A Tool Concept for Execution Time Analysis of Legacy Systems

Johan Erikson, Peter Funk, Jan Gustafsson and Björn Lisper

Mälardalen University, Department of Computer Engineering
P.O. Box 883, Västerås, Sweden
johan.erikson@mdh.se, peter.funk@mdh.se, jan.gustafsson@mdh.se, bjorn.lisper@mdh.se

Abstract

In industry, real-time requirements sometimes arise a long time after the system was initially developed. The systems can be large, and the code may be unstructured. This makes it hard to use existing execution time estimation methods and it may not be economically feasible to rewrite the code. These features of the code, together with the complexity of the systems, also complicate program comprehension.

In this paper we present a tool concept based on a novel combination of DJ graphs, execution time estimates and a graphical notation. The method enables analysis and visualization of the above mentioned systems and is intended to give programmers a good overview of execution paths and their corresponding execution time estimates. The analysis consists of loop identification and execution time estimation. A graphical viewer uses output from the compile-time analysis.

The tool concept has been partly implemented, and it has been used to analyse a part of a large telecommunication legacy system.

I. Introduction

In the development of new systems, real-time requirements can be considered already during the design phase, e.g. by using special real-time languages. But if the requirements arise in legacy systems, there are a number of possible solutions. For example to rewrite the system, remove unstructured code, add manual annotations and perform WCET analysis as proposed by e.g., [PS95, LM95], or to use ad hoc methods such as programmer guidelines.

Telecommunications systems are often legacy systems, and they tend to be too large to be rewritten or annotated by hand. A typical telecommunication system can have more than 10 million lines of code [HM01]. This is in stark contrast to embedded system codes, which are the usual candidates for execution time analysis.

Legacy code typically has to be maintained and updated to meet new functional requirements. When the code is large and unstructured it can be very hard to ensure also that non-functional properties, such as real-time properties, are met. Tools that help understanding the structure of the code and estimating its execution time are thus sorely needed.

In this paper we present a tool concept consisting of a number of components that enable execution time estimation and loop identification in legacy systems implemented with unstructured code[1]. We particularly target systems that do not allow a complete WCET calculation with current methods, due to unstructured code or size. The approach is not restricted to a particular hardware configuration since the analysis works on a higher level as shown in Section III-B (Execution Time Estimation). The tool concept consists of an analysis part, where loops in the code are identified in a hierarchical manner and given symbolic execution times parameterised in loop counts, and an interactive graphical environment which gives developers a hierarchical view over the system in terms of the control flow of the code, with the identified loops and their estimated execution times.

In Section II we discuss related work. Section III describes the different components and the approach enabling execution time estimation using a control flow graph produced by a compiler back-end as input. Section IV discusses future work. The paper is wrapped up in Section V with a summary and conclusions.

II. Related Work

In [HM01], code abstraction and reverse engineering is dealt with and the same target system as ours is explored, but execution time analysis is not addressed.

The graph-theoretical method, [SGL96], used in our framework has been discussed in [UM01] where it is explored as a tool for an optimising compiler to handle irreducibility. The method has also been compared with other methods to handle irreducibility [Ram00]. But to our knowledge, it has not been used in the context of WCET analysis.

The idea of showing execution time characteristics in a graphical environment and enable users to change between different abstraction levels has been presented by Pospischil, Puschner, Vrchoticky and Zainlinger in the Mars project [PPVZ92]. The main difference to the approach proposed in this paper is our need to handle unstructured code, a necessity in the class of systems we target.

[1] This research was partly funded by Ericsson AB and the Swedish Knowledge Foundation. The prototype tool components are implemented for the PLEX language used in the AXE telecommunication system. The AXE system and the language PLEX are introduced in [HM01].

Traditional WCET Methods

The basic methods how to calculate a safe and tight WCET using static analysis for imperative programming languages (like C) and simple hardware (like MC68000) was presented around 1990 [PK89, PS91]. The Timing Schema approach used in these methods assumes a structured program. It basically assigns a constant execution time for each atomic statement in the language. The WCET for a program is found by recursively combining these execution times in the language constructs.

In the presence of loops and recursion, finite loop or recursion bounds must be given to the Timing Schema method. Most often, they are given as manual annotations by the programmer. Optional annotations (like information on infeasible paths) may also be given, to reduce the over-estimation of the calculated WCET. The annotations can be written as comments in a special format or in a separate information file. A limitation of the annotation method is that correct annotations require good knowledge of the structure of the code. If the code is unstructured, with many jumps, then it can be hard to even identify the loops that are to be annotated with bounds. This is particularly true if legacy code is analysed.

During the 90-ies, WCET research has concentrated on the following areas:

- Flow analysis to replace manual annotations in the code with automatically calculated values for, e.g., loop bounds and infeasible paths.
- New methods that cope with modern, complex hardware properties like pipelines, caches, and branch prediction which introduce new difficulties in the WCET calculation.
- Different approaches to calculate the WCET for optimized code.
- Powerful methods, like ILP, for WCET calculation.

