

An Analysis of Superscalar Sorting Algorithms on an R8000 Processor *

Josep-L. Larriba-Pey Daniel Jiménez
Juan J. Navarro

Computer Architecture Dept., Universitat Politècnica de Catalunya
Jordi Girona 1-3, Modul D6, E-08071 Barcelona, (Spain)
e-mail: {larri, djimenez, juanjo}@ac.upc.es

Abstract

In this paper we compare and analyze different in-memory sorting algorithms to understand their behavior on a superscalar MIPS R8000 processor. We explore Quick sort, Heap sort and an implementation variant of Radix sort that we propose. We compare the methods isolated and combined with Multiway merge and Bucket sort. The combination of methods helps to check for potential use of locality. We describe and analyze the models of the most significant algorithms.

Some conclusions can be drawn from this work. First, Radix sort is the fastest algorithm. Second, the use of combined methods does not help to exploit locality. Third, with the help of the models and an analysis of the codes, it is possible to understand that Radix sort is the most promising of the methods studied here for future superscalar architectures.

1 Introduction

Sorting is necessary in many important applications like Databases [Dat95]. Because of its importance, sorting has been included in different database benchmarks [Bit94], [Ano89]. Therefore, it is necessary to optimize sorting algorithms for up to date computers.

Present computers have three very important characteristics. First, most of them use off-the-shelf superscalar processors both in parallel and single processor implementations [SGI94a]. Second, they incorpo-

*This work was supported by the Ministry of Education and Science of Spain under contract TIC-0429/95 and by CEPBA. Daniel Jiménez was supported by a research grant from the Ministry of Education and Science of Spain for pre-graduate students.

rate different levels in the memory hierarchy [Hen96]. Third, given that the price of memory gets cheaper constantly, the size of the memory will keep growing. As a consequence, it will be possible to run in memory, applications that used to be run out-of-core [Gar92].

So, the objective of this paper is to analyze different strategies to sort in-memory sets of data on one superscalar processor MIPS R8000 and its memory hierarchy. We have restricted our problem to sets of data that fit in the second level of cache.

We have not found a comparison of this type in the most recent literature and we intend to perform it with this piece of work. Recent work in the area focusses on overlapping I/O with sorting operations which considerably differs from our approach [Aga96] [Nyb94]. Other work focusses on sorting on distributed memory computers with superscalar processors [San97].

The strategies analyzed here are Quick sort, Heap sort and a variant of Radix sort that we propose. We study them as isolated methods and combined with Multiway merge and Bucket sort. With the study of the combined methods we search for potential use of locality. That is, we want to see if sorting small sets of data by Quick sort, Heap sort and Radix sort reduces the memory access time sufficiently as to compensate for the extra work introduced by Multiway merge or Bucket sort. The comparisons show that locality is not exploited and that Radix sort is the fastest method.

We have made models of the most representative methods (Radix sort, Quick sort and Bucket sort combined with Radix sort) to understand their behavior.

The analysis of the models allows us to draw some conclusions and perform an extrapolation of the behavior of the methods on future superscalar processors. First, the amount of work done by the processor for Radix sort is around 1/4 of that done for Quick sort and

1/2 of that done for Bucket sort combined with Radix sort. Second, the amount of memory contention cycles for the three methods is similar. A prefetch strategy might be good for them, although a study on prefetch strategies has not been possible here because the R8000 does not allow prefetch. As a global extrapolation, we can say that Radix sort has a code that adapts better to multiple issue superscalar processors than other methods. Also, the disadvantage of Radix sort is the amount of memory it requires which is double than Quick sort or Heap sort.

The paper is structured as follows. Just below, we give an account of the characteristics of the SGI Power Challenge and the MIPS R8000. Then, in section 3, we describe the sorting methods, propose a variant of Radix sort and compare them. In section 4 we describe the models that we have built and validate them. Finally, in section 5 we conclude.

2 The target computer

The target computer is an SGI Power Challenge with four MIPS R8000 processors [SGI94a]. We have used one of those processors to run our algorithms.

The R8000 [Yan94] can issue up to 4 instructions per cycle that can be any combination of two data movement (two Load or one Load and one Store), two Integer or two Floating Point ones. We deal with integer sorting algorithms so we are limited to two data movement and two Integer instructions per cycle. Also, the chip has separate instruction and integer data first level caches of size 16 Kbytes each. Those caches are direct mapped. The size of the cache line is $L_1 = 32$ bytes.

Each processor of the Power Challenge has its external private 4 set associative 4 Mbyte second level cache. The size of its cache line is $L_2 = 128$ bytes. The processors are connected to main memory by means of a unique bus. The main memory is organized in 8 modules and its size is 512 Mbytes.

There are several issues that we took into account when building the models of our algorithms for the R8000. First, the latency of its arithmetic operations. Second, the penalty cycles caused by its branch prediction mechanism. Third, the latency of the memory instructions for the levels of its memory hierarchy. Table 1 shows the features that we use in the paper.

