A Model for the Effect of Caching on Algorithmic Efficiency in Radix based Sorting

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

Stein Gjessing
Department of Informatics
University of Oslo
Norway
steing@ifi.uio.no

***Abstract*— This paper demonstrates that the algorithmic performance of end user programs may be greatly affected by the two or three level caching scheme of the processor, and we introduce a general but simple model that estimates this effect with good accuracy. Using 4 different processors and two simple tests, we demonstrate that a number of random reads and writes in an array can have a penalty factor of 40-70 compared to the exact same number of sequential array accesses. These effects from cache misses first occur when a randomly accessed array is larger than the L1 cache and even more so when the array does not fit in the much larger L2 cache. A similar but much smaller effect between sequential reads and writes in large arrays compared with smaller arrays is also reported. In the second part of this paper three versions of the well known Radix algorithm is used to demonstrate that the 'artificial' results from the first part definitely occur in practice. These versions use one, two and three digits, and hence one, two or three full passes to sort an array. For small arrays the two and three digit Radix algorithms of course are much slower than the one digit version, but for larger arrays the former outperform the latter. For comparison, the radix sorting execution times are compared to Quicksort and Flashsort, a different one-digit sorting algorithm, both of which are outperformed by the two and three digit Radix algorithms. Finally, a simple model based on the penalty of cache misses is introduced, and we use this model to explain the demonstrated differences in algorithmic efficiency.**

Keywords: caches, cache models, cache friendly algorithms, radix, sorting

I. INTRODUCTION

When users execute their programs, they are given the illusion that its data and instructions reside in a 'large' linear memory where there is a unit cost for reading or writing to any of these locations. Nothing could now be further from the truth. User programs are allocated in virtual memory space, initially on disk and brought into main memory when referenced and brought further up to the (L3), L2, and L1 cache memories and finally to the CPU-registers before operations actually take place. The purpose of this paper is not to go into any great detail describing this memory hierarchy, but to investigate how the huge speed difference between the processor and the main memory affects real user programs. What are the effects of caching on sequential and random reads and writes in arrays? We will use radix sorting as a test case and demonstrate how one version of this algorithm executes almost 3 times as many instructions as another version but at the same time runs 4-5 times as fast – a 10 fold speed increase per performed instruction! The only difference between the two programs is that the fastest version does random access among far fewer array elements compared with the slow version that does its random reads and writes all over in a single, much larger array. We then introduce a simple model using sequential and random access in arrays for predicting algorithmic performance The essence of this paper is that while sequential access in arrays is almost for free; our programs must as far as possible try to localize random access to data in order to be fast.

II. RELATED WORK

The effects of caches on comparison based sorting as well as Least Significant Digit Radix (LSR) has been investigated by [1], distribution based sorting with Least Significant Radix (LSR) by [2] and [3], and the Most Significant Radix (MSR) sorting by [4] and [5]. While [1] finds Radix slower than Quicksort [6], the other papers find Radix algorithms in both variants faster and also faster than Flashsort [7]. The reason that [1] finds Radix slower is that it uses uniformly drawn 64 bit integers $U(2^{64})$, while the other papers use a U(n) distribution, where n is the length of the sorted array. The references [3] and [5] model a one level caching scheme and in [3] makes a detailed analysis of TLB-misses and introduces algorithms that are either cache optimized or TLB-miss optimized – these two concerns can not be optimized at the same time. In [2] a more empirical based approach to the effects is advocated, and in [4] a cache friendly, in place sorting MSD Radix algorithm with variable sized radixes is introduced.

III. CACHED MEMORIES

The ratio of CPU to main memory speed has increased for the last 25 years and is now a factor 2000 larger than in 1980 [8]. To bridge this huge performance gap a number of smaller, more costly and faster cache memories have been introduced between the CPU and main memory. These caches are not addressed directly by the user, but operate behind the scene to

make the program almost as fast as if it was executed in L1, the first level cache memory.

