Operating Mode Specific WCET Analysis*

Philipp Lucas, Oleg Parshin, Reinhard Wilhelm
Compiler Design Lab
Universität des Saarlandes
66123 Saarbrücken, Germany
[phlucas|oleg|wilhelm]@cs.uni-sb.de

Abstract

Many embedded control systems work in different operating modes, for example start-up, stand-by, shut-down and failure mode. These different operating modes usually have different timing requirements, and the different functional behaviour also leads to differences in timing behaviour. A Worst-Case Execution Time (WCET) analysis of such a system needs to determine mode-specific bounds on execution times, because a single overall bound may be too pessimistic. Mode determination and mode-specific analysis also form a prerequisite for analysing system behaviour during mode transitions, a most critical phase of a system.

The operating modes of embedded control systems are often not precisely represented, neither in hand-written code nor in code synthesised by model-based design tools nor on the model level. In this paper, we outline the use of operating modes for WCET analysis. Furthermore, we describe ongoing work on semi-automatically deducing mode information from C source code and using that information in WCET analysis.

Keywords: *operating modes, timing analysis, WCET, embedded systems, hard real-time*

1. Introduction

A safe estimation of a task's *Worst-Case Execution Time (WCET)* is necessary to verify that hard real-time applications meet their deadlines [8]. The actual execution time of a task depends on a variety of factors:

(1) *Task inputs* determine the sequences of executed instructions and of memory accesses;

(2) *dynamic hardware properties* such as cache content and pipeline state determine the hardware execution trace (e. g., cache hit or miss, speculation results);

(3) *static hardware properties* such as speed of execution units determine the time for the actual execution.

Tools which statically bound Worst-Case Execution Times in general take specific static system properties as given (they fix (3)) and compute the worst-case over the dynamically possible states (over (1) and (2)). Partitioning task inputs into classes restricts (1), leading to a more accurate WCET estimation [1].

The subsets of inputs we are considering here are those corresponding to *operating modes* of a software. Operating modes can be considered on several levels. Modes on the task set level require different functionalities or certain functionality implemented by *alternative tasks*. These modes give rise to different scheduling decisions or resource allocations, but they also determine the behaviour of and may pose specific requirements on each single task. We are considering this behaviour, which is not always specified explicitly: We strive to *statically extract* operating modes of a single task, to distinguish those modes which are *important* for WCET analysis, and to compute *mode-specific WCET bounds* by specialising the analysis to the execution context of each mode.

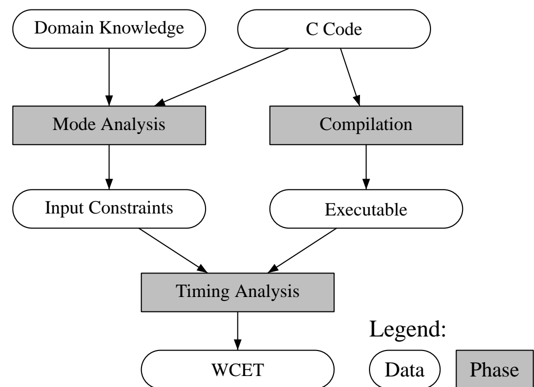


Figure 1. Operating modes in timing analysis.

Figure 1 shows how to fit mode analysis into a general WCET analysis framework. For the mode analysis, modes are mutually exclusive sets of input variable value ranges leading to specific system behaviour.

The remainder of this paper briefly introduces operating modes (Section 2) and our mode analysis tool under

*Supported by the European Network of Excellence *ArtistDesign* and the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement number 216008 (*Predator*).

development (Section 3), gives an overview of related approaches (Section 4) and outlines future work (Section 5).

