Run-Time Guarantees for Real-Time Systems—The USES Approach*

Christian Ferdinand[1], Daniel Kästner[2], Marc Langenbach[2], Florian Martin[2], Michael Schmidt[2], Jörn Schneider[2], Henrik Theiling[2], Stephan Thesing[2], and Reinhard Wilhelm[2]

[1] AbsInt Angewandte Informatik GmbH, Universität des Saarlandes, Starterzentrum—Gebäude 45, 66123 Saarbrücken
[2] Fachbereich Informatik, Universität des Saarlandes, 66123 Saarbrücken, Germany

Abstract. The USES group follows an approach to compute reliable run-time guarantees which is based on well-understood theoretical foundations, practical in use, and efficient.
Our worst case execution time (WCET) prediction tool is modularly structured: the task of determining the WCET is divided into a sequence of subtasks. The structure has been developed by taking into account the results of previous work. The subtasks value analysis, cache analysis, and pipeline analysis are tackled by abstract interpretation. Path analysis is performed by integer linear programming (ILP).
The analyses will be embedded in a framework which allows to generate (or adapt) the analyses from one target description and provides well defined interfaces and exchange formats.

1 Introduction

An important characteristics of hard real-time systems is that they are subject to timing constraints which determine when computations have to be accomplished. Failure of a safety critical real-time system can lead to considerable damage or even a loss of lives. Therefore, a schedulability analysis has to be performed in order to guarantee that all timing constraints will be met. All existing techniques for schedulability analysis require the worst case execution time (WCET) of each task in the system to be known. Since these are in general not computable, estimations of the WCET have to be calculated. These estimations have to be safe, i. e., they may never underestimate the real execution time, and they should be tight, i. e., the overestimation should be as small as possible.

For processors with fixed execution times for each instruction there are established methods to compute sharp WCET bounds [PK89,PS91]. However, in modern microprocessor architectures caches and pipelines are key features for improving performance. The consequence is that the execution behaviour of the instructions cannot be analysed separately since it depends on the execution history. Therefore, the classical approaches to worst case execution time prediction are not directly applicable or lead to results exceeding the real execution time by orders of magnitude.

* Work of the USES group (**U**niversity of the **S**aarland **E**mbedded **S**ystems group) is partially supported by TFB 14 of the Deutsche Forschungsgemeinschaft.

2 Overview

This paper describes the overall design of an integrated WCET prediction tool. The next two sections summarise previous work done by other groups and by the USES group that have influenced our design decisions and represent basic components of the final system. An outline of the planned developments is given in Section 4.1 and Section 4.1 summarises characteristic aspects. The underlying framework in which all components are embedded and which provides for well defined interfaces and easy retargetability is presented in Section 4.3. Section 4.4 discusses industrial relevance and Section 4.5 concludes.

3 Previous Work

3.1 Some Approaches to WCET Prediction

The following approaches to predict the WCET of programs have influenced the design of our approach.

Arnold, Mueller, Whalley, and Harmon [MWH94,Mue96] describe a data flow analysis for the prediction of instruction cache behaviour of programs for direct mapped caches. This analysis has been one of the starting points of our cache analysis as described below.

In [LMW96] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe describe an integrated method to determine the worst case execution time of a program and to model architecture features like instruction caches and/or pipelines. The problem of finding an accurate worst case execution time bound is formulated as an integer linear program that must be solved.

For direct mapped instruction caches and programs whose execution paths are well defined and not very input dependent the predictions can be computed fast and are very accurate. Increasing levels of associativity where the cache behaviour of one memory reference depends on more (other) references and less defined execution paths lead to prohibitively high analysis times.

In [LBJ+95], Lim et al. describe a general framework for the computation of WCETs of programs in the presence of pipelines and cache memories based on the timing schema approach of Park and Shaw [PS91]. From the program structure a timing equation system can be derived which must be explicitly solved. An approximation to the solution for the set of timing equations has been proposed. The usage of an input and output state provides a way to modularise the timing analysis and makes especially the pipeline analysis very intuitive.

3.2 Own Work

The USES approach evolved from the following previously studied components or aspects of a WCET tool:

- Cache analysis technique based on abstract interpretation and implemented in a PAG (Program Analyser Generator) generated analyser. It receives as

input the control flow graph of a program and a cache description and produces a categorisation of the instruction/context pairs of the input program [Fer97,FMW97,FMWA98] (*see* Section 3.2).
- Pipeline analysis for the super-scalar SuperSPARC I. The results of the cache analysis are used as a parameter [Sch98,SF99] (*see* Section 3.2).
- Program path analysis based on integer linear programming. It uses the results of the cache analysis and calculates the estimated WCET together with an execution profile [TF98,The98] (*see* Section 3.2).

