

Comparing Different Worst-Case Execution Time Analysis Methods

Jakob Engblom^{†*}
IAR Systems AB
Box 23051, SE-750 23 Uppsala
Sweden
email: jakob.engblom@iar.se

Andreas Ermedahl[†]
Dept. of Computer Systems
Uppsala University
Box 325, SE-751 05 Uppsala
Sweden
email: andreas.ermedahl@docs.uu.se

Friedhelm Stappert[‡]
C-LAB
Fürstenallee 11
33102 Paderborn
Germany
email: fst@c-lab.de

Abstract

Knowing the Worst-Case Execution Time (WCET) of a program is necessary when designing and verifying real-time systems. A correct WCET calculation method must take into account the possible program flow (like loop iterations and function calls), as well as effects of hardware features like caches and pipelines.

In this paper we describe our work on comparing two different calculation methods for finding the WCET of a real-time program, namely the Implicit Path Enumeration Technique (IPET) and the Path-based calculation method.

We investigate a structured way to fairly compare the computational complexity, as well as the expressiveness, i.e. the set of possible program flow constraints that can be handled by the two methods.

Keywords: WCET, hard real-time, embedded systems, ILP, path search.

1. Introduction

The purpose of *Worst-Case Execution Time* (WCET) analysis is to provide a priori information about the worst possible execution time of a program before using the program in a system. WCET estimates are used in real-time systems development to perform scheduling and schedulability analysis, to determine whether performance goals are met for periodic tasks, and to check that interrupts have sufficiently

short reaction times. The goal of WCET analysis is to generate a *safe* (i.e. no underestimation) and *tight* (i.e. small overestimation) estimate of the worst-case execution time of the program.

To generate a WCET estimate, we consider a program to be processed through the phases of *program flow analysis*, *low level analysis* and *calculation*.

The program flow analysis phase determines the dynamical behaviour of the program. The result is information about which functions get called, how many times loops iterate, dependencies between *if*-statements, etc.

The purpose of low-level analysis is to determine the execution time for each atomic unit of flow (e.g. an instruction or a basic block), given the architecture and features of the target system. Examples of features to consider are pipelines and caches.

In this paper we focus on the calculation phase which generates the final WCET estimate for the program, given the program flow and the low-level analysis results. We investigate two different calculation approaches, namely the *Implicit Path Enumeration Technique* (IPET) and the *Path-based* method.

Several results using the respective method have been previously published (see Section 2). However, it is necessary to objectively compare the different calculation methods in order to find out which one is most appropriate for which particular problem. E.g., it may turn out that one method is generally better than the other, or, more likely, that there are cases where one approach performs better than the other and vice versa. Therefore, we are developing a strategy to evaluate the two proposed methods. We are particularly interested in their ability to handle various program flow informations and in their computational complexity.

The main contribution of this paper is to propose a way of doing a fair and structured comparison of the Path- and IPET-based calculation methods.

*Jakob is an industrial PhD student at IAR Systems (<http://www.iar.com>) and Uppsala university, sharing his time between research and development work.

[†]This work is performed within the Advanced Software Technology (ASTEK, <http://www.docs.uu.se/astec>) competence center, supported by the Swedish National Board for Industrial and Technical Development (NUTEK, <http://www.nutek.se>).

[‡]Friedhelm is a PhD student at C-LAB (www.c-lab.de), which is a cooperation of Paderborn University and Siemens AG.

2. Related Work

There are three main categories of WCET calculation methods proposed in literature: Path-, Tree-, or IPET (Implicit Path Enumeration Technique)-based.

In a Path-based calculation [6, 11], the final WCET estimate is generated by calculating times for different paths in a program, searching for the path with the longest execution time. The defining feature is that possible execution paths are *explicitly* represented.

In Tree-based methods [1, 8], the final WCET is generated by a bottom-up traversal of a tree representing the program. The analysis results for smaller parts of the program are used to make timing estimates for larger parts of the program.

IPET-based methods [5, 7, 9, 10] express program flow and atomic execution times using algebraic and/or logical constraints. The WCET estimate is calculated by maximizing an objective function, while satisfying all constraints.

To our knowledge, this is the first paper that deals with a comparison of different calculation methods.

3. Properties of Calculation Methods

In the following, we describe the IPET and Path-based calculation methods in more detail.

3.1. IPET-based Calculation

In IPET the flow of a program is modeled as an assignment of values to *execution count variables*. The values reflect the total number of executions of each node for an execution of the program.

Each entity with a count variable (x_{entity}) also has a time variable (t_{entity}) giving the contribution of that part of the program to the total execution time (for each time it is executed).

The possible flows given by the structure of the program are modeled using *structural constraints*. For each node, the sum of the incoming flows is equal to the outgoing flows.

In addition to the structural constraints, there are constraints given by the information about possible and impossible program flows. Some constraints are mandatory, like upper bounds on loops, while others will only tighten the final WCET estimate, like information on infeasible paths through the program.

The WCET estimate is generated by maximizing the sum of the products of the execution counts and execution times (subject to the flow constraints):

$$WCET = maximize\left(\sum_{\forall entity} x_{entity} \cdot t_{entity}\right)$$

This maximization problem can be solved using constraint solving or integer linear programming (ILP).