Finally, in the case of a miss in the first level cache, the cache stops executing any other request until that being processed is served. In this case, a set of two arithmetic operations can be issued in the cycle after the miss, then the processor stops issuing instructions until the memory hierarchy serves the requested data item. This does not allow to perform data prefetch.

Operation	Cost in Cycles
<i>branch misspredict</i>	3
<i>load/store</i> (1st level cache)	1
<i>load/store</i> (2nd level cache)	10
<i>load/store</i> (main memory)	60

Table 1. Branch misspredict penalization and memory latency for the R8000.

Compiler issues

Compilers use Unrolling [DoHi79] or Software Pipelining [All95] to tune sequential codes. Both allow a proper scheduling to exploit the instruction pipeline.

We use the MIPSPro Fortran compiler for one processor of the SGI Power Challenge [Hog94]. The results given in this paper are obtained with the maximum level of optimization for one processor (-O3) which performs Software Pipelining and Unrolling.

3 The algorithms

We start this section by describing Multiway Merge and Bucket sort. Then, we compare Quick sort, Heap sort and the version we propose of Radix sort with their combined versions. Finally, we make a global comparison of the algorithms.

We focus on sorting vectors of N keys and pointers. In database applications, sorting is done on records that may be very large. So, to avoid moving large sets of data, the key is extracted from the record, a pointer is created and they are both copied to a new vector of key and pointer tuples. This has been suggested in [Sha94] and [Nyb94]. So, we assume that this operation has already been done before starting to sort.

The size of the key and pointer is 4 bytes each. They are declared as integers though they only take positive values. The measures are given for vectors ranging in number from 1K up to 200K tuples which means that they fit in the second level cache.

The sets of keys that we use are distributed at random, are not replicated and are sorted ascendingly.

In this section, measures are given in terms of speed, i.e. Number of Elements sorted Per Second (NEPS).

3.1 Multiway merge

Multiway merge has traditionally been used as an external sorter using the disk as external storage. Here, we use it as a first level cache external sorter with the second level cache as external storage.

With Multiway merge, a set of q pre-sorted runs of size n ($q = N/n$) are merged by means of a binary selection tree [Knu73]. At the beginning, the tuple with smallest key of each run becomes a leaf of the selection tree. Then, sets of 2 leaves are compared, the smallest of each set is promoted to the parent node and the following tuple in the run is promoted as a leaf of the tree. This is done recursively until the tree is full. From then on, the tuple on top of the tree is extracted and placed in the final sorted vector, and the tuples of the runs are promoted correspondingly.

The tree has a total of $2 * q - 1$ tuples. Thus, Multiway merge requires a total amount of operations proportional to $N \log(2 * q - 1)$ ¹. This strategy requires a total amount of memory proportional to $2N$.

3.2 Bucket sort

Bucket sort is based on the fact that the key can be used as the index for the tuple in the final sorted structure. So, an extra vector of size 2^B is required where B is the number of bits of the key. With this strategy, the final sorted vector would have empty positions unless the number of tuples was 2^B .

In our case, we have 32 bit keys and the memory needed would be too large. So, to get around this problem one can use the most significant b bits of the key as index. With the b indexing bits, we can group the tuples into 2^b buckets. The number of tuples per bucket depends on the distribution of the key values.

After the distribution of the tuples into 2^b buckets, each bucket has to be sorted. The operation count for Bucket sort is proportional to N .

Bucket sort can be implemented in different ways. For instance, with a tuple matrix of size $N \times 2^b$ or a set of 2^b linked lists that require a pointer vector. We use a linked list which requires $2N$ memory positions.

3.3 Quick sort

Knuth describes Quick sort simply [Knu73]: “The idea (...) is to take one record, say R_1 , and to move it to the final position it should occupy in the sorted file, say position s . While determining this final position, we will also rearrange the other records so that there will be none with greater keys to the left position of s , and none with smaller keys to the right. Thus the file will have been partitioned in such a way that the original sorting problem is reduced to two simpler smaller problems, namely to sort R_1, \dots, R_{s-1} and (independently) to sort R_{s+1}, \dots, R_N . We can apply the same technique to each of the subfiles, until the job is done.”

¹Logarithms in this paper are always assumed to be base 2.

The iterative version of Quick sort that we implement requires an auxiliary structure managed as a stack to keep track of the partitions that have to be sorted. The size of the stack must be $O(\log N)$ in the worst of the cases [Wir76].

There are different ways to choose one key to divide each partition [Knu73]. The simplest one is to take the first element of each partition. This strategy gives a good operation count for a randomly distributed set of keys $O(N \log N)$. Nevertheless, if the original vector is already sorted, this strategy gives a worst operation count $O(N^2)$. There are different solutions to this problem and they are very well documented in [Knu73]. This is the strategy we implement in this paper.

Execution of Quick sort

Figure 1 shows the NEPS obtained for Quick sort, three cases for the combination of Quick sort and Multiway merge and a combination of Quick sort with Bucket sort where the $b = 14$ most significant bits of the key are sorted with Bucket sort (16384 buckets).