Program Analysis to Predict Run-time Behaviour Program analysis is a widely used technique to determine runtime properties of a given program without actually executing it. A program analyser takes a program as input and computes an approximation of an often undecidable or very hard to compute program property.

There is a well developed theory of program analysis, *abstract interpretation* [CC77,NNH99]. This theory states criteria for correctness and termination of a program analysis. A program analysis is considered an abstraction of a standard semantics of the programming language. Abstract interpretation amounts to performing a program's computations using *value descriptions* or *abstract values* instead of concrete values.

One reason for using abstract values instead of concrete ones is computability: to ensure that analysis results are obtained in finite time. Another is to obtain results that describe the result of computations on a set of possible (e. g., all) inputs.

The behaviour of a program (including its cache and pipeline behaviour) is given by the semantics of the program. A standard (operational) semantics of a language is given by a *domain* of data and a set of functions describing how the statements of the language transform data. To predict the run-time behaviour of a program, we approximate its *collecting semantics*.

The collecting semantics gives the set of all program (cache and pipeline, resp.) states for a given program point. Such information combined with a path analysis can be used to derive WCET-bounds of programs. This approach has successfully been applied to predict the cache behaviour of programs [Fer97,FW98,TF98] and can naturally be adapted to predict the pipeline behaviour [Fer97,Sch98,SF99].
The approach works as follows:

- In a first step, the *concrete semantics* of programs is defined. The concrete semantics is a simplified (auxiliary) semantics that describes only the interesting aspects of computation but ignores other details of execution. E. g., for the cache analysis, the concrete cache semantics only determines the resulting cache state for a given path in the program, but ignores for example register values. In this way each real state is represented by a concrete state.
- In the next step, an *abstract semantics* that "collects" all possibly occurring concrete states for each program point is defined. An abstract semantics consists of an abstract domain and a set of abstract semantic functions, so called

Category	Abb.	Meaning
always hit	ah	The memory reference will always result in a cache hit.
always miss	am	The memory reference will always result in a cache miss.
persistent	ps	The referenced memory block will be loaded at most once.
not classified	nc	The memory reference could neither be classified as ah, am, nor ps.

Table 1. Categorisations of memory references and memory blocks.

transfer functions, for the program statements computing over the abstract domain. They describe how the statements transform abstract data. They must be monotonic to guarantee termination. An element of the abstract domain represents sets of elements of the concrete domain. The subset relation on the sets of concrete states determines the complete partial order of the abstract domain. The partial order on the abstract domain corresponds to precision, i. e., quality of information.
The partial order determines the *least upper bound* operation, $\sqcup$, on the abstract domain. This operation is used to combine information stemming from different sources, e. g. from several possible control flows into one program point.

The abstract semantics is constructed in such a way that an abstract state at a given program point "represents" at least all concrete states that are included in the collecting semantics [Fer97].

The computation of the abstract semantics can be implemented with the help of the program analyser generator `PAG` [Mar98], which allows to generate a program analyser from a description of the abstract domain and of the transfer functions.

Cache Analysis Three analyses have been designed and implemented.

The analyses are used to compute a categorisation for each memory reference describing its cache behaviour. The categories are described in Table 1.

Must Analysis The must analysis determines a set of memory blocks that are in the cache at a given program point whenever execution reaches this point. Good information is the knowledge that a memory block is in this set.

May Analysis To determine whether a memory block will never be in the cache we compute the complementary information, i. e., sets of memory blocks that *may* be in the cache. Precise information is that a memory block is not in this set, because this memory block can be classified as definitely not in the cache whenever execution reaches the given program point.

Persistence Analysis For memory references which are not categorised as `ah` or `am`, *persistence* of the referenced memory block is a valuable property. A memory block is called *persistent* if no execution will remove it from the cache. Knowledge about persistence allows to bound the number of cache misses to 1. The first reference may result in a cache miss, but *all* further references cannot result in cache misses.

Data Caches and Combined Caches [FW98,Fer97] generalises the persistence analysis to sets of possibly referenced memory locations, e. g., arrays. This generalisation determines memory locations that survive in the cache thus providing effective and efficient means to compute an upper bound of the number of possible cache misses.

