Real-Time Performance of Sorting Algorithms *

PETER P. PUSCHNER
peter@vmars.tuwien.ac.at
Institut für Technische Informatik, Technische Universität Wien
A-1040 Wien, Austria

Abstract. In hard real-time systems tasks must meet their deadlines under guarantee. Soft real-time tasks may miss deadlines occasionally, as long as the entire system can provide the specified quality of service.

In this paper we investigate the hard and soft real-time performance of sorting algorithms and compare it to their average performance. We show in which way the adequacy of an algorithm depends on the demanded performance criterion (hard, soft, or non real-time). The results provide a guideline to select the right sorting algorithm for a given application.

Keywords: real-time systems, timing analysis, worst-case execution time, performance, algorithms, sorting

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 mainly the average number of operations or the average execution times of these algorithms have been investigated and compared.

While the average execution time of a program is suited as a performance measure for non real-time applications, different performance criteria apply for the construction of real-time systems: Hard real-time tasks must meet their deadlines under guarantee; soft real-time tasks must meet their deadlines most of the time, i.e., only a given, limited fraction of executions may miss the deadline. This paper investigates the worst-case (hard real-time) and soft real-time 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 have been broadly discussed in literature (Mehlhorn, 1984, Knuth, 1973, Aho, Hopcroft, and Ullman, 1983, Sedgewick, 1989, Sedgewick, 1996, Cormen, Leiserson, and Rivest, 1990). These works describe the properties of different sorting strategies, discuss their memory consumption, and characterize the number of operations needed for each algorithm. With respect to the number of operations the focus is on average performance, while the worst-case scenarios are only shortly presented. Lately those performance figures were complemented with performance evaluations of sorting algorithms on parallel computers, e.g., (Karp, 1988).

* This work was supported in part by Digital Equipment Corporation under contract EERP/AU-038.
Blelloch et al., 1991]. To this date there exists, however, no work that explicitly
investigates and compares the worst-case execution times (WCETs) and soft real-
time (SRT) performance of sorting algorithms. In literature, WCET computations
for real (sample) programs were only used to demonstrate the quality of prototype
WCET analysis tools (Park and Shaw, 1991, Harmon, Baker, and Whalley, 1992,
Park, 1993, Li, Malik, and Wolfe, 1995). The purpose of this paper is to fill the
gap: it investigates the WCETs and the SRT performance 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 one 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 (Puschner, 1996) and experimentally by construction of worst-case in-
put data for each sorting algorithm. To assess the SRT performance we determined
soft execution time bounds, for short soft bounds. Soft execution time bounds are
execution time bounds that the algorithms may exceed with a specified probability.
Like average execution times, soft bounds were determined experimentally.

The paper is structured as follows. Section 2 describes our experiments in detail.
Section 3 presents the average execution times, the WCET bounds, and the soft
bounds of the analyzed sorting algorithms for data sets of different size. 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 algo-
rithms that have been compared and describes the hardware/compiler configuration
used for the evaluation. In the second part we explain, how we derived approxima-
tions for the average execution times and both worst-case execution times and soft
bounds 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 (Mehlhorn, 1984, Sedgewick, 1996). 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 and 1000 different keys.
2.2. Experimental Setup

We used simulations to derive the average execution times and soft bounds 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 01. 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. Since we do not use parallel hardware and since the data to be sorted cannot be predicted, the logical control structure of the sorting programs, determining the numbers and sequences of compare and swap operations, remains unchanged by optimizations. Thus, 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 between one and 1000 elements. As a measure for the average execution times we used the mean value of the execution times of 100000 executions of the sorting algorithms with random permutations (see Section 2.4). WCETs were derived both analytically and experimentally for each configuration (Section 2.5). As SRT bounds we took the value of the order statistics of the sample at the 95\% quantile assuming that 1 deadline miss in every 1000 executions was allowed (Section 2.6).

2.4. Average Execution Times

In our experiments 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 100000 sets of random permutations. The execution frequency information produced during these executions was stored.

6. In the last step we determined the execution times of the 100000 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 $X T_i$ were computed:

$$XT_i = \sum_{j=1}^{N} \text{frequency}_{j,i} \times xt_j,$$

where $N$ stands for the number of basic blocks, $\text{frequency}_{j,i}$ 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 $XT_i$, $(1 \leq i \leq 100000)$ 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 non-trivial, 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 (Puschner, 1996). 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 \textit{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. \textit{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. 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).