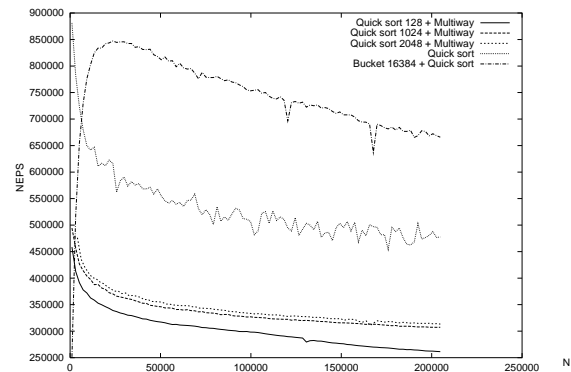


Figure 1. Speed (Number of sorted Elements Per Second, NEPS) as a function of the set size for different implementations of Quick sort.

The combined use of Quick sort with Multiway merge is slower than Quick sort alone even for large runs that fit in cache (2048 tuples). Also, the combination of Quick sort with Bucket sort is faster than Quick sort alone. This shows that the amount of work performed by Bucket sort combined with Quick sort is smaller than that for the other methods. We analyze this later with the models of the algorithms.

3.4 Heap sort

Heap sort is based on first creating a max-heap of the run. A max-heap is defined as a structure where

the keys (K_j) have the following relation [Knu73]:

$$K_{j/2} > K_j \text{ for } 1 \leq j/2 < j \leq n$$

In other words, it is a binary tree where the key of each node is larger than the key of its two sons.

After the creation of a max-heap, it is easy to sort the tuples in a second phase. In order to sort the largest key, it is necessary to exchange the tuple on top of the tree with the last tuple in the vector. This way, the max-heap has one tuple less. Now, the largest tuple is in its final sorted position and the tuple on top of the tree has to be sifted down to its correct place in the tree. The same strategy can be applied to every new tuple on top of the tree with the second, third, ..., last tuples of the vector.

The operation count for this method is $O(N \log_2 N)$.

Execution of Heap sort

Figure 2 shows the NEPS obtained for Heap sort and four different combinations of Heap sort and Multiway merge. The combinations shown are for run sizes of 32, 64, 256 and 1024 tuples per run. Here, we do not show the combination of Bucket sort plus Heap sort because Heap sort is very inefficient and we do not analyze it in following sections.

In this case, the combination Heap sort plus Multiway merge is faster than Heap sort alone. Also, the smaller the runs (32, 64 tuples per run), the faster the execution. We foresee that the combination of Bucket sort plus Heap sort, is far from the speed achieved by the rest of the methods.

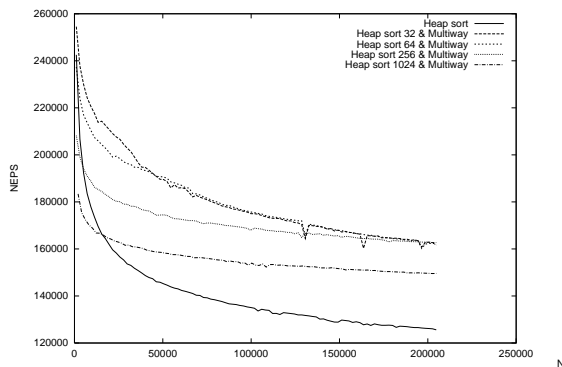


Figure 2. Speed (NEPS) of the implementations of Heap sort.

3.5 Radix sort

Radix sort performs three phases for each digit of the key starting from the least significant one. First,

it counts the number of occurrences for each possible value of that digit. Then, the counters are accumulated in such a way that they can be used as indices to store the tuples in an auxiliary structure. Finally, the tuples are moved accordingly to the indices.

Figure 3 shows an example of this procedure for 2 decimal digit keys. In the example, it is necessary to create a vector with 10 counters, one for each possible value (0 to 9) of one digit. Step 1 is performed on the least significant digit of the key. The counters show, for instance, that value 1 of the least significant digit has 6 occurrences in the source vector S and value 7 has 2 occurrences. The accumulators show that tuples with value 0 in the least significant digit have to be stored starting at position 0 of the destination vector D , tuples with value 1 starting at position 1, tuples with value 2 starting at position 7, etc.. The process after creating the accumulated structure, uses those accumulators to compute the effective address of the tuple in the destination vector D for that step. In the example, we show the situation of the accumulators after moving 9 tuples from vector S to vector D . Note that, after the movement of each tuple, the corresponding accumulator is incremented.

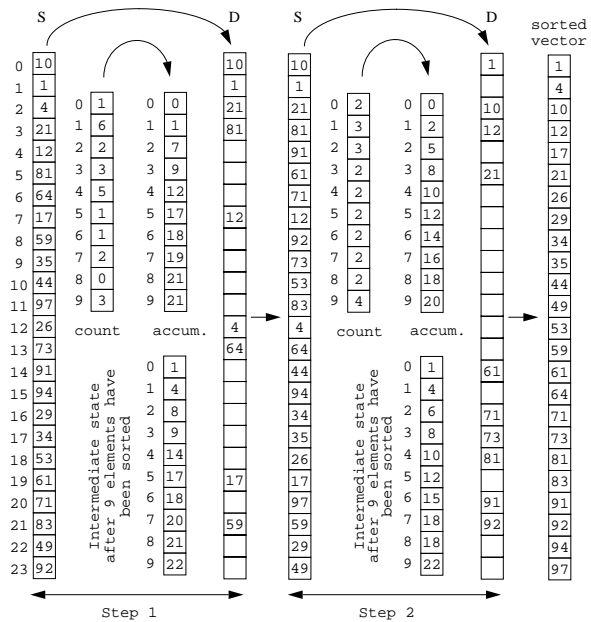


Figure 3. Steps of Radix sort for 2 decimal digit keys.