A recent overview of WCET research can be found in [Pusch00].

When Integer Linear Programming (ILP) was introduced as a tool for WCET calculation [PS95, LM95], also unstructured code could be handled. But still, the programmer had to provide some kind of manual annotations to bound the number of loop iterations. There is also a practical limit to how big codes can be to be analyzed in this way, since ILP is an NP-complete problem.

Unfortunately, in the targeted application domain it is economically infeasible to provide manual annotations for more than a few critical sections. Our aim is to do a kind of re-engineering of the code, present the loop structure to the programmer, and aid in estimating the execution time without requiring manual annotations.

III. Proposed Tool Concept

The tool concept comprises three components:

- Loop identification
- Execution time estimation
- Visualisation of results

In the current implementation, the loop identification and execution time estimation is an extension to the back-end of the PLEX compiler for the AXE system. They both use the internal control flow graph representation in the back-end. The graphical environment is a stand-alone component that uses the output from the extended compiler back-end and source code as shown in Fig. 1.

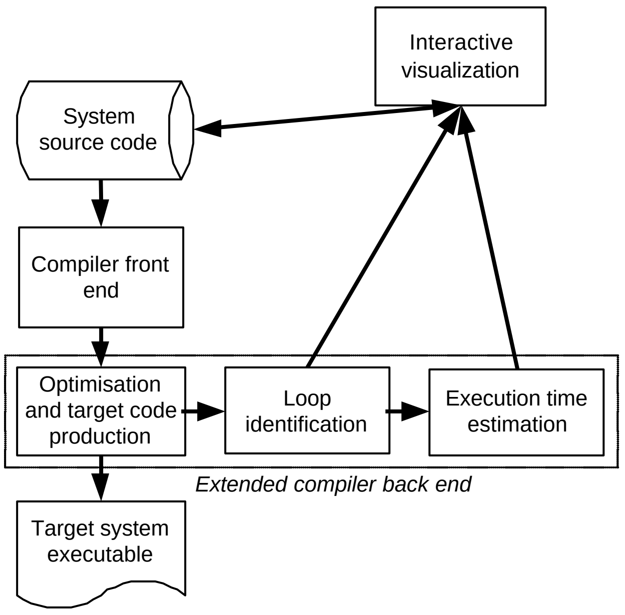
Interactive visualization
System source code
Compiler front end
Optimisation and target code production
Loop identification
Execution time estimation
Extended compiler back end
Target system executable

Fig. 1 *Organisation of the system.*

A. Loop Identification

The loop identification component identifies all loops in the system. This is achieved by using the control flow graph from the compiler back-end, which represents the true control flow in the resulting code, after possible compiler optimizations.

A limitation of all analyses based on static control flow graphs is that they cannot describe execution flows with dynamic jump addresses, like calls to subroutines and functions. However, if the language under consideration disallows recursion, then this limitation can always be overcome by inlining the control flow graphs for all subroutines at each call site, (at the expense of some code expansion).

The loop identification component uses *DJ graphs* [SGL96]. The advantage with DJ graphs is that they can describe irreducible loops. The method repeatedly collapses each identified loop into a single node. By this, the input to the second phase of the framework, the execution time estimation, is a "de-loopified" control flow graph with loops hierarchically hidden in single nodes. Fig. 2 illustrates the concept.

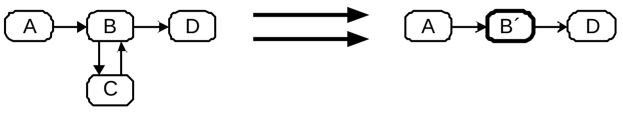

Fig. 2 *The identified loop B®C®B at the left side is represented by B´ (right side).*

This hierarchical hiding is also planned to be used in the graphical visualisation as an abstraction of the system, see section C.

B. Execution Time Estimation

The execution time estimation yields information about how the execution time varies with the maximal number of iterations for the loops in the analysed program/system.

The analysis does not take hardware aspects, such as caches and pipelines, into account even if these mainly affect the execution time. The reason is that programmers maintaining the target system claims that the formulas presented are valuable and sufficient information even if no exact WCET calculation is performed. But, as will be mentioned in Section IV (Future Work), these aspects could be considered at a later stage.

The control flow graph is traversed and minimum and maximum execution times are assigned to every node (i.e., to every basic block). In the current prototype, the execution time for a node is calculated in a rudimentary way:

- If the node corresponds to straight-line code, the minimum and maximum time will equal the number of statements in the basic block.
- If the node represents a collapsed loop, then the minimum time will be the time for the loop header (i.e., the original basic block), as calculated above. The maximum time will be the estimated time for the nodes that constitute the loop body. It will be presented as a formula with the execution time of one iteration multiplied with an unknown variable (i.e., the number of iterations).