2.6. Soft Bounds

To establish soft bounds for the considered algorithms we had to answer the question, how much time we have to allocate for each sorting algorithm for n elements if a deadline miss rate $\mu$ is acceptable, i.e., on the average 1 out of $1/\mu$ executions need more time to complete than the time allocated. Ideally one would derive the length of such a time interval by constructing the cumulative distribution function (CDF) of its execution times. The execution time at which the CDF reaches $1 - \mu$ would be the time to be reserved, i.e., the soft bound (see Figure 1).

Since the construction of the CDFs is difficult for the complex algorithms like merge sort, quick sort, and heap sort, we used approximate values as soft real-time bounds. We used a statistic test for quantiles of continuous distributions (DeGroot, 1986) for our estimation: From the 100000 execution time samples obtained from our random permutations (see Steps 1 to 6 in Section 2.4) we computed a soft real-time bound with confidence coefficient 0.95 as the value of the
order statistics of the execution time samples at the 0.95 quantile of the binomial distribution for \( n = 100000 \) trials and probability \( p = 1 - \mu \). The index of the element of the order statistics which bounds the 0.95 quantile of the binomial distribution was computed with the help of the Mathematica software package (Wolfram Research, 1992).

In our experiments we assumed that one in 1000 executions is allowed to miss its deadline. For \( \mu = 0.001 \) the index of the 0.95 quantile of the binomial distribution for \( n = 100000 \) and \( p = 1 - \mu \) is \( 99916 \). Therefore, the \( 99916^{th} \) execution time sample of the order statistics was taken as approximate value for the soft real-time bound of each investigated scenario.

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 \( 100000 \) random input data sets for array sizes in the range between one and 1000 elements.

Table 1 shows selected 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 in the table,
Table 1. Average Execution Times

<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubble</td>
<td>1.10</td>
<td>4.26</td>
<td>26.5</td>
<td>106.2</td>
<td>239.1</td>
<td>425.8</td>
<td>2658.8</td>
<td>10675.5</td>
<td>42554.7</td>
</tr>
<tr>
<td>insertion</td>
<td>1.00</td>
<td>3.18</td>
<td>16.4</td>
<td>60.6</td>
<td>132.6</td>
<td>232.1</td>
<td>1412.5</td>
<td>5506.9</td>
<td>22292.4</td>
</tr>
<tr>
<td>selection</td>
<td>1.40</td>
<td>4.77</td>
<td>26.5</td>
<td>101.5</td>
<td>224.8</td>
<td>396.4</td>
<td>2440.4</td>
<td>9719.6</td>
<td>38797.4</td>
</tr>
<tr>
<td>merge</td>
<td>2.52</td>
<td>5.88</td>
<td>17.9</td>
<td>41.9</td>
<td>71.8</td>
<td>96.8</td>
<td>277.6</td>
<td>623.1</td>
<td>1382.7</td>
</tr>
<tr>
<td>quick</td>
<td>1.62</td>
<td>3.61</td>
<td>10.8</td>
<td>24.9</td>
<td>40.4</td>
<td>56.7</td>
<td>265.7</td>
<td>398.3</td>
<td>810.9</td>
</tr>
<tr>
<td>heap</td>
<td>2.15</td>
<td>6.02</td>
<td>20.9</td>
<td>59.8</td>
<td>83.9</td>
<td>119.8</td>
<td>353.6</td>
<td>787.7</td>
<td>1735.8</td>
</tr>
<tr>
<td>mdix</td>
<td>16.59</td>
<td>21.43</td>
<td>40.4</td>
<td>67.9</td>
<td>96.9</td>
<td>123.4</td>
<td>280.1</td>
<td>560.1</td>
<td>1130.1</td>
</tr>
<tr>
<td>d500</td>
<td>54.74</td>
<td>55.73</td>
<td>58.7</td>
<td>63.6</td>
<td>68.6</td>
<td>73.5</td>
<td>92.4</td>
<td>151.1</td>
<td>248.3</td>
</tr>
<tr>
<td>d1000</td>
<td>168.39</td>
<td>169.37</td>
<td>112.3</td>
<td>117.3</td>
<td>122.2</td>
<td>127.1</td>
<td>156.1</td>
<td>204.7</td>
<td>362.0</td>
</tr>
</tbody>
</table>

The average execution time of insertion sort for five elements, 1398 CPU cycles. (In the table d500 and d1000 stand for sorting by distribution counting for an input data range of 500 and 1000, respectively.)

