Avoiding Timing Problems in Real-Time Software [*]

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

Abstract

To safely establish the correct timing of a real-time processing node, adequate architectural structures have to be used. This refers to the hardware architecture of the processing node as well as the software architecture of its operating system and application software.

This paper presents architectures that allow for a well structured and simple timing analysis. First, it presents solutions for cleanly splitting the overall timing analysis into schedulability analysis and task worst-case execution time analysis. Second, it presents a programming strategy that yields software that is highly temporally predictable and easy to analyze for its worst-case execution time.

1 Introduction

Real-time computer systems consist of a single processing node or a number of cooperating processing nodes that interact with the environment. In this interaction with the environment the processing nodes have to obey the timing constraints imposed by the application. Only if the system meets all timing requirements the correct and safe operation of the entire real-time system can be guaranteed.

In this paper we explain how the correct timing of a single processing node of a real-time system can be established. In our argumentation we follow the widely-used strategy to separate the overall timing analysis into two parts, schedulability analysis and task execution-time analysis. First we discuss the problems that have to be considered when separating the two subproblems. Second, we propose a task programming strategy that yields tasks that are highly temporally predictable and easy to analyse for their worst-case execution time (WCET). This paper does not deal with scheduling techniques and schedulability analysis, because these topics have gained broad coverage elsewhere, see, e.g., [15, 9].

Section 2 discusses the classical separation of timing analysis into schedulability analysis and task timing analysis. Section 3 discusses hardware and operating system design considerations to allow for a clear and simple separation of the two steps of the analysis. Section 4 proposes a programming strategy that is well suited for writing temporally predictable and WCET-analysable code.

[*]This work has been supported by the IST research project “High-Confidence Architecture for Distributed Control Applications (NEXT TTA)” under contract IST-2001-32111.

2 Structured Timing Analysis

To reduce complexity of real-time systems, programming is usually split into two levels: Programming in the large decomposes the overall application into smaller components, e.g., modules and modes of a system and the respective tasks, and defines the relationships and interaction between the tasks. Programming in the small focuses on the programming of the well-defined task components that have been separated out during the programming-in-the-large decomposition.

To establish the correct timing of a real-time program, each of the two programming levels needs to be supported by adequate techniques for evaluating the temporal properties of the artefacts it created. At the programming-in-the-small level WCET analysis computes a bound c_i for the maximum time comsumption of each task T_i.

At the programming-in-the-large level schedulability tests are used to test whether the selected set of tasks $\{T_i\}$ can be scheduled on the system. To test for example whether such a task set can be scheduled by rate monotonic scheduling [8], the utilisation factor μ of the system has to meet the following test:

$$\mu = \sum_{i=1}^{n} c_i/p_i \leq n(2^{1/n} - 1))$$

where n denotes the number of tasks and p_i the period of task T_i.

Without taking care it can be difficult to establish a structured timing analysis based on these two levels. On modern processors that use pipelines and caches, the execution of a certain piece of code leaves its footprints in the internal state of the processor. The temporal execution behaviour of code subsequently executed depends on these footprints, i.e., the timing of different executions varies. This implies that it becomes potentially impossible to calculate a safe and precise bound for the maximum time consumption of each task T_i in isolation of the overall system configuration. A detailed discussion of properties in conventional systems that prevent a precise timing prediction of a single task at programming-in-the-small level is given by Schneider [14]. To weaken this limitation, Schneider proposes to use static configuration parameters (e.g., number of tasks and their memory mapping) that are fixed for a particular application. For the remaining challenges in execution time analysis Schneider proposes to develop a comprehensive WCET- and schedulability-analysis approach. The drawback of this approach is that timing analysis becomes quite complex and without support for structured analysis.

It is an interesting topic of research to find more systematic ways to restrict the system design so that a higher predictability is achieved. A well structured system would then allow us to apply a structured timing analysis at a reasonable cost. In the following we describe research achievements in this area.

3 Separating the Timing-Analysis Steps

The use of caches in real-time systems in general makes the prediction of the WCET of tasks quite complicated. Therefore, Hand argued that real-time applications must be run with the cache disabled when determining the guaranteed level of performance offered by the system [3].

An approach to improve the overall predictability of a system is to make caches more predictable. Kirk et al. proposed a hardware based partitioning of caches [4, 5]. In this scheme, each task is assigned an equal-sized portion of the cache. A further portion of the cache is allocated for data shared between tasks. Using hardware partitioned caches prevents the modification of the cache partition of a task due to preemptions. The drawback of this approach is that the partition size is fixed and specific hardware is required. Another constraint is that the overall memory allocation of a task is determined by the relative size of its assigned cache portion.

