Which Sorting Algorithms to Choose for Hard Real-Time Applications

D. Mittermair and P. Puschner
Institut für Technische Informatik
Technische Universität Wien
A-1040 Wien, Austria
peter@vmars.tuwien.ac.at

Abstract

This paper compares the worst-case performance of eight standard sorting algorithms. It investigates how well-suited these algorithms are for hard real-time systems.

In a series of experiments we determined the average and worst-case execution times of the sorting algorithms for different numbers of elements to be sorted (in the range between 7 and 1000 elements). Average times were extracted from test runs with random data whereas the worst-case times were determined both analytically with an analysis tool and experimentally by construction of the worst-case input data for each algorithm. The experiments demonstrate that algorithms that are well-suited for normal needs are not necessarily suited for hard real-time systems. Thus the results help to choose the right sorting algorithm for real-time applications.

1. Introduction

The problem of sorting is a problem that arises frequently in computer programming. Many different sorting algorithms have been developed and improved to make sorting fast. As a measure of performance the average number of operations or the average execution time of these algorithms has been investigated and compared.

While the average execution time of a program is suited as a performance measure for non real-time applications, the construction of hard real-time systems requires the knowledge of the worst-case execution times (WCETs) of programs [15]. In some cases the minimal and the worst-case execution times are of concern [14, 8]. This paper investigates the worst-case performance of eight standard sorting algorithms (bubble sort, insertion sort, selection sort, merge sort, heap sort, quick sort, radix sort, and sorting by distribution counting) and compares it with their average performance.

Sorting algorithms and their average performance have been broadly discussed in literature [2, 5, 7, 12, 13]. Lately those results were complemented with performance evaluations of sorting algorithms on parallel computers, e.g., [1, 4]. To this date there exists, however, no work that investigates and compares the WCETs of sorting algorithms. In literature, WCET computations for real (sample) programs were only used to demonstrate the quality of prototype WCET analysis tools [3, 6, 9, 10]. The purpose of this paper is to fill the gap: it investigates the WCETs of sorting programs.

For our experiments we implemented the sorting algorithms and evaluated their timing behavior for different numbers of elements (in the range between seven and 1000). Approximations for the average execution times were derived from test runs of the programs with random input data. The worst-case execution times were determined both analytically with a WCET analysis tool according to the theory described in [11] and experimentally by construction of worst-case input data for each sorting algorithm. The comparison of the so-derived average and worst-case data shows that algorithms that perform well in the average case do not necessarily perform well in the worst case. On the other hand, an algorithm with mediocre average performance may turn out to be well-performing if the worst-case behavior is of concern.

The paper is structured as follows. Section 2 describes our experiments in detail. Section 3 presents the average and worst-case execution times of the analyzed sorting algorithms for data sets of different size.

0-8186-8034-2/97 $10.00

Section 4 provides a discussion of the results and Section 5 concludes the paper.

2. Experiment Description

This section gives a detailed description of our experiments. First it lists the algorithms that have been compared and describes the hardware/compiler configuration used for the evaluation. In the second part we explain, how we derived approximations for the average execution times and the bounds for the worst-case execution times of the sorting programs.

2.1. Used Sorting Algorithms

We chose eight classic array sorting algorithms for our experiments: Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Heap Sort, Quick Sort, Radix Sort, and Sorting by Distribution Counting. All algorithms were implemented in C to sort arrays of type integer (2 bytes). The iterative algorithms were implemented as described in standard literature [7, 13]. The recursive algorithms were transformed into iterative programs so we could analyze them for their WCETs with our WCET analysis tool.

We chose very simple versions of all sorting algorithms, e.g., merge sort splits lists down to length one and quick sort simply takes the first element of each list as pivoting element. Our version of radix sort sorts arrays in four passes. In every pass it compares four bits of the two byte integers. Distribution counting was implemented and evaluated for ranges of 500, 1000, and 10000 different keys.

2.2. Experiment Setup

We used simulations to derive the average execution times of the sorting programs. We used both simulation runs and an analytical method to assess the WCET of the programs. The Motorola MC68000 processor with zero wait state memory served as the target architecture for our experiments. We chose the MC68000 processor, since its architecture is fairly simple. It does not use a cache and the effects of pipelining can be considered easily both in simulation and WCET analysis. So we could model the processor with sufficient accuracy to avoid that the computed WCET bounds become too pessimistic and impair the quality of the results.