Table 2. Worst-Case Execution Time Bounds

<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubble</td>
<td>1.26</td>
<td>5.00</td>
<td>31.5</td>
<td>126.4</td>
<td>284.7</td>
<td>506.6</td>
<td>3170.8</td>
<td>12680.0</td>
<td>50773.3</td>
</tr>
<tr>
<td>insertion</td>
<td>1.35</td>
<td>4.92</td>
<td>28.9</td>
<td>133.3</td>
<td>253.1</td>
<td>448.4</td>
<td>2784.0</td>
<td>11111.6</td>
<td>44397.7</td>
</tr>
<tr>
<td>selection</td>
<td>1.41</td>
<td>4.83</td>
<td>27.0</td>
<td>103.6</td>
<td>229.8</td>
<td>405.7</td>
<td>2903.0</td>
<td>9968.5</td>
<td>39786.8</td>
</tr>
<tr>
<td>merge</td>
<td>2.69</td>
<td>6.26</td>
<td>18.9</td>
<td>43.9</td>
<td>74.8</td>
<td>106.8</td>
<td>237.0</td>
<td>641.9</td>
<td>1430.2</td>
</tr>
<tr>
<td>quick</td>
<td>2.96</td>
<td>5.37</td>
<td>22.1</td>
<td>70.5</td>
<td>143.6</td>
<td>242.2</td>
<td>1344.8</td>
<td>5127.0</td>
<td>19857.3</td>
</tr>
<tr>
<td>heap</td>
<td>2.86</td>
<td>8.30</td>
<td>31.0</td>
<td>78.6</td>
<td>132.5</td>
<td>191.2</td>
<td>585.0</td>
<td>1344.5</td>
<td>3039.2</td>
</tr>
<tr>
<td>mdix</td>
<td>16.59</td>
<td>21.43</td>
<td>40.4</td>
<td>67.9</td>
<td>96.9</td>
<td>123.4</td>
<td>290.1</td>
<td>590.1</td>
<td>1130.1</td>
</tr>
<tr>
<td>d500</td>
<td>54.74</td>
<td>55.73</td>
<td>58.7</td>
<td>63.6</td>
<td>68.6</td>
<td>73.5</td>
<td>92.4</td>
<td>151.1</td>
<td>248.3</td>
</tr>
<tr>
<td>d1000</td>
<td>168.39</td>
<td>169.37</td>
<td>112.3</td>
<td>117.3</td>
<td>122.2</td>
<td>127.1</td>
<td>156.1</td>
<td>204.7</td>
<td>362.0</td>
</tr>
</tbody>
</table>

Table 2 shows selected upper bounds for the WCETs we derived analytically, again relative to the average execution time of insertion sort for five elements. The programs had been analyzed for their behavior to get the maximum execution frequencies for each program 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](image)

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

<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick</td>
<td>1.04</td>
<td>1.08</td>
<td>1.11</td>
<td>1.10</td>
<td>1.08</td>
<td>1.08</td>
<td>1.05</td>
<td>1.03</td>
<td>1.02</td>
</tr>
<tr>
<td>heap</td>
<td>1.08</td>
<td>1.06</td>
<td>1.04</td>
<td>1.03</td>
<td>1.02</td>
<td>1.02</td>
<td>1.02</td>
<td>1.01</td>
<td>1.01</td>
</tr>
</tbody>
</table>

The simulation results for quick sort and heap sort were always smaller than the values returned by WCET analysis. For 5 and 10 elements the simulation results are the real WCETs (1.99 resp. 5.03 for quick sort and 2.64 resp. 7.80 for heap sort) obtained from simulations with all possible permutations of the input data. For the larger input data sets the values acquired 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 and soft bounds.