Mueller proposed a software based cache partitioning technique with compiler support [10]. The compiler rearranges the code so that each task fits into a unique set of cache lines. Larger blocks are split by inserting unconditional jumps. It remains to evaluate the overall performance impact of this approach via quantitative analysis.

An approach for software-based cache partitioning performed at the operating system level is described by Liedtke et al. [7]. Their approach assigns a cache partition to a certain task via the translation of virtual memory. This requires no hardware modification of processors or caches. The approach can therefore only be applied to a cache outside the processor (e.g., second level cache). Liedtke et al. use a technique they call *free coloring* to relax the dependence between allocated memory and relative size of cache partition. *Free coloring* requires to change the wiring of the processor board.

An alternative strategy to improve the predictability is to replace the cache by more predictable mechanisms. Cogswell et al. have described the Multiple Active Context System (MACS) that avoids the necessity of caches by managing multiple task contexts by the processor pipeline [2]. The processor executes a different task on every cycle by repeatedly rotating through all task contexts. The delay of memory accesses of one context thus overlaps with execution cycles of the other contexts. Only extensive accesses to the same memory bank degrade performance.

Another technique to avoid the necessity of caches, called *threaded prefetching* is described by Lee et al. [6]. They use instruction prefetching instead of caching. Each block is assigned a designated prefetching block which is calculated by the compiler via analysis of the worst-case execution path.

To conclude, the support for structured timing analysis is preferred to reduce the complexity of WCET analysis. The whole system design has to be done carefully to achieve this structural reduction of complexity in analysis.

4 Obtaining WCET-Analysable Code

We demand from WCET analysis to produce reliable and tight estimates of the worst-case execution times of the tasks. Besides this, it is important that WCET analysis is simple, i.e., it must be both easy to apply and its results must be easy to trace and understand.

WCET analysis is not simple per se. On the contrary, without the use of an appropriate programming strategy, one can run into one of the severe problems of WCET analysis [12]. We therefore propose a task programming strategy that yields tasks that are highly temporally predictable and easy to analyze for their WCET.

4.1 WCET-Oriented Programming

We observe that today real-time programs, like most non real-time programs, are often designed and coded to achieve a good average performance to allow for a high throughput. The primary performance goal of such a programming strategy is the speed optimization for the most probable scenarios. In order to be able to favour the frequent cases, the code tests the properties of input-data sets (inclusive any data that store a state between different activations of code) and chooses the actions to be performed during and execution based on these input data.

Using input-data dependent control decisions is an effective way to achieve short execution times for favoured input-data sets. This approach is therefore well suited for optimizing the average case. On the other hand, a programming style that is based on input-data dependent control decisions adversely affects the quality of the achievable WCET. The code optimizations that favour some inputs come at the cost of a substantially higher execution time for the other inputs. Further, the time it takes to execute the input-data dependent control decisions adds up to the total execution time. While the fast code executed for favoured inputs makes up for this additional time, the durations of these tests increase the execution time without compensation for all other situations. As stated above, this increases the WCET.

In order to write code that has a good WCET, the shortcomings of the traditional, performance-oriented programming style and algorithms have to be avoided. A programming strategy for real-time code must use a completely different coding approach. We call this coding approach WCET-oriented programming, see [12]:

> *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, it restricts operations that are only executed for a subset of the input-data space to a minimum.

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. In these situations the programmer has to try to keep input-data dependencies local to small portions of the code.

WCET-oriented programming produces unconventional algorithms that may not look straightforward at the first sight. The resulting pieces of code, however, are characterized by competitive WCETs. Besides keeping the WCET down, WCET-oriented programming 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.

4.2 Generating Single-Path Code

The WCET-oriented programming approach provides a programming strategy that aims at avoiding input-data dependent code or at least keeping input-data dependencies local to a limited number of operations. Still, WCET-oriented code cannot be guaranteed to be free of input-data dependencies. On the contrary, we have to be prepared that WCET-oriented code will in general not get along entirely without input-data dependent control flow. To eliminate the remaining input-data dependent branches and thus reduce the complexity of WCET analysis, we conceived the single-path programming paradigm [13]. The following paragraphs give a summary on single-path programming.

As mentioned before, one reason for the complexity of WCET analysis is that different input data cause the code to execute on different execution paths with differing execution times. The single-path approach avoids this complexity by ensuring that the code has only a single execution path. This approach uses code transformations to transform input-data dependent loops and branches [11]. It transforms loops with input-data dependent termination conditions into loops with invariable iteration counts. Input-data dependent branches with the semantics of *if* or *case* statements and their alternatives are transformed into strictly sequential code. To be precise, the code resulting from the transformation of branches avoids data dependencies in execution times by keeping input-data dependent branching local to single operations with data-independent execution times.

The conversion of *if* or *case* statements into sequential code is called if-conversion [1]. The sequential code generated by the above-mentioned loop transformation and if-conversion uses so-called predicated operations, i.e., operations that implement branches within single machine instructions. A number of modern microprocessors (e.g., Alpha, IA-64, Motorola M-Core, Pentium P6) realize predicated execution to allow compilers to generate code that avoids conditional branch instructions and thus potential pipeline stalls. The idea of predicated execution is that instructions are associated with predicates. An instruction is only executed if its predicate evaluates to true. If the predicate evaluates to false the microprocessor internally replaces the instruction by a no-operation (NOP) instruction.

In summary, the combination of WCET-oriented programming and the single-path conversion produces code that is well suited for real-time systems: WCET-oriented programming yields highly competitive WCETs. Due to the single-path conversion the execution time of the code is constant and therefore fully predictable. Because there is only a single path it is possible to obtain the WCET in a single measurement run. The latter allows us to get around the problems that are inherent to static WCET analysis.

5 Summary and Conclusion

In this paper we discussed the problem of building real-time processing nodes whose timing is predictable and can be easily assessed. We observed that it is highly desirable that the correct timing of a processing node can be established in two clearly separated steps: WCET analysis assesses the timing of single tasks and schedulability analysis tests the correct timing of the whole task set of the node using the results of WCET analysis.

A cleanly structured timing analysis has to be supported by an adequate choice of hardware and operating system. It is important that the hardware and operating-system mechanisms are laid out such that task preemptions and run-time decisions taken in single tasks do not have side effects on the execution time of the other tasks. Different approaches that aim at independent task timing have been introduced.

In the second part of the paper we focused on task development. The WCET-oriented programming strategy allows programmers to write code with good worst-case timing. The single-path transformation of WCET-oriented code further yields code with only a single execution path. This code has a constant and therefore fully predictable execution time, which equals its WCET. Because there is only one possible path to choose, the WCET of the code can be safely obtained by measuring the execution time of the code with an arbitrary set of input data. This implies that no static analysis is needed and there is no danger of running into the problems of static WCET analysis.

References

[1] J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In *Proc. 10th ACM Symposium on Principles of Programming Languages*, pages 177–189, Jan. 1983.

[2] B. Cogswell and Z. Segall. Macs: A predictable architecture for real time systems. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 296–305, 1991.

[3] T. Hand. Real-Time Systems Need Predictability. *Computer Design (RISC Supplement)*, pages 57–79, Aug. 1989.

[4] D. B. Kirk. Smart (strategic memory allocation for real-time) cache design. In *Proc. 10th Real-Time Systems Symposium*, pages 229–237, Santa Monica, CA, USA, Dec. 1989.

[5] D. B. Kirk and J. K. Strosnider. Smart (strategic memory allocation for real-time) cache design using the mips r3000. In *Proc. 11th Real-Time Systems Symposium*, pages 322–330, Lake Buena Vista, Florida, USA, Dec. 1990.

[6] M. Lee, S. Min, C. Park, Y. Bae, H. Shin, and C. Kim. A Dual-mode Instruction Prefetch Scheme for Improved Worst Case and Average Case Program Execution Times. In *Proc. 14th Real-Time Systems Symposium*, pages 98–105, 1993.

[7] J. Liedtke, H. Hartig, and M. Hohmuth. OS-Controlled Cache Predictability for Real-Time Systems. In *Proc. 3rd IEEE Real-Time Technology and Applications Symposium*, pages 213–223, June 1997.

[8] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[9] J. W. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[10] F. Mueller. Compiler support for software-based cache partitioning. In *Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 137–145, La Jolla, CA, USA, June 1995.

[11] P. Puschner. Transforming execution-time boundable code into temporally predictable code. In B. Kleinjohann, K. K. Kim, L. Kleinjohann, and A. Rettberg, editors, *Design and Analysis of Distributed Embedded Systems*, pages 163–172. Kluwer Academic Publishers, 2002. IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002).

[12] P. Puschner. Algorithms for dependable hard real-time systems. In *Proc. 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Jan. 2003.

[13] P. Puschner and A. Burns. Writing Temporally Predictable Code. In *Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 85–91, Jan. 2002.

[14] J. Schneider. Why You Can't Analyze RTOSs without Considering Applications and Vice Versa. In *Proc. of the 2nd International Workshop on Worst-Case Execution Time Analysis (WCET 2002)*, June 2002.

[15] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*. Kluwer Academic Publishers, 1998.