We compiled the sorting programs with the GNU GCC compiler. In order to simplify the analysis of the compiler-generated code, we did not generate fully optimized code but compiled the program at optimization level O1. The resulting code was simple enough to identify the correspondences with the C source programs. So we could insert annotations for WCET analysis, that describe the possible execution frequencies of the program parts, into the assembly language program easily. Note that the fact that we did not use fully optimized code for our experiments does not change the quality and principal trends of our results. In principle, further optimization of the software or the use of faster hardware only changes the scale of the resulting data.

2.3. Assessment of the Sorting Algorithms

The experimental data were extracted the following way. We assessed the average execution times and WCETs for each sorting algorithm for array sizes of 7, 10, 25, 50, 75, 100, 250, 500, 750, and 1000 elements. As a measure for the average execution times we used the mean value of the execution times of 10000 executions of the sorting algorithms with random data. WCETs were derived both analytically and experimentally for each configuration.

2.4. Average Execution Times

In our experiment the average execution times of all configurations of the algorithms were approximated by the mean value of the execution times of the algorithms with random data. We performed the following steps for every sorting algorithm and number of elements to obtain these approximate values:

1. The C source code of the sorting program was compiled with the GNU GCC compiler to generate code for the MC68000 processor.

2. We analyzed the 68000 assembler code with a tool that computed the execution times of all basic blocks of the code.

3. In the next step we analyzed the control structure of the program. We identified those parts whose execution frequencies we would need to compute the execution time of an execution of the program with sample data.

4. The C source code of the sorting program was modified. We added code for counting the execution frequencies of program parts required for the computation of execution times during program execution.

5. The modified program was compiled to run on a DEC alpha station. It was executed on the DEC Alpha with 10000 sets of random input data. The execution frequency information produced during these executions was stored.

6. In the last step we determined the execution times of the 10000 generated test cases for the MC68000 processor. From the knowledge of the execution times of the basic blocks obtained in Step 2 and the frequency information of Step 5 the execution times XT_i were computed:

$$XT_i = \sum_{j=1}^{N} frequency_{j,i} \times xt_j,$$

where N stands for the number of basic blocks, $frequency_{j,i}$ is the execution frequency of block j in the i-th execution, and xt_j is the execution time of block j on the MC68000. We used the mean value of the XT_i, $(1 \leq i \leq 10000)$ as an approximation of the average execution time.

The reason why we did not measure execution times for the sample executions on the MC68000 hardware but calculated them on the DEC Alphas is that the results of the calculations can be better compared to the bounds from static WCET analysis. For each instruction we used the same execution time in the execution time calculations as was used in WCET analysis. Thus the WCET analysis of a program that fully characterizes the behavior of the implemented algorithm and the execution of the program with worst-case inputs are guaranteed to yield the same (worst-case) execution time. If we compared the computed WCET with measured execution times that would not necessarily be the case. This is due to the pessimistic assumption WCET analysis makes about the execution times of instructions with variable execution times (see Section 2.5.1).

2.5. Derivation of WCETs

The WCETs of all scenarios were determined both analytically and by an execution of the sorting algorithms with input data that had been constructed to generate the worst case. A combined analytic and constructive assessment of WCETs is preferable to a pure experimental investigation for the following reasons:

- The execution of an algorithm with random input data cannot be guaranteed to generate the worst case (see Section 3).
- A systematic investigation of all possible execution times of an algorithm that would necessarily yield the WCET is infeasible. For an array of size n it would require $n!$ test runs to survey all permutations of elements.
- The generation of input data that force the program to execute on a worst-case path is nontrivial, unless the analyzed program is very simple.

The combined analytic and constructive investigation of WCETs allows us to bound the WCET of the sorting algorithms. The results of the static WCET analysis are pessimistic, i.e., WCET analysis yields upper bounds for the WCET. On the other hand, the execution time of the scenarios that had been constructed to generate the worst case (or at least a bad case) are lower bounds for the WCET. The difference of the two bounds serves as an indicator for the quality of both values. It is an upper bound on how much both values might miss the real WCET. A small difference indicates a good quality of the results, equality of the two bounds demonstrates that the real WCET has been found.

