CerCo

INFORMATION AND COMMUNICATION
TECHNOLOGIES
(ICT)
PROGRAMME

Project FP7-ICT-2009-C-243881 CerCo

Report n. D2.1
Addendum: survey of related work

Version 1.0

Main Author:
Nicolas Ayache

Project Acronym: CerCo
Project full title: Certified Complexity
Proposal/Contract no.: FP7-ICT-2009-C-243881 CerCo

The CerCo project addresses three different problems that are often considered separately in the literature: formal compiler certification (section 1), the determination of upper bounds on processors' execution time, also known as worst-case execution time (WCET, section 2), and the automatic inference of upper bounds for recurrence relations (section 3). In the following, we survey the state of the art of each of these problems, and we then relate them to the CerCo's project in section 4.

1 Formal compiler certification

Early work on formal compiler certification can be traced back to 1967. In [1], McCarthy presents the compilation of a language composed of arithmetic expressions to a simple assembly language, and gives formal evidence that this translation is *sound*, *i.e.*, that the generated assembly code indeed computes the result of the original arithmetic expression. The proof includes a formalisation of the semantics of the source and object languages and a related *simulation property* of the compilation function.

With the increasing number and complexity of critical embedded software, compiler verification has gained a great interest over the years (see [2] for a rather complete bibliography up to 2003), and a number of research projects and approaches are nowadays dedicated to this task.

Amongst them, many still follow McCarthy's approach to certified compilation. A notable example is the Piton project which relies upon the ACL2 (Boyer-Moore) prover [10]. It concerns the formalization of a series of compilers from high-level languages to an assembly language. Because, reasoning is conducted in a fragment of first-order logic, the formalization of the simulation property in ACL2 is based on a so-called *clock function* that, given a high-level program and a status, returns the number of low level steps required to simulate the execution of the next high level instruction. Interestingly, this is similar to the so called *direct approach* to certifying cost annotations that we describe in deliverable D2.1 and that we eventually reject because of its poor modularity properties.

More recently, Strecker [3] certifies a compiler from Java to bytecode in the Isabelle system proof assistant, and the CompCert project [4], led by Leroy, certifies a moderately optimising compiler from C to the PowerPc assembly language in the Coq proof assistant. In both cases, the challenge is to decompose the compilation in several elementary passes, where each pass is proved sound in a sense close to McCarthy's (but also taking into

consideration potential errors in the source program). Then the soundness of the whole compiler is obtained by the composition of the soundness of each pass; modularity is a crucial issue in the engineering of the proof.

Compiler certification is also a central task in the Verisoft project (see, *e.g.*, [9]) which aims at a so called *pervasive* verification of computer systems including the hardware and the operating system. Their results are based on the Isabelle/HOL proof assistant and concern the compiler for a fragment of the C language called C0 (close to Pascal).

Another trend in certified compilation is that of *translation validation* [5]: instead of proving once and for all that the translation is correct whatever the input program, a checker verifies *after* compilation that the produced code correctly implements the source program. For instance, Necula [6] applies this technique to an optimizing GNU C compiler. The formal certification of the validators has also been considered, *e.g.*, in [7].

Rival [8] considers a related approach where an invariant property on the source program is translated into a corresponding invariant property on the assembly code [8]. The translation is based on debugging information (a correspondence between variables and memory location, and a correspondence between some program points) and abstract interpretation. Then, abstract interpretation is used again to verify that the property on the assembly code indeed holds.