![Table 4](image)

Table 4. Soft Bounds ($\mu = 0.001$)

<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubble</td>
<td>1.26</td>
<td>4.81</td>
<td>28.7</td>
<td>12.3</td>
<td>250.2</td>
<td>442.5</td>
<td>2727.3</td>
<td>10830.2</td>
<td>43105.4</td>
</tr>
<tr>
<td>insertion</td>
<td>1.35</td>
<td>4.53</td>
<td>22.3</td>
<td>76.9</td>
<td>162.7</td>
<td>277.8</td>
<td>1504.9</td>
<td>6133.7</td>
<td>23752.9</td>
</tr>
<tr>
<td>selection</td>
<td>1.41</td>
<td>4.81</td>
<td>20.6</td>
<td>101.7</td>
<td>225.0</td>
<td>396.7</td>
<td>2441.1</td>
<td>9711.9</td>
<td>38739.6</td>
</tr>
<tr>
<td>merge</td>
<td>2.69</td>
<td>6.22</td>
<td>18.5</td>
<td>42.9</td>
<td>73.0</td>
<td>98.3</td>
<td>279.6</td>
<td>636.0</td>
<td>1366.9</td>
</tr>
<tr>
<td>quick</td>
<td>1.99</td>
<td>4.62</td>
<td>13.1</td>
<td>29.8</td>
<td>47.7</td>
<td>66.9</td>
<td>191.7</td>
<td>420.3</td>
<td>912.2</td>
</tr>
<tr>
<td>heap</td>
<td>2.64</td>
<td>7.32</td>
<td>23.9</td>
<td>55.6</td>
<td>90.3</td>
<td>126.9</td>
<td>366.5</td>
<td>812.7</td>
<td>1785.9</td>
</tr>
<tr>
<td>radix</td>
<td>16.59</td>
<td>21.43</td>
<td>40.4</td>
<td>67.9</td>
<td>96.9</td>
<td>123.4</td>
<td>250.1</td>
<td>509.1</td>
<td>1130.1</td>
</tr>
<tr>
<td>d500</td>
<td>54.74</td>
<td>55.73</td>
<td>58.7</td>
<td>63.6</td>
<td>68.6</td>
<td>73.5</td>
<td>92.4</td>
<td>151.1</td>
<td>248.3</td>
</tr>
<tr>
<td>d1000</td>
<td>168.39</td>
<td>169.37</td>
<td>112.3</td>
<td>117.3</td>
<td>122.2</td>
<td>127.1</td>
<td>156.1</td>
<td>204.7</td>
<td>302.0</td>
</tr>
</tbody>
</table>

Table 4 presents the soft bounds for the selected scenarios from the tables above. Again, the times obtained from our experiments are scaled relative to the average execution time of insertion sort for five elements.
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., (Knuth, 1973, Sedgewick, 1988). 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 $O(N \log N)$ grow faster than linearly. The execution times of radix sort and sorting by distribution counting is nearly linear with the number of elements.

![Figure 2. Average Execution Times](image)

Figure 2 shows the average execution times of the sorting programs in the range between 5 and 75 elements. For sorting by distribution counting only the version with input range 500 is displayed. The curve for a range of 1000 would be parallel to that curve.

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 $O(N \log N)$ programs quick sort has the best average performance, followed by merge sort and heap sort.
The reader may have noticed the steps in the execution time plot for merge sort. They always occur between array sizes of $2^i$ and $2^i + 1$. The reason for these steps is that for every $2^i + 1$ array elements the number of sorting passes through the whole array increases by one. This increase in the number of passes substantially influences the increase of the execution times and thus causes the steps.

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 five 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 3 and Table 2).

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 21 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. **Soft Bounds**

The figures for the soft bounds show an interesting development: For very small numbers of elements to be sorted the soft bounds of the sorting algorithms are equal to the WCETs (see WCETs and soft bounds for five elements). From some array size on, the soft bounds get smaller than the WCETs, and for medium to large array sizes the curves of the soft bounds rather resemble the curves for the average execution times than the WCET curves (see Figure 4). The reason for this
The phenomenon is as follows: for small array sizes only a small number (less than $1/\mu$) of possible input permutations exist. Since for such a small number of different inputs the deadline miss for every possible permutation would imply a violation of the miss rate, the soft bound must be greater or equal than the execution times for all possible inputs. Thus it equals the WCET. For larger arrays the number of possible input permutations exceeds $1/\mu$ and the permissible miss rate makes possible to ignore the small number of permutations with extremely high execution times (see also the sample CDF in Figure 1). The resulting soft bounds are closer to the average execution times than to WCETs for medium to large array sizes.