2.5.1. Static WCET Analysis

For the static WCET analysis of the sorting programs we used an approach that is based on the theory described in [11]. We performed the following steps for every configuration of the sorting algorithms:

1. We analyzed the algorithm to find out how often every part of the program is maximally executed in one execution of the sorting program. This analysis yielded a set of linear inequalities characterizing the possible maximum execution frequencies for both single program parts and mutually dependent program parts.

   To obtain a very detailed description, our analysis took into account the lazy evaluation of conditions in C. We investigated how often every part of a compound condition would be evaluated in the worst case.

2. We compiled the sorting program with the GNU GCC compiler and annotated the resulting assembler program with the knowledge about the program behavior we gained in Step 1. For the latter we used a simple annotation language which could be further processed with our tools (see next step).

3. We used the tool *ass2lp*, which had been developed at our department, to generate an integer linear programming problem for the computation

of the WCET bound from the annotated assembly language program. *ass2lp* reads an assembler program, looks up the execution time for each instruction, and builds a graph that represents the flow through the program, the execution times of the program parts, and the constraints on execution frequencies of the program parts. From the graph description it generates an ILP problem: The goal function reflects how much every program part contributes to the program execution time. The constraints of the programming problem describe both the structure of the program and the knowledge about possible execution frequencies which stems from the user's annotations.

4. The ILP problem was solved with *lp_solve*.[1] The solution of the problem yielded the WCET bound for the analyzed program and the execution frequency settings of all program parts for a worst-case execution. The information about the execution frequencies is valuable since it characterizes execution paths that are relevant for the WCET. In fact, this information was helpful for the generation of input data that forced a worst-case execution of some of the sorting programs (see below).

The *ass2lp* tool used in Step 3 takes the instruction code and the operand types into account to determine the execution time of an instruction. For the shift instructions, whose execution times depend on the number of bits to be shifted, *ass2lp* also takes into account shift counts of constant type. If the shift count is stored in a register, the execution time for the worst-case count is taken. For the other instructions with data-dependent execution times (bit manipulation instructions, multiplication, division) *ass2lp* uses the maximum execution time.

2.5.2. WCET Assessment with Constructed Scenarios

To test the quality of the computed WCET bounds we tried to construct input data that force programs to execute on worst-case paths. We used both our knowledge about the operation of the sorting algorithms and the information about the worst-case paths returned by the WCET analyzer to produce promising input data sets. Finally we executed the sorting programs with the generated input data. The corresponding execution times were obtained with the same simulation method we used to obtain execution times for the approximations of average execution times (see Section 2.4).

[1] *lp_solve* was developed by Michel Berkelaar. It can be retrieved from *ftp.es.elle.tue.nl* via anonymous FTP.

3. Results

This section presents the results of our experiments. We obtained approximate values for the average execution times from simulations of the program executions. Each sorting algorithm was executed with 10000 random input data sets for array sizes of 7, 10, 25, 50, 75, 100, 250, 500, 750, and 1000 elements.

Table 1. Average Execution Times

	7	10	25	50	75	100	250	500	1000
bubble	1.22	2.5	15	61	138	246	1537	6148	24592
insert.	1.00	1.8	10	35	77	134	816	3234	12876
select.	1.45	2.8	15	59	130	229	1358	5404	21558
merge	1.94	3.4	10	24	42	56	160	360	799
quick	1.37	2.1	6	14	23	33	96	213	468
heap	2.00	3.5	12	29	49	69	204	459	1017
radix	10.4	12.4	23	39	56	71	168	329	653
d500	31.9	32.2	34	37	40	43	59	87	144
d1000	62.9	63.2	65	68	71	74	90	118	175
d10e4	621	621	623	626	629	632	648	676	733

Table 1 shows the average execution times from our simulations. Since we are interested in the relation of the execution times and not so much in the actual values, execution times are given relative to the shortest execution time that occurred, the average execution time of insertion sort for seven elements, 2419 CPU cycles. (In the table *d500*, *d1000*, and *d10e4* stand for sorting by distribution counting for an input data range of 500, 1000, and 10000, respectively.)