Up till now, it has been usual to have two levels of caches, the L1 cache, typically 12-128 Kb, and a much larger and slower L2 cache with 1- 4 MB. On more recent processors we also find a L3 cache with a 3-16 Mb. The L1 cache might have from 1 to 3 CPU cycle access time; the L2 has a 10-20 cycles access time and main memory have a memory access time of several hundreds of CPU-cycles. The penalty for not finding what you are looking for in a cache, a cache miss, hence easily incurs a 10 times access penalty.

Data and instructions that are used by the CPU is first looked up in the L1 cache, and if it isn't there a cache miss occur, and a request is sent to the L2, and if it is not there a second cache miss occur from L2 to main memory (or to the L3 cache if present). When data eventually is found, a cache line of 32-64 bytes is copied upwards to L2 and then to L1 before it is used by the CPU. The reason for copying more than you asked for is the assumption that most programs usually reference data and instructions sequentially and then the program will not get a cache miss from the next few accesses. Also a prefetch unit is often present on the CPU trying to find out how the program uses data (and instructions) and issues read for data before a cache miss occur. This prefetch, which we will test here, is however always limited by the memory bandwidth compared with the processors ability to process data.

The above description only scratches the surface of the memory system. Just one more feature of a modern CPU should be mentioned, the Translation Lookaside Buffer (TLB), a cache memory in the CPU that holds the most recently used parts of the translation tables. These tables keep track of where the program is, on disk or in memory and is involved in the virtual to physical address translation. These tables usually have one entry for every 4 - 16KB of program [8] and are loaded from main memory to the TLB when used. The point here is not the fetching of a block from disk to main memory, but the much more frequent event of loading parts of the TLB itself from main memory to the CPU when needed, a TLB miss. The TLB miss effects on radix sorting have been investigated by [3], and might be significant but are less frequent than a memory cache miss.

In this paper we take a empirical approach – we run two simple user programs that experience the combined effect of the compiler and memory system with its TLB, the L1 and L2 and the prefetch unit for four different CPUs. One reason for not trying to model the inner working of these CPUs is that they often come in new designs every few months, and not all parts of the memory system is publicly documented. Especially the Pentium4/Xeon processor has had so many features in its core, that a simulation or mathematical model could only capture some of its ever changing features. The main focus for either approach, our empirical or a model-based, seems to be the sizes and speeds of the L1 and L2 (and L3) caches. The only two deviations we find from this will be commented on later. It must also be noted that both AMD and Intel in their latest announced models introduce Quad core design with three levels of caching where only the L3 is shared by the four processor cores.

IV. Two Cache Tests

The first test executes Code fragment 1 for n= 100, 200, 400,.., 52M and for 4 different CPUs. For the smaller array lengths, n, a number of executions are made such that all results represent sorting of 52M integers. First, the array b is filled with a random permutation of 0:n-1. We then see that this random access will for large values of n perform a cache miss for almost every access to b. Then we set b[i]= i, and execute Code fragment 1 for each value of n as the first time. For a given i, the nested references to b[i] will all reference the same element that certainly will be in L1. We call this sequential access. The reason for a 16 level deep nesting of access to the array b and not just one, is that this part of the loop should dominate the total execution time of Code fragment 1. If the nesting is only 2 deep, we get a maximum delay factor of 15, if it is one deep we get a factor of 10, not 60, because of the loop overhead and the one sequential access, the innermost b[i] is mixed with the random access (results not presented in figures).

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

Code fragment 1. Performed for n=100, 200, 400,..,52M. First filled with a random permutation of 0:n-1 in b[] - called random access, then filled with b[i] = i – called sequential access.

Figure 1. The effect of caching on the execution of the same number of array accesses of code fragment 1 with length of the arrays a and b = 100, 200,400,..,52M. With a random permutation of 0,1,..n-1 in the array b, the curves displays the number of times slower this executes compared to a sequential fill of the same array (b[i] = i).

