An Evaluation of Constant Execution Time Sorting Algorithms

Kristian Slabbekoorn (1228196)

Delft University of Technology
k.slabbekoorn@tudelft.nl

Abstract. Constant execution time (CET) programming approaches are known to significantly reduce execution time variability for specific algorithms, but how different types of CET algorithms perform in relation to their time-optimized versions and each other is not immediately apparent. This paper evaluates the performance of CET programming on four common sorting algorithms. We find that performance overhead incurred from CET design transformations is largely dependent on algorithm complexity and that there exists an inverse relation between this complexity and performance on small input sizes – simple, computationally inefficient algorithms are observed to outperform more sophisticated algorithms with tighter upper bounds on time complexity.

1. Introduction

Along with the advancement of computing technology, our lives are becoming increasingly reliant on embedded real-time systems – from smartphones to GPS navigation systems to pacemakers. An essential property for real-time systems is timeliness; processes have deadlines before which they need to finish. If this deadline is unexpectedly missed, the consequences can be severe, depending on the application. It is therefore critical for programs running on embedded systems to be temporally predictable. This prediction is usually performed through best-case execution time (BCET) analysis and particularly worst-case execution time (WCET) analysis. There are, however, some limitations to these approaches, and making an accurate WCET analysis can be a difficult task. WCET analysis is a research domain of itself, and over the years, many approaches for it have been developed [1].

One particular approach of increasing the predictability of real-time systems is by applying WCET-oriented programming in the development stage of the application [2]. With this paradigm, applications are programmed in a way that minimizes the possible execution paths. Optimizations in the code, such as a break out of an iteration of an array once a certain condition is met, are removed, assuring that every time this piece of code is traversed, it will perform the same or equivalent operations and take the same time to execute. Performance is therefore sacrificed to increase predictability. The extent to which performance is impeded differs per algorithm, and while the theoretical worst-case, best-case and average-case running times are usually known, it is not directly clear how constant time versions of these algorithms perform

with regard to their originals: there is a need to explore the relations that exist between execution time, algorithm code complexity, code transformations applied and input sizes on which the algorithms are run.

This paper is structured as follows. In section two, we provide background information on time analysis and highlight some existing work on constant execution time design. Next, in section three, we describe how CET programming works and explain some alterations we made to existing methods. In section four we describe the experiments performed, as well as our evaluation of the results. We conclude the paper with a short summary and some suggestions for possible future work in section five.

2. Time analysis of real-time applications

There are several methods available to predict the execution times of programs. With real-time systems we usually focus on WCET analysis, since finishing before a deadline is often a critical factor in embedded real-time systems. With WCET analysis, we try to find an upper bound on the execution time of a procedure or application that is as close as possible to the *actual* worst-case execution time. There are two main established ways of doing this.

2.1 Static WCET analysis

With static WCET analysis we determine the structure of a program by analyzing its source code or assembly code. By calculating and summing the maximum possible cost of each specific code branch or procedure we can find out what the theoretic longest possible running time can be, and take this as an upper bound for our WCET. Drawbacks to this approach is that this process is cumbersome to perform by hand for large systems and too complex to fully automate by covering each and every possible state of the program. What further complicates it is that compilers may perform unexpected optimizations, or that hardware may be unpredictable (e.g. due to having multiple processor cores, caching, etc.).

2.2 Dynamic WCET analysis

A different approach is the dynamic analysis of an application. An application is tested by running a variety of inputs in order to cover each possible or relevant computation path, either exhaustively or by means of some kind of heuristic. For complex applications, this is a very costly procedure, since there could be an astronomical number of possible computation paths if the program contains many conditionals.

2.3 WCET-oriented programming approach

The approach we take in this paper is the WCET-oriented approach. Rather than analyzing an existing program, this approach makes sure that the program under analysis is already easy to predict. This is done by minimizing the amount of possible execution paths for a program to follow in the programming stage, or by transforming existing source code into a form that always executes at a constant time. A drawback to this approach is that significant overhead is incurred since the resultant programming code is inherently inefficient in the time dimension.