Table 2. Worst-Case Execution Time Bounds

	7	10	25	50	75	100	250	500	1000
bubble	1.42	2.9	18	73	165	293	1833	7334	29343
insert.	1.45	2.8	17	66	146	259	1609	6421	25657
select.	1.47	2.8	16	60	133	235	1394	5553	22165
merge	2.05	3.6	11	25	43	58	166	371	821
quick	1.88	3.1	13	41	83	140	777	2963	11534
heap	2.68	4.8	18	45	77	111	338	777	1756
radix	10.4	12.4	23	39	56	71	168	329	653
d500	31.9	32.2	34	37	40	43	59	87	144
d1000	62.9	63.2	65	68	71	74	90	118	175
d10e4	621	621	623	626	629	632	648	676	733

Table 2 shows the upper bounds for the WCETs we derived analytically, again relative to the average execution time of insertion sort for seven elements. The programs had been analyzed for their behavior to get the maximum execution frequencies for each pro-

gram part. From the knowledge about the static control structure and the possible execution frequencies we generated ILP problems to compute the execution time bounds.

In order to certify that the computed WCET bounds are not too pessimistic, we compared the computed WCET bounds with the simulated worst-case execution times of selected input data. We tried to construct inputs that would generate the worst case or a scenario close to the worst case for each configuration of the sorting algorithms.

For all sorting algorithms, except quick sort and heap sort, worst-case inputs were easy to generate and the simulations yielded the same results as WCET analysis. Thus, for all algorithms, except quick sort and heap sort, WCET analysis yielded the exact maximum execution time, not just a bound. Note, that for radix sort and sorting by distribution counting the execution times are independent from the input data; the average and worst-case execution time are identical.

Table 3. Ratio between Calculated Execution Time Bounds and Maximum Measured Execution Times

	7	10	25	50	75	100	250	500	1000
quick	1.05	1.08	1.11	1.10	1.08	1.08	1.05	1.03	1.02
heap	1.07	1.06	1.04	1.03	1.02	1.02	1.02	1.01	1.01

The simulation results for quick sort and heap sort were always smaller than the values returned by WCET analysis. For 7 and 10 elements the simulation results are the real WCETs (we ran simulations with all permutations of the input data). For the larger input data sets the values obtained from the simulations form a lower bound for the WCETs, i.e., the real WCETs are somewhere between the simulated lower and the computed upper bound. Since the differences between simulated and computed values were relatively small (see Table 3), we considered these results good enough to use the computed (save) WCET bounds for our comparison of average and worst-case execution times.

4. Discussion of Results

4.1. Average Performance

Our results for the average execution times correlate with the data on the algorithmic complexity described in standard literature, e.g., [5, 12]. The average execution times of the three $O(N^2)$-algorithms, bubble sort, insertion sort, and selection sort, increase quadratically with the number of elements to be sorted. The execution times of the algorithms with complexity $O(N\mathrm{ld}N)$ grow faster than linearly. The execution times of radix sort and sorting by distribution counting is nearly linear with the number of elements.

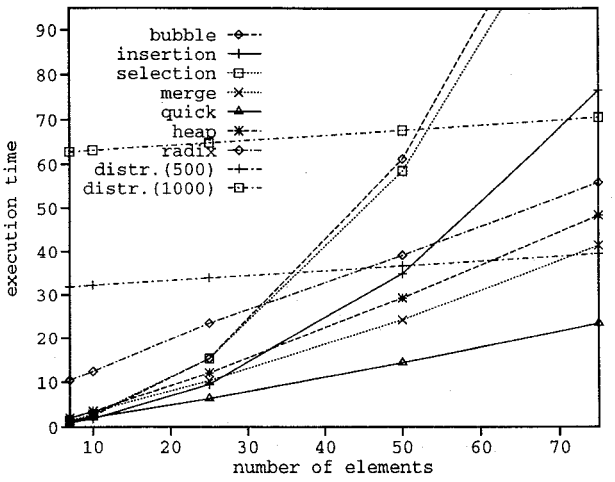

Figure 1. Average Execution Times

Figure 1 shows the average execution times of the sorting programs for 7, 10, 25, 50, and 75 elements. For sorting by distribution counting only the versions with input range 500 and 1000 are displayed. For an input range of 10000 the execution times are too high to be shown in the same figure.