The same procedure is performed on the most significant digit in step 2. In this case, vectors S and D change their role. The final result is shown in Figure 3.

The operation count for this method is proportional to N times the number of digits of the key d , $d * N$.

Given that we assume no values repeated for the keys, $d > \log N$. So, $d * N$ is larger than $N \log N$ for Quick sort or Heap sort.

Implementation of Radix sort

For the implementation of Radix sort, we schedule its loops so that they reduce the complexity of the method. We give more details in the following.

The tuples we are sorting have 32 bit keys. This implies 32 steps of those explained above and two counters. This is very inefficient.

One way to reduce the number of steps is to group the bits in sets so that less steps have to be performed. Here, we make 4 groups of 8 bits. So, it will be necessary to use a counter vector with 256 positions. We chose to make groups of 8 bits for two main reasons. First, 8 bits imply a reasonable number of counters and are a multiple of 32. Second, we tried other combinations like 8 groups of 4 bits and 3 groups of 10, 11 and 11 bits and were slower.

The strategy chosen requires loading all the tuples 8 times, one for each count phase and one for each movement phase of each step. The dependence graph of the 4 steps of the method is shown in Figure 4.a.

Now, we propose a different scheduling of the steps of Figure 4.a that reduces the number of tuple loads.

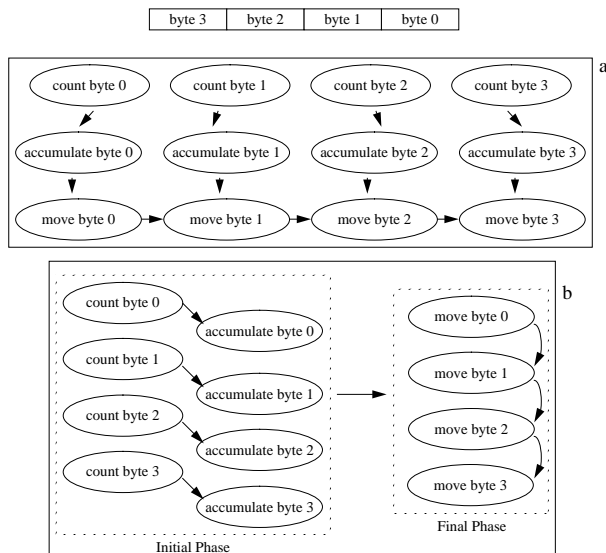


Figure 4. Radix sort. (a) Dependence graph of the different steps, (b) Radix 2 Phase (R2P).

This scheduling works in a 2 phase fashion and its dependence graph is shown in Figure 4.b. We call it

Radix 2 Phase (R2P). This approach requires 4 counter vectors with 256 positions each. During the first phase, we load once each tuple and count for each byte of the key of that tuple. After that, we accumulate all the counters. During the second phase, the sorting is performed in a 4 hop data movement that requires 4 loads per tuple. The total amount of loads is 5. This will be clearly faster than the baseline method of Figure 4.a.

This approach requires a total of $16 * N + 4096$ bytes of memory. This is, 2 vectors of size $8 * N$ bytes plus 4 counters of 256 positions of 4 bytes each.

Figure 5 shows the NEPS achieved by R2P and its combination with Bucket sort. The algorithm that implements the combination works as follows. Bucket sort is performed on the 8 most significant bits of the key creating 128 buckets. Remember that we suppose that the keys are unsigned but we represent them as signed integers, so, the most significant bit is always 0 and for this reason we create 128 buckets instead of 256. After that, each bucket is sorted for the 24 least significant bits with Radix sort. This means that we only need to perform 3 steps of Radix sort.

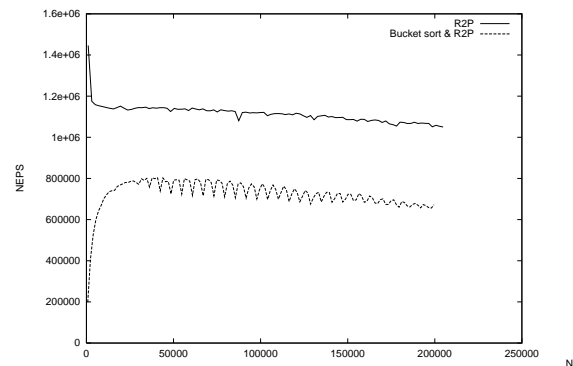


Figure 5. Speed (NEPS) of Radix sort (R2P) and its combination with Bucket sort.

