Classification of Code Annotations and Discussion of Compiler-Support for Worst-Case Execution Time Analysis [*]

Raimund Kirner, Peter Puschner
Institut für Technische Informatik
Technische Universität Wien
Treitlstraße 3/182/1
A-1040 Wien, Austria
{raimund,peter}@vmars.tuwien.ac.at

Abstract

Tools for worst-case execution time (WCET) analysis request several code annotations from the user. However, most of them could be avoided or being annotated more comfortably if the compilers would support WCET analysis.

This paper provides a clear categorization of code annotations for WCET analysis and discusses the positive impact on code annotations a compiler-support on WCET analysis would have.

1 Introduction

The knowledge of the worst-case execution time (WCET) is a mandatory prerequisite for the design of safety-critical embedded systems, since embedded systems have to fulfill the temporal requirements imposed by their physical environment.

Current research on compilers for embedded systems mainly focuses on issues like reduction of energy consumption, resource-aware code generation, or retargetable code generators. Program execution time is typically covered - as in traditional compiler construction - by performance-oriented code optimizations. The real-time behavior of programs is rarely covered.

As a consequence, a WCET analysis tool has to request the user for numerous code annotations, mostly at object code level, which could be avoided if the compilers explicitly support WCET analysis. Furthermore, the compiler-support would allow to specify code annotations at the source code level instead of burdening the user with object code annotations. Existing mechanisms like debug information is not sufficient in case of code optimizations performed by the compiler.

This paper categorizes in Section 2 the code annotations used by WCET analysis frameworks. Methods for providing such annotations are described in Section 3. The positive impact a compiler providing explicit support for WCET analysis would have on usage of these code annotations is discussed in Section 4.

2 Code Annotations for WCET Analysis

The calculation of the worst-case execution time (WCET) for a piece of code in general requires further information about the possible execution context or runtime behavior of the code. For example, the chosen configuration of the hardware platform has to be specified. Furthermore, the program analysis method may fail to predict the full execution behavior of a program with complex control flow and therefore, explicit assertions about the program behavior are required. These examples give an idea of what information is required by a WCET analysis tool additionally to the input program code. The specification of this additional information is done by code annotations. This section categorizes the different classes of code annotations required for WCET analysis and discusses possible methods to specify them.

Due to limitations on computability, a WCET analysis framework that is capable to analyze industrial code within a realistic software production process requires interfaces for the explicit specification of miscellaneous parameters. Some of these parameters are not

[*]This work has been partially supported by the FIT-IT research project "Model-Based Development of distributed Embedded Control Systems (MoDECS)" and the ARTIST2 Network of Excellence of IST FP6.

directly related to the WCET calculation itself, but are required to parse and interpret the program code. Therefore, we also looked at code annotation mechanisms provided by commercial WCET analysis tools like `aiT`[1] [3, 6] or `Bound-T`[2] [8, 7].

The code annotations for WCET analysis can be categorized as follows:

1. Platform Property Annotations (PPA)
2. CFG Reconstruction Annotations (CRA)
3. Program Semantics Annotations (PSA)
4. Auxiliary Annotations (AA)

Auxiliary annotations are constructs of annotation languages that are used to reference certain locations or control-flow edges in a program code. For example, a symbolic name that will be used later on within other code annotations, is assigned to a specific code location. The other categories of code annotations are described in the following subsections.

2.1 Platform Property Annotations

Platform Property Annotations (PPA) are application-independent annotations, which are used to characterize the target platform. A WCET analysis framework supports one or more target platforms. In case of a strictly static WCET analysis tool, it uses a built-in hardware model for each target platform. However, a computing platform typically can be configured in many ways. For example, there may be caches available with different layouts, or, as another example, the assignment of data and code to the available memory configuration can be done in different ways. Furthermore, to represent the calculated WCET bound as real time instead of processor cycles it is required to annotate the selected clock frequency for the processor.

The PPA annotations described above are used, for example, to parameterize the hardware models of caches and pipelines. Since in this case the annotations are not directly bound to the application code, there is no need of compiler support for such annotations. However, PPA annotations may be attached to the program code for the sake of code optimizations. For example, annotations about the use of read-only or write-only memory regions can be combined with annotations about their assignment to program code. This may allows a compiler to optimize the access operations for these data areas.

