to have all central data structures in the L1 and L2 caches. We have found by testing that r =11 is a balanced choice for the present sizes of caches, [Maus02]). If s is the number of elements sorted by the current recursive instance of ALR, then numBits is set such that : $2^{numBits} \geq s > 2^{numBits-1}$ if possible within the maximum of r bits – otherwise a digit of r bits is used. The first step in ALR, as in any Radix algorithm, is to count how many there are of each of the $2^{numBits}$ possible values of the sorting digit. We then say that the sorting digit defines $2^{numBits}$ ***boxes***, each of length =

the number of elements in the subsection of the array having that value. In short, a sorting digit of width numBits defines $2^{numBits}$ boxes (We will later use the index numbers of these boxes: b1, b2,.. to make ALR stable)

4. Making ALR stable, pseudo code

We now want to make the ALR stable by changing step 3. The ability to do that rests on the following two observations:

1. Even though ARL is not stable, same valued elements from the same box are not reordered relative to each other by the permutation cycle, because we pick elements from a box from the top and downwards. Such equal valued elements from the same box also end up in the same box after sorting on the current digit in the same relative order.
2. When we sort by recursive decent, the part of the array a recursive instance of ALR is sorting, have for all digits to the left of the current digit we sort on, the same value for all digits. These values are not used further on in the recursion. Furthermore, after having found a value **vi** for an element **ei** in a box **bi** for the current sorting digit, the place this value **vi** occupied in **ei** is not used/read any more during the ARL-sorting, and can be replaced with any other value temporarily (if we on backtrack can reintroduce the original value **vi** in that digit and that element). We note that this place that **vi** occupied is numBits wide – the same width as the box-indexes.

On the recursive decent phase, we substitute the values in the elements with the box indexes *these elements came from* for each digit as we descend. This is illustrated in figure 1. When we have sorted on all digits in the array, then all elements will only contain the box-indexes they came for every recursive step of the ALR algorithm.

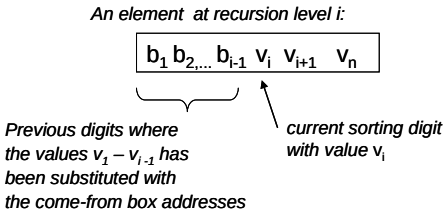

Figure 1. *An element during recursive descend at recursive level i.*

After having sorted on all digits, we sort with a stable sorting algorithm (on these substituted come-from-box-indexes in the elements) all boxes of the last sorting digit of length >1. These boxes are the equal valued subsequences. Then any reshuffling of equal valued elements that might have taken place during the permutation cycles, will be rearranged to their original order from input. (The reason we need a stable sorting algorithm here, is that equal valued elements that start up in the same box might end up reversed on input if this final sorting stage is not stable). The pseudo code for StableALR is presented in program 2.

The reason that program 2 specifies a stable sort, is that we now only sort with a stable sorting algorithm those subsections that contains equal valued elements, and among

these: If they started the same come-from box, they will not be reshuffled (by the observation 1), and if they appeared in different boxes, the stable sorting will reorder them according to their original ordering from input.

1. First find the maximum value in a, and hence the leftmost bit set in any number in the array a.

2. For the segment of the array a we sort on, decide a digit size: numBits less or equal r bits for the next sorting digit and count how many there are of each value for this digit (all digits to the left of the segment of 'a' you sort, now have the same value)

3. We have now found the $2^{numBits}$ 'boxes' and their sizes for each value of current digit:
4. Move elements in permutation cycles until all have been moved, repeat:
    i. Select the first element e1 (from top) that hasn't been moved, from the position p1 in the array (in the box b1, the first box from top that hasn't had all its elements moved). The value of the sorting digit in e1, we call v1.
        a) **Replace v1 with b1 in e1:**
        b) Find out where (in which box b2) it should be moved
        c) Find the first not moved position p2 in b2
        d) Place e1 in p2 and repeat a-c, now with the element e2 (with digit value v2) from p2 . **and replace v2 with b2 in e2** (later e3 with digit value v3 from p3,... and replace..) until we end up in some bk = b1 (where the cycle started), and we place ek there in position p1.

5. For each box of length > 1of the current digit (all elements in each box now having the same value **v** for the sorting digit): call ALR recursively (to sort on the next digit) – i.e. perform 2,3 and 4 until there are no more digits to sort on, **and then sort the elements in all boxes of the last digit of length >1 with some stable sorting algorithm – say Insertion sort (the sorting is then done on all the substituted b-values in the elements)**.