Figure 1 shows for each n the ratio of the random access time divided by the sequential access time. The same number of instructions is executed, but random access in large arrays end up 50 to 65 times slower than sequential access. This we find for all the CPUs investigated (AMD Opteron254 2.8GHz, Intel Xeon 2.8GHz, Intel Core Duo U2500 1.16Ghz and Sun UltraSparc IIIi 1.1 GHz). We also see on the curves when cache misses start from L1 to L2 and from L2 to main

memory. For the small L1 data cache of 8KB for the Xeon (Pentium4) CPU shows performance decrease much sooner than for the other CPUs that have a 32 or 64 KB L1. Again the smaller L2 of the Xeon CPU makes it also thrash must sooner than the other CPUs.

Notice how the Xeon CPU performance increases above n = 2M. This is because the processor then accesses main memory directly without trying the L1 and L2 caches first. This is a wise decision since worst case for a cached memory is worse than a directly mapped CPU to main memory.

We cited [8] that there is a factor of several hundreds between access to L1 and main memory, but in fig. 1 we 'only' find a factor of say 60. How can this be explained? If we look closer at the 16-deep nested accesses to array b, by definition Java will check each of these references to b if the index is within the array limits. Then on each array access it will first load the length of the array from L1, then check the actual displacement against 0 and the array length before doing the actual read operation. This is at least an overhead of 5 CPU cycles. In the sequential access this might then be 5 + 2 cycles (for accessing b in L1) = 7 cycles. In the random access case this might be: 5 + 400 cycles = 405 cycles, and the ratio between these two numbers is 405/7= 57. The actual numbers here might be off the mark, but the essence remains; even a tight random access in an array is intermixed with code that does not generate a cache miss, so the user will experience at worst a degeneration of factor 50-60, not several hundred.