[1]`http://www.absint.de`
[2]`http://www.bound-t.com`

2.2 CFG Reconstruction Annotations

The CFG Reconstruction Annotations (CRA) are used as guidelines for the analysis tool to construct the control flow graph (CFG) of a program. Without these annotations it may not be possible to construct the CFG from the object code of a program.

On the one side, annotations are used for the construction of syntactical hierarchies within the CFG, i.e. to identify certain control-flow structures like loops or function calls. For example, a compiler might emit ordinary branch instructions instead of specific instructions for function call or return. In such cases it might be required to annotate a branch instruction whether it is a call or return instruction. A work around that sometimes helps avoiding code annotations is to match code patterns generated by a specific version of a compiler. However, such a “hack” cannot cover all situations and may also have the risk of incorrect classifications, for example, if a different version of the compiler is used.

On the other side, annotations may be needed for the construction of the CFG itself. This may be the case for branch instructions where the address of the branch target is calculated dynamically. Of course, static program analysis may identify a precise set of potential branch targets for those cases where the branch target is calculated locally. In contrast, if the static program analysis completely fails to bind the branch target, it has to be assumed that the branch potentially precedes each instruction in the code, which obviously is too pessimistic to be able to obtain a useful WCET bound. In such a case, code annotations are required that describe the possible set of branch targets.

2.3 Program Semantics Annotations

Program Semantics Annotations (PSA) are used to guide the calculation of a program’s dynamic behavior. In contrast, the annotations of Section 2.2 and 2.1 provide mostly static information about the program to be analyzed and its intended target platform.

To obtain a precise WCET bound, it is mandatory to accurately calculate the possible dynamic behavior of the program. For example, a static WCET analysis tool calculates the dynamic behavior of the program by *exec-time modeling* and by performing *path analysis* (as described in Chapter 2 of [10]). Exec-time modeling means the assignment of execution time to instructions for a given execution context.

To calculate a WCET bound, it is at least necessary to get iteration bounds for every loop or recursive call structure in the program. A quality improvement of the resulting WCET bound is possible if infeasible paths can be excluded from the calculation of the longest path. Annotation languages that allow the explicit specification of flow constraints are described in [9, 1] (also `aiT` and `Bound-T` allow the specification of flow constraints). However, in case the static analysis of the WCET tool performs a semantic analysis of the program, it may be sufficient to indirectly specify the feasible paths by describing properties like value constraints or invariants of program variables.

Another kind of PSA annotation is the description of possible addresses of memory references. Such annotations may improve the path analysis as well as the exec-time modeling.

3 Annotation Methods

This section discusses different methods how to annotate the code. First of all, to get precise results, it is important that WCET analysis is performed at a program representation level close to the executable program format. We call the program representation level where the analysis is performed *object code* level. Following the ongoing trend in embedded systems development, the representation level where the program is developed is much more abstract. By *source code*, we denote the representation level of program development. The whole tool chain that transforms the program from source code to object code is summarized as *compiler*. Following these definitions we can describe things in common terms without loosing generality.

3.1 Separate Annotation Files

One way to annotate code is to use a separate annotation file. This is especially useful for annotating the object code, as there are no common tools to add such information to the object code. Since `aiT` and `Bound-T` are primarily designed to analyze object code, they both support the use of separate annotation files. Both tools have to provide such an annotation technique, due to the missing compiler-support for WCET analysis. Another reason is that the WCET analysis framework should also be able to analyze code that is only available as object code. The obvious drawback of this procedure is that the developer has to look at and understand the object code, which is only an intermediate representation where code locations might change each time the source code is modified and re-compiled.

The support of annotations referring to code locations relative to symbolic labels reduces the amount of code annotations that have to be checked again whenever a single module has been re-compiled.

A more practical way to refer to the program code is to describe the referring code location structurally. For example, `aiT` allows to refer to loops by their order within a function. `aiT` also allows to annotate loops at the source code, but this represents the same mechanism, since the source code locations of these annotations are translated into structural locations. Further, `Bound-T` provides a quite generic pattern matching language that allows to refer to code locations based on various criteria. Using structural references allow the user to annotate for the object code while looking at the source code. However, the drawback of this technique is that it fails in case that the code optimizations performed by the compiler change the structure of the code.

