Towards Resource Consumption-aware Programming

Andreas Holzer
Technische Universität Darmstadt
Formal Methods in Systems Engineering
Darmstadt, Germany
Email: holzer@forsyte.de

Visar Januzaj
Technische Universität Darmstadt
Formal Methods in Systems Engineering
Darmstadt, Germany
Email: januzaj@forsyte.de

Stefan Kugele
Technische Universität München
Institut für Informatik
Garching bei München, Germany
Email: kugele@in.tum.de

***Abstract*—In order to check the fulfilment of non-functional requirements at an early system design and development stage, we provide a framework that facilitates the combination of platform-independent and platform-specific information in a query-based manner to calculate estimates for the resource consumption of the software under investigation at fine grained levels of code. Based on an already optimised intermediate representation of the source code, using a testing infrastructure for C code, we count the occurrence of instructions during program executions in a platform-independent manner. These instruction counters can be determined at program or function level. By combining these counters with cost information of a hardware platform we can provide resource consumption estimates. This allows the software developer to tailor the code steadily towards the non-functional characteristics of the software.**

***Index Terms*—Code Instrumentation, Embedded Systems, Execution Time**

I. Introduction

Non-functional system properties, like execution time and power consumption, are of importance in many application areas. They range from computer games to safety-critical embedded real-time systems such as those heavily used in the automotive and avionics domain. During the last decade, the development of the latter system domains has been affected by the paradigm of model-driven development (MDD) which is gaining in importance. Nevertheless, in many cases source code for those systems is still either completely handwritten, due to contractual provision, or the code is generated in a MDD fashion and afterwards optimised by system engineers. Therefore, code inspection is still indispensable during the development process of safety-critical systems.

Unlike the rather easier understanding of functional aspects, it is usually hard for a software developer to get an intuition of the non-functional characteristic of source code. We want to improve this situation by providing a framework that facilitates an efficient estimation of the non-functional behaviour of the whole software system or even only of certain parts of it. In this manner, developers can be tool-supported right at the outset of the overall development process, by providing them with various estimation types such as performance estimations for a piece of code. Thus, real-time requirements can roughly be checked immediately at design time and therefore enable their traceability which is considered as a challenge in the automotive domain [1]. Given specific requirements, a software developer can *ask questions* about the code at hand by writing queries in an SQL-like fashion. The envisioned framework responds to such queries (questions) with a resource consumption estimate for the specified source code. In this way one can check whether the code of interest meets its non-functional requirements. Such queries can amount to complex interactions between functions reflecting, e. g., operating modes [2], [3] as used in embedded systems. By allowing to query for non-functional properties of specific parts of the software system, a systematic analysis of the impact of single code fragments on the non-functional behaviour of the overall system is possible.

Furthermore, the separate treatment of hardware and software in our approach opens the possibility to tailor the hardware and software in an easy way in order to meet the specified non-functional requirements. To achieve this goal, one can either optimise the software in order to meet the non-functional requirements or choose a different hardware platform or combine both choices. In this way our framework enables the reduction of the gap between requirements, design, and implementation of a system.

We introduce our framework in Section II. In Section III we describe related approaches. Finally, in Section IV we conclude our work and give some ideas for future work.

II. Framework

Figure 1 shows the skeleton of our framework. A user gives as input a *resource cost estimation (RCE) query*, e. g.,

```
ESTIMATE time
OF foo IN bar.c
ON PPC
```

Such a query refers to source code (function `foo` in source file `bar.c`), to a resource type (`time`) and the platform configuration (`PPC`) the software is considered to run on. For the given query a resource consumption estimate will be calculated. The *query processing* module analyses the query and retrieves the *resource cost information* for the given platform from the *resource cost repository*, cf. II-D. These resource cost information are essentially weights for variables that count the occurrence of instructions during a program execution. Thus,