In the second test, shown in Code fragment 2, we investigate how well the CPUs can do sequential reads and writes. The code fragments 2a and 2b are executed for n = 100, 200,…,52M. Both code fragments do an ordinary copy between two arrays. In code fragment 2a the length of these arrays is n, while in code fragment 2b both arrays have fixed length = 100. In the repeated copy of 100 elements, both code and data should all be in the L1 cache in all CPUs. The copying of 100 elements are executed the necessary number of times to ensure that the two code fragments perform the same number of elementary copy operations. In both versions we also repeat all operations 100 times to get measurable execution times. Hence, for each data point on the curves, 100* 52M elements are copied sequentially from b to a.

```
// code fragment 2a
int iter = 52428800;

for (int i= 0; i < iter; i++)
  for(int k= 0; k<100; k++)
    a[k] = b[k];

// code fragment 2b
int iter = (52428800/n)*100;

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

Code fragment 2 – Sequential read and write in arrays. Both fragments performed for n=100,200,400,..,52M.

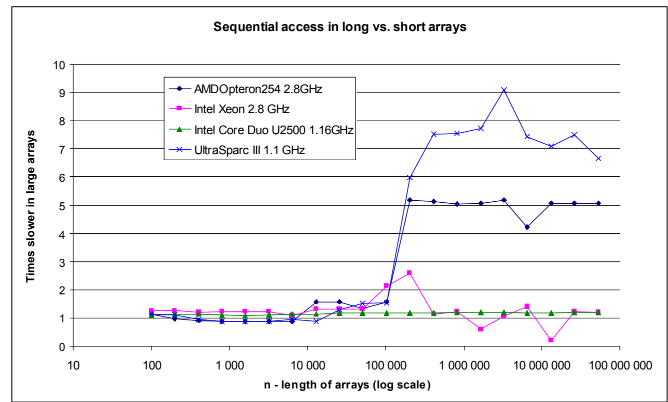

Figure 2. The ratio of accessing sequentially two arrays (one read, one write) compared with the same number of reads and writes in small arrays of fixed size (n=100)

The ratio between executing code fragments 2a and 2b is presented in fig. 2. We see that for the AMD Opteron and especially the Sun Ultra SparcIIIi, the sequential access to large arrays is from 5 to 8 times slower than the same number copy operation in short arrays. We also note that the two Intel processors do not experience any degradation when accessing large arrays. The erratic performance of the Xeon processor is difficult to explain. Both the execution times for the short and long array copy have unexpected values for large values of n. We guess that it might be caused by a varying number of TBL misses (a few trivial changes to this test did not change this erratic behavior for the Intel Xeon – the effects are real and puzzling). It must here be noted that the Pentium 4 versions will be phased out, and that new CPUs from Intel will be based on the Pentium III/ PentiumM/Dual Core line of processors [14].

To conclude these two tests, random access in arrays should only be done in small arrays that fit into L1. Sequential access is much faster than random access, and on some CPUs, especially for the Intel Duo Core, sequential access time per element is constant, independent of array size

V. TESTING THE EFFECT OF CACHING ON RADIX SORTING

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 [9] or in standard reference works [10,11]. New sorting algorithms are still being developed, like Flashsort [7] and 'The fastest sorting algorithm" [12]. The most influential sorting algorithm introduced since the 60'ies is undoubtedly the distribution based 'Bucket' sort which can be traced back to [13].

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 proven [9] that the time complexity of a comparison based algorithm is at best O(n logn). Well known comparison based algorithms are Heapsort and Quicksort [6]. Distribution based algorithms on the other hand, sort by using the values of the elements to be sorted directly. Under the assumption that the numbers are (almost) uniformly distributed, these algorithms can sort in O(n) time (under the uniform cost assumption). Well known distribution based algorithms are Radix sort in its various implementations and Bucket sort.

The above cache tests might seem somewhat artificial. While copy between arrays is common in algorithms, indexes that are 16 levels deep nested array accesses are certainly not realistic. To test if these results carry over to actual programs, three versions of the LSRadix algorithm were designed – sorting on the digits from right to left. The first algorithm uses only a 1-pass with one very 'large' digit (as large as the maximal element in the array), the second is a 2-pass algorithm that splits this radix into two equally sized smaller digits, while the last 3-pass Radix algorithm splits this original digit into 3. The comments in code fragments 3 and 4 further explain these 3 algorithms. They use a single method 'radixSort' to do the actual sorting.

```
static void radix1 (int [] a, int left, int right) {
      // 1 digit radixSort: a[left..right]
      int max = 0, numBit = 1, n = right-left+1;

      for (int i = left ; i <= right ; i++)
           if (a[i] > max) max = a[i];

      while (max >= 1<<numBit) numBit++;

      int [] b = new int [n];
      radixSort( a,b, left, right, numBit, 0);
 }

 static void radix2(int [] a, int left, int right) {
      // 2 digit radixSort: a[left..right]
      int max = 0, numBit = 2, n = right-left+1;

      for (int i = left ; i <= right ; i++)
           if (a[i] > max) max = a[i];

      while (max >= 1<<numBit) numBit++;

      int bit1 = numBit/2,
          bit2 = numBit-bit1;

      int[] b = new int [n];
      radixSort( a,b, left, right, bit1, 0);
      radixSort( a,b, left, right, bit2, bit1);
 }

 static void radix3(int [] a, int left, int right) {
      // 3 digit radixSort: a[left..right]
      int max = 0,numBit = 3, n = right-left+1;

      for (int i = left ; i <= right ; i++)
          if (a[i] > max) max = a[i];

      while (max >= 1<<numBit) numBit++;

      int bit1 = numBit/3,
          bit2 = bit1,
          bit3 = numBit-(bit1+bit2);

      int [] b = new int [n];
      radixSort( a,b, left, right, bit1, 0);
      radixSort( a,b, left, right, bit2, bit1);
      radixSort( a,b, left, right, bit3, bit1+bit2);
   }
```

Code fragment 3 – The radix1, radix2 and radix3 algorithms