2 Worst-case execution time

Wilhelm et al. [11] provide a rather complete survey of the worst-case execution time (WCET) problem and its tools (up to 2008). Not only is the problem undecidable, but also, today's complex architectures are challenging in precisely predicting the execution time of a sequence of instructions, because of features such as caches [12, 13], pipelines [14], and branch prediction that may contribute to timing anomalies [15, 16]: the local worst case may not be the global one. There exist two main approaches to solving the WCET problem: static analysis and measurement-based tools (and sometimes hybrid). Static analysis tools intend to compute *safe* timing bounds, *i.e.*, an upper bound of the actual WCET, but at the cost of precision, and relying on a model of the architecture. On the other hand, dynamic tools are often able to find precise unsafe bounds, *i.e.* close but sometimes under-estimated, by measuring the executions on a concrete microprocessor. The WCET analysis problem is often broken down into two sub-problems:

data-dependent control flow (what path will be executed) and context of execution (state of the cache for instance).

Static analysis tools — which are clearly more relevant to *certified* cost bounds — all rely on more or less advanced techniques borrowed from abstract interpretation, the main issues being the abstraction of the architecture, *e.g.*, the cache and the replacement policy, and the computation of a bound for the number of iterations of each loop [17, 18]. Some tools that are dedicated to specific languages can infer these bounds automatically [19, 20], but tools for general purpose languages require the user to insert explicit bounds. In the cited work, Wilhelm *et al.* present several tools dedicated to WCET analysis, be they static or measurement-based. In the following we shall focus on two belonging to the first class which are representative of the state of the art and whose functionalities are of particular interest to us.

AbsInt **[21].** This commercial tool performs stack consumption and WCET analyses. It works at the assembly level by control flow graph (CFG) reconstruction and abstract interpretation, and targets several architectures. Also, it is able to consider complex architectural features, such as caches, pipelines, branch prediction. For WCET analysis, AbsInt is able to give tight upper bounds on the number of processor cycles needed to execute a program from its entry point.

AbsInt has been successfully integrated in the Scade tool suite [20], a model-based environment used to design synchronous systems. The integration consists in translating the high level model into C code (this is a feature of Scade), compiling it to assembly code, performing the WCET analysis and finally sending the result back to Scade where it is presented to the user. The efficiency of the integration comes from the properties of the generated C code from Scade; in particular, bounds on the number of iterations of loops are statically known, which greatly simplifies the analysis.

AbsInt can be used on generic C files, though it may face imprecision inherent to abstract interpretation. In this case, the kind of information that the WCET analysis needs for precision can be added manually through specification files.

Chronos **[22].** It is an open source tool for computing safe WCET of C programs. The analysis operates at the assembly level through CFG reconstruction and handles complex architectural features, such as caches, pipelines, and branch prediction. The tool builds a mapping between source

and assembly program points, estimates the cost of basic blocks of code, and then computes the WCET relying on *integer linear programming*. One of the strengths of Chronos, as claimed by the authors, is the fact that it targets the SimpleScalar [23] computer architecture simulator which is a widely used model of a processor with CPU, cache memory, and memory hierarchy.

3 Tools for automatic cost inference

Cost inference consists in statically determining an upper bound on the usage of some resource during program execution. Most analyses found in the literature operate on the source language (hence at a higher level than WCET analysis) and count some supposedly $O(1)$ operations such as assignments, function calls, memory allocations,... while ignoring the details of the compilation process. The techniques used to infer the costs vary from the analytic solution of recurrence equations to abstract interpretation. In the following we mention some examples which are representative of the state of the art.

The COSTA tool [27] is concerned with the analysis of Java code (actually of Java bytecode). COSTA is based on the classical approach to static cost analysis which consists of two phases. First, given a program and a description of the resource, the analysis produces cost relations, which are sets of recursive equations. Second, closed-form solutions are found, if possible and for this task a form of abstract interpretation is used.

Barthe and Pavlova [25] work on memory consumption annotations for Java. They define the Bytecode Specification Language, resembling JML, that allows to describe memory consumption properties in bytecode programs with annotations (comments) à la Hoare. Then, a weakest precondition calculus computes some verification conditions that, if discharged through automatic provers for instance, ensure that the program verifies its memory consumption specification. They also design an algorithm that automatically infers the memory consumption annotations in some simple cases. On the same topic, Cachera et al. [26] design a certified memory usage analyser in Coq for bytecode-like programs, that mixes abstract interpretation techniques and CFG-based loop detection.