6. **On backtrack after every call to ARL, substitute the values v back in the current digit in each element.**

Program 2*. The pseudo code for the Stable ALR sorting algorithm. Additions to the UnstableALR algorithm are highlighted*

One might ask what's the point about a second sorting phase with a stable sorting algorithm, why don't we just sort the array with the stable sorting algorithm in the first place. The answer to that is twofold. First, this second sorting phase is only performed for equal valued subsections. For most distributions we sort, these subsection are short and relatively far between (any such section is in practice at most 10-50 elements long, and is easily sorted with no extra space overhead by Insertion sort). The second argument is that we now sort (much) smaller subsections independently and that is much faster than sorting the whole array in one piece since the execution time of our algorithms are basically of order O(*n logn*), larger than linear. The sorting time of the distribution U(n/10) in figure 5 underpins this argument – all elements here belong to some equal valued subsection of the array, but since we now effectively have divided the array into subsections of an average length of 10, they are each stably sorted very

efficiently by Insertion sort. However, for many common distributions, most elements don't belong to such a section, and are hence only sorted once. Notable exceptions to this will be treated in section 6.

The obvious optimization in using Insertion sort if the length of a box is less some constant and terminate this recursive branch is also done here, but before we do Insertion sort on the elements now containing some box addresses in its leftmost digits and some values not sorted on in its right digits (figure 1), we interchange the box-parts and value parts in all elements before sorting. By making the box addresses the least significant part, we get a stable sorting if this subsection contains equal values elements. After Insertion sort of such a short section, we interchange the value and box address parts again in the elements.

5. A worked example

To illustrate the stable sorting ALR, we will demonstrate how it works for sorting 2,1,1. This short example will demonstrate all essential features of ALR (but not a permutation cycle longer than 2 element and not sorting subsequences shorter than 30 with Insertion sort). To get two digits to sort on in this example, will use a one bit digit size and hence two sorting digits. We also represent the numbers in binary notation to show their binary digits, First we do the unstable ARL with only permutation cycles, and then stable ALR with substitution of values with come-from box addresses, and back substitution of box-addresses with values on backtrack.

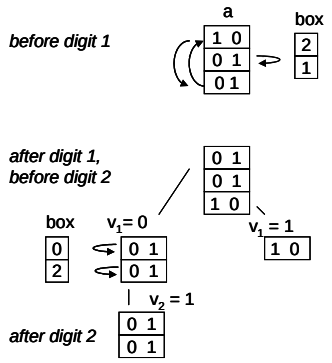

Figure 2. *Unstable ALR. We first sort on leftmost bit and the permutation cycle exchanges the 2 and the last 1. Then recursively call for the second digit on the subsection containing the two 1 s in the second digit (exchange them with themselves) and return with an unstable sorted array.*

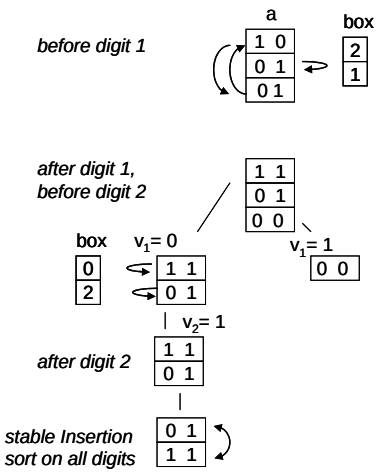

Figure 3. *Stable ALR. We first sort on the leftmost bit. The permutation cycle will exchange the 2 and the last 1. Then we substitute the box addresses for the values v_1 in all elements (since the last 1 came from the 1 box, it gets a 1 substituted for a 0 in the first digit. Like wise the 2 came from box 0 in the first digit and gets a 0 substituted for the value 1) Then we recursively call for the second digit on the subsection containing the two 1s in the second digit (exchange them with themselves). Again we substitute the come-from box addresses for the values for the second digit (since these are all 1, it can't be seen as a change in the figure). We note that we don't do a recursive call for the value=1 for the first digit since its box has length =1. We then stable sort the section having value = 1 based on the substituted box addresses. On backtrack we substitute the value v_2 =1 for digit 2,and v_1 =0 for digit 1 as we backtrack and end up with a stable sorted array.*

6. Comparing stable and unstable ALR, Flashsort and Quicksort

To test how fast StableALR is and how much overhead we introduce compared to UnstableALR, we tested these two sorting algorithm with Quicksort and Flashsort, a bucket type sorting algorithm claiming to be faster than Quicksort.