In Figure 5 we can see that Radix sort is 1.5 times faster than its combination with Bucket sort. This means that it is not possible to exploit locality with the combination of those methods when the size of the data structures is smaller than the cache size. For example, for 100K tuples, our implementation of Bucket sort generates 128 buckets with approximately 700 tuples each. A set of 700 tuples requires less than 16K bytes in data structures which means that the buckets would fit in cache. The plot of Figure 5 does not show any difference between data sets larger and smaller than 100K tuples which makes us think that there is no such exploitation of locality.

3.6 Comparison of the methods

Now, we compare the most significant implementations of the algorithms. Figure 6 shows the NEPS obtained by Quick sort, Heap sort on runs of size 32 tuples combined with Multiway merge, Bucket sort with $2^b = 2^{14} = 16384$ buckets combined with Quick sort to sort each bucket, Radix sort in two phases (R2P) and Bucket sort with 128 buckets combined with R2P.

Our implementation of Radix sort (R2P) is the fastest algorithm. This version is 1.5 times faster than Bucket sort combined with either Quick sort or Radix sort. Also, it is twice as fast as Quick sort.

The combination of Radix sort and Bucket sort achieves a similar speed than the combination of Bucket sort and Quick sort. The former is implemented with 128 buckets which adapts to our problem, as we said. The latter is implemented with 16384 buckets which is the implementation that behaves better.

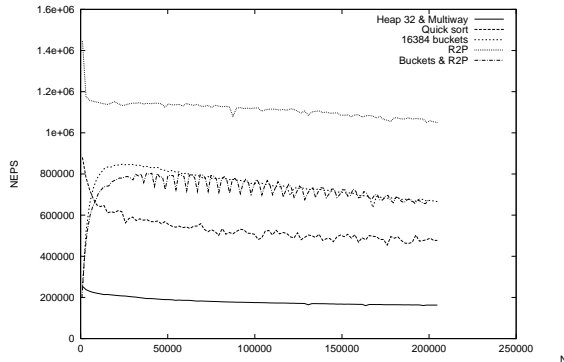


Figure 6. Speed (NEPS) of the best versions of all the methods.

4 Analysis of the methods

In this section we analyze the models of the most representative methods: Radix sort, Quick sort and the combination of Bucket and Radix sort.

The reasons for choosing those methods are as follows. Radix sort is the fastest of the methods studied here so, we want to know why it behaves so well. Quick sort is the fastest representative of those methods with complexity $O(N \log N)$ and we want to know about its exploitation of resources. On the other hand, Bucket sort combined with Radix sort has complexity $O(N)$. We want to know why this combination does not exploit locality as we supposed.

The model that we use for the algorithms can be expressed in terms of total number of cycles spent by

each algorithm

$$C = C_{cpu} + C_{cache_1} + C_{cache_2} + C_{ov}$$

The amount of cycles spent in the CPU (C_{cpu}) can be expressed as the sum of the cycles spent in each loop times the number of iterates for that loop

$$C_{cpu} = \sum_{loops} N_{iter} * N_{cycles}.$$

For each of the algorithms we have obtained the CPU model with the help of the SGI MIPSPro compiler comments. With those comments the compiler tells us about the cycle at which each instruction starts its execution. In order to validate those figures, we have compared the results to hand made Software Pipelining versions of the loops we are modelling.

The number of cycles spent in memory level i (C_{cache_i}) can be expressed as the number of misses for that level times the latency of the following level $i + 1$

$$C_{cache_i} = N_{miss_i} * C_{lat_{i+1}}.$$

A particular issue of our implementations is that data fit in the second level cache. So, only compulsory misses have been taken into account for that level.

The cycles spent on overheads of the algorithm (C_{ov}) like routine calls, initialization, I/O, etc. are not taken into account in our models.

Here, we explain the complete model for Radix sort and the C_{cpu} model for Quick sort.

4.1 A model of Radix sort

Our implementation of Radix sort is divided into an initial phase (count and accumulation) and a final phase (4 hop data movement). The loops for the count and data movements perform N iterations and, the loop for the accumulation only performs 256 iterations. Given this, we only take into consideration the loops with N iterations. Although this is incorrect for small sets of data we consider it sufficient as we are interested in the global behavior of the algorithm.

CPU

We have based our algorithm, on the use of arithmetic and shift instructions. We could have based it on compares and branches but those instructions partition the execution flow of a loop which reduces the degree of Instruction Level Parallelism.

The number of CPU cycles spent in each of those phases is $11N$ for the count and, $3.5N$, $4N$, $4N$ and $4N$ for each of the data movements. Those values are

obtained with the Software Pipelining of the compiler that performs an unrolling of 2 iterates for the data movement loops. So, the number of CPU cycles is

$$C_{cpu} = 26.5N.$$

Memory hierarchy

Now we discuss a model for the memory hierarchy. Radix sort uses a total of 3 data structures; two vectors (S and D) of size N to hold the 8 byte tuples and 4 sets of 256 counters of 4 bytes each. Vector S is read sequentially while vector D is written depending on the position dictated by the counters.