Pipeline Analysis [Fer97,SF99] describe the foundations of pipeline analysis and [Sch98,SF99] describes an experiment done for the super-scalar pipeline of the SuperSPARC I. The results of a preceding cache analysis are used, so the cache state need not explicitly be considered in the concrete pipeline semantics.

The abstract pipeline update function reflects what happens when a new instruction enters the pipeline. It takes into account the current set of pipeline states, in particular the resource occupations, the contents of the prefetch queue, the grouping of instructions, and the classification of memory references as cache hits or misses.

Analysis of Loops and Recursive Procedures Loops and recursive procedures are of special interest since programs spend most of their runtime there. Frequently in programs, the first execution of the loop body loads the cache and subsequent executions find most of their referenced memory blocks in the cache. This has to be taken into account when analysing the behaviour of a loop on the cache. Therefore, we distinguish the first iteration from the others.

In the `USES` approach a method has been designed and implemented in the program analyser generator `PAG`, which virtually inlines the first pass through a recursive procedure and virtually unrolls loops once, the so-called VIVU approach [MAWF98]. Memory references are now considered in different execution contexts, essentially nestings of first and non-first iterations of loops (recursive procedures, resp.). Experiments have shown that the VIVU approach can largely increase the precision [Fer97].

Worst Case Path Analysis Using Integer Linear Programming (ILP) By using ILP the structure of a program and the set of program paths can be described in a very natural way. A set of constraints describes the overall structure of the program. Solving these constraints yields very precise results [TF98,LMW96].

A problem formulated in ILP consists of two parts: the objective function and constraints on the variables used in this function. The objective function represents the number of CPU cycles in the worst case. Correspondingly, it has to be maximised. Each variable in the objective function represents the execution count of one basic block of the program and is weighted by the execution time of that basic block.

The integer constraints describing how often basic blocks are executed relative to each other can automatically be generated from the control flow graph. However, additional information about the program provided by the user is usually needed, as the problem of finding the worst case program path is unsolvable in the general case.

4 The `USES` approach to WCET-Computation

4.1 Planned Work

The results of previous work have lead the DFG (German National Science Foundation) to fund the pre-product development of a tool for analysing the WCET of programs (*Transfer Project* 14). The overall goal is an integration of all components in a development environment for microcontroller applications on modern architectures. This way, the calculated information concerning the runtime behaviour can be efficiently used during software development. The development is done in cooperation with two innovative companies, AbsInt Angewandte Informatik GmbH and HighTec EDV Systeme GmbH. The *Transfer Project* 14 runs for three years. The projected development includes:

- Implementation of frontends for appropriate architectures.
- Development of a generic static pipeline analysis.
- Development of a value analysis. The knowledge of the ranges of values of registers (on executable level) allows to resolve indirect accesses to memory. The knowledge of the ranges of values of variables (on source level) allows to minimise the amount of user annotations and to check user provided information.
- Integration of the value, cache, and pipeline analysis with a path analysis.
- Integration of all components into a graphical user-interface.

The cache and the pipeline analyses assume an uninterrupted execution. However, most real-time systems are implemented as reactive systems where task execution is initiated by external events as, e. g., interrupts or exceptions. The consequences of such interrupts on the cache and pipeline behaviour are non-trivial and require new analysis approaches.

In [KT98] a pre-runtime scheduling method has been developed which precisely incorporates the effects of task switching on the processor cache into its decisions. This approach will also be followed in the transfer project.

4.2 Rationale and Essentials

Several elements are characteristic to the `USES` approach. *Modularity*: the task to determine the WCET of a program is structured into a sequence of subtasks, *Adequacy of methods*: the substasks are tackled with appropriate methods, and *Genericity*: generic and generative methods are used whenever possible. These principles lead to an understandable, maintainable, efficient, and provably correct approach.