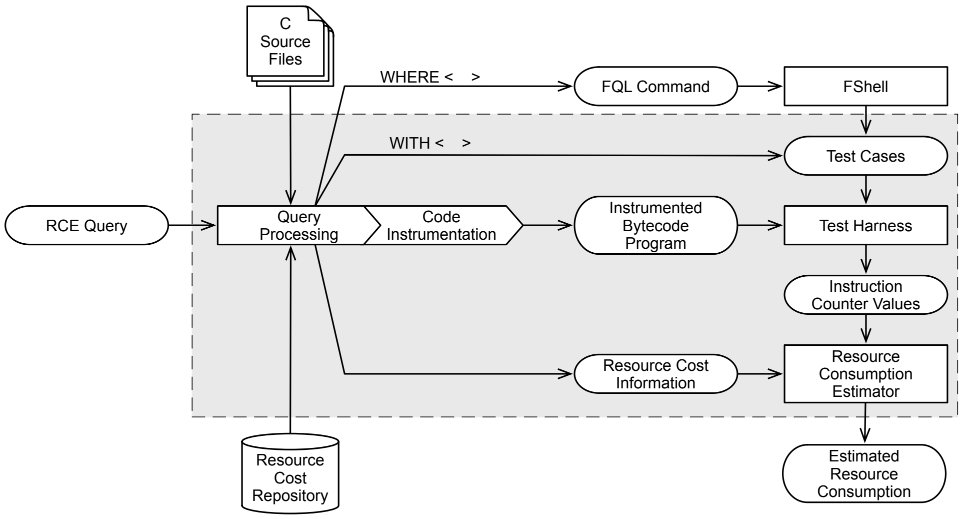
C Source Files
WHERE < >
FQL Command
FShell
WITH < >
Test Cases
RCE Query
Query Processing
Code Instrumentation
Instrumented Bytecode Program
Test Harness
Instruction Counter Values
Resource Cost Information
Resource Consumption Estimator
Resource Cost Repository
Estimated Resource Consumption

Figure 1. Resource Consumption Estimation Framework.

to calculate an estimate we have to determine the number of executions for each instruction type, e. g., `ADD`, `MULT`, `JMP`, and `CALL`. The distinction between instruction types is reasonable, since instructions of different classes contribute with varying weights to the overall non-functional behaviour. This means, that in the case of the timing behaviour of a program a division affects the runtime significantly more, i. e., its execution needs more cycles than an addition and therefore these instructions are considered separately. To determine the number of occurrences for each instruction type, counter variables are added to the source files that are used in the query. Every time an instruction is executed the corresponding counter variable is incremented. The values for the counter variables are then determined in a test-based manner, i. e., the test harness executes the instrumented program on some reference platform with test inputs. For this purpose, the test harness builds a wrapper around the code to ensure that the program can be executed with given test inputs, e. g., it creates a main function that calls a function *foo* with its necessary input data. Finally, by combining the instruction counters and the resource cost information of the designated target platform a *resource consumption estimate* is calculated. The time the program needs to run on the reference platform is irrelevant for the resource consumption estimate, since only the counter values are of interest.

A. Query Processing

An RCE query has the following form:

```
ESTIMATE <resource type>
OF <source range information>
ON <platform descriptor>
(WITH <test cases> | WHERE <constraints>)
```

The information given by the *resource type*, e. g., *time* or *power*, and the *platform descriptor*, e. g., *ARM* or *PPC*, are used to determine which information should be loaded from the resource cost repository. In order to run the test harness it is necessary to provide test cases. This can be done either by directly specifying *test cases* in the query using the `WITH` keyword or by giving *constraints* with the `WHERE` keyword. Both keywords cannot be used together and both are optional, i. e., in case neither `WITH` nor `WHERE` is given, the query is considered to have no constraints on the source code. In case no test cases are given directly, the *source range information*, e. g., *foo* `IN` *bar.c* or *line 5* `IN` *bar.c*, and the given constraints, e. g., *operation_mode* `==` *parking*, are used to form an FQL [4] command that is delegated to the FShell [5] test case generator, see II-C.