2.4 Related work

There has not been a lot of work on WCET-oriented programming in the academic community. The paradigm was originally devised by Puschner [1]. His work provides some rules to transform traditional program code into their single-path equivalents. These rules were further generalized and presented in a step-wise approach by Horn Lopes in [4]. By following the CET design approach lined out in his paper, it is possible to convert common conditional programming principles to structures that assure constant time traversal. In this paper we build upon the work of Horn Lopes by using these principles to convert sorting algorithms to their CET equivalents.

3. Principles of constant time programming

Before we can start creating algorithms that run in constant time, it is essential for a methodology to be devised that describes in robust, structured steps how to convert standard code to CET code. As mentioned, one such methodology is presented in [4], which builds on the work of Puschner and incorporates techniques such as if-conversion [2]. We will make use of this methodology, albeit with a few modifications to suit our specific needs.

3.1 Constant execution time design

CET design breaks up the conversion process in three steps. First, statements that disrupt the program flow such as continue and break statements are removed. Next, while statements are made constant by iterating them a predetermined number of times. This is done because the number of iterations can be dependent on the input size. For our purposes, however, we are not interested in retaining constant time for different input sizes. It is arguable, in fact, whether this is relevant for WCET analysis at all, as often the input sizes are predetermined, or execution time increases predictably relative to the input size. We *do* convert while statements to iterate a constant number of times if they contain a conditional that can lead to variable execution times, for instance when there is a break from the loop once a certain element in an array has been found.

In the third and last step, a five-step sequence is provided for the conversion of if-statements: nested if-statements are combined, conjunctions of conditions are evaluated separately, else clauses get new if statements with negated conditions, values calculated in the body get calculated before the if statement and saved in temporary variables, and for each assignment statement in the body, equivalent assignments are made in alternative paths. Missing from this sequence, however, is the case of a series of "if, else-if, else" clauses, i.e. a string of conditionals of which only one is supposed to execute. We handle this as shown in figure 1. An explanation of the transformation is as follows.

Since we need to make sure that all conditionals are evaluated, they are moved in front of the if-else-if sequence. The results of the evaluations are saved to temporary variables. The same is done to values calculated in the body of each conditional. Next, each `if`, `else if` or `else` clause is transformed into an `if` clause containing as conditional the intersection of the truth-value of its own original condition (as saved in the temporary variable) with negated truth-values of the remaining conditions. The final else statement will then be the intersection of the negation of all conditions before it. The bodies are filled as described in the fifth step of the original sequence in [4]. This eliminates any variability in the execution time, albeit at the cost of wieldier code.

```
                                t1 = c1; t2 = c2;
                                t3 = c3;
                                v1 = f1(); v2 = f2();
if c1 && c2:                    v3 = f3();
  a1 = f1();                    if (t1 && t2) && !t3:
else if c3:          ->           a1 = v1; a2 = a2;
  a2 = f2();                      a3 = a3;
else:                           if !(t1 && t2) && t3:
  a3 = f3();                      a1 = a1; a2 = v2;
                                  a3 = a3;
                                if !(t1 && t2) && !t3:
                                  a1 = a1; a2 = a2;
                                  a3 = v3;
```

Fig. 1. Transforming an if-else if-else sequence.

Based on this preliminary work, we have chosen four sorting algorithms to convert to CET versions. The experiments and their evaluations are described in the next section.

4. Experiment on sorting algorithms

In this section we describe the four algorithms analyzed and the experiments performed. We explain the sorting algorithms we selected for this experiment, describe our experiment setup and input types used, and finally, we list the results obtained and provide our evaluation of these results.

4.1 Selected algorithms

We have selected four algorithms to analyze from three classes of sorting algorithms to get a varied selection of sorting techniques:

- *Exchange sort class*: Bubble sort
- *Selection sort class*: Selection sort
- *Insertion sort class*: Insertion sort, shell sort

Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order.

Selection sort finds the minimum value in the list, swaps it with the value in the first position, then repeats these two steps for the rest of the list (each time incrementing the starting index by one).