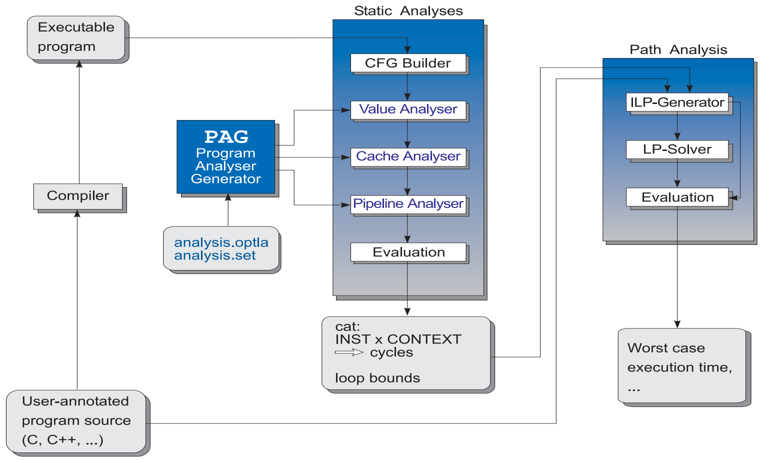

Fig. 1. The structure of the analysis.

Separation into Phases The determination of the worst case execution time (WCET) of a program is composed of several different tasks, i. e., classification of memory references as cache misses or hits, done by the *cache analysis*, predicting the behaviour of the program on the processor pipeline, achieved by the *pipeline analysis*, and the determination of the worst case execution path of the program, which is done by the *path analysis*. Combining them into one analysis adds complexity.

The sequence: value prediction, cache analysis, pipeline analysis, path analysis has been chosen in the `USES` approach, each analysis step uses the results of the preceding ones. The separation of WCET determination into several phases has the additional effect that different methods tailored to the subtasks can be used. In our case, the value prediction, the cache analysis and the pipeline analysis are done by *abstract interpretation*. The path analysis is done by *integer linear programming*.

By serialising the tasks we see a chance of losing precision. Our prototypes show, however, that this loss is very small.

4.3 Underlying Framework

Composition of the Framework Starting point of our analysis framework (*see* Figure 1) is C source code annotated with user provided information about number of loop iterations, upper bounds for recursion, etc. The C code is then translated into an executable or assembler file with a standard C compiler.

In a next step a parser is created which reads its input instruction by instruction (executables and assembler files can be analysed) and translates them into corresponding `CRL` instructions (*see* Section 4.3) grouping them to basic blocks.

The annotated control flow graph is the input for the microarchitecture analysis which determines execution times for instructions in various contexts. The result of this step is used by the path analysis to determine one longest execution path in the given control flow graph.

The address analysis [Sic97] determines ranges of values of registers and by this it can resolve indirect accesses to memory involving registers. The cache analysis classifies the accesses to main memory, i. e., whether the data needed resides in the cache or not. The pipeline analysis models the pipeline behaviour to determine execution times for a sequential flow of instructions. The result is an execution time for each instruction in each distinguished execution context.

Based on the results of the microarchitecture analysis, the path analysis derives a safe estimation of the WCET using ILP.

`TDL` Specification Language All target-specific information required for the generic program parts is expressed by a dedicated description language `TDL` (*target description language*). Two aspects must be covered. First, the properties of all hardware resources which are relevant for the analyses have to be specified. Second, the specification must allow for the automatic generation of a frontend for the assembly language of the target machine. There are numerous existing hardware description languages, e. g. [FVPF95,Had98,BCRS97]. However, the requirements of retargetable postpass analyses and optimisations (not discussed here) associated with this approach made the development of a dedicated specification language necessary.

Control Flow Representation Language `CRL` CRL (*control flow representation language*) is designed to represent the control flow of a given program in a hardware-independent way, but with the ability to include architecture-specific information. In CRL a control flow graph is hierarchically composed of routines, blocks, and instructions.

4.4 Industrial Perspective

The market for embedded applications is growing rapidly. Integration efforts (system-on-a-chip) and new applications require more and more computational performance. The hardware manufacturers have reacted to these requirements: according to their road maps for the next few years basically all performance oriented microcontrollers (cores) and even DSPs will features cache memories and (complex) pipelines. For many current designs this is already the case.

The `USES` approach allows to give runtime guarantees (due to the underlying theories, formally provable) also for modern hardware architectures.

The microprocessor architectures are subject to rapid evolution. The use of generic and generative methods support the modelling of latest hardware features and provide for easy retargetability of the tools.

Another important aspect is the productivity of an embedded systems programmer. Stringent time to market requirements necessitate the use of higher programming languages, like C instead of assembly or C++ or Java instead of C. While the assembly programmers for the classical architectures without cache memories and pipelines was usually well aware of the execution time of their code, the users of higher programming languages have much more problems to determine the timing behaviour. Within the Transfer Project 14, the `USES` group has the goal to built an integrated and easy-to-use WCET tool that is an integral part of a real IDE (Integrated Development Environment) so that the embedded systems programmer is provided with immediate feedback on the execution time.

