PROJECT PERIODIC REPORT

PUBLISHABLE SUMMARY

Grant Agreement number: 243381
Project acronym: CERCO
Project title: Certified Complexity
Funding Scheme: STREP
Date of latest version of Annex I against which the assessment will be made:

Periodic report: 1st ☐ 2nd ☐ 3rd X 4th ☐
Period covered: from 01 February 2012 to 31 March 2013

Project coordinator: Prof. Claudio Sacerdoti Coen
Alma Mater Studiorum – Università di Bologna
Tel: +39 051 2094973
Fax: +39 051 20 9 4510
E-mail: claudio.sacerdoticoen@unibo.it
Project website address: http://cerco.cs.unibo.it

1 Publishable summary

Project context and objectives

The CerCo project (Certified Complexity) aims to construct a formally verified complexity preserving compiler from a large C subset to some microcontroller machine language, of the kind traditionally used in embedded systems.

The work consists of the definition of cost models for the input and target languages, and the machine-checked proof of preservation of complexity (concrete, not asymptotic) along the compilation chain. The compiler will also return tight, certified cost annotations for the source program (expressed in terms of clock-cycles) for basic blocks. It is then up to the user, assisted by automatic tools, to use this (trusted) information to state and to prove precise complexity assertions on the program. The user can exploit tools for invariant generation and cost inference for imperative programs, with two additional benefits: a guarantee that the inferred intensional properties will carry over to machine code; and the adoption of a cost model that is absolutely precise, being induced from the generated object code.

The main focus of the current project is the certified, cost annotating compiler. We are also developing a prototype Frama-C plugin, to show how cost annotations can be exploited semi-automatically in an industrial tool.

The major breakthrough of the project is the generation of tight performance estimates for executable code (not relying on operating systems), a task that is currently regarded as highly visionary in the compiler community. The essential paradigm shift consists in creating the (certified) infrastructure allowing us to draw conclusions on the target code, whilst comfortably reasoning on the source.

The project also complements work done by the Worst Case Execution Time (WCET) community that focuses on techniques for computing approximate costs of object code programs for more complex architectures, using a blend of static and empirical analyses. The most sophisticated attempts try to relate estimated costs for the object code to the intermediate or source languages of programs, as we do. This is usually done with external tools that are not integral to the compiler and are not certified. On the contrary, we do cost inference as part of compilation, and will be fully certified. Combining the two methodologies will also be investigated as part of the project.

The compiler will be open source, and all proofs will be public domain. More information on CerCo can be found at the project website:

http://cerco.cs.unibo.it.

Work performed so far and main results achieved

First period During the first period of the project we focused on a methodology for building a compiler which can lift in a provably correct way information on the execution cost of the object code to cost annotations on the source code. We needed a clear and flexible picture of: (i) the meaning of cost annotations, (ii) the method to prove them sound and precise, and (iii) the way such proofs can be composed. After examining several approaches, we have devised one that we name the *labelling approach*.

In the labelling approach, we augment the operational semantics of source, intermediate and target languages to record traces of execution. The traces roughly record the number of times each basic block is executed. The proof shows that the traces are essentially preserved during compilation. The proof is quite symbolical, since no algebraic computation is performed on the traces. No approximation is introduced. Thus, any cost that is computed on the target code and that is associated only to a trace is automatically lifted to the source language without losing precision. The methodology has been applied to a toy compiler and partially verified with a proof assistant.

We have also developed a moderately optimising compiler from a large subset of C, providing two backends, one to the Mips assembly language and one to the MCS-51 8-bit object code. The compiler is written in O'Caml and uses a similar compilation chain to that of CompCert, which represents the state of the art in compiler certification, but whose proofs are not free software.

We have implemented and tested the labelling approach on top of this prototype compiler and we have produced experimental evidence that the labelling approach can handle a realistic C compiler with a moderate level of optimisation (comparable to level 1 optimisation of GCC).

The version of the compiler with the MCS-51 backend is the one that will be certified. The MCS-51, still manufactured *en masse* by a host of vendors, has a simple architecture, unencumbered by advanced features of modern processors, making it an ideal target for formalisation.

Second period In the second period the development of CerCo has followed three parallel routes.

The first route extended the methodology developed during the first period. In particular, the basic labelling approach suffered from two major limitations.

The first limitation is related to loop optimisations that may duplicate cost labels in the body of loops. The duplicated bodies are compiled and optimised separately. Therefore duplicated occurrences of cost labels corresponding to different loop iterations should be assigned different costs. By taking the maximum the upper bound is still safe, but the loss of accuracy is so severe to make the analysis useless.

The second limitation has different causes, but has the same symptoms. On modern processors a loss of precision occurs when applying the basic labelling approach. Due to caches, pipelines, speculative execution and so on, the cost of a basic block is a function of the status of the processor at the beginning of the block. In particular, every iteration of a loop can have a different cost. WCET techniques based on abstract interpretation can be used to associate the costs to loop iterations. The basic labelling approach forces us to assign a single cost to every iteration, ignoring the effects of the cache in the analysis.

To overcome both limitations at once, we have studied a more elaborate labelling approach where traces are a sequence of labels paired with indexes for each enclosing loop that identifies the iterations that emitted the label. It is thus possible to associate a different cost to the occurrence of labels as a function of the execution history. For example, the first iteration of a loop can be more costly than the following ones. The different costs are then amalgamated to a dependent cost expression that is presented to the user. We name this new technique the *dependent labelling approach*.

In order to gain confidence in the dependent labelling approach, we have extended the proof on paper for the toy compiler with two kinds of loop optimisations, peeling and unrolling. The untrusted prototype has been extended with the same optimisations and dependent costs.

Independently of the extension to dependent costs, we have extended the CerCo labelling approach described in D2.1 to a standard compilation chain from a higher-order functional language of the ML family to C. This shows that the approach is sufficiently general to be applied to higher-order programs whose concrete complