Sorting by generating the sorting permutation, and the effect of caching on sorting.

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

Abstract
This paper presents a fast way to generate the permutation p that defines the sorting order of a set of integer keys in an integer array 'a'- that is: a[p[i]] is the i'th sorted element in ascending order. The motivation for using Psort is given along with its implementation in Java. This distribution based sorting algorithm, Psort, is compared to two comparison based algorithms, Heapsort and Quicksort, and two other distribution based algorithms, Bucket sort and Radix sort. The relative performance of the distribution sorting algorithms for arrays larger than a few thousand elements are assumed to be caused by how well they handle caching (both level 1 and level 2). The effect of caching is investigated, and based on these results, more cache friendly versions of Psort and Radix sort are presented and compared.

Introduction

Sorting is maybe the single most important algorithm performed by computers, and certainly one of the most investigated topics in algorithmic 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 [Dahl and Belsnes, Weiss, Goodrich and Tamassia] or in standard reference works [Knuth, van Leeuwen]. New sorting algorithms are still being developed, like Flashsort [Neubert] and 'The fastest sorting algorithm" [Nilsson]. 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].

Sorting algorithms can be divided into comparison and distribution based algorithms. Comparison based methods sort by 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 [Weiss] that the time complexity of a comparison based algorithm is at best O(n logn).Well known comparison based algorithms are Heapsort and Quicksort. 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) time. Well known distribution based algorithms are Radix sort in its various implementations and Bucket sort.

Radix and Bucket sort

Since the distinction between the various Radix sort and Bucket sort implementations is important in this paper, we will define this difference as how they deal with the problem of elements in 'a' having the same value. Radix sort first count how many there are of each value, and will then afterwards be able to do a direct placement of the elements

based on this information. Bucket sort will, on the other hand, do a direct placement of each element in its 'bucket' based directly on its value. But since there might be more than one element with this value, a list-structure is usually employed to connect these equal valued elements together. A last phase is then need in Bucket sort to connect these lists to generate the sorted sequence.

The sorting permutation and Psort

This paper introduces algorithms for generating the sorting permutation 'p' of 'a' - defined as: a[p[i]] is the i'th sorted element of 'a' in ascending order. The original array, 'a', is not rearranged. The reason for this is that in real life problems, we are not only interested in sorting a single array, but rearranging a number of information according to some key – say writing a list of students names and addresses etc. sorted by their student number, or writing information on bank accounts (names, addresses, payments and balance) sorted by account number. If these accompanying information are kept in arrays b, c,.., finding the sorting permutation p, will make it trivial to write out this as a[p[i]], b[p[i]], c[p[i]],…. This is the motivation for finding the sorting permutation.

The alternatives to use the sorting permutation, are either to sort by rearranging the elements in the accompanying arrays b,c,.. together with the elements in 'a' in an ordinary sort, which takes a heavy time penalty, or alternatively store the information belonging to logical units in objects and then sort these objects based on the key in each object. Sorting objects can, under the assumption that the objects are generated outside of the algorithm, be made just as fast as sorting integer arrays, as demonstrated by Bucket sort in Fig. 1. If the objects to be sorted are also generated by the algorithm (which is assumed not to be a likely situation), then object based Bucket sort becomes 3 to 5 times as slow as Quicksort (not shown here). Anyway, object sorting takes a rather high space overhead (pointers for lists and system information for each object). Generating the sorting permutation can thus be a good alternative in many practical situations.

The generation of the sorting permutation is not, however, a new idea. Even though I am not able to find any published paper on it, nor finding a published algorithm, at least two public software libraries have these functions [HPF, Starlink/PDA]. The implementation of these library functions are not (easily) available. Code-fragment 1 is my first implementation of Psort. We see that it borrows from a full one-pass Radix sort. After having found the maximum value, Psort counts how many there are of each value and accumulates these counts into indexes (logical pointers). The final stage is, however, new. Instead of rearranging 'a', it generates p from a (third scan) of the array 'a' and these logical pointers.

Comparison with other sorting algorithms

We observe by inspecting the code, that Psort obviously has a time complexity of O(n), and uses two extra arrays of length O(n) - assuming that the max value in 'a' is of order n. To be exact, Psort does 6n reads and 6n writes in total (also counting the zero-fill).We now compare Psort with Heap-sort, Quick-sort, Bucket sort and ordinary least-significant Radix-sort with a 'digit'size of 10 bits. The implementation of Heapsort is taken from [Dahl and Belsnes], Quicksort is taken from [Hoare], but optimized by calling insertion

sort if the segment to sort is shorter than 10 elements. Bucket sort is my own implementation with objects and one list (stack) for every possible value, and a final step of pop'ing all these stacks , starting with the last , and pushing them onto a result-list (stack).

```
int [] psort1 ( int [] a )
 // assumes max unknown
{ int [] p = new int [n];
  int [] count ;
  int localMax = 0;
  int accumVal = 0, j, n= a.length;

     // find max
     for (int i = 0 ; i < n ; i++)
     if( localMax < a[i]) localMax = a[i];

     localMax++;                // upper limit of array
     count = new int[localMax]; // zero-filled by def.

     for (int i = 0; i < n; i++) count[a[i]]++;

     // Add up in 'count' - accumulated values
     for (int i = 0; i < localMax; i++)
     { j = count[i];
       count[i] = accumVal;
       accumVal += j;
     }

     // make p[]
     for (int i = 0; i < n; i++)
      p[count[a[i]]++] = i;

    // now a[p[i]] is the sorted sequence
     return p;

  }/* end psort1 */
```

Code fragment 1 *– Psort1, first version (not optimized for cache usage)*
Finds the sorting permutation

The tests were run on a Dell Precision Workstation 410 Pentium III 450MHz with 32 KB (16-KB data cache; 16-KB instruction cache) level 1 cache and a second-level cache of 512-KB pipelined burst; 4-way set-associative, write-back ECC SRAM. The results of sorting arrays of lengths varying from 50 to 1 million elements is presented in Figure 1. Note the non linear scale on the x-axis and that results are normalized with Quicksort = 1 for each length of the sorted array.

```
void radix(int [] a)
{ // initial operations
    int max = 0, i, n = a.length;

    for (i = 0 ; i < n; i++)
         if (a[i] > max) max = a[i];
    b = new int [n];
    c = radixSort( a,b, 0, max);

    // copy array if odd number of calls
    if ( a != c)
       for (i = 0; i < n; i++)  a[i] = c[i];
}

int [] radixSort ( int [] fra, int [] til , int bit, int max )
{   int numBit = 10, rMax = 1024-1;
    int  acumVal = 0, j, n = fra.length;
    int [] count = new int [rMax+1];

    for (int i = 0; i < n; i++)
      count[((fra[i]>> bit) & rMax)]++;

    // Add up in 'count' - accumulated values
    for (int i = 0; i <= rMax; i++)
    { j = count[i];
      count[i] = acumVal;
      acumVal += j;
    }

    // move numbers between arrays
    for (int i = 0; i < n; i++)
        til[count[((fra[i]>>bit) & rMax)]++] = fra[i];

     if ( (1 << (bit + numBit)) < max )
         return radixSort ( til, fra, bit + numBit, max);
    else return til;

 }/* end radixSort */
```

Code fragment 2 *– Radix sort, sorting by 10 bit least-significant-first Radix.*

We see that the three O(n) time complexity distribution sorting algorithms do better than Quicksort and Heapsort, for number to be sorted < 100 000 - up to 3 times as fast as Quicksort, and approx.- 7 times faster than Heapsort (except that Radix performs badly when n < 100 because it uses a fixed digit size of 10 bits, which is an overkill for such small lengths of a). In the end Radix emerges as the winner with Psort in second place and that Bucket sort degenerates for large n because of the space overhead incurred by objects.

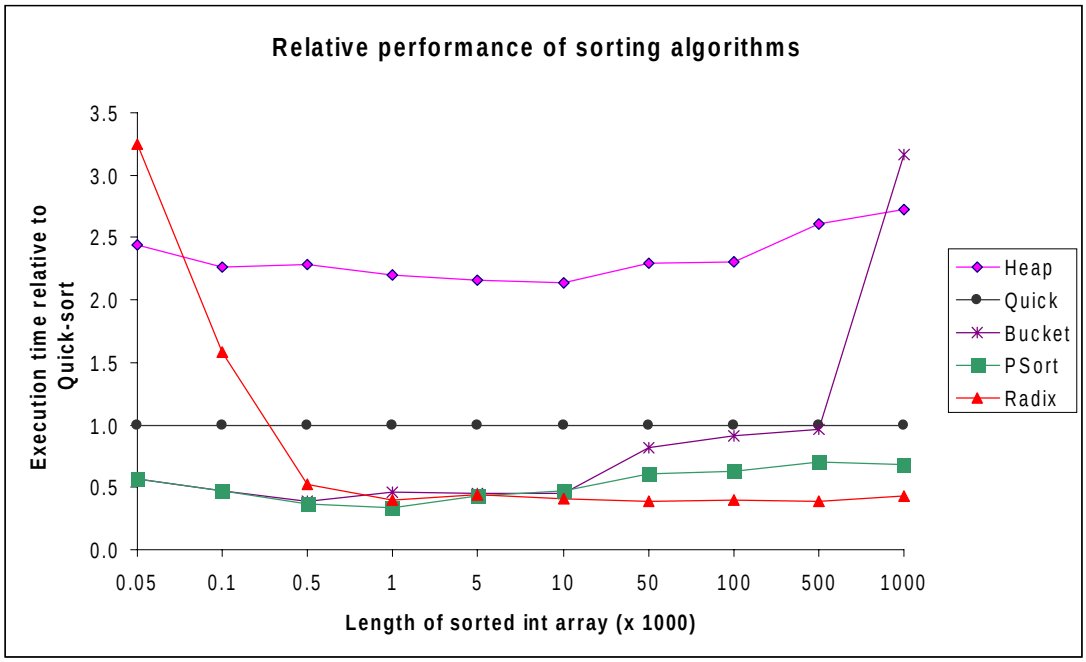

Figure 1. *Comparison of Psort versus Quicksort, Heapsort, Radix sort and Bucket sort. The results are normalized for each length of the array with Quicksort = 1.*

If we inspect the code for Radix in Code fragment 2, we see that for length of array a > 1024, we use 4 reads and 3 writes of O(n) in each method invocation plus one initial O(n) read and one write, and with two invocations, a total of 9 reads and 7 writes – compared with 6 reads and 5 writes for Psort. Why is then Radix a faster algorithm? We will explain this by the effect of caching – both by the very fast 16 kByte level 1 cache (able to hold a little more than 4000 integers) and by the 512 kByte level 2 cache that is able to hold a little more than 100 000 integers. In the next section we investigate how large the effect of cache misses might be.

The effect of caching

It is well known that the speed difference between the CPU, that soon operates with a cycle time of less than 1 nanosecond, and the main memory, that has a cycle time of approx. 20 nanoseconds, is increasing. An access to main memory from CPU goes through the system bus (typically clocked at 100MHz) and takes more than 50-100 nanoseconds, or approx. 3 memory cycles to complete. To ease this speed discrepancy, data are read and written a cache line at a time. The size of a cache line is typically 32 bytes (8 integers). Two levels of memories, level 1 cache integrated on the CPU, and level 2 cache between the CPU and the system bus, are employed to ease this speed difference. The effect of caching in general is extensively investigated, usually by estimated by simulations or pure theoretical considerations.

Also the effect of caching on sorting has previously been investigated in [LaMarca & Ladner] where Heapsort, Mergesort and Quicksort are compared to Radix sort. They work along similar lines as this paper, and report on improved performance on cache-friendly (in their words: memory tuned) versions of Heapsort, Mergesort and Quicksort. They report negative findings on Radix sort. This paper tries cache-friendly versions of

Radix sort and Psort. The main reason that their findings are not the same as here (here: Radix is best versus LaMarca & Ladner: Radix is worst), is that they have random 64 bit values in 'a', while I have a uniform random distribution of the values {0..length of 'a'}. Comparison based methods are heavily favored by such sparse distributions, while distribution based sorting are favored by a dense set of keys.

This paper also takes an empirical approach. The number of methods employed at the different hardware levels, are many. They are not very well documented – if at all, and their combined effect is difficult to assess. Also, new hardware is introduced every few months. We will therefore empirically try to find the effect of caching by repeatedly executing a typical combined array access statement found in distribution based sorting algorithms. This statement is of the form a[b[…b[i] ..]] = 'some simple expression' That is, the array 'a' is indexed by a number of nested references to an array b. Each reference to 'b' is a read operation, and the reference to 'a' is a write operation. This operation is performed for the lengths of a and b = 10, 50, 100,…,5 million , and iterated enough times to get reliable time measurements. The number of reads (number of nested references to b) is varied = 1,2,4,8 or 16.

```
for(int i= 0; i < n; i++)
 a[b[b[i]]]]  = i ;
```

Code fragment 3 *– Cache miss test code example (1 Write, 2 Reads).*

The important difference in the tests is the contents of b. The base case is that b[i] = i for all values of i. A nested reference to b[i] will then be cached in the level 1 and level 2 cache, and will reference the same cached value repeatedly. The other content of 'b' tested, is a random constructed permutation of the numbers 0 to n-1. A nested reference to such an array will almost certainly generate cache misses for (almost) all references when n is large enough. The effect investigated is for which 'n' the effect of cache misses are felt, and how large it is.

The results are given in figure 2 and 3. In the figures, the results with b[i] = i, is denoted sequential, and the randomly filled b is denoted Rdom or random, and the number of reads is the number of nesting of the b references. The number of writes is 1 in all cases.

In figure 2 we see that the execution time of the random case increases far faster than sequential access as expected, but since we have a linear scale on the x-axis, it is difficult to see when the two curves diverge. In figure 3, with a log scaled x-axis, we first see that up to n = 1000, there are no noticeable differences between sequential and random access of arrays. This I explain by the fact that both arrays 'a' and 'b' are very soon fully cached in both caches after a small number of references. When n = 5000, we obviously don't have room for all of 'a' and 'b' in the level 1 cache, and the random access pattern generates many cache misses. In the range n = 5000 to 100000 , with typical expressions found in sorting algorithms like Psort, with one write and two reads, we will experience an execution time 2-3 times larger with random access of arrays than with a sequential reference to the same arrays.

For arrays longer than 100000, we also generate lots of level 2 cache misses with both patterns, every 8'th time with the sequental pattern and for every access with the random access pattern, because there is not enough memory for the arrays in the level 2 cache. We see that the typical 1 write, 2 reads expression executes approx. 5 times faster with sequential than random access. When taken to extreme, with 1 write and 16 reads per operation, a more than 20 times slower random execution is achieved with n = 5 million. The lesson learned is that it is not necessarily the algorithm with the fewest number of performed statements that is the fastest - if we keep references to data localized or sequential, we could still make a faster algorithm even if it performs 2 to 3 times as many statements.

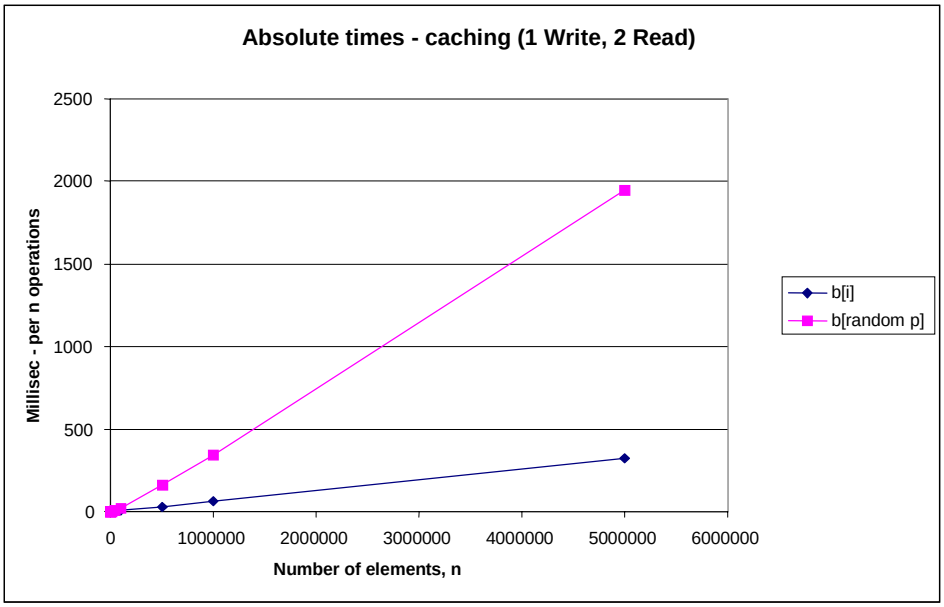

Figure 2*. The absolute running times of caching on a modern PC, comparing the same access to two arrays a and b for different lengths of these arrays – the Sequential does generate far fewer cache misses, than random access that (almost) certainly generates a cache miss for every reference(1 write and 2 reads per expression) for large n.*

A new Psort and Radix algorithm

Inspired by the fact that a (more) sequential access pattern generates far fewer caches misses and that smaller arrays stays longer in the caches regardless of their access pattern, I decided to redesign the Psort and Radix algorithms used in this paper to be more cache friendly. Being inspired by the better performance of Radix sort, I propose a two-pass Psort algorithm, Psort2, as given in Code fragment 4.

The redesigned Radix sort, called Block sort, has a code that is too long for this paper - it would cover almost 3 pages. It can best be described as a combination of a pass 1 that is a kind of buffered bucket sort on the most significant bits. For each bucket, a dynamic linked list of integer arrays (with a maximum size of 1024 elements) is kept, and the elements a[i] are placed sequentially into the linked buffer-list corresponding to the value of the most its significant bits. The number of elements in each list is also counted as the elements are placed in this list of buffers.

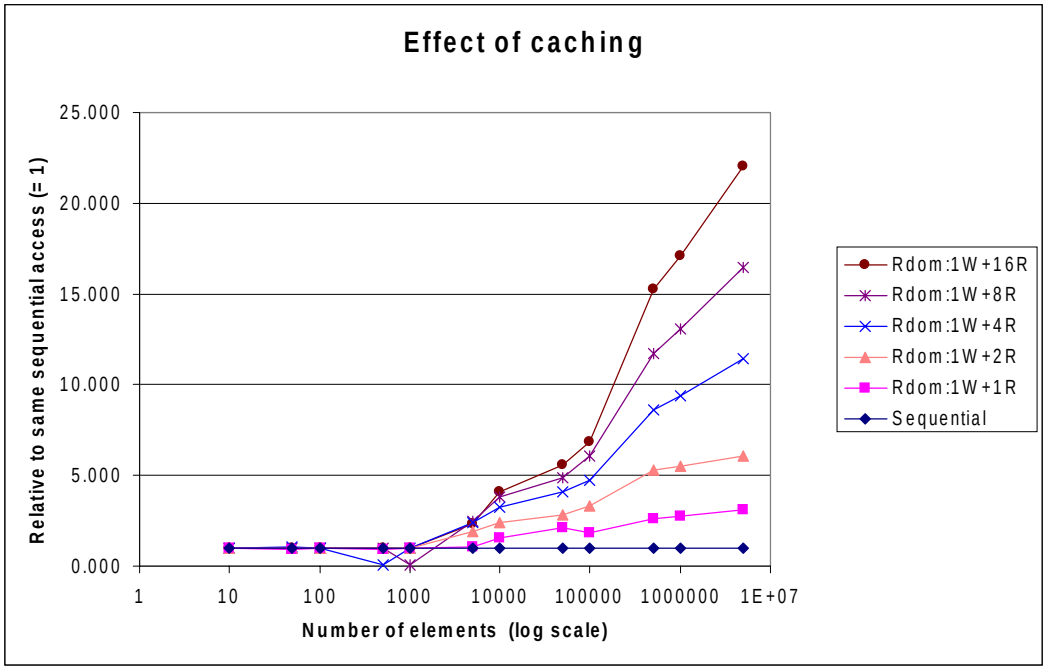

Figure 3. *The relative effect of caching on a 450 MHz PC, comparing the same accesses to two arrays a and b for different lengths of these arrays – the Sequential does generate few cache misses, while the Rdom (random) filled almost always generates a cache miss for every reference made for large n. The results are normalized with the sequential = 1 for each length of n.*

```
int [] psort2 ( int [] a)
{   int n = a.length, maxNumBit1 = 10;
    int [] p = new int [n];
    int [] til = new int [n];
    int [] ant ;
    int localMax = 0;
    int numbit=0, mask=0,mask1=0,mask2=1;
    int numbit1, numbit2, num1,num2, num;
    int accumVal = 0, j;

       // find max
       for (int i = 0 ; i < n ; i++)
       if( localMax < a[i]) localMax = a[i];

       if (localMax < 4) localMax =4;

       // find bitnumb & masks;
       while ((1<<numbit) < localMax) numbit++;
       mask = (1<<numbit) -1;
       numbit1 = numbit /2;
       numbit2 = numbit - numbit1;
```

```
 while (numbit1 > 1 && numbit2 < maxNumBit1)
{ numbit1--; numbit2++;}
 mask1 = (1<<numbit1) -1;
 mask2 = mask - mask1;
 num1 = 1<<(numbit1);
 num2 = 1 <<(numbit2);
 if (num1 < num2 ) num = num2; else num = num1;
 localMax++; // upper limit of array

// ---- pass 1--------------
 ant = new int[num1];

for (int i = 0; i < n; i++)
  ant[a[i] & mask1 ]++;

// Add up in 'ant' - accumulated values
for (int i = 0; i < num1; i++)
{ j = ant[i];
  ant[i] = accumVal;
  accumVal += j;
}
// move numbers - pass 1
for (int i = 0; i < n; i++)
    til[ant[a[i] & mask1]++] = i;

// ---- pass 2 --------------
 ant = new int[num2];
 accumVal =0;

 for (int i = 0; i < n; i++)
    ant[(a[til[i]] & mask2)>> numbit1]++;

 // Add up in 'ant' - accumulated values
 for (int i = 0; i < num2; i++)
 { j = ant[i];
   ant[i] = accumVal;
   accumVal += j;
 }
 // make p[]
 for (int i = 0; i < n; i++)
  p[ant[(a[til[i]] & mask2)>> numbit1]++] = til[i];

// now a[p[i]] is the sorted sequence
 return p;

}/* end psort2 */
```

Code fragment 4. *The 2 pass Psort algorithm.*

The second pass then takes over which for each such list, first counts the number of elements of each possible value in the least significant bits. After this counting, a direct placement back into 'a' can be made, starting with the first list having all zeros in its most significant bits.

In the design of both Psort2 and Block sort, care was taken so that all access to long arrays of length O(n) was sequential, and that all supporting array that might be accessed in a random fashion were short - i.e • 1024 elements. Such small arrays would then entirely be cached in the level 1 cache. In short, all data structures would be cache friendly.