B. Code Instrumentation

We use the LLVM compiler framework [6] for *code instrumentation* to perform a translation of the source code into LLVM bytecode intermediate representation. This intermediate representation, compared to C, has a reduced RISC-like instruction set and its instructions are of a three-address-code form. LLVM offers a modified version of the gcc compiler that allows the inclusion of platform-independent optimisations during the translation step. Hence improving the resource consumption estimate by narrowing the gap between high-level C code and the instructions that are actually executed on the machine. On the LLVM bytecode level we introduce counter variables that keep track of the number of instruction executions for each instruction type. Currently, these counters can be recorded for each function invocation, i. e., the number of the executed instructions for each function run is tracked separately. The source code location, where the instrumentation has to take place, is derived from the source code fragment information given in an RCE query.

C. Testing Environment

There are two ways test cases are derived for executing the instrumented program. First, the user can give test cases directly as part of a query (`WITH`), and second, the query processing generates an FQL command (WHERE) for the FShell tool. FShell is an automatic test case generator for C code that can handle complex specifications of test suites. For example, one can ask for test cases that yield a certain execution order on function calls or can fix some variable values at some certain point of code. This allows to query for test cases that cover certain execution modes of the software under investigation. Given a test suite, a test harness executes the program (code fragment) for each test case and determines the counter values for the program (code fragment) execution.

D. Resource Consumption Estimation

Once the occurrences for each LLVM bytecode instruction type are determined, these counter values are then multiplied with the corresponding weight and summed up, similar to the approach presented in [7]. The weights are the resource cost information taken from the resource cost repository and are specific for each instruction type.

The resource cost repository can be filled in different ways: one way is to determine the resource cost information from datasheets [8]. This approach is practical for the timing (cycle) information of existing processors but remains problematic for other kinds of resources, e. g., power consumption, where such information is not available with respect to single instructions. Another way to get the resource cost information is by running benchmark programs on the hardware of interest, cf. [7], [8].

III. Related Work

Closest to the spirit of our approach is [7], where the LLVM framework is used to perform code instrumentation in order to extract the structure of the given source code and to calculate its estimated execution time, so called *software analysis*. In addition they perform a separate test-based *hardware analysis* by applying a number of benchmarks. These measurements are stored in the corresponding hardware profile. However, their main focus is the system deployment analysis. In contrast to their approach we facilitate a more integrated framework in order to automatically analyse specific software runs.

In [8] every basic block of some given source code is translated into values that count a corresponding number of executions of virtual processor instructions. Their virtual processor instruction set can have two different interpretations: one that accurately models a specific combination of a compiler and a CPU, which leads to accurate timing estimations, and a second one, that estimates the timing behaviour by roughly modelling the processor platform. The second interpretation is close to the approach with LLVM that we are currently using in our framework. They determine the timing estimation by executing the program on some platform retrieving the execution counts for different kinds of virtual instructions. The authors provide different methods to derive a timing estimate from the determined counter values. Either one takes cycle information for a virtual instruction from the processor data sheet, or one estimates the cycles from previously ran benchmark programs, as was also done in [7].

Wang et al. [9] describe a software performance estimation framework that is related to the work described in [7], [8] as well as to ours. For every instruction in the source code the number of cycles that are needed for executing the machine instructions that result from compiling the sources for some specific processor are determined. Then, the original source code is instrumented. In principle they introduce a counter variable that counts the number of cycles that are executed so far. For each source code instruction the corresponding increment is added to the counter variable. Next, they compile the instrumented sources together with a detailed microarchitecture model and run the program using test data. By this, they get an estimate of the number of cycles the program needs for execution. Their approach is limited to timing analysis and lacks an flexible querying mechanism. Furthermore, the increment for the counter variable for every instruction has to be determined for every compiler/processor combination.

IV. Conclusion and Future Work

We have presented a framework for query-driven resource consumption estimation. Its flexible querying mechanism facilitates an efficient tracing of requirements throughout the development process of the software. Therefor, for a given requirement, the software developer can select the corresponding piece of source code and a corresponding RCE source range information is generated fully automatically. This technique offers a very easy tracing functionality, which is desired by the industry. Being still under development, our framework cannot be automatically tested to the full extent. However, we prepared and manually ran some tests, since parts of the framework are already implemented. We can currently perform code instrumentation (cf. II-B), resource consumption estimation (cf. II-D) and establish a resource cost repository, as proposed in [7]. In order to be able to put together the whole tool chain of our framework, query processing (cf. II-A) and the testing environment (cf. II-C) need to get implemented.