The two linear algorithms have a higher constant overhead than the other algorithms. Their average execution time is higher than the execution times of the other algorithms for a small number of elements. Among the $O(N^2)$-algorithms insertion sort performs better than the other two sorting programs. For $N > 25$ Elements insertion sort is more than 60 percent faster than the two other algorithms. From the three $O(N\mathrm{ld}N)$ programs quick sort has the best average performance, followed by merge sort and heap sort.

Summarizing the results over the range of array sizes investigated we can observe: If high average performance is of central concern insertion sort performs best for a small number of elements, quick sort is fastest for medium size arrays (between 25 and 100 elements in our experiments), and distribution counting is best suited for large number of elements, provided the range of values to be sorted is small.

4.2. Worst-Case Performance

The WCETs of all three $O(N^2)$ algorithms bubble sort, insertion sort, and selection sort grow fastest with the number of elements. Among the other three non-linear programs quick sort shows the fastest growth of the WCET with an increasing number of elements to be sorted. While it is faster than merge sort and heap sort for seven and ten elements it is clearly outperformed by these algorithms for arrays of large sizes (from 75 elements on in our experiments). The worst-case behavior of the two linear algorithms equals their average behavior. For small arrays their WCETs are higher than the WCETs of the other algorithms and the execution time of radix sort increases faster with the number of elements than the execution time of distribution counting. For medium to large array sizes both linear algorithms show a better worst-case performance than all other algorithms (see Figure 2).

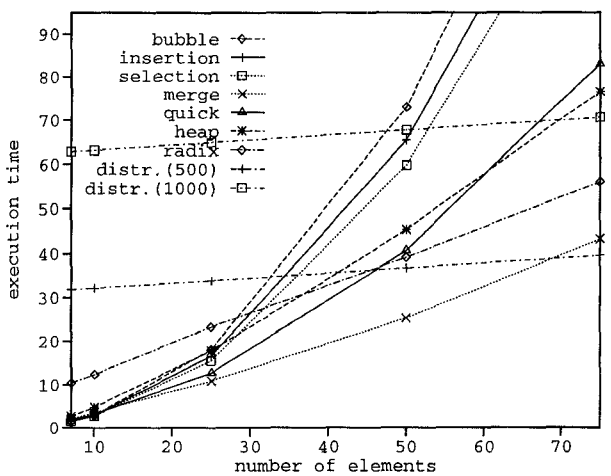

Figure 2. Worst-Case Execution Times

Note that the execution time of distribution counting heavily depends on the range of the input data. It increases much slower than for all other algorithms with the number of elements to be sorted. Therefore it performs better than all other algorithms from some array size on (To determine that size one has to regard the range of the input data).

As an overall rating of the algorithms with respect to WCETs we observe: For a small number of elements the simple $O(N^2)$ algorithms perform best. For a medium and high number of elements (more than 10 elements in our experiments) merge sort outperforms all other non-linear programs. If the range of the input data values is small, the number of elements to be sorted is high, and there is enough memory, radix sort or distribution counting might be good alternatives.

4.3. Comparison of Average and Worst Case

Figure 3 illustrates the comparison of the average and worst-case execution times of bubble sort, insertion sort, and selection sort. To support readability the curves of both the average execution time and the WCET of each algorithm are drawn in the same line style. In the plots of average execution times points are marked with crosses, in the WCET plots points are marked with triangles.