Second Level Cache misses. We only consider the compulsory misses given that data fit in cache level two. So, the total number of misses is equal to the number of bytes occupied by the data structures divided by the size of the second level cache line $L_2 = 128$

$$N_{miss_2} = (16N + 4096) \frac{1}{L_2} = \frac{N + 256}{8}$$

It is possible to appreciate that the independent term is not significant for the range of values that we are considering and we do not take it into account. So, the cycles for misses in cache level two are

$$C_{cache_2} = N_{miss_2} C_{lat_{mem}} = \frac{N}{8}(T_2 - T_1) = 6.25N$$

where T_1 and T_2 are the latencies of the first and second levels of cache. Note that all misses to the second level cache are also misses to the first level cache, so, the latency of first level cache misses is taken into account in C_{cache_1} .

First Level Cache misses. Now, we discuss the amount of misses for the initial and final phases. For each loop we count the number of misses incurred by each data structure due to interferences with itself and due to interferences with other data structures. We suppose that vectors S and D do not fit in this cache.

During the count of the initial phase we read the N tuples of vector S sequentially which implies a miss every fourth tuple given that $L_1 = 32$. The counters fit in cache and their compulsory misses are small, $4096/L_1$.

The interferences between data structures are as follows. The counters should exploit temporal locality but vector S flushes them from the cache every time 2K tuples are read ($N/2048$). The total amount of misses of the counters due to the interference of vector S is $((N/2048) - 1)4096/L_1$.

Given that vector S only exploits spatial locality, only one line of the cache is busy for it. The interference of a counter with the busy cache line of vector S

is improbable, so, we do not consider this case. So, the number of misses for the count are

$$N_{miss_c} = \frac{8N}{L_1} + \frac{2N}{L_1} = \frac{5N}{16}$$

Now we concentrate on the final phase. Given that all data movements have a similar memory cycle count, we only describe one of them.

We start by the self interferences of data structures

- **Vector S .** For each data movement, vector S is read sequentially. This implies $8N/L_1$ misses.
- **Counters.** We suppose that the counters are in cache from the previous loop when each data movement starts. This means that the 4 sets of 256 counters are in cache although only one is being used at each data movement. This is because we declared them so that the same counter of each set maps in the same cache line.
- **Vector D .** A tuple is written onto vector D depending on the counters, which are indices to memory locations. There are 256 pointers being used during one data movement, so, there will be 256 cache lines busy at one time for vector D . Thus, $8N/L_1$ misses.

We will describe the interferences of each data structure with the rest of data structures

- **Vector S .** The counters occupy 4096 bytes. So, vector S interferes with them as in the initial phase. Vector D occupies 256 cache lines as we said before. So, Vector S interferes with vector D as to generate $(N/2048)8192/L_1$ misses.
- **Counters.** Given that the counters fit in cache, we suppose that they do not interfere with the other structures.
- **Vector D .** Given that vector S keeps one only cache line busy at a time, we consider that vector D does not interfere with vector S . As for the counters, we consider that the interferences of vector D are similar to those of vector S .

So, the total number of misses for each data movement is

$$N_{miss_M} = 2 \frac{8N}{L_1} + 2 \frac{2N}{L_1} + \frac{4N}{L_1} = \frac{3N}{4}$$

Now, we can compute the total number of cycles due to misses to the first level of cache. It is important to recall now that when a miss occurs, it is possible to issue two arithmetic instructions in the cycle after the

load or store that produces the miss. The analysis of the codes for Radix sort tells us that such situation does not occur in any of the loops that we deal with. The cycle count for the first level of cache is

$$C_{cache_1} = (N_{miss_C} + 4N_{miss_M})(T_1 - 1) = 29.8N$$

Now we can show the global model for Radix sort

$$C = C_{cpu} + C_{cache_1} + C_{cache_2} = 62.5N$$

Figure 7 shows the cycles per element measured for Radix sort and the cumulative values of each part of the model, C_{cpu}/N , C_{cache_1}/N and C_{cache_2}/N . The error of the model is around a 5%.

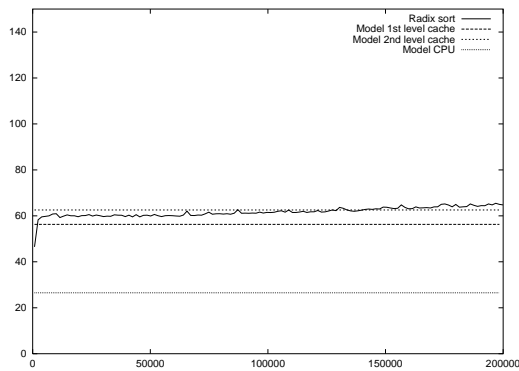


Figure 7. Cycles per element sorted for Radix sort. The plot shows processor cycles, cycles caused by first and second level cache misses and cycles for real measures.

We can observe in Figure 7 that the CPU cycle count amounts to a 42% of the total cycle count per element. This is important as it shows that Radix sort has a lot to gain in processors that allow prefetch.