2. Operating Modes

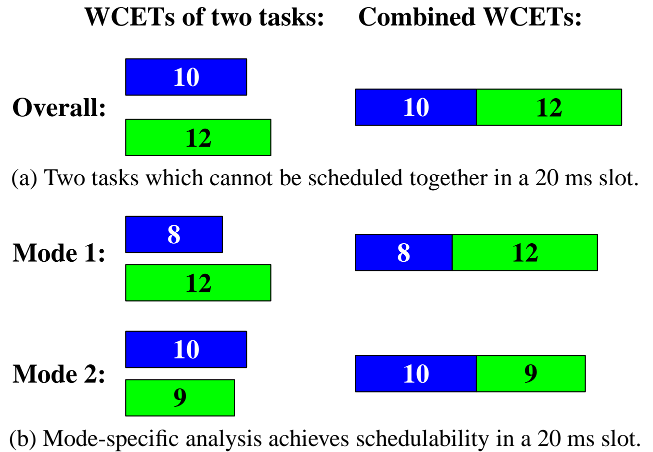
In the most general sense, we speak of *operating modes* when a software can exhibit different behaviour under different circumstances, if these behaviours are mutually exclusive and if they are determined at runtime. For example, an error mode is triggered by reading a sensor variable: If the variable denoting sensor input is in the range $[0, \infty)$, the software operates in normal mode; if the variable has the value -1, the software operates in error mode. Thus, an operating mode is defined by a *constraint on input variable values*.

An operating mode as such is a *logical* concept which does not necessarily relate to timing behaviour, but can relate to resource usage, signal flow, or even very subtle differences in behaviour. In this work, we strive to semi-automatically derive modes from C code in a general sense, but with focus on relevance for timing analysis: We approximate logical modes and identify those which exhibit significant variations in timing behaviour.

One usage for modes in timing analysis is *within a single task*: Different pieces of code are executed depending on the operating mode. Mutual exclusivity of modes translates into specific subsets of code executed per mode, and thus a WCET analysis can give *different* WCET bounds specific to modes, in contrast to a conservative overall WCET bound.

This is useful and even necessary because tasks also may have different timing *requirements* in different situations. Consider a task which controls fuel injection of an engine. Because it is triggered by the crankshaft sensor, the deadlines are stricter at higher RPMs. Over a certain RPM threshold, the task has to switch to a computationally less demanding mode. If only one overall WCET bound were considered, then the task would become formally unschedulable under these circumstances. Thus, the task has a WCET motivated mode distinction which needs to be taken into account in the timing analysis phase to ensure schedulability.

Another use of modes for timing analysis comes from considering the combined mode-specific behaviour of a set of tasks. If one task needs to perform more computations in start-up mode (e.g., initialisation of some system), and another one in run-time mode (e.g., monitoring some parameters), then their overall WCETs are mutually exclusive. Combining these WCETs would unnecessarily constrain schedulability analysis if the tasks run on the same processor (Figure 2). Mode-specific WCETs allow to ensure schedulability in the depicted case. Also, mode-specific access patterns to global resources may be considered to gauge the possible interferences. Generally, modes of tasks do not occur in arbitrary combinations, but form a *global mode* determining the behaviour of the complete system.



(a) Two tasks which cannot be scheduled together in a 20 ms slot.

(b) Mode-specific analysis achieves schedulability in a 20 ms slot.

Figure 2. Operating modes

3. Heuristical Mode Analysis

Here we explain the basic ideas behind a prospective mode-analysis tool designed for incorporation into a general WCET analysis framework. Conceptually, modes are derived by identifying *patterns* in the code hinting at mode-dependent behaviour, by establishing the *conditions* for the derived modes, and by *clustering* modes according to the desired properties (e.g., differences in WCET).

The analysis works on C code instead of binary level for two reasons. For one, the modes resulting from the analysis shall approximate natural notions of operating modes. A mode derived from the binary level that would not be *found* on source code level or that is not easily *expressible* in terms of input variables is unlikely to be of much use. Also, variable naming conventions or user annotation rely on debug information or the availability of the source code. Although mode analysis on an even higher level such as ASCET or Simulink is also useful [4], we focus here on C code to cover the vast amount of hand-coded legacy applications. For synergies between Simulink analysis and mode analysis, see [9], which also considers modes as mutually exclusive control flow path choices, leading to a *single, lower* WCET.

3.1 Mode Conditionals and Mode Variables

As different behaviour is determined by different evaluations of conditionals, the rough idea of mode estimation is to find out *which conditionals* are likely to govern modes, and *which values* on the input variables give rise to the varying evaluations of the conditional.

We call mode-governing conditionals *mode conditionals* and the input variables influencing their evaluation *mode variables*. Both features need to be derived heuristically by static analysis.

Heuristics for *mode variables* include:

- A variable is probably used as a mode variable if it is directly used for control in largely disparate parts of the source code.