Figure 4 shows the soft bounds of quick sort for different array sizes and different miss rates. The smaller the value $\mu$ the larger arrays must be for soft bounds to become smaller than the WCETs. On the other hand, the trend of the soft bounds is similar for all $\mu$ for growing array sizes. The higher the number of elements to be sorted the smaller gets the difference between soft bound and average execution time relative to the difference between WCET and soft bound.

4.4. Comparison of Worst Case, Soft Bounds, and Average Case

Figure 5 illustrates the comparison of the average, worst-case execution times, and soft bounds of bubble sort, insertion sort, and selection sort. To support readability the same line style is used for the three curves of each algorithm. In the plots of average execution times some selected points are marked with crosses, in the WCET
plots selected points are marked with triangles. The plots for soft bounds use lines without points.

We make the following observations. The WCET of selection sort is only little higher than its average execution time (0.7 percent for 5 elements to 2.7 percent for 500 elements). For bubble sort the difference lies between 15.0 (5 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 higher for 100 elements, 99 percent higher 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. For each of the three algorithms the soft bound increases slightly faster than its average execution time. The soft bound of selection sort is so close to its average execution time that the respective curves seem to coincide in Figure 5.

Figure 6 shows the average, worst-case execution times and soft bounds of merge sort, heap sort, and quick sort. As before the average, worst-case, and soft-bound curves of each algorithm are drawn in the same line style. Again, selected points are marked with crosses in average plots and with triangles in worst-case plots. The plots for soft bounds use lines only, no points.

While quick sort performs with $O(N\log N)$ 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 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. Merge sort has the best worst-case performance among the three $O(N \log N)$ programs, heap sort is second, and quick sort performs worst.

Similar to the $O(N^2)$ algorithms, the soft bounds of the three $O(N \log N)$ algorithms rather resemble the average execution times than the WCETs. More generally, we want to emphasize that at least for medium and large $N$ the comparison of the soft bounds of the considered sorting algorithms yields the same results as the comparison of the average execution times. This has the practical consequence that, as a rule of thumb, one can draw on his/her knowledge about the average performance of sorting algorithms when selecting an algorithm for a soft real-time application.

The execution times of radix sort and sorting by distribution counting are not included in the comparative figures. For those algorithms the average execution times, soft bounds, 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 5 and 6 we observe, that (except for small arrays) in both groups of programs, $O(N^2)$ and $O(N \log N)$, 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 5 and 6 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, soft bounds, and the worst-case execution times, which are of relevance for 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, under each of the three performance criteria. For a large number of elements their execution times are higher than the execution times of all other algorithms.

For the $O(N \log N)$ algorithms quick sort, heap sort, and merge sort we get the following result. Quick sort performs best if the average execution time or the soft real-time performance is of interest, but worst with respect to the WCET. Merge sort has the most stable behavior. While its average and soft real-time performance are 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, soft bound, 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, 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. As for soft bounds, for small amounts of data the soft real-time performance of the algorithms is equal or close to the worst case. For medium to large amounts of data, soft bounds get closer to average execution times than to WCETs and as a rule of thumb knowledge about average execution times can be used to judge the soft real-time performance of sorting algorithms for these array sizes. For all, from 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 $O(N \log N)$ 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 hard real-time applications.

Acknowledgments

The author would like to thank Dietmar Mittermair for his help in computing the WCET bounds for the sorting algorithms. He is grateful to Emmeric Fuchs and Hermann Kopetz for their valuable comments on earlier versions of this paper.
Notes

1. Especially for large arrays, data caching influences the execution time of a sorting program. Investigating the effects of different cache types and data access patterns of the sorting algorithms on the performance might thus yield interesting results. An evaluation of the effects of caching on the performance of sorting algorithms is, however, beyond the scope of this paper.

2. *lp_solve* was developed by Michel Berkelaar. It can be retrieved from ftp.es.ele.tue.nl via anonymous FTP.

References