Insertion sort removes an element from the input data every iteration, then inserts it into the correct position in a newly built list until no input elements remain.

Shell sort is a generalization of insertion sort, improving on it by allowing the comparison and exchange of elements that are far apart [3]. It does this by drawing upon a sequence of positive integers, known as the *gap sequence*. Since the gap sequence [1,2,4,8,16,...,2^k] originally devised by Shell has been found to be far from optimal, we use the optimal sequence as discovered by Ciura [5].

We have chosen to omit well-known powerful algorithms such as merge sort and quick sort from this comparison, as recursion is an integral part of their implementation. Recursive statements cannot be converted to constant time equivalents easily. It would therefore be necessary to convert the recursive versions of these algorithms to iterative ones, which would in turn increase their time complexity and nullify what made them efficient in the first place. This is a significant limitation of the CET design approach in its current form.

An overview of the known theoretical worst-case, best-case and average-case performance for each of the selected algorithms is shown in table 1 [6]. By knowing these we can make meaningful comparisons to CET versions of the sorts.

	Worst-case	Best-case	Average-case
Bubble sort	$O(n^2)$	$O(n)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n)$	$O(n^2)$
Shell sort	$O(n\log^2 n)$ (Ciura's gap sequence)	$O(n)$	$O(n\log^2 n)$ (Ciura's gap sequence)

Table 1. Known theoretical time complexities for the selected sorting algorithms.

4.2 Experimental conditions

We performed the execution time analysis in C on an Intel® Xeon® CPU W3530 at 2.80GHz running SuSe Linux. The source code was compiled with gcc without

compiler optimizations. However, even when explicitly telling gcc not to perform optimizations with the –O0 option, some elementary optimizations are still performed. These optimizations are minor, however, and the discrepancies produced are not deemed significant for our interests.

We ran both the original and the CET versions of the algorithms, each time measuring the time elapsed in microseconds using C's built-in `gettimeofday()` function. As input we used three types of lists:

- Randomly generated integers (to simulate an average-case input)
- Integers sorted in reverse order (to simulate a worst-case input)
- Integers sorted in the proper order (to simulate a best-case input)

For each of these types, we prepared input arrays of 100, 1,000 and 10,000 units in length. We ran each type twenty times for each algorithm with each input size and recorded their running times by taking the average. It should be noted that sorted and reverse-sorted lists are best-case and worst-case inputs respectively for bubble sort and insertion sort, but this is not the case for selection sort and shell sort.

4.3 Experiment results and evaluation

In this section we will list the results of our experiments. Table 2 and 3 show the results obtained for the original and CET algorithms respectively.

Comparing table 2 with table 1, we see that the running times of the simulated best, average and worst cases indeed increase at roughly the expected rate relative to the input size. Only the best-case bubble sort is not $O(n)$ as expected; this is due to the bubble sort algorithm we implemented not being optimal (every extra n will cause another iteration of the full list). For selection sort and shell sort, sorted and reversed lists can be approximately treated as average-cases as well.

	Bubble sort			*Selection sort*		
Input size	**100**	**1000**	**10000**	**100**	**1000**	**10000**
Sorted	18,4	1841,7	181004,3	17,9	1744,0	164618,8
Random	36,1	4092,2	417824,6	21,3	1789,6	165490,7
Reversed	37,6	3662,8	370473,3	19,3	1772,2	175002,2
	Insertion sort			*Shell sort*		
Input size	**100**	**1000**	**10000**	**100**	**1000**	**10000**
Sorted	0,8	6,6	66,3	2,7	44,6	616,0
Random	12,0	1098,1	110336,1	8,5	132,3	1989,5
Reversed	24,4	2211,0	220071,7	5,1	68,1	987,5

Table 2. Execution times in microseconds for the original, time-optimized sorting algorithms.

