Algorithms for Dependable Hard Real-Time Systems *
(Extended Abstract)

Peter Puschner
Institut für Technische Informatik
Technische Universität Wien
A1040 Wien, Austria
Email: peter@vmars.tuwien.ac.at

Abstract

Hard real-time code is special purpose code whose worst-case performance (worst-case execution time \( \ldots \) WCET) needs to be good and easy to predict. Despite these specific demands on the temporal properties of hard real-time code, real-time programmers often use the same algorithms and programming techniques that have proven effective for non real-time applications.

This paper explains the different temporal requirements imposed on real-time resp. non real-time code, and outlines why traditional (non real-time) programming tends to produce code that (a) has a high WCET and (b) is hard to analyse for its WCET. Based on these observations the paper proposes an unconventional programming strategy that avoids the shortcomings of traditional coding and yields code that is well-suited for hard real-time systems, i.e., its WCET is short and predictable. The evaluation of a number of examples demonstrates the advantages of the proposed programming strategy on WCET and its predictability.

1 Introduction

The Worst-Case Execution Time (WCET) is one of the central parameters characterizing a piece of hard real-time code. Typically, the program code for dependable time-critical systems needs to have a short WCET. Further the code is required to be WCET-analyzable, i.e., a safe and tight WCET bound must be computable.

Despite these special demands imposed on the temporal properties of (hard) real-time code, no dedicated discipline for developing real-time code has evolved to this date. Still, real-time programmers use the same algorithms and programming techniques that have proven to be effective for non real-time applications.

---

\*This work has been supported by the IST research project “High-Confidence Architecture for Distributed Control Applications (NEXT TTA)” under contract IST-2001-32111.
In this paper we compare the different performance requirements imposed on real-time and non real-time code. The speed optimization for the most probable (i.e., frequent) execution scenarios is the primary goal in non real-time applications. (Hard) real-time programming, in contrast, focuses on producing code with good and easy-to-analyse worst-case timing. These diverse performance and analyzability requirements typically favour different solutions to functional problems in real-time respectively non real-time contexts, e.g., a functionally correct piece of code that performs well in a non real-time context may not meet the performance criteria for a hard real-time application, and vice versa. As a consequence, real-time program development must use different strategies than conventional (average) performance-oriented programming. The typical optimization pattern used in non real-time programming relies on input-data dependent decisions to achieve good performance. We analyze why this strategy is disadvantageous for obtaining code with a good and analyzable WCET. As a solution, we propose an alternative, WCET-oriented programming approach.

2 Non Real-Time versus Real-Time Programming

The performance requirements that are imposed on non real-time respectively real-time code are usually quite diverse. In a system that does not operate in a time-critical context, a high throughput is the central performance goal. To achieve a high throughput the typical (average) execution time of each task has to be short.

In a (hard) real-time system, in contrast, the average execution time of a task is of minor importance. What is crucial is that all time-critical tasks meet their deadlines under all anticipated circumstances, even under peak load. In order to be able to guarantee the latter and still keep the resource needs reasonable, real-time tasks need to have a short WCET.

More formally the different performance requirements are described as follows. Assume that we have to write code that solves a problem for a space of possible input data \( \Delta = \{ \delta_i \} \) on a machine \( \mu \). The corresponding set \( P = \{ p_i \} \) denotes the probabilities with which each of the input-data scenarios occurs, i.e., \( \delta_i \) occurs with probability \( p_i \). Further assume that \( \Pi = \{ \pi_i \} \) represents the (infinite) set of correct code fragments or programs that solve the given problem and the function \( c(\pi_j, \delta_i, \mu) \) returns the execution time of \( \pi_j \) with input data \( \delta_i \) on \( \mu \). Then, non real-time and real-time programming aim at finding different solutions \( \pi_{avg}^\mu \) resp. \( \pi_{wcet}^\mu \) that yield the best result according to the following diverse optimization criteria:

\[
\pi_{avg}^\mu \in \{ \pi_s \mid \forall \pi_j \in \Pi : \sum_{\delta_i \in \Delta} p_i c(\pi_s, \delta_i, \mu) \leq \sum_{\delta_i \in \Delta} p_i c(\pi_j, \delta_i, \mu) \}
\]

\[
\pi_{wcet}^\mu \in \{ \pi_s \mid \forall \pi_j \in \Pi : \max_{\delta_i \in \Delta} c(\pi_s, \delta_i, \mu) \leq \max_{\delta_i \in \Delta} c(\pi_j, \delta_i, \mu) \}
\]

Note that, in contrast to (2), the selection criterion for \( \pi_{avg}^\mu \) pays only limited attention to the execution times of individual input-data scenarios. It is the majority of execution scenarios that determines the outcome of (1). In particular, a WCET scenario that occurs with very low probability has only a very small influence on \( \pi_{avg}^\mu \), even if its WCET value is much larger than the typical execution time.

In the following we will discuss the diverse coding strategies for \( \pi_i \) that yield good solutions for the two different programming domains. In particular we will demonstrate in which way real-time code
typically has to look different from code that aims at a good average performance. As for the hardware \( \mu \), we assume that the execution time of a piece of code increases monotonically with the number of instructions being executed. Otherwise, we do not make any assumption about the hardware architecture and its features.