One important point that can be observed from the models is that the accesses to both vectors S and D show a considerable amount of spatial locality. While for S this is achieved because the accesses are sequential, for D this is achieved because we group elements in 256 sets and each of them is traversed sequentially.

Another important point is that the loops of the algorithm have a large amount of instructions which allow a good scheduling using Software Pipelining.

4.2 A model for Quick sort

In this section we describe the CPU model for Quick sort. This is an interesting model to describe because of the special features of its inner loop shown in Figure 8.

In the code of Figure 8, loop 1 searches for one key value larger than $S1.key$ starting from the beginning

of vector S . Loop 2 searches for one key value smaller than $S1.key$ starting from the end of vector S . Once both elements are found, they are interchanged. With this strategy, value $S1.key$ can be put in its final place when the two indices (i and j) cross. At that point, the problem is divided into two subproblems that can be solved in the same way. This procedure is repeated in a binary tree fashion. At each level of the tree, the N tuples of the vector are traversed completely. Thus, the complexity of the method is $N \log N$.

```

S1.key:=S(1).key;
S1.pointer:=S(1).pointer;
i:=initial_element + 1;
j:=final_element;
while (j>i) do
  while (S(i).key < S1.key) do i:=i + 1; /*(loop 1)*/
  while (S(j).key > S1.key) do j:=j - 1; /*(loop 2)*/
  aux.key:=S(i).key;
  aux.pointer:=S(i).pointer;
  S(i).key:=S(j).key;
  S(i).pointer:=S(j).pointer;
  S(j).key:=aux.key;
  S(j).pointer:=aux.pointer;
enddo
S(1).key:=S(i).key;
S(1).pointer:=S(i).pointer;
S(i).key:=S1.key;
S(i).pointer:=S1.pointer;

```

Figure 8. Inner loop of Quick sort.

We can divide the model of the CPU cycles into three parts. First, the search of elements to interchange. Second, the interchange of the elements. Third, the penalization cycles due to branch misspredictions in loops 1 and 2 of Figure 8.

For the search of the elements, we have that the two loops traverse all the vector at every level of the execution tree. Thus, we have a total of 3 cycles per iterate times $N \log N$.

The interchange of two elements takes 5 cycles. Now, the elements to be interchanged in each loop are approximately one out of two per iterate because they are distributed randomly in vector S . This is equivalent to one interchange per pair of elements ($(N/4) \log N$ in total). The amount of cycles for the interchanges is $(5N/4) \log N$.

A branch misspredict takes 3 cycles in the R8000. A prediction strategy is useful when there is a tendency in taking or not taking the branch. In our case, the probability of taking the branch in both loops is the same as that of not taking it. This is due to the random distribution of key values. So, we can consider that half out of the $N \log N$ iterations for both loops will be misspredictions. This gives a total of $(3N/2) \log N$

cycles. So, the CPU model of Quick sort is

$$C_{cpu} = 4,75N \log N$$

Figure 9 shows the cycles per element for Quick sort. The error of the model is less than 10%. The model shows that the percentage of time spent by the processor is approximately a 75% of the total amount of time. It is important to note here that this amount of cycles can barely be reduced with the scheduling. The reason is that the very little amount of instructions within the internal loops does not allow to exploit Instruction Level Parallelism optimally.

The memory cycles are a 25% of the total and most of them are due to first level cache misses. This means that an ideal first level cache prefetch policy would improve the performance of the method very little.

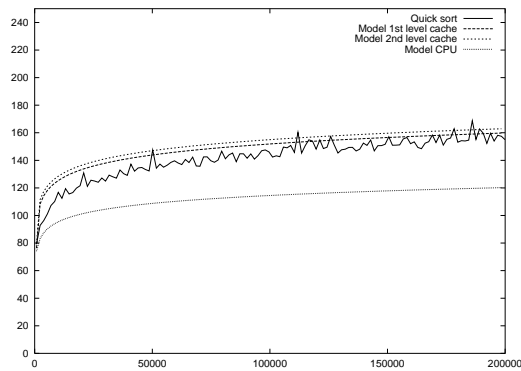


Figure 9. Cycle count per element sorted for measurements and models of Quick sort.

4.3 A model for the combination of Bucket and Radix sort

We do not explain the model of Bucket sort combined with Radix sort but we comment on it. Figure 10 shows the cycle count for this method. The error of the model is less than 15%.

Note in the plot that the amount of CPU cycles is around 50 which is almost double than for Radix sort. An analysis of the code shows that the accesses to linked lists by Bucket sort are very painful due to the amount of indices to be computed.

One other important point to make is that the amount of memory cycles (50 in average) is larger than the amount of memory cycles for Radix sort (35 in average). This tells us that far from exploiting the locality of Radix sort, Bucket sort's data structures are conflicting with Radix sort's structures and increment the memory misses.

So, the explanation we wanted to find to the low speed of this method is both in the memory and the processor behavior.