To test sorting of n integers, and when the largest number of n elements in a we test for is nMax, then an array a of length nMax is declared, and n/nMax sections of length n and are each filled with the selected distribution of the numbers. Then all sorting methods are given these subsections to sort, one after the other, and the execution time to sort n integers is then the average for sorting these n/nMax sections of the array. All performance for each value of n are presented relative to the Quicksort execution time, i.e. we divide the absolute execution time by the execution time for Quicksort for each n. Lower than 1 values for a sorting algorithm then shows that it sorts faster than Quicksort. The version of Quicksort used is Arrays.sort from the java.util package, and the Java source code for Flashsort is taken from its homepage [Neubert]. All sorted sections of a is checked for $a[i] \leq a[i+1]$, i=0,1,..,n-2, and that StableALR really is stable, an accompanying array p, initiated :1,2,...,n was declared. By doing exactly the same interchanges in p as in a, we could in the end check that equal values sequences in a was sorted in the right order. All tests were performed for n= 10,50, 250,...., 19531250, 97656250. The only exception to this is the figures for a Pentium M laptop, where the last data point could not be calculated because of memory limitations.

6.1 The standard test – a uniform U(n) distribution

In figure 4 we used the standard test case, sorting n numbers that are taken from the uniform 0..n-1 distribution – U(n). We here see that for n > 50, both stable and unstable ALR are much faster than Quicksort. Flashsort however starts out faster than Quicksort, but ends up slower. For all other distributions tested (not all reported here) Flashsort displayed this (or worse) behavior and thus no more results are displayed for Flashsort. We also conclude that the overhead in making ALR stable is in the range 10-30% for this distribution. Since the U(n) distribution produced many, but short sequences of equal valued elements, we see that sorting these with Insertion sort is effective.

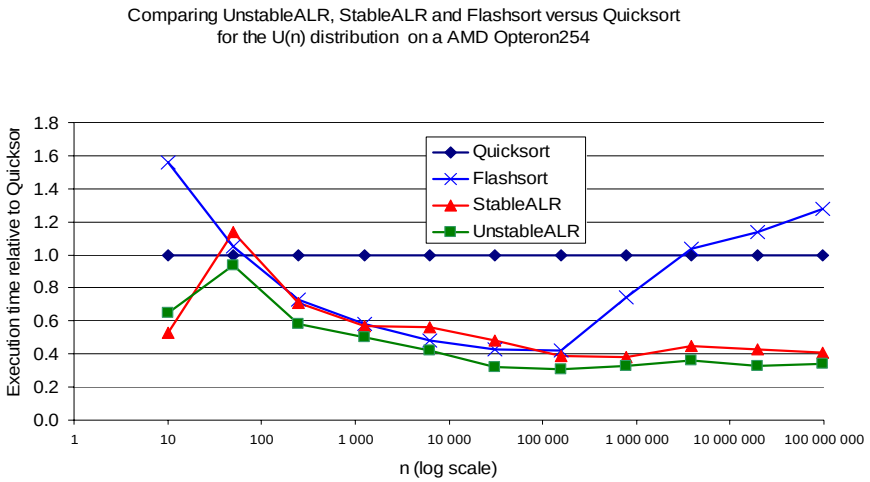

Figure 4. *Comparing four sorting algorithms, n=10,50,..., 97656250 with a uniform 0..n-1 distribution on a 2.8 GHz Opteron254 CPU.*

6.2 Testing distributions with longer equal valued sequences

To test how well StableALR will compare with Quicksort when the equal valued sequences becomes longer, it was tested with the following distributions:

1. U(n/3) – Uniformly distributed in the range 0..n/3-1 (each equal sequence is on the average 3 elements)
2. U(n/10) - Uniformly distributed in the range 0..n/3-1 (each equal sequence is on the average 10 elements)
3. fixed i%3 – the numbers 0,1,2 randomly distributed in the array of length n (each equal sequence is on the average n/3 elements)
4. fixed i%29 – the numbers 0,1,2,...,28 randomly distributed in the array of length n (each equal sequence is on the average n/29 elements)
5. fixed i%171 – the numbers 0,1,2,...., 170 randomly distributed in the array of length n (each equal sequence is on the average n/171 elements)

The last three distributions are not as uncommon as one may think. Say you want to sort people by sex (3 values: male/female/unknown), district and occupation. One might get these distributions where there are only a fixed number of values for the sorting key regardless of the number of elements sorted. We note that the equal valued sequences might get very long, in the fixed i%3-distribution more than 30 million elements each. To cope with these fixed distributions, a rather standard Right Radix implementation that does a back copy from the auxiliary buffer to 'a' for each digit was used as stable subalgorithm.