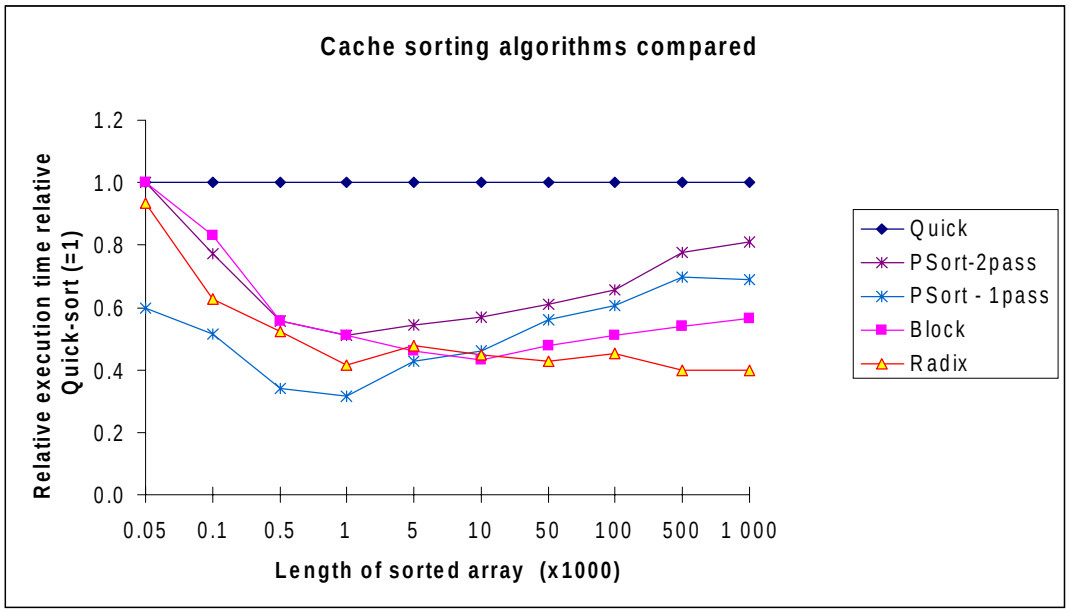

Figure 4. *Comparison of the reworked cache-friendly algorithms Block sort and P sort 2 pass, compared with Quicksort, Psort 1 pass and the least-significant-digit-first Radix sort.*

The results are plotted in Figure 4, normalized with Quicksort = 1. (Radix sort is a little reworked from the code presented in Code fragment 2. It now adjusts down the number of bits it sorts on if the largest element to be sorted has fewer than 10 bits. Radix sort therefore does not in Figure 4, as in Figure 1, perform badly for short arrays.)

We see that the two new algorithms perform well compared with Quicksort, but are both somewhat slower than the algorithm they are supposed to be an improvement upon. Psort 2 pass is slower than Psort 1 for all lengths of the sorted array and Block sort is slower than Radix except for some lengths in the range 5000-10000, where it is only marginally faster. How do we explain this?

First, the assumption that I based this rework on, is not all correct. As previously observed, the Radix algorithm in Code fragment 2 does 9 reads and 7 writes (when sorting arrays of length between 2^{10} and 2^{20}). Not all of these reads and writes are cache unfriendly. A closer inspection reveals that only 2 of the writes and none of the reads are random in the O(n) arrays - the rest are cache friendly accesses. Even though these 2

writes may take say 5-20 times as long time as a cache friendly write, the possible improvement factor for an all-cache-friendly-algorithm is then certainly not a factor 3 to 5, but more a factor of 2. This factor can be observed by comparing the Radix sort execution time for n=5000 (1.469 msec.) where we can assume that almost all read and write operations are cache friendly, with the execution time for n = 1 000000 (432.7 msec.). This gives a factor of 294 longer execution time. Since Radix in both cases uses two passes, the added execution time that can't be contributed to the extra length (a factor of 200), that is a factor of 1.47, that can only be caused by cache misses.