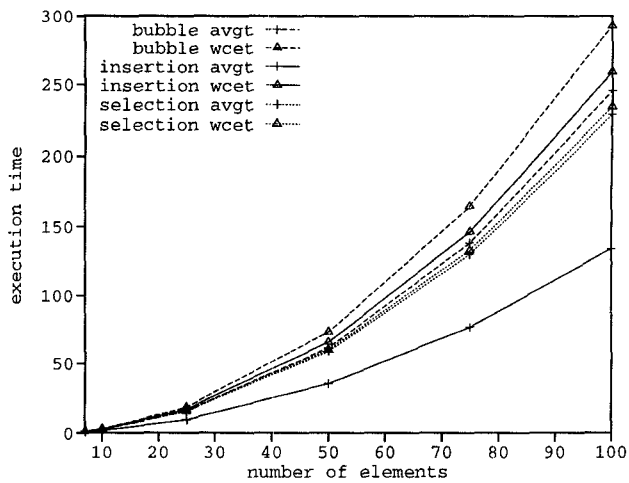

Figure 3. Average vs. Worst-Case Execution Times

We make the following observations. The WCET of selection sort is only little higher than its average execution time (1 percent for 7 elements to 2.8 percent for 500 elements). For bubble sort the difference lies between 16.5 (7 elements) and 19.3 percent (500 elements). Interesting is the timing of insertion sort. Its WCET is nearly two times the average execution time for a medium to a large number of elements (93 percent for 100 elements, 99 percent for 1000 elements). While it outperforms the other $O(N^2)$ algorithms on the average, it is inferior to selection sort with respect to the WCET.

Figure 4 shows the average and worst-case execution times of merge sort, heap sort, and quick sort. As before the average and worst-case curves of each algorithm are drawn in the same line style. Points are marked with crosses in average plots and with triangles in worst-case plots.

While quick sort performs with $O(N\mathrm{ld}N)$ on the average, the worst-case number of steps is proportional to N^2. Our results reflect that behavior in a dramatic increase of WCETs in comparison to average execution

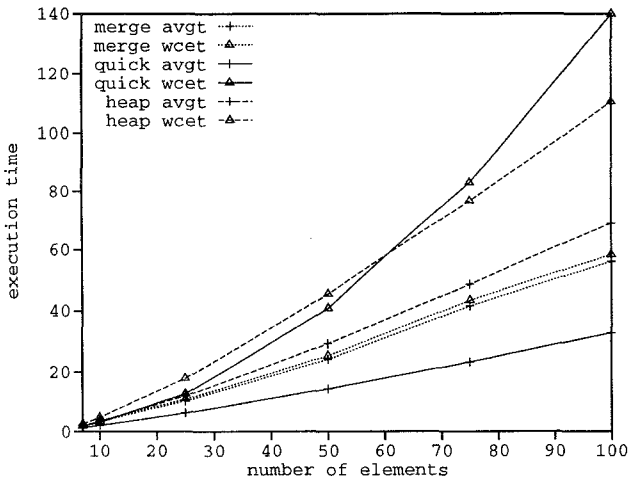

Figure 4. Average vs. Worst-Case Execution Times

times. Heap sort shows a similar behavior, however the gap between worst case and average is not so big as for quick sort. On the other hand, the execution characteristics of merge sort are comparable stable. The difference between worst-case and average execution time is small.[2] Merge sort has the best worst-case performance among the three $O(N\text{ld}N)$ programs, heap sort is second, and quick sort performs worst.

The execution times of radix sort and sorting by distribution counting are not included in the comparative figures. For those algorithms the average and worst-case execution times are identical. Their execution times are independent from the actual input data values. One has, however, to consider that these programs have a much higher memory consumption than the other sorting algorithms.

From the curves shown in Figures 3 and 4 we observe, that (except for small arrays) in both groups of programs, $O(N^2)$ and $O(N\text{ld}N)$, the programs with the best WCET are those with the smallest difference between worst-case and average execution time, selection sort and merge sort, respectively. Although one must not conclude that all programs with a small execution time variance have a short WCET, this observation is worth being discussed.

Traditional programming aims at a good average performance. Based on the knowledge about the distribution of input data, programmers write data-dependent code which performs good for those input data that occur frequently at the price of a bad performance for rare cases. Obviously this programming style neglects the performance of some cases and is inadequate for hard real-time programming. In hard real-time programming it is right the worst case that is crucial. In contradiction to traditional programming goals, real-time programming aims at an optimization of the worst case, even at the price of a degraded average performance. So the observation from Figures 3 and 4 may give us a clue for the selection of programs that are adequate for real-time applications: use programs with a small spread of possible execution times and avoid programs, which favor execution with certain data at the cost of a bad performance with other data.