3.2. Path-based Calculation

The idea of a Path-based calculation method is to apply standard graph algorithms for finding longest paths in the control flow of a given program.

In our case, the basic path calculation deploys an algorithm for finding the k shortest paths as described in [2]. The algorithm works as follows: First, the longest path in the graph is sought using a standard longest path algorithm (e.g. Dijkstra's algorithm). This path is then checked for feasibility, and if it isn't feasible, it is excluded from the graph, and the search begins again, now finding the second-longest path. This is repeated until a feasible path is found. The first feasible path found is the longest executable path in the flow graph and its length is the WCET of the program.

3.3. Main Characteristics

IPET does not find the worst-case execution path but just gives the worst-case count on each node. There is no information about the precise execution order. In contrast to this, the Path-based approach explicitly computes the longest executable path in the program. This may be a valuable information for the programmer, e.g. for tuning and debugging purposes.

The IPET method computes an execution count for each node in the control flow graph, i.e. how often it is executed in the worst case. In contrast to this, the Path-based method needs to know the execution counts of nodes *before* the path search starts, since a standard longest path search algorithm is not able to derive this kind of information. Therefore, it is necessary to collect this information before the actual path search, e.g. by first investigating several subpaths of the control flow (comparable with a Tree-based calculation), in order to find out how often a node can be executed considering all given flow information.

4. Comparing Calculation Methods

When investigating different calculation methods for WCET Analysis, this must be made with respect to criteria of a 'good' calculation method:

- *expressive*: A calculation method should be able to handle detailed flow information coming from the program flow analysis phase. Furthermore, it should be capable of handling all results coming from the low level analysis.
- *efficient*: It should have reasonable execution times and memory consumption.
- *safe and tight*: It should not add any pessimism by itself and the WCET estimate should not be unsafe if the low- and high-level analyses used are safe.

We are investigating how well the two proposed methods fulfil these demands, in particular their handling of program flow informations and their efficiency.

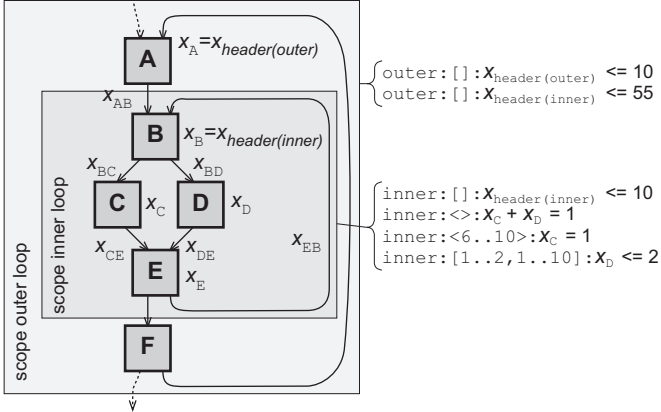


Figure 1. Scopes with attached Flow Facts

To be able to compare the two methods, we have to provide them with a variety of information about possible and impossible flows in the program. For this purpose, we use a flow fact specification language [4].

The control flow of the program is modeled as a *scope graph* (Figure 1). Intuitively, each scope corresponds to a certain repeating or differentiating execution environment in the program, like a function call or loop. Each node of the control flow graph belongs to exactly one scope, and represents the execution of a certain basic block in the program in the environment given by a scope and its superscopes. Edges represent the flow of the program between the basic blocks.

Each scope has as a set of associated *flow information facts*. Each flow information fact consists of three parts: the name of the *scope* where the fact is defined, a *context specifier*, and a *constraint expression*.

The context specifier specifies during what iterations the fact should be valid and how the constraint expression should be interpreted. We distinguish between facts that should be valid for all iterations of a scope or just for a range of iterations (e.g. 1..10). We can interpret the constraint expression as a constraint either on each single specified iteration (using $\langle \rangle$) or on all specified iterations together (using $[]$).

The constraint is given as a relation over count variables (x_{entity}) and integers, specifying constraints on the number of times corresponding nodes and edges can be taken.

The specification of this input language for the calculation methods implies that the given flow information first have to be converted in order to make them applicable for the respective method. Thus, we divide the calculation into two steps: Firstly a *conversion phase* which compiles the given flow information to a form suitable for the calculation, and secondly the *calculation phase* producing the final WCET estimate.

The first step mainly determines what types of flow information can be considered by the calculation

method, while in the second step a calculation method shows its computational complexity.

The following paragraphs describe the conversion phase of the IPET and the Path calculation method.

IPET conversion phase The basic idea for the conversion phase of the IPET approach is to introduce *virtual scopes* for different ranges in flow facts. Since a fact specified for a certain range is not valid for all iterations of a scope, we treat each range as a virtual scope in itself. Each virtual scope is represented by additional variables and constraints that are added to the input for the constraint solver.