	Bubble sort			*Selection sort*		
Input size	**100**	**1000**	**10000**	**100**	**1000**	**10000**
Sorted	42,9	4304,5	448599,3	41,6	4145,8	393921,0
Random	43,0	4307,1	448469,3	48,8	4138,6	395630,1
Reversed	42,9	4304,2	448291,2	43,9	4001,6	394195,1
	Insertion sort			*Shell sort*		
Input size	**100**	**1000**	**10000**	**100**	**1000**	**10000**
Sorted	115,2	11522,9	1161756,0	84,2	1400,0	14663,2
Random	114,4	11437,6	1143096,0	86,6	1371,6	15791,1
Reversed	109,7	10924,5	1088955,0	83,7	1308,5	14930,1

Table 3. Execution times in microseconds for the CET versions of the algorithms.

If we subsequently look at table 3 for the CET versions of the algorithms, and compare them with their originals in table 2, we can see some interesting results. In line with previous work on CET design, variability of execution times between different inputs of the same size is minimal. The trade-off is that, for every input type, the time complexity witnessed is that of the worst-case listed in table 1.

Aside from this worst-case time complexity, there is a static amount of overhead incurred from the inherently time-inefficient code produced by CET design. This is different per algorithm and is related to the amount of unnecessary operations that need to be performed on top of the optimal ones. Based on the average-case with n = 10000, CET bubble sort has 105.9% the original running time, selection sort 239%, insertion sort 1036% (roughly an order of magnitude) and shell sort 794%. There likely exists a correlation between overhead and code complexity of any given algorithm, but more research is needed to see exactly how CET transformations translate to overhead increase.

The most interesting result we see is that, while shell sort has the lowest worst-case time complexity and performs best on all input sizes in its optimized form, the CET version performs second-worst on an input size of 100. Despite this, it performs best again even out of the CET algorithms on input sizes 1000 and 10000. This seems solidify the theory that overhead increases with code complexity, as shell sort is the most complex algorithm under test (to give an idea, the original fit in 16 lines of code, yet the CET version required 78 – nearly 5 times the original amount). Additionally, the simplest algorithm under test, bubble sort, performs the worst in all cases originally yet is the best performer out of the CET algorithms on small inputs. There is likely to be an inverse relation between running time and code complexity at small input sizes, since more complex code creates more overhead, but further work is needed to find out how this relation can be formally expressed. Furthermore, it is unclear where the fine line lies where certain algorithms stop being efficient compared to less sophisticated algorithms given their complexity and input size.

5. Summary and future work

In this paper we adopted the CET design paradigm described in [4], made some minor alterations to it and applied it to four common sorting algorithms. We then evaluated the performance of these newly created CET algorithms in relation to their time-optimized originals. We found that this performance is not easy to predict, and that there exists some relation between the complexity of an algorithm and the amount of overhead incurred by making CET transformations. This overhead can cause sophisticated algorithms to underperform simple, time complexity-wise inefficient algorithms when input sizes are sufficiently small.

Points of further research in this field have been hinted at throughout the paper. One thing that is still unclear is how recursion can be efficiently transformed to a form that runs in constant time. Which CET design transformations cause how much overhead under what circumstances is another open question. Lastly, a formal relation between code complexity, input size and overhead needs to be discovered so that a "break-even" point can be found where complex algorithms stop being efficient compared to less sophisticated algorithms for a given input size.

6. References

[1] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., and Stenstrom, P. The worst-case execution-time problem—Overview of methods and survey of tools, ACM Trans. Embed. Comput. Syst. 7, 3, 1–53, 2008

[2] Puschner, P. The single-path approach towards WCET-analysable software, 2003 IEEE International Conference on Industrial Technology, 2003

[3] Shell, D.L. A high-speed sorting procedure, Communications of the ACM 2 (7): 30–32, 1959

[4] Horn Lopes, J. Improving Temporal Predictability by Applying Constant Execution Time Design, Delft University of Technology Real-time Systems course 2009/2010, 2010

[5] Ciura, M. Best Increments for the Average Case of Shellsort, Proceedings of the 13th International Symposium on Fundamentals of Computation Theory, pp: 106—117, 2001

[6] http://en.wikipedia.org/wiki/Sorting_algorithm