We see that StableALR is faster than Quicksort for the U(n/3) , U(n/10) and fixed i%171 distribution,, on the same level as for the fixed i%29 distribution and a little more than twice as slow for the fixed i%3 distribution. Note that we are here comparing relative execution times. Both StableALR and Quicksort are much faster when sorting the fixed distributions, but the speed increase for Quicksort is even greater. The actual execution times in milliseconds for sorting 97 million numbers for the five distributions are given in table 1.

	U(n/3)	U(n/10)	fixed i%171	fixed i%29	fixed i%3
Quicksort	20 781	18 984	6 031	3 969	1 891
StableALR	8 936	8 922	4 066	4 048	4 066

Table 1. *Absolute execution time in milliseconds for sorting approx. 97 million numbers for Quicksort and StableALR on Opteron 254 for five distributions of the elements. Note the speedup for the fixed distributions, especially for Quicksort.*

Actually, both Qucksort and StableALR achieves in practice linear performance O(n) when sorting the fixed distributions. This is expected when using StableALR, because it is O(n*logM) where M is the largest value in the array. Since M is fixed, this is O(n). We only observe that Quicksort is O(n) in practice for the fixed distributions even though it is $O(n^2)$ worst case.

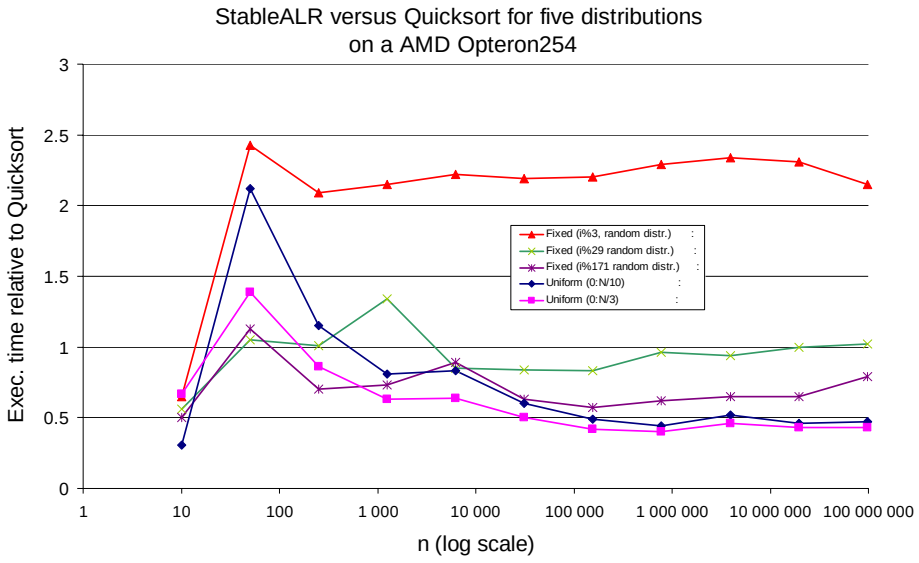

Figure 5. *The execution times of StableALR relative to Quicksort (=1) for five different distributions, n= 10,50,250, .., 97656250. on Opteron254.*

6.3 Testing StableALR vs. Quicksort for three CPUs

It is well known that the relative performance of algorithms vary with the CPUs or rather the CPU with its memory system which now has at least 2 levels of caches [Maus00]. Some algorithms are more cache-friendly than others. A cache miss to main memory may be more than 100 slower than accessing the same element in the L1 cache [Maus00]. After sorting on the first digit, StableALR is much more cache friendly than Quicksort because the data set is divided by StableALR into many, each much shorter

sequences that fits better into the L1 and L2 caches than the halving strategy of Quicksort. It is also explained that the reason for Radix sorting being so much faster than Quicksort on a cached CPU, is that it uses far fewer moves to sort the same array.

Below in figure 6 we test StableALR versus Quicksort on three different machines: a 0.6 GHz Pentium M, a 2.8 GHz Pentium 4(Xeon), and a 2.8 GHz Opteron254 CPU. In table 2 we present the absolute execution times for Quicksort on these CPUs. The main performance difference between these CPUs, it may be reasonable to attribute to the fact that the L1 data cache is only 8 kB in a Pentium 4, while it is 32kB on a Pentium M and 64kB on an Opteron. The effect of caching on sorting has also been investigated in [LaMarca & Ladner, Maus00].