A similar analysis of the possible improvement factor for Psort with an execution time of 1.312 msec. for n= 5000, and 746.9 msec. for n= 1 000000, gives a possible improvement factor of 2.85 for Psort that can not be contributed to the increased length.
In both cases, Psort as presented in Code fragment 1, uses only one pass, while Psort2 uses 2 passes.

We could therefore hope that at least there would be possible to improve on Psort, but as we see from Code fragment 2, the Psort2 has much longer and more complex code with more nested expressions. The final and most complex expression in Psort1:

```
p[count[a[i]]++] = i;
```

has its counterpart in Psort2:

```
p[ant[(a[til[i]] & mask2)>> numbit1]++] = til[i];
```

We see a longer, more complex expression with 4 reads and 2 writes compared with 2 reads and 2 writes. Obviously, some of the possible gains are lost in the code. The length of the code itself is also an issue, since code is cached in the 16kByte instruction cache.

If we do a similar analysis of Radix, the Block sorting algorithm is much longer, but each expression is (almost) as simple. Only the length of the code and the length of each loop may be blamed for more code cache misses. The cache-miss factor for Block sort is 614.0/(1.416*200) = 2.16, which is worse than for Radix sort.

The Psort2 algorithm, with an execution time of 1.672 msec. for n = 5000 and 876.6 msec. for n = 1000 000 has itself a cache-miss factor of 2.62, which is almost as bad as Psort1. The explanation for the fact that the cache-friendly Psort2 algorithms have as bad a cache factor as Psort1 that it is supposed to replace, is, apart from the increased instruction cache misses, that Psort1 only scans the O(n) arrays twice, but Psort2 basically doubles the number of reads and writes. And when these large arrays are larger than the level 2 cache, every 8’th access for Psort2 will also generate a cache miss even with sequential reading of these arrays. Cache-friendly code is certainly not the same as cache-miss-free code.

Conclusion

This paper has presented two themes. First, a new algorithm Psort, for generating the sorting permutation has been presented, and it has been demonstrated to be almost as fast as any sorting algorithm. It has also been pointed out why this is a practical sorting algorithm when more than one array has to be brought into the same sorting order.

Secondly, I have empirically investigated the effect of caching on sorting algorithms, and demonstrated that cache misses might account for a slowdown when sorting large arrays by a factor of 1.5 to 3 depending on which sorting algorithm is used. Then two

new algoritms, Block sort and Psort2, has been presented that was designed to be more cache friendly. They did not on the computer used for testing, outperform their simpler, but more cache unfriendly counterparts, Radix and Psort1.

This last conclusion needs however, a final remark. Any standard algorithm is a mixture of cache friendly and cache unfriendly code, and the effect of removing all cache unfriendly code in the way it was done here, did not pay off today. That does not rule out these new algorithms once and for all. New machines with a far worse ratio between the CPU cycle time (with more than 1000 MHz) and main memory, will soon be brought to the marketplace. Since new memory architectures, like RAMBus, up till now has not demonstrated better performance than more standard memory architectures, it seems reasonable to assume that the penalty for doing a cache miss soon will double from the machine used in this paper. It will therefore be interesting to test Psort2 and Block sort on new machines in the years to come.

References

[Dahl and Belsnes] - Ole-Johan Dahl and Dag Belsnes : „*Algoritmer og datastrukturer*“ Studentlitteratur, Lund, 1973

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

[Goodrich and Tamassia] - Michael T. Goodrich and Roberto Tamassia: *„ Datastructures and Algorithms in Java“*, John Wiley & Sons, New York, 1998

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

[HPF] - the GRADE_DOWN routine at: http://hazmat.fms.indiana.edu/manpages/g/grade_down.3hpf.html

[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

[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

[Starlink/PDA] see software | programming | mathematical at : http://star-www.rl.ac.uk/

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