For example, the two facts $s : [1..10] : x_A = 2$ (which means: Node A must, for each entry of scope s , execute exactly two times during the first ten iterations), and $s : \langle 11..30 \rangle : x_A \geq x_B$ (i.e. for each iteration 11 thru 30 of scope s an execution of B *implies* an execution of A. Node A can still get executed on its own) will create the two virtual scope variables $x_A^{s:1..10}$ and $x_A^{s:11..30}$, each holding the number of times node A is executed in the respective iteration space. If node A only can get executed during these iterations, the global IPET variable x_A (holding the total number of times that A gets executed) is equal to the sum of these two created virtual scope variables.

For a more detailed description of the flow representation language and the conversion phase of the IPET calculation, we refer to [4].

Path-based conversion phase For the Path-based analysis we use the same algorithm as in IPET to derive virtual scopes. However, in contrast to the IPET method, the virtual scopes are *explicitly* represented in the graph as scopes containing nodes and edges. E.g. for our two example facts we will produce two new scopes, $s : 1..10$ and $s : 11..30$, containing copies of the nodes and edges of the original scope s . This conversion prepares the program flow information in a natural way for the graph algorithms that will be applied in the calculation phase.

Flow facts are checked during the path search. For example, when a longest path is found, it is checked if it contains nodes that are mutually exclusive, like in $s : \langle i..j \rangle : x_A + x_B \leq 1$. If the path contains both nodes A and B, it is infeasible and thus the algorithm continues with the next-longest path as described in Section 3.2.

Clearly, not all kinds of flow facts can be handled efficiently by the respective calculation methods, and different types of facts have different influence on the complexity of the calculation step. We therefore propose the following structured way to make a fair comparison of the effectiveness and expressiveness of the two methods: We start by adding flow facts which just

give finiteness bounds on loops and successively add more complicated types of flow facts to the input in order to see how they affect the calculation phase:

- Add flow facts which introduce infeasible paths for all iterations of a scope, e.g. specifying mutually exclusive nodes (not creating virtual scopes).
- Add flow facts which introduce virtual scopes, but no constraints on feasible paths.
- Add combinations of the above, i.e. facts that let a virtual scope have different longest paths for different iterations. This means that a local search is necessary within each scope in order to find out how many times each path can be taken and to calculate a local WCET for each virtual scope.
- Add facts that affect several scopes. This implies that a local search is insufficient, and that it is necessary to consider several virtual scopes simultaneously.
- Add multi-dimensional facts (i.e. facts about loops with subloops [4]), and facts about nodes located in lower scopes.
- Add variability in the iteration count of a scope, and maybe other features as we discover them in tested programs.

For each step, we compare the complexity of the calculation. In order to measure the computational complexity, we collect the following information:

- The size of the generated virtual scope graph
- The size of the constraint system (i.e. no. of variables and no. of constraints)
- The total calculation time to find the WCET
- The no. of paths investigated by the path method

The precision of the calculations can be checked by comparing the WCET that is computed by the two methods.

Note that we consider the effects of the low-level analysis to be fixed, i.e. its influence on the calculation methods is the same for both the IPET- and the Path-based approach. As we focus on the differences of the calculation methods, we can e.g. assume that each basic block has a WCET of 1. Thus, the total WCET is found by just counting the number of nodes executed.

5. Conclusions and Future Work

We have proposed a strategy for objectively comparing two calculation methods with regard to their expressiveness of flow information and also their calculation power.

We are currently doing some measurements on the computational complexity of the IPET- and the Path-based calculation. We use several example programs

(some of which have been used by other WCET analysis research groups). We also use some large test programs in order to push the calculation methods to their limits.

For the Path-based approach, we are further investigating algorithms for performing local searches on parts of the scope graph, in order to derive the worst-case execution counts of nodes.

Furthermore, it would be interesting to find out what kinds of flow facts are really useful for real-time programs. Therefore, it is necessary to find out what kinds of flow information can actually be provided by designers or by automatically analysing the code. This could be achieved by doing further analysis of the programs collected during the MARE project [3], and new programs collected.

In the future, it might also be interesting to include comparisons with Tree-based calculation methods.

References

- [1] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Journal of Real-Time Systems*, May 2000.
- [2] Ernesto de Queiros Vieira Martins and Jose Luis Esteves dos Santos. A New Shortest Paths Ranking Algorithm. *Investigacao Operational*, 20(1):47–62, 2000.
- [3] J. Engblom. Static properties of embedded real-time programs, and their implications for worst-case execution time analysis. In *Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99)*. IEEE Computer Society Press, June 1999.
- [4] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. 21th IEEE Real-Time Systems Symposium (RTSS'00)*.
- [5] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.
- [6] C. Healy, R. Arnold, F. Müller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), January 1999.
- [7] Y-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proc. of the 32:nd Design Automation Conference*, pages 456–461, 1995.
- [8] S.-S. Lim, Y. H. Bae, C. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Ki. An accurate worst-case timing analysis for risc processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.
- [9] G. Ottosson and M. Sjödin. Worst-Case Execution Time Analysis for Modern Hardware Architectures. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, June 1997.
- [10] P. Puschner and A. Schedl. Computing maximum task execution times with linear programming techniques. Technical report, Technische Universität, Institut für Technische Informatik, Wien, April 1995.
- [11] F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.