- Mode variables are more likely directly used in a conditional or with a small indirection than only after a very long derivation.
- Naming conventions may enforce special names for mode variables.
- User annotations directly identify mode variables.

A syntactical heuristics for *mode conditionals* is given by the familiar implementation of a function body being guarded completely by an `if`-statement. More heuristics, however, are available if we take semantic information into account and predict the conditional's impact on runtime behaviour: If a conditional influences whether a significant part of the code is executed, this is more likely to signify a mode than a choice between two slightly different arithmetic operations. For example, consider a conditional choosing between the computation of an output by searching through a lookup table (in normal mode) or by passing on a default constant (in an exceptional mode). Thus, a conditional is deemed a mode conditional, if the *difference* in behaviours, such as predicted WCET influence or access to global variables, is large enough, that is, if the conditional is *unbalanced* according to some criterion. Note that even when taking semantic information into account, the heuristics still derive *logical* modes; *importance* of modes according to a criterion such as WCET impact is handled in a subsequent phase (Section 3.2).

The impact of a branch may be estimated simply by examining the operations in its branches.

- The conditional execution of loops, such as in the implementation of lookup tables with binary or linear search, hints at mode-dependent behaviour.
- External functions may be annotated by the user to signify whether they are special cases. For example, such functions may be those that communicate with external devices.
- Accesses to global variables, be it constant inputs or sensor values, signify differing behaviours.

In all cases, it is furthermore possible to exploit *synergies* with higher levels of the toolchain. For example, conditionals that are already present in the high-level tools, such as switch blocks, are more natural candidates for mode conditionals than `if` statements arising as artifacts of the code generation process.

Determining mode conditionals and mode variables are linked: Mode conditionals lead to mode variables by backward slicing, and mode variables lead to mode conditionals by forward slicing. Thus, an iteration of the slicing phases leads to an extension and refinement of mode determination.

In each of the analysis steps, human intervention is beneficial or even necessary. For example, the analysis needs annotations for external sensor reading library functions. On the other hand, annotations of known mode variables or mode conditionals simplifies the analysis. Such annotations can be provided by the programmers or tool users, but more likely can be passed onto the tool from higher-level specifications.

A *reduced* control-flow graph arises by abstracting those conditionals which are not mode conditionals out of the control-flow graph. The statically possible set of paths through the reduced control-flow graph contains the *significantly different* paths. Thereby, we have *divided* the input space of a task in various possible operating modes according to syntactic and semantic criteria.

Let us consider a single path in the reduced control-flow graph. By backward interpretation from the mode conditionals to their governing mode variables (in the complete graph), one can gain a specification of the value ranges of these variables giving rise to the one or the other choice. *Must* information as well as *may* information are useful: An *under-approximation* of the input state *guarantees* that these values definitely give rise to the behaviour as determined by one path; an *over-approximation* ensures that at least some paths are not taken in this state.

3.2 Clustering

What remains to be done is to *cluster* the paths in the reduced control-flow graph, for not all such logical modes form significant modes for the analysis at hand. This is an optional post-processing phase; the determination of logical modes, e. g., for program understanding, is complete at this point. For usage in timing analysis, for example, not all modes have to be treated differently: If their predicted WCETs do not differ significantly, they can be handled together. To this end, the paths in the reduced control-flow graph can be assigned a variety of properties to be used for classification into modes. Such properties include:

- the very control-flow choices defining the paths;
- read and write accesses to communication variables;
- static length, including presence of loops;
- calls to specified external functions.

More properties corresponding to different heuristics can be added into this framework. For a discussion of possible heuristics, see also [4].

To ensure that paths can be compared to each other, we need these properties to be comparable. If the comparability is ascertained, the paths can be sorted into *clusters* by graph-theoretic clustering or gravitational clustering algorithms [6].

This approach has the benefit that sensitivity to certain properties can be changed by the gravitational constants. For example, if a certain conditional is already known to be mode defining, then paths representing different control-flow choices shall not cluster at all; or the sensitivity between path length and communication patterns can be adjusted by balancing the respective constants. In this way, the importance of modes can be established with respect to other criteria such as the communication fabric.

Furthermore, the clustering can also directly be used for *sub-mode* determination, by running the clustering step again for the set of paths forming a mode with different gravitational constants.