The Speed tool [28] synthesizes complexity bounds for a Pascal-like programming language. Its strength appears to be the handling of loops for which the tool proposes various kinds of loop transformations and abstract

interpretation techniques that turn out to be effective even in the presence of *non-linear* operators (logarithm, exponential, square root, maximum, etc).

4 CerCo's approach

The CerCo project is an original combination of the problems and the techniques we have described.

CerCo vs. compiler certification CerCo appears to be the first large scale attempt at certifying the *intentional* properties of a compiler. To this end, it proposes a so called *labelling approach* to the certification of cost annotations which has very good modularity properties. It also proposes a refined approach called *dependent labelling* that can account for sophisticated optimisations.

Another original feature of the CerCo approach is the heavy exploitation of *dependent types* and *executable semantics* in the formalization. Small examples of compilers implemented using dependent types already exist, but the CerCo compiler is the first large-scale formalization to employ them. The combination of both techniques yields a new proving style ("Russell-style") where CerCo's developers simply write the code and the system opens relevant proof obligations. At the moment this approach is supported only by the Coq and Matita interactive theorem provers. Whilst support for the former is implemented in an external layer of the system, this style is implemented in the latter at the so-called *refiner level* and is therefore much more flexible.

CerCo vs. WCET From a formal certification viewpoint, the approaches to WCET described in section 2 are not quite satisfying. First, there is no formal certification of the fact that the abstract interpretation method does indeed produce correct results for a given processor. This in turn supposes a formal modelling of the processor and a proof that the abstract interpretation method does indeed abstracts the processor's behaviour. Second, there is no formal certification of the fact that the annotations on the loops are indeed correct. More generally, the relationship between the source code and the generated binary is not even formalised.

However, the most severe limitation seems to be that the WCET analysis are not *compositional* (at least in the case of AbsInt and Chronos tools we have considered) in that it is not possible to have a form of assume-guarantee

reasoning on the WCET of, say, a procedure. Instead, the whole program must be analysed at once.

For this reason, the CerCo project focuses on processors with a simple architecture for which manufacturers can provide accurate information on the execution cost of the binary instructions. In particular, CerCo's experiments are based on the 8051[1]. This is a widely popular 8-bits processor developed by Intel for use in embedded systems with no cache and no pipeline. An important characteristic of the processor is that its cost model is 'additive': the cost of a sequence of instructions is exactly the sum of the cost of each instruction.

An interesting benefit of this approach is that cost information on short sequences of instructions can be lifted to the level of the source C code and general purpose tools can be used to reason on them. Thus while AbsInt builds an abstract interpretation for each processor, we can just work with one abstract interpretation tool for the source C language. Clearly this avoids an annoying (but commercially justified) duplication of work and increases the trust we can have in the results.

CerCo vs. automatic cost inference Here CerCo's contribution is not in providing new abstract interpretation techniques or new methods for solving recurrence relations but rather in building an extensible infrastructure which can handle *realistic* C programs as found in embedded applications and *certify* the validity of the asserted bounds with respect to the *compiled code*.

In particular, the CerCo's approach builds upon the Frama − C software analyzers and it consists of a so-called plug-in which in first approximation takes the following actions: (1) it receives as input a C program, (2) it applies the CerCo compiler to produce a related C program with cost annotations, (3) it applies some heuristics based on abstract interpretation to produce a tentative bound on the cost of executing the C functions of the program as a function of the value of their parameters, (4) it calls an automatic tool (Jessie) to discharge the related proof obligations.

Unlike usual cost analysis tools which act as black boxes that may succeed or fail, the proposed approach provides the user with human readable information at the level of the source code and supports interactive theorem proving.

[1]The recently proposed ARM Cortex M series would be another obvious candidate.

References

[1] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Math. aspects of Comp. Sci. 1*, vol. 19 of Symp. in Appl. Math., AMS, 1967.