The reason that figures 4 and 5 are presented with figures for the AMD Opteron254 and not the Pentium 4, is that Intel has announced that it will not continue its Pentium 4 line of CPUs, but base its new processors on the Pentium M design, mainly because of the heat problems with the Pentium 4. The choice between the 0.6 GHz Pentium M (in a laptop) and the 2.8 GHz AMD Opteron, it was clear that the results from the Opteron are more industrial relevant. The relative performance differences between the Pentium M and the Opteron are also small.

Anyway, the StableALR is anything from 20-60% faster than Quicksort when $n \geq 250$.

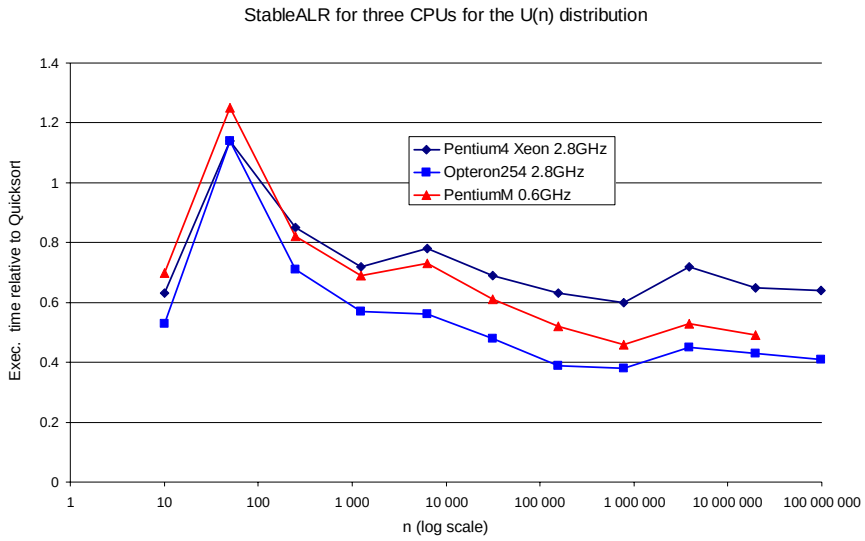

Figure 6. *The relative performance of Stable ALR versus Quicksort on three different CPUs Pentium M, Pentium 4(Xeon) and Opteron 254 for the U(N) distribution of n integers, n=10, 50,..., 19531250, 97656250. (only up to 19531250 for the Pentium M).*

	50	1 250	31 250	781 250	19 531 250	97 656 250
Pentium4, 2.8GHz	0.004	0.143	4.917	157.720	4 812.600	26 250.000
Opteron254 2.8GHz	0.003	0.108	3.862	127.320	3 971.800	21 859.000
PentiumM 0.6GHz	0.012	0.506	17.681	594.300	19 172.500	

Table 2. *Absolute execution times for every second data point in figure 6 (n=50,1250,..) in milliseconds for Quicksort on sorting the U(n) distribution for 3 different CPUs*

7. Analysis of Unstable ALR and StableALR

The execution time for both the UnstableALR and the StableALR are O(n*logM), where M is the maximum element in input. The UnstableALR needs only constant extra space, two arrays of length of at most 2^r (actually, there are two implementations of ALR, one that needs two such supporting arrays and one that only needs one supporting pointer array of length of at most 2^r. The latter is somewhat faster, especsially for shorter arrays, but clearly more difficult to implement and explain. Thus all results in this paper is from the two-supporting pointer arrays implementation). The StableALR needs in addition an auxiliary array of length K when we use ordinary Radix as a subalgorithm, where K is the longest equal valued subsection of 'a' – i.e. of O(n) as a worst case with the fixed distributions.

8. Sorting negative numbers, other data types and on a multi core CPU

It is well known that integer sorting algorithms also are also well suited for other data types such as floating point numbers and strings. Since StableALR is a bit manipulating algorithm, I will give some comments on some of these issues.

Sorting a mixture of positive and negative number can be done by first partition the array in the negative numbers negated and positive numbers in a one-bit sort. These two sections can then be sorted separately as positive numbers and as a final stage, the negative section is again negated (and reversed).

Sorting floating point numbers can be done as integers by the IEEE 754 standard. In a strongly typed language as Java this is difficult because the logical bit operations are only defined for integer data types, but a type cast from double to long without conversion might be done through the C-interface in Java.