Deducing constraints on the input variables giving rise to the paths of a cluster is more involved, however. Although it is reasonably easy to specify the clusters by exhaustive enumeration of properties and property combinations, be it an under- or an over-approximation, this is not likely to be a relevant information for the users. Future research needs to investigate whether existing approaches (e.g, [2]) yield sufficient results in our setting.

3.3 Usage of Mode Information

With several modes identified, the WCETs specific to each mode need to be computed. The obvious way to do so is to conduct several timing analyses with the inputs specifically fixed according to the modes—a costly process. A second possibility is to only solve different ILPs for combining the basic blocks' WCETs into the mode-specific task WCET by creating flow constraints from the mode governing conditionals. These two approaches thus enable a choice between precision and speed.

A third choice, *trace partitioning* [7], is a promising approach for a middle-ground approach. Partitioning according to a specific mode leads to a separate instance of timing analysis for each mode. This yields not only to a special execution time bound for each mode, one also specialises the whole of the analysis to the mode from the earliest phase on, without requiring several full analyses. If several mode conditionals are congenerous, trace partitioning takes care of exploiting this similarity automatically. Mode analysis thus is used to provide the split points for trace partitioning and the information to exploit the different results.

4. Other Approaches

Operating modes have been studied before in the literature, with differing foci and differing definitions of what constitutes a mode. For a discussion of various notions of modes, see [5].

Closely related to our approach is the work of [3, 4], which reports on a tool to semi-automatically derive operating modes from ASCET-MD models. Similar to our approach, various heuristics are employed to arrive at important modes, taking properties such as syntactic patterns, naming conventions and differences of measured execution times into account. The modes are then used to visualise mode-dependent signal flow and also for mode-dependent timing and schedulability analysis.

Another work strongly related to ours is [2]. The authors consider modes on a very low level, regarding different paths through functions as different modes, and derive a symbolic expression of WCET estimations. We concentrate on larger programs, distinguishing only between important modes according to some heuristics, and compute numeric restrictions on variables for use by WCET analysis.

The authors of [1] group those invocations of a component which lead to similar execution times into modes (*clusters*, in their terminology). Clusters are determined by a process of iterative refinement starting from one cluster representing the complete input space: A cluster is subdivided if its computed BCET and WCET bounds differ too greatly. Performing these analyses is very costly, however, and clustering in a blind search process which only considers the span between BCET and WCET need not lead to operating modes which are useful for scheduling or program understanding.

5. Conclusion and Future Work

We have described a process to determine operating modes by static analysis of C code and a usage scenario for operating modes in timing analysis. As of now, a source code analysis and heuristics for determination of mode variables and conditionals are implemented. We are developing the tool further to cover the full process, including the derivation of input values and the clustering phase.

References

[1] J. Fredriksson, T. Nolte, A. Ermedahl, and M. Nolin. Clustering worst-case execution times for software components. In *Proceedings of the WCET Workshop*, 2007.

[2] M.-L. Ji, J. Wang, S. Li, and Z.-C. Qi. Automated worst-case execution time analysis based on program modes. *The Computer Journal*, 52(5):530–544, 2009. Online 2007.

[3] J. E. Kim, R. Kapoor, M. Herrmann, J. Härdtlein, F. Grzeschniok, and P. Lutz. Software behavior description of real-time embedded systems in component based software development. In *Proceedings of ISORC*, pages 307–311, 2008.

[4] J. E. Kim, O. Rogalla, S. Kramer, and A. Hamann. Extracting, specifying and predicting software system properties in component based real-time embedded software development. In *Proceedings of ICSE*, pages 28–38, 2009.

[5] P. S. M. Pedro. *Schedulability of Mode Changes in Flexible Real-Time Distributed Systems*. PhD thesis, University of York, September 1999.

[6] T. V. Ravi and K. C. Gowda. Clustering of symbolic objects using gravitational approach. *IEEE TSMC-B*, 29(6):888–894, 1999.

[7] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM TOPLAS*, 29(5), 2007.

[8] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The determination of worst-case execution times—Overview of methods and survey of tools. *ACM TECS*, 7(3), 2008.

[9] R. Wilhelm, P. Lucas, O. Parshin, L. Tan, and B. Wachter. Improving the precision of WCET analysis by input constraints and model-derived flow constraints. In S. Chakraborty and J. Eberspächer, editors, *Advances in Real-Time Systems*. Springer-Verlag, 2010. To appear.