C. The Graphical Environment

The task for the graphical component is to give the programmer visual information on execution paths, loops and execution time estimates[2]. A prototype is implemented which displays the "de-loopified" control flow graph and allows one level of "zoom", i.e. the user may select and open a node to see the possibly hidden nodes inside.

The programmer can use the graphical information and the estimated execution time information in a number of ways:

- To get an overview of the program structure to identify parts that needs optimisation.
- To see the symbolic formula for the execution time estimates for the basic blocks and the inner loops.
- To calculate execution time estimates, assuming the iteration counts are known.

IV. Future Work

We plan to evaluate our approach together with programmers to get an estimate of its efficiency and quality improvements. In order to do so, we have to:

- Extend the current prototype with a more detailed and fine-grained execution time estimation. Currently some coarse simplifications are made (e.g. in the case of if-then-else statements).
- Decide the graphical notation that is to be used and extend the implemented zoom function to handle several levels of abstraction.
- Perform a full integration of the execution time calculation component with the interactive graphical visualisation component (today a stand-alone Java application).

Possible extensions

Abstract interpretation as used by [Gus00] is an interesting extension since this method can calculate the maximum number of iterations automatically (i.e., the unknown variable mentioned in Section III-B).

Data flow analysis may be used to detect value dependent constraints [HW99]. These constraints make it possible to identify the maximum number of times a certain path can be executed in a loop, which can be used to increase the accuracy in execution time estimations for some loops. We see this as a possible extension to our approach.

[2] The development of the graphical environment is described in [AGG99].

Cache dependencies and processor pipelines have not been considered so far (as mentioned in Section III-B). An obvious extension is to take these issues into account to make the execution time estimation tighter.

V. Conclusion

In industry, real-time requirements sometimes arise a long time after the system is implemented. It is not always economically feasible to reengineer the systems to make them amenable to WCET analysis.
This paper presents a tool concept that offers a solution in providing basic execution time estimates without requiring reprogramming or source-code annotation.

A number of programmers maintaining the target system have been interviewed, and they claim that such a tool would enable large improvement on system quality and efficiency if used during the maintenance, of source code.

The method gives an overview of execution paths and the corresponding execution time. By using the internal control flow graphs from the compiler, all loops are identified and therefore the estimated execution time also takes compiler optimisations into account. The prototype has been used to explore a number of modules for a large telecommunication system with about 10 million lines of code and more than 1000 modules.

References

[AGG99] A. Arnström, C. Grosz and A. Guillemot. GRETA: a tool concept for validation and verification of signal based systems (e.g., written in PLEX). Master's thesis, Mälardalen University, Sweden, 1999.

[Gus00] J. Gustafsson. Analysing Execution-Time of Object-Oriented Programs Using Abstract Interpretation. PhD thesis, Uppsala University, Sweden, 2000.

[HM01] D. Herzberg and A. Marburger. E-CARES research project: understanding complex legacy telecommunication systems. In Proceedings of the IEEE Fifth European Conference on Software Maintenance and Reengineering, 2001. Also http://hobbes.informatik.rwthaachen.de/research/projects/ecares/Main.html

[HW99] C. Healy and D. Whalley. Tighter timing prediction by automatic detection and exploitation of value-dependent constraints. In Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium, 1999.

[LM95] Y-T.S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In Proceedings of the ACM Workshop on Languages, Compilers and Tools for Real-Time Systems, May 1995.

[PB00] P. Puschner and A. Burns. A Review of Worst-Case Execution-Time Analysis (Editorial). The Journal of Real-Time Systems, 18(2/3), 2000.

[PK89] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. The Journal of Real-Time Systems, 1(2):159-176, September 1989.

[PPVZ92] G. Pospischil, P. Puschner, A. Vrchoticky and R. Zainlinger. Developing Real-Time Tasks with Predictable Timing. IEEE Software, 9(5), 1992.

[PS91] C.Y. Park and A.C. Shaw. Experiments with a program timing tool based on a source-level timing schema. IEEE Computer, 24(5):48-57, 1991.

[PS95] P. Puschner and A. Schedl. Computing Maximum Task Execution Times with Linear Programming Techniques. Technische Universität, Institut für Technische Informatik, Wien, 1995.

[Ram00] G. Ramalingam. On loop, dominators, and dominance frontier. ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 233-241, 2000.

[SGL96] V.C. Sreedhar, G.R. Gao and Y-F Lee. Identifying Loops Using DJ Graphs. ACM Transactions on Programming Languages and Systems, 18(6), 1996.

[UM01] S. Unger and F. Mueller. Handling Irreducible Loops: Optimized Node Splitting vs. DJ Graphs. Technical Report, Humbolt-University, 2001.