Our framework can serve as a building block for a system deployment analysis, where the system analyst queries for timing estimates of a given software regarding different hardware platforms. In this way, one can define possible mappings between hardware and software. These mappings are then used for the calculation of a feasible system deployment, cf. [7].

The modified gcc compiler of the LLVM framework provides frontends for a multitude of languages and the extension of our framework to more languages would be a natural next step. The code instrumentation of our framework would remain unchanged, whereas only the test case generation and the test harness would be those parts that need to be extended and adapted to the new languages.

The envisioned framework is motivated by real-time constraints, such as deadlines, found in requirement specification documents of safety-critical embedded systems. Since the presented framework is designed to be extensible, we believe that our approach is also feasible for other application domains as well as for other non-functional properties, e. g., power consumption, which will be evaluated in more detail in future work.

In principle, the application of a static code analysis instead of a dynamic analysis, i. e., the analysis of all possible execution paths by looking at the source code instead of executing the software on a reference platform, is an interesting point for future work. This would completely decouple the resource estimation from actual program executions on hardware, enabling trade-offs between the precision of an estimate (a quick coarse static analysis versus a path-sensitive detailed analysis or actual program runs) and the calculation cost. For instance, at the beginning of the software development process a quick coarse estimation of the execution time would suffice to get an idea of how the program will behave on a chosen hardware platform, in order to be sure that the development moves towards a suitable end product. This would also support design decisions at an early stage of development which could lead to cost reductions by preventing wrong architectural decisions. The static analysis would replace the test case generation and test execution parts in our framework.

Using a more complex platform-independent and platform-specific model than counters and weights could improve the estimates that our framework produces.

References

[1] P. Braun, M. Broy, M. V. Cengarle, J. Philipps, W. Prenninger, A. Pretschner, M. Rappl, and R. Sandner, *The automotive CASE*. Wiley, 2003, pp. 211 – 228.

[2] A. Bauer, M. Broy, J. Romberg, B. Schätz, P. Braun, U. Freund, N. Mata, R. Sandner, and D. Ziegenbein, “AutoMoDe — Notations, Methods, and Tools for Model-Based Development of Automotive Software,” in *Proceedings of the SAE 2005 World Congress*. Detroit, MI, USA: Society of Automotive Engineers, April 2005.

[3] F. Maraninchi and Y. Rémond, “Mode-automata: a new domain-specific construct for the development of safe critical systems,” *Science of Computer Programming*, vol. 46, no. 3, pp. 219–254, 2003.

[4] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith, “Query-Driven Program Testing,” in *Proceedings of the Tenth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2009)*, ser. Lecture Notes in Computer Science, N. D. Jones and M. Müller-Olm, Eds., vol. 5403. Savannah, GA, USA: Springer, Jan. 2009, pp. 151–166.

[5] ——, “FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement,” in *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008)*, ser. Lecture Notes in Computer Science, vol. 5123. Princeton, NJ, USA: Springer, Jul. 2008, pp. 209–213.

[6] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *CGO ’04: Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2004, p. 75.

[7] V. Januzaj, R. Mauersberger, and F. Biechele, “Performance Modelling for Avionics Systems,” in *Proceedings of EUROCAST 2009, to appear*, 2009.

[8] P. Giusto, G. Martin, and E. Harcourt, “Reliable estimation of execution time of embedded software,” in *DATE ’01: Proceedings of the Conference on Design, Automation and Test in Europe*. Piscataway, NJ, USA: IEEE Press, 2001, pp. 580–589.

[9] Z. Wang, A. Sanchez, and A. Herkersdorf, “Scisim: a software performance estimation framework using source code instrumentation,” in *Proceedings of the 7th International Workshop on Software and Performance (WOSP ’08)*. New York, NY, USA: ACM, 2008, pp. 33–42.