[2] M. A. Dave. Compiler verification: a bibliography. 2003

[3] Martin Strecker. Formal Verification of a Java Compiler in Isabelle. In *proceedings of the 18th International Conference on Automated Deduction (CADE)*. 2002

[4] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107-115, 2009.

[5] Amir Pnueli, Michael Siegel and Eli Singerman. Translation Validation. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. 1998

[6] George C. Necula. Translation validation for an optimizing compiler. In *Programming language design and implementation (PLDI)*. 2000

[7] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: a case study on instruction scheduling optimizations. In *Principles of programming languages (POPL)*. 2008

[8] Xavier Rival. Abstract Interpretation-Based Certification of Assembly Code. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2003.

[9] E. Alkassar, M. A. Hillebrand, D. C. Leinenbach, N. W. Schirmer, A. Storastin and A. Tsyban. Balancing the Load: Leveraging a Semantics Stack for Systems Verification. *Journal of Automated Reasoning: Special Issue on Operating Systems Verification*, 42(2-4) 2009.

[10] Matt Kaufmann and J Strother Moore. Design goals for ACL2. Technical report. 2000

[11] R. Wilhelm et al. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[12] Franck Mueller. Timing Analysis for Instruction Caches. In *Real-Time Syst.*, 18(2/3). 2000

[13] Jan Staschulat and and Rolf Ernst. Worst case timing analysis of input dependent data cache behavior. In *Euromicro Conference on Real-Time Systems (ECRTS)*. 2006

[14] Stephan Thesing Safe and precise WCET determination by abstract interpretation of pipeline models. PhD thesis. 2004

[15] Thomas Lundqvist and Per Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Real-Time Systems Symposium (RTSS)*. 2009

[16] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger and B. Becker A Definition and Classification of Timing Anomalies In *Worst-Case Execution Time Analisys (WCET)*. 2006

[17] C. Healy, M. Sjödin, V. Rustagi and D. Whalley. Bounding Loop Iterations for Timing Analysis. In *Real-Time Technology and Applications Symposium (RTAS)*. 1998

[18] C. Healy, M. Sjödin, V. Rustagi, D. Whalley and E. R. Van. Supporting timing analysis by automatic bounding of loop iterations. In *Real-Time Systems*, 18(2-3). 2000

[19] R. Kirner, R. Lang, G. Freiberger and P. Puschner. Fully Automatic Worst-Case Execution Time Analysis for Matlab/Simulink Models. In *Euromicro Conference on Real-Time Systems (ECRTS)*. 2002

[20] C. Ferdinand, R. Heckmann, T. Le Sergent, D. Lopes, B. Martin, X. Fornari, and F. Martin. Combining a high level design tool for safety-critical systems with a tool for WCET analysis of executables. In *Embedded Real Time Software (ERTS)*, 2008.

[21] AbsInt Angewandte Informatik. `http://www.absint.com/`.

[22] Xianfeng Li, Yun Liang, Tulika Mitra and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. Science of Computer Programming. 2007

[23] SimpleScalar LLC. `http://www.simplescalar.com`

[24] Karl Crary and Stephanie Weirich. Resource Bound Certification. ACM-POPL, 2000

[25] Gilles Barthe and Mariela Pavlova. Precise Analisys of Memory Consumption using Program Logics. In *Software Engineering and Formal Methods (SEFM)*, 2005.

[26] David Cachera, Thomas Jensen, David Pichardie and Gerardo Schneider. Certified Memory Usage Analysis. In *Formal Methods (FM)*, 2005

[27] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, D. Ramírez, G. Román and D. Zanardini. Termination and Cost Analysis with COSTA and its User Interfaces. *Electron. Notes Theor. Comput. Sci.*, 258(1) 2009.

[28] Sumit Gulwani. SPEED: symbolic complexity bound analysis. In *Computed Aided Verification CAV*. 2009