4.5 Conclusion

We have presented the design of a system for reliable run-time guarantees, which is currently under development in the `USES`-group.

Abstract interpretation is used for the value, cache, and pipeline analysis. The approach is semantics based, allows for correctness proofs, allows to use generative support (`PAG`), and allows for a high level of genericity.

Integer linear programming is used for the path analysis. It provides an intuitive and flexible user interface for program annotations and allows for virtually arbitrary precision in the path description by the user.

Both abstract interpretation and integer linear programming are well explored theories. Together with the well defined interfaces and intermediate representations, they allow for easy extensibility (e. g. to model branch prediction schemes found in recent microprocessors).

Portability is supported by a common language for the specification of the underlying hardware properties. Our approach directly analyses executables. No special compiler or linker are required. Furthermore, standard *optimising* compilers can be used.

The results of experiments done so far are convincing. This gives reason to hope that the separation of analyses as presented above will give precise results within very moderate analysis times for the overall integrated WCET tool.

References

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[BCRS97] F. Bodin, Z. Chamski, E. Rohou, and A. Seznec. *Functional Specification of SALTO: A Retargetable System for Assembly Language Transformation and Optimization. rev. 1.00 beta*. INRIA, June 1997.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, January 1977.

[Fer97] C. Ferdinand. Cache Behavior Prediction for Real-Time Systems. PhD Thesis, Uni. d. Saarlandes, September 1997.

[FMW97] C. Ferdinand, F. Martin, and R. Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *LCTRTS*, pages 37–46, June 1997.

[FMWA98] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache Behavior Prediction by Abstract Interpretation. *Science of Computer Programming, Elsevier*, 1998.

[FVPF95] A. Fauth, J. Van Praet, and M. Freericks. Describing Instruction Set Processors Using nML. In *European Design and Test Conference*. IEEE, 1995.

[FW98] C. Ferdinand and R. Wilhelm. On Predicting Data Cache Behavior for Real-Time Systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, June 1998.

[Had98] George Hadjiyiannis. ISDL: Instruction Set Description Language Version 1.0. Technical report, MIT RLE, April 1998.

[KT98] D. Kästner and S. Thesing. Cache Sensitive Pre-Runtime Scheduling. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, Montreal, Canada, June 1998.

[LBJ+95] 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. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[LMW96] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *RTSS*, December 1996.

[Mar98] F. Martin. PAG—an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1), 1998.

[Mar99] F. Martin. *Generation of Program Analyzers*. PhD thesis, Uni. d. Saarlandes, 1999. to appear.

[MAWF98] F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. Analysis of Loops. In *Proceedings of the International Conference on Compiler Construction (CC'98)*. Springer, 1998.

[Mue96] F. Mueller. Generalizing Timing Predictions to Set-Associative Caches. TR96-66, Institut f. Informatik, Humboldt-University, July 1996.

[MWH94] F. Mueller, D. B. Whalley, and M. Harmon. Predicting Instruction Cache Behavior. In *LCTRTS*, 1994.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[PK89] P. Puschner and Ch. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1:159–176, 1989.

[PS91] C. Y. Park and A. C. Shaw. Experiments with a Program Timing Tool Based on Source-Level Timing Schema. *IEEE Computer*, 24(5):48–57, May 1991.

[Sch98] Jörn Schneider. Statische Pipeline-Analyse für Echtzeitsysteme. Dipl. Thesis, Universität des Saarlandes, October 1998.

[SF99] J. Schneider and C. Ferdinand. Pipeline behavior prediction for superscalar processors. Technical Report A/02/99, Uni. d. Saarlandes, February 1999.

[Sic97] Martin Sicks. Adreßbestimmung zur Vorhersage des Verhaltens von Daten-Caches. Dipl. Thesis, Uni. d. Saarlandes, 1997.

[TF98] H. Theiling and C. Ferdinand. Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis. In *RTSS*, pages 144–153, Madrid, Spain, December 1998.

[The98] H. Theiling. Über die Verwendung ganzzahliger linearer Programmierung zur Suche nach längsten Programmpfaden. Dipl. Thesis, Uni. d. Saarlandes, Sep. 1998.

[WM95] R. Wilhelm and D. Maurer. *Compiler Design*. International Computer Science Series. Addison–Wesley, 1995. Second Printing.