3.2 Annotations within Program Code

Code annotations within the program code provide the advantage that the developer can annotate the program behavior directly where the program is coded. The preferred annotation method from the developer's point of view is to directly annotate the source code.

The concrete syntactical realization of these code annotations is not of stringent importance within this paper. Even the approach of extending the programming language with code annotation constructs allows the compilation by conventional compilers that do not support these language extensions. This can be realized by deactivating the annotations by a preprocessing pass prior to compilation [9]. The more relevant question is whether the compiler provides support for maintaining the consistency of code annotations in case of code optimizations that change the structure of the code. As shown by Exler, the consistency of code annotations may not be maintained without the help of the compiler in case of code optimizations that change the structure of the code [2].

The code annotation within the source code is especially interesting for PSA annotations since this provides the most seamless annotation interface for analyzing and annotating the code manually by the user. PPA annotations are natural candidates for separate annotation files since they refer to the low-level details of the target platform. As a further argument, the PPA annotations are often application independent.

4 Compiler Support for WCET Analysis

The compiler (and all related tools as defined in Section 3.2) transforms the code from the source code representation level to the object code level, at which

WCET analysis is applied. There are several reasons why a compiler can contribute to and improve the calculation of a WCET bound:

- The compiler has the control and knowledge over all code transformations that are performed before emitting the object code. For a number of code optimizations it is not possible to recognize the effect of the optimization by comparing the structure of the object code with that of the source code.

- The compiler has the view on both, the source code and the object code. Typically, the execution behavior of a program is easier to obtain from the source code than from the object code. This is because the instructions in the object code reflect low-level implementation issues enforced by the characteristics of the target hardware optimized for low resource consumption. For example, distinct variables in the source code can become aliased as spilled registers in the object code.

However, due to their lack of support for WCET analysis, compilers are currently not considered as a helpful tool for calculating a WCET bound. Instead, the policy often is to turn off most of the features of a compiler for the sake of generating object code that maintains properties found in the source code. The result is an object code that shows a poor runtime performance and a WCET that is typically much higher than in the fully optimized code.

The intention of having a compiler supporting WCET analysis is to get WCET analyzable code with a seamless interface for code annotations. The support by the compiler can be twofold. First, a seamless integration of code annotations into the source code representation level can be provided. Second, the need for code annotations can be reduced by emitting properties about the object code by the compiler. The following lists several possibilities how compilers could support WCET analysis.

Emit Description of CFG Structure: A static WCET analysis tool has to use CRA annotations at the object code level for reconstructing the CFG of a program. Using such code annotations is a burden for the user of the tool since it forces him to look at the object code level of a program, maybe each time the code is re-compiled.

The compiler knows about the CFG structure at the same precision as it is given by the syntactic structure of the source code. Therefore, the compiler could automatically annotate the generated object code by CRA annotations that will guide the WCET analysis tool to reconstruct the CFG of the program.

Currently, there is an initiative under way by the cluster *Compilers and Timing Analysis* of the ARTIST2 Network of Excellence of the IST FP6. The aim of this group is to define a common format for the specification of object code and code annotations. As a natural consequence, compilers could be extended to directly generate such a code specification file.

Maintain Consistency for Code Annotations: The natural interface for PSA annotations is the source code representation level, because this would allow the developer to do the implementation of the program logic and the code annotation at the same representation level.

A framework that allows to maintain consistency of control-flow annotations in case of code optimizations performed by the compiler is described in [10]. This framework maintains the consistency of the annotations for arbitrary code transformations. Such a framework can be complemented by static program analysis as a preprocessing step to calculate control-flow annotations from the code semantics and the provided annotations about code invariants. An example for such a static program analysis based on abstract interpretation has been described by Gustafsson and Ermedahl [4].

Emit Properties of Execution Behavior: PSA annotations provide hints about the execution behavior of a program. This information can be used for the *exec-time modeling* and *path analysis* phase of a WCET analysis tool (see Section 2.3).