2.1 Traditional Performance-Oriented Coding

Non real-time programmers typically aim at a good average performance to allow for a high throughput. Therefore, the primary performance goal of non real-time programmers is the speed optimization for the most probable (i.e., frequent) scenarios, see (1). In order to be able to favor the frequent cases, the code tests the properties of input-data sets and chooses the actions to be performed during an execution based on input data.

Using input-data dependent control decisions is an effective way to achieve short execution times for the favored input-data sets. This approach is therefore suitable for optimizing the average execution time. In contrast to this, a programming style that is based on input-data dependent control decisions adversely affects the quality of the achievable WCET. This is due to the following reasons:

- **Tests to identify the current input data**: Even if an input-data set is not among the “favored” inputs, it has to be tested at the points where the control flow between favored and non-favored inputs splits. While the fast code makes up for the cost of the control decisions in the case of favored inputs, the execution time of the input-data tests add up to the execution time without compensation for all other data.

- **Branching costs**: Similarly to the previous argument, not just the costs for testing the properties of input data but also the costs for branching to the respective code sections increase the total execution time of non-favored cases.

- **Information-theoretical imbalance**: Every functionality on a defined input-data space and available data memory has a specific complexity. The overall problem complexity determines the number and types of operations needed to solve the problem for the given input-data space. Performance-oriented, non real-time programming spreads this overall complexity unevenly over the input-data scenarios: to facilitate a high throughput, the frequent input-data scenarios are treated at computational cost that are below the complexity that would result if the total complexity would be evenly distributed to all scenarios. As the complexity inherent to a problem is constant, a cost reduction for some part of the input-data space necessarily causes higher costs for the rest of the inputs. Again, this impairs the achievable WCET (example: average versus worst-case transmission time of a string coded in Huffman code respectively a constant-length code).

Data-dependent control decisions are the results of traditional performance-optimization patterns. The strategy followed in such optimizations – looking for solutions that have the shortest expected completion time – seems to be quite similar to the optimization patterns we use in every-day life. In the following, we show that we have to apply a completely different and not so common optimization strategy if we aim at optimizing the worst-case completion time.
2.2 Programming for the Worst Case

As shown in the previous section, traditional (non real-time) programming tends to produce code that has a high WCET. We observe that it is the different treatment of scenarios, i.e., favouring certain input-data sets over others, that causes an increased WCET. The reasons for this were detailed above. In order to write code that has a good WCET the shortcomings of the traditional programming style have to be avoided. A novel programming strategy is needed.

WCET-oriented programming (i.e., programming that aims at generating code with a good WCET) tries to produce code that is free from input-data dependent control flow decisions or, if this cannot be completely achieved, restricts operations that are only executed for a subset of the input-data space to a minimum.

Note that in some applications it is impossible to treat all inputs identically. This can be due to the inherent semantics of the given problem or the limitations of the programming language used. WCET-oriented programming needs a way of thinking that is quite different from the solution strategies we normally use. As a consequence, it produces unconventional algorithms that may not look straightforward at the first sight. The resulting pieces of code, however, are characterized by competitive WCETs due to the small number of tests (and branches) on input data and the minimal information-theoretical imbalance.

A small number of input-data dependent alternatives does not only keep the WCET down. It also keeps the total number of different execution paths through a piece of code low. Identifying and characterizing a smaller number of paths for WCET analysis is easier and therefore much less error-prone than dealing with a huge number of alternatives. In this way, WCET-oriented programming does not only produce code with better WCET performance but also yields more dependable WCET-analysis results and thus more dependable real-time code than traditional programming.

3 Experiments

The table shows the average and worst-case execution times (in number of CPU cycles) for two different versions of each of three different programs: bubble sort, find-first (finds the first occurrence of a key in an array), and bin-search (binary search for a data item in an array). The first version of each of the programs is coded for good average-case performance (traditional), the second version has been programmed according to the worst-case programming style. Array sizes are 9 elements for bubble sort and 10 elements for the other programs.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>traditional AVG</th>
<th>WCET</th>
<th>WCET-oriented AVG</th>
<th>WCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubble</td>
<td>599</td>
<td>724</td>
<td>609</td>
<td>663</td>
</tr>
<tr>
<td>find-first</td>
<td>68</td>
<td>122</td>
<td>103</td>
<td>103</td>
</tr>
<tr>
<td>bin-search</td>
<td>94</td>
<td>124</td>
<td>105</td>
<td>106</td>
</tr>
</tbody>
</table>

For all three programs the traditional, average-case performance oriented version outperforms its competitor in the average execution time. On the other hand, the versions that have been programmed for a good worst-case performance are indeed superior with respect to WCET.
4 Summary and Conclusion

In this extended abstract we showed that hard real-time programming, that aims at code with a short WCET, needs a programming style that is completely different from traditional, average-performance oriented programming. Performance-oriented programming benefits from input-data dependent decisions to determine the control flow during code execution and reduce the execution time of selected scenarios. In contrast, Programming for the Worst Case tries to treat all cases of input data equally and avoids control-flow decisions based on input data wherever possible. This programming style yields code with a WCET that is short and predictable and facilitates a highly dependable WCET analysis. The different execution-time characteristics of traditional and WCET-oriented programming were demonstrated in a number of experiments.