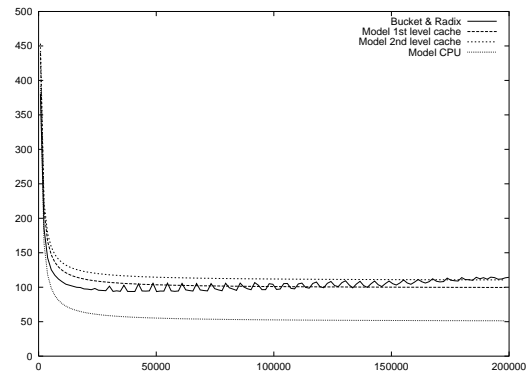


Figure 10. Cycle count per element sorted for measurements and models of Bucket sort combined with Radix sort.

5 Conclusions

In this paper we compared Quick sort, Heap sort, Radix sort and their combination with Multiway merge and Bucket sort. The objective of the paper was to understand the main features of those methods when implemented on a MIPS R8000 processor.

Some conclusions can be drawn from the study we have carried on:

- The fastest algorithm of those analyzed here is the implementation of Radix sort that we propose.
- The combination of Multiway merge or Bucket sort with the methods does not help to improve the locality of the fastest algorithms when implemented in memory.
- The cycle count per sorted element of the analyzed methods shows that the most promising algorithm is the version of Radix sort that we propose. This algorithm spends around 27 processor cycles per element sorted compared to the 80 to 120 for Quick sort and the 50 for Bucket sort combined with Radix sort.

On the other hand, the three algorithms spend a similar amount of time in memory cycles. Thus, in the case of an ideal prefetching policy (which is not supported by the R8000), Radix sort would be 2 to 4 times faster than the other algorithms.

- With this work we have learnt that there are two types of sorting algorithms. First, those algorithms like Quick and Heap sort with a complexity $O(N \log N)$ that have loops with a little amount of instructions. Those instructions are mostly comparisons and branches that depend on the data to be sorted which do not allow a good exploitation of Instruction Level Parallelism (ILP). Second, Radix sort that can be modified to reduce its complexity to $O(N)$. The loops of Radix sort have a large number of arithmetic and logic instructions that allow a good exploitation of ILP.

The behavior of the algorithms on future superscalar processors with a higher ILP than present processors depends strongly on the type and number of instructions within their loops. While the instructions for Radix sort allow a larger ILP than it is exploited in the R8000, those for Quick or Heap sort are very few in number and do not allow for a larger ILP exploitation. So, Radix sort is a good candidate to exploit the features of future superscalar processors while Quick and Heap sort are not.

- One important aspect of the methods analyzed here is the memory space their data structures require. In particular, Radix sort requires more than double the original set of elements to be sorted. On the opposite side, Quick sort is an in-place method, so it does not require any additional structure.

References

- [Aga96] R. Agarwal, A Super Scalar Sort Algorithm for RISC Processors, IBM Research Report, Jan. 1996.
- [All95] V. Allan et al. Software Pipelining, ACM Comp. Surveys, Vol. 27, No. 3, pp. 367-431, Sept. 1995.
- [Ano89] Anon et al., A Measure of Transaction Processing Power, Datamation, V.31(7), pp. 112-118.
- [Bit94] D. Bitton and C. Turbyfill, A retrospective of the Wisconsin Benchmark, in Readings in Database Systems, M. Stonebreaker ed. 1994 (2nd ed.).
- [Dat95] C. J. Date, An Introduction to Database Systems, Addison-Wesley, 1995 (6th ed.).
- [DoHi79] J. Dongarra and A. Hind, Unrolling Loops in Fortran, Software-Practice and Experience, Vol. 9, No. 3, 1979.
- [Gar92] H. Garcia-Molina, K. Salem, Main Memory Database Systems: An Overview, in IEEE Trans. on Know. and Data Eng., Vol. 4, No. 6, 1992.
- [Hen96] J. Hennessy and D. Patterson, Computer Architecture: a quantitative analysis, Morgan Kaufman, 1996 (2nd ed.).
- [Hog94] C. Hogue, MIPSpro Fortran 77 Programmers Guide, Insight, SGI, 1994.
- [Knu73] D. Knuth, The art of Computer Programming; Volume 3/Sorting and Searching, Addison-Wesley, 1973.
- [Nyb94] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray and D. Lomet. AlphaSort: A RISC Machine Sort. Proceedings of the Sigmod 94 Conference, pp. 233-242.
- [San97] D. Sanders, Y. Park and V. Govindan, Parallel Sorting on the NEC Cenju-3 and IBM SP2, Proceedings of HPC on the Information Superhighway (HPC Asia-97) pp. 214-219 IEEE Press (1997).
- [SGI94] R8000 microprocessor chip set, MIPS, Open Risc Technology, August 1994. (<http://www.mips.com>).
- [SGI94a] Silicon Graphics Inc., Power Challenge Technical Report, SGI, August 1994. (<http://www.mips.com>).
- [Sha94] A. Shatdal, C. Kant and J. Naughton, Cache conscious Algorithms for Relational Query Processing. Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994.
- [Wir76] N. Wirth, Algorithms + Data Structures = Programs, Prentice-Hall, Inc. 1976.
- [Yan94] Peter Yan-Tek Hsu, Design of the R8000 Microprocessor. Silicon Graphics Incorporated.