The compiler may reduce the amount of required PSA code annotations by automatically calculating and emitting some of these code properties. For example, the compiler may know the memory area potentially referenced by a specific pointer operation. Research on compiler extensions to emit code annotations about control flow and memory access addresses is described in [11, 5].

Improve Predictability of Code: A compiler may indirectly support WCET analysis by features not directly related to code annotations. For example, by using the *single path conversion* the execution-time jitter of real-time programs may be reduced while at the same time the WCET analyzability of the program will be improved [12]. This conversion may be also applied to local program segments instead of the whole program, giving an effect similar to *wcet-oriented programming* [13].

4.1 Using Optimizing Compilers to Produce Safety-Critical Code

There is often the argument that code optimizations have to be prohibited for the production of safety-critical code. This argument is strengthened by the fact that it is very hard to prove formal correctness of a compiler. However, recent research is focusing on analysis techniques to verify the semantic equivalence between the original and the optimized version of a program [14]. Maybe, such verification techniques in the future can be used to weaken the prohibition of code optimizations on safety-critical software, and at the same time providing a stronger argument for the support of WCET analysis by optimizing compilers.

5 Summary and Conclusion

This paper provides an analysis of how compiler-support would improve the use of WCET analysis tools.

First, a categorization of code annotations for WCET analysis has been done, resulting into four categories: platform property annotations (PPA), CFG reconstruction annotations (CRA), program semantics annotations (PSA), and auxiliary annotations (AA).

Second, it has been discussed what impact compiler-support could have on these annotations. PPA annotations are program-independent, therefore no compiler-support is needed to support WCET analysis. However, the compiler may use PPA annotations for platform-dependent code optimizations. The CRA annotations address the object code level. A compiler may be extended to output additional program properties in order to reduce the need for manual CRA annotations. The most important impact a compiler supporting WCET analysis provides, is for PSA annotations. It will free the user from the burden of manually analyzing and annotating the behavior of the object code (provided that the source code of the program is available).

Acknowledgments

The authors would like to thank C. Ferdinand, R. Heckmann, and H. Theiling from AbsInt and N. Holsti from Tidorum for fruitful discussions about code annotations for WCET analysis.

References

[1] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. 21st IEEE Real-Time Systems Symposium (RTSS)*, Orlando, Florida, USA, Dec. 2000.

[2] M. Exler. Propagierung von Pfadinformation für die Analyse von Programmlaufzeiten. Master's thesis, Technische Universität Wien, Vienna, Dec. 1999.

[3] C. Ferdinand, R. Heckmann, and H. Theiling. Convenient user annotations for a WCET tool. In *Proc. 3rd Euromicro International Workshop on WCET Analysis*, pages 17–20, Porto, Portugal, July 2003.

[4] J. Gustafsson and A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. *Parallel and Distributed Computing Practices*, 1(2), June 1998.

[5] C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon. Bounding Pipeline and Instruction Cache Performance. *IEEE Transactions on Computers*, 48(1), Jan. 1999.

[6] R. Heckmann and C. Ferdinand. Combining automatic analysis and user annotations for successful worst-case execution time prediction. In *Embedded World 2005 Conference*, Nürnberg, Germany, Feb. 2005.

[7] N. Holsti. *Bound-T Application Note ERC32.* Space Systems Finland Ltd, Espoo, Finland, 1 edition, Jan. 2002.

[8] N. Holsti. *Bound-T User Manual.* Space Systems Finland Ltd, Espoo, Finland, 2 edition, Mar. 2003.

[9] R. Kirner. The programming language WCETC. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[10] R. Kirner. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis.* PhD thesis, Technische Universität Wien, Vienna, Austria, May 2003.

[11] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C.-S. Kim. An accurate worst case timing analysis for RISC processors. *Software Engineering*, 21(7):593–604, 1995.

[12] 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).

[13] P. Puschner. Algorithms for Dependable Hard Real-Time Systems. In *Proc. 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Jan. 2003.

[14] R. van Engelen, D. Whalley, and X. Yuan. Automatic validation of code-improving transformations on low-level program representations. *Science of Computer Programming*, 52:257–280, Aug. 2004.