```
static void radixSort
( int [] a, int [] b ,int left, int right,
                          int maskLen, int shift){
     int  acumVal = 0, j, n = right-left+1;
     int mask = (1<<maskLen) -1;
     int [] count = new int [mask+1];

  // a)count=the frequency of each radix value in a
     for (int i = left; i <=right; i++)
        count[(a[i]>> shift) & mask]++;

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

  // c) move numbers in sorted order a to b
     for (int i = 0; i < n; i++)
       b[count[(a[i+left]>>shift) & mask]++] =
                  a[i+left];

  // d) copy back b to a
     for (int i = 0; i < n; i++)
       a[i+left] = b[i] ;

   }/* end radixSort */
```

Code fragment 4 – The radixSort algorithm

In this paper we compare sorting of arrays of length n = 50, 100, 200,…,52M. These arrays are filled with the numbers 0:n-1 drawn with equal probability – the U(n) distribution. Also, when n < 52M, the necessary number of such arrays are sorted such that every data point on the curves represents the sorting of 52M numbers. For comparison the three radix algorithms are compared with Quicksort (The java API: Arrays.sort) and Flashsort, a one digit bucket sort that claims to be faster than Quicksort. The code for Flashsort is taken from its homepage [15]. Flashsort also have a 2 pass version, but that turned out to be much slower. Figures for these five sorting algorithms are given in fig. 4 – 8 for the four different CPUs tested. For comparison, the execution times are normalized to the sorting time for Quicksort for that n and that CPU.

To get a grip of the absolute execution times, we first present in figure 3 the actual execution times for Quicksort (as nanoseconds per sorted element as function of array length). From this figure we see that the AMD Operon254 clearly is the fastest machine, and that it also scales best because it has the slowest slope as n increases. Quicksort is characterized by many (log n) sequential passes through the

array, both forwards and backwards and does on the average n/2* log n swaps of elements in the array. In the 1970'ies, when machines did not have caches, Quicksort was considered the fastest sorting algorithm. From the figures we see that this is no longer the case. Quicksort is in all cases outperformed by the radix2 and radix3 algorithm, and for moderate values of n also by the radix1 and Flashsort. The main reason for this is that radix does more on the CPU itself (shifts, adds and ands) that is now fast, but far fewer reads and writes to main memory. Quicksort on the other hand does 25 full passes of the array when n = 52M but only one simple comparison operation on the CPU. Even though Quicksort does only sequential reads and writes, the caching system can't quite hide the speed gap between CPU and main memory. (Since this is not primarily a sorting paper, it must be mentioned that none of the radix-algorithms are optimal. A radix algorithm that uses 1 digit for small values of n, 2 digits for moderate values,… is obviously better).