StableALR should be well suited for sorting strings since it sorts from left to right if it uses only Insertion sort as a subalgorithm for stable sorting. For strings it should be reasonable not to expect too many equal strings.

StableALR is also well suited for multi core CPUs. Here I suggest a first pass with a first sorting digit that splits the array in at least as many subsections as there are processing cores. Then one starts as many threads as there are processing cores, and in a loop hand out these subsections until they all have been sorted on the remaining digits with one instance of StableALR for each thread. This we do until all subsections from the first digit has been processed. Since starting treads are considered expensive, this should only be done when the length of the input set is larger than some constant (say 100 000). This description is only a sketch, and more research remains to be done in this area.

9. Conclusions

A stable sorting algorithm, StableALR, has been defined as a modification to an in place most significant radix sorting algorithm. In most cases it has been demonstrated to be from 20-60% faster than Quicksort, which is not stable. In worst case this stable sorting ability comes at a price of extra space of O(n) needed for the stable sorting phase, but as argued, this worst case (as the worst case execution time for Quicksort)

will seldom or never happen in practice. Thus, StableALR should preferably be used as a general purpose sorting algorithm, possibly replacing Quicksort.

10. Acknowledgement

I would like to thank Stein Krogdahl for comments to earlier drafts of this paper.

11. References

[Andersson & Nilsson] Arne Andersson & Stefan Nilsson . *Implementing Radixsort.* Journal of Experimental Algorithmics (JEA), Volume3, 1998.

[Aho] V.A. Aho , J.E. Hopcroft., J.D. Ullman, *The Design and Analysis of Computer Algorithms*, second ed, .Reading, MA:Addison-Wesley, 1974

[Dobosiewicz] - Wlodzimierz Dobosiewicz: *Sorting by Distributive Partition,* Information Processing Letters, vol. 7 no. 1, Jan 1978.

[Franceschini & Geffert] – Gianni Franceschini & Viliam Geffert. *An In-Place Sorting with O(n log n) Comparisons and O(n) Moves.* Journal of the ACM, Vol 52, No.4, July 2005.

[Hildebrandt and Isbitz] - P. Hildebrandt and H. Isbitz *Radix exchange - an internal sorting method for digital computers.* J.ACM 6 (1959), p. 156-163

[Hoare] - C.A.R Hoare : *Quicksort*, *Computer Journal* vol 5(1962), 10-15

[Knuth] - Donald E. Knuth: *The art of computer programming - vol.3 Sorting and Sear*ching, Addison-Wesley, Reading Mass. 1973

[LaMarca & Ladner] - Anthony LaMarcha & Richard E. Ladner: *The influence of Caches on the Performance of Sorting,* Journal of Algorithms Vol. 31, 1999, 66-104.

[van Leeuwen] - Jan van Leeuwen (ed.), *Handbook of Theoretical Coputer Science - Vol A, Algorithms and Complexity,* Elsivier, Amsterdam, 1992

[Maus00] – Arne Maus, *Sorting by generating the sorting permutation, and the effect of caching on sorting*, NIK'2000, Norwegian Informatics Conf. Bodø, 2000, www.nik.no

[Maus02] – Arne Maus, ARL, *a faster in-place, cache friendly sorting algorithm,* NIK'2002, Norwegian Informatics Conf. Kongsberg, 2002. Tapir, ISBN 82-91116-45-8. www.nik.no

[McIiroy et al. 93] – McIlroy, P.M., Bostic, K. and McIlroy M.D. Engineering Radix Sort. Computing Systems6, 1(1993) p. 5-27.

[Neubert] – Karl-Dietrich Neubert, *Flashsort,* in Dr. Dobbs Journal, Feb. 1998

[Nilsson] - Stefan Nilsson, *The fastest sorting algorithm,* in Dr. Dobbs Journal, pp. 38-45, Vol. 311, April 2000

[Sins & Zobel] - Ranjan Sinha and Justin Zobe*l Cache-Conscious Sorting of Large Sets of Strings with Dynamic Tries.* ACM Journal of Experimental Algorithms, Vol 9, Art. 1.5, 2004, p. 1-31

[Sedgewick] – Robert Sedgewick, *Algorithms in Java*, Third ed. Parts 1-4, Addison Wesley, 2003

[Weiss] - Mark Allen Weiss: *Datastructures & Algorithm analysis in Java,* Addison Wesley, Reading Mass., 1999