5. Conclusion

In this paper we compared the average execution times, which is the standard performance measure for software programs, and the worst-case execution times, which is of relevance for hard real-time applications, of eight standard sorting algorithms — bubble sort, insertion sort, selection sort, merge sort, quick sort, heap sort, radix sort, and sorting by distribution counting. We demonstrated that algorithms with a good average performance are not necessarily suited for real-time systems, in which the worst case is of importance.

We showed that the simple $O(N^2)$ algorithms bubble sort, insertion sort, and selection sort perform good for a small number of elements, both in average and worst case. For a large number of elements their execution times are higher than the execution times of all other algorithms.

For the $O(N\text{ld}N)$ algorithms quick sort, heap sort, and merge sort we got the following result. Quick sort performs best if the average execution time is of interest, but worst with respect to the WCET. Merge sort has the most stable behavior. While its average performance lies between quick sort and heap sort, merge sort has the best worst-case performance.

For both radix sort and distribution counting the average execution time and the worst-case execution time are equal. The execution times of these algorithms only depend on the number of elements to be sorted and on the range of the input values. They are, however, independent of the actual input data values. The minimal,

[2] Notice that the gradient of the execution time curves of merge sort steadily increases in each interval between 7, 10, 25, 50, and 75 elements. The gradient between 75 and 100 is, however, lower than between 50 and 75 elements. The reason for that phenomenon is as follows: In each interval, except the interval [75, 100], the number of passes through the whole array increases by one (from size 2^i+1 on). This increase in the number of passes substantially influences the increase of the execution times and thus the gradient of the curves. Since there is no increase in the number of passes in the interval [75, 100], the gradient of that part of the curve is naturally lower.

constant overheads of both algorithms are higher than for all other algorithms. So they perform bad for a small number of elements. For a larger number of elements the execution time of radix sort increases faster than that of distribution counting. Both algorithms are characterized by high memory needs.

The results can be summarized in the following recommendation. If algorithms have to be judged for their worst-case performance, the $O(N^2)$ algorithms are best suited for a small number of elements to be sorted and merge sort for medium and large amounts of data. For both average and worst case, distribution counting is best suited for large amounts of data if the range of the input data is small and the additional memory consumption is not of concern.

In a comparison of average and worst-case performance for both $O(N^2)$ and $O(N\mathrm{ld}N)$ algorithms we observed that those algorithms whose execution paths (and times) were least dependent on input data (selection sort and merge sort) yielded the best WCET results. We think that this phenomenon can in general be helpful for the selection of algorithms that are suited for real-time applications.

Acknowledgements

This work has been supported by Digital Equipment Corporation under contract EERP/AU-038.

We are grateful to Emmerich Fuchs, Hermann Kopetz, and the anonymous reviewers for their valuable comments on earlier versions of this paper.

References

[1] G. E. Blelloch, C. E. Leierson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine cm-2. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, pages 3–16, 1991.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press and McGraw Hill, 1990.

[3] M. G. Harmon, T. P. Baker, and D. B. Whalley. A retargetable technique for predicting execution time. In *Proc. 13th Real-Time Systems Symposium*, pages 68–77, Phoenix, AZ, USA, Dec. 1992.

[4] R. M. Karp. A survey of parallel algorithms for shared-memory machines. Technical Report CSD-88-408, University of California Berkeley, 1988.

[5] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, MA, USA, 1973.

[6] Y. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proc. 16th Real-Time Systems Symposium*, pages 298–307, Dec. 1995.

[7] K. Mehlhorn. *Data Structures and Efficient Algorithms*, volume 1. Springer, EATCS Monographs, 1984.

[8] L. Motus and M. Rodd. *Timing Analysis of Real-Time Software.* Pergamon, Elsevier Science, 1994.

[9] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, 1993.

[10] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, 24(5):48–57, May 1991.

[11] P. Puschner and A. Schedl. Computing maximum task execution times — a graph-based approach. *Real-Time Systems*, to appear 1997.

[12] R. Sedgewick. *Algorithms.* Addison-Wesley, Reading, MA, USA, 1989.

[13] R. Sedgewick and P. Fajolet. *An Introduction to the Analysis of Algorithms.* Addison-Wesley, Reading, MA, USA, 1996.

[14] A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, SE-15(7):875–889, July 1989.

[15] J. A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *IEEE Computer*, 21(10):10–19, Oct. 1988.