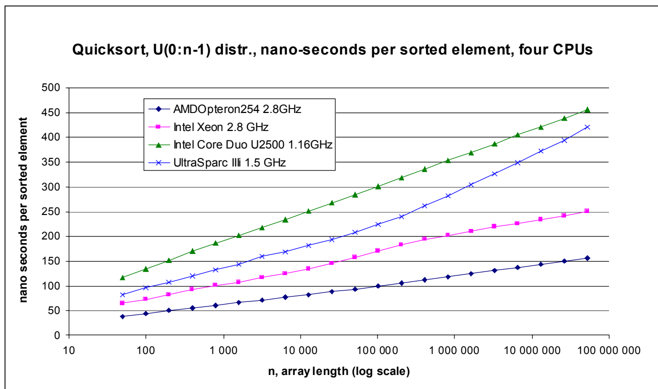

Figure 3. Absolute execution times, nanoseconds per sorted element, for Quicksort on four CPU, as function of the length of he sorted array n =50, 100, 200,…, 52M.

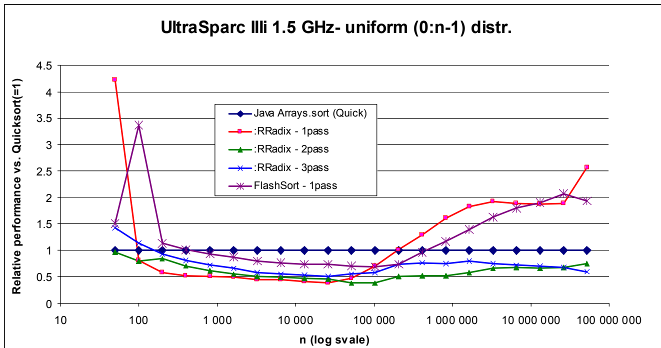

Figure 4. Four sorting algorithms relative to Quicksort on a Sun UltraSparc IIIi based machine with L1 data=64KB, L2= 1Mb, 1.5 GHz.

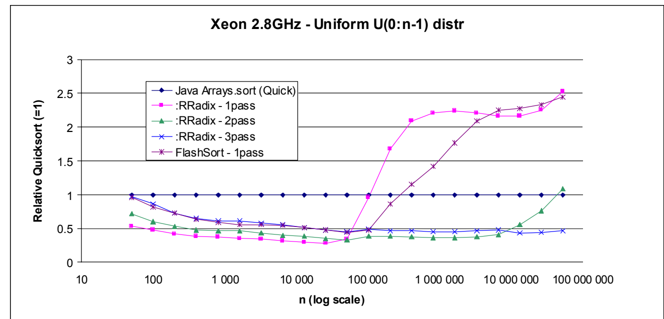

Figure 5. Comparison of five sorting algorithms on a Intel Xeon(Pentium 4) based machine with L1 data=8KB, L2= 0.5Mb, 2.8 GHz.

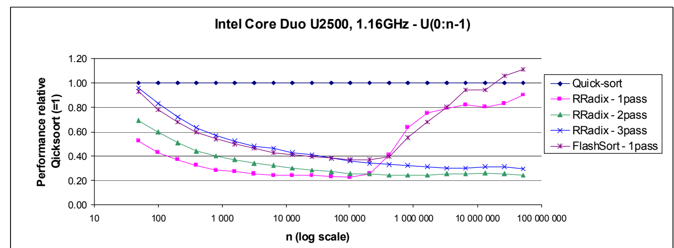

Figure 6. Four sorting algorithms relative to Quicksort on a Intel Core Duo U2500 based machine with L1 =64KB, L2= 2Mb, 1.16 GHz.

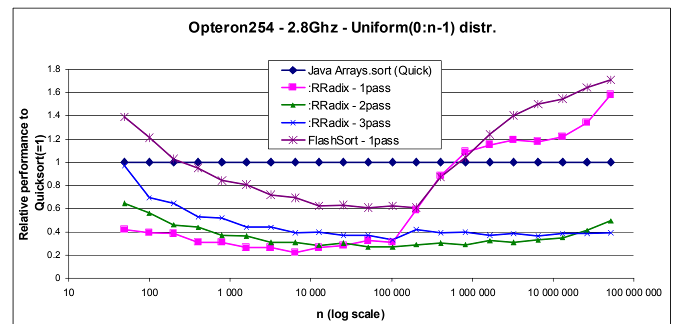

Figure 7. Four sorting algorithms relative to Quicksort on an AMD Opteron254 based machine with L1 =64KB, L2= 1Mb, 2.8 GHz.

From the above empirical tests on all CPUs, radix1 is almost twice as fast as radix2 and almost tree times as fast as radix3 when n < L1. But when random accesses in step a) and c) in method radixSort (Code segment 3) start to fail in the L1 and L2 caches, we see that radix1 gets slower. On all CPUs we see that this slowdown increases steeply when the count array is larger than the L2 cache. The execution time then gets the penalty of cache misses from L2 to main memory. We also note that on all CPUs except on the CoreDuo with 2 MB L2, also the radix2 starts to get slower than the radix3 algorithm for the same reason.

VI. A MODEL FOR ALGORITMIC PERFORMANCE

We will model the execution time of the three radix algorithms based on the performance figures for random versus sequential access in fig. 1. Let E_k denote the number of the different operations for a k-pass radix algorithm (k=1,2,..), S denote a sequential read or write, and R_k a random read or write in m different places in an array where $m = 2^{(\log n)/k}$ (= $\sqrt[k]{n}$)– then:

$$E_k = nS + k[\, n(2S + R_k) + 2mS + n(R_k + R_k + 3S) + n2S]$$
$$= nS + nk\,(7S + 3R_k) + 2mkS$$

Two important notes here: 1. In step c) the array b of length n is accessed in a random way , indexed by random elements of array count. So why do we use R_k and not R_1 ? The reason for this is that we count the different number of possible points of random access, the number of possible cache lines, which is k, not the length of the array itself. 2. Some of the CPUs had a degradation when doing sequential access (fig. 2). Should this not be taken into account here? The answer is that fig.1 already takes this into account since it displays the ratio between sequential and random access, not absolute random access times. Hence we get:

$$E_1 = 10nS + 3nR_1$$
$$E_2 = nS + 2n\,(7S + 3R_2) + 2S\,2^{(\log n)/2}$$
$$E_3 = nS + 3n\,(7S + 3R_3) + 3S\,2^{(\log n)/3}$$

When n is not small, both $2^{(\log n)/2}$ <<n and $2^{(\log n)/3}$<<n, terms with these are omitted and we get:

$$E_2/E_1 = (15S + 6R_2)/(10S + 3R_1)$$

$$E_3/E_1 = (22S + 9R_3)/(10S + 3R_1)$$

The analytical values for the ratio for the execution times for the radix algorithms with small and large values of n are with constants taken from fig. 1:

	n small: n < L1, $R_1 = R_2 = R_3 = S$	n large: n >>L2, $2^{(\log n)/3} < L1 < 2^{(\log n)/2} < L2$ $R_1 = 10R_2 = 50R_3 = 50S$
E_2/E_1	21/13 = 1.61	(15+6*10)/(10*1+3*50) = 0.46
E_3/E_1	31/13 = 2.38	31/(10*1+3*50) = 0.19

We see that this simple model is in very good accordance with the results in figures 4 to 7.

VII. CONCLUSION

This paper has demonstrated that one has to be very careful when writing code in programs and algorithms that accesses memory. Random access in large arrays can slow your software down by a factor 10 or more. For speed, it is necessary to design central algorithms so randomly accessed data fit well into the cache memories. The number of instructions performed is no longer a good measure of software efficiency. We have also demonstrated that the execution time of an algorithm can realistically be modeled by the weighted sum of the number of sequential and random accesses it performs.

REFERENCES

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

[2] Maus, Arne. "Sorting by generating the sorting partition, and the effect of caching on sorting". In. NIK'2000. Norwegian Informatics conference (ISBN 82-7314-308-2), Trondheim: Tapir,2000:

[3] 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

[4] Maus, Arne,"ARL, a faster in-place, cache friendly sorting algorithm". in NIK'2002 - Norsk Informatikkonferanse NIK'2002 – Norwegian Informatics conference (ISBN 82-91116-45-8) Trondheim: Tapir, 2002: p.85-95

[5] Naila Rahman and Rajeev Rahman, "Analysing cache effects in distribution sorting", Dec. 2000, Journal of Experimental Algorithmics (JEA), Vol. 5, ACM Press, NY, USA

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

[7] Karl-Dietrich Neubert: „Flashsort", in Dr. Dobbs Journal, Feb. 1998

[8] John L. Hennessy , David A. Patterson, Computer architecture a quantitative approach, (3nd ed.): Morgan Kaufmann Publishers Inc., San Francisco, CA, 2003

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

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

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

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

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

[14] Wikipedia.com

[15] http://www.neubert.net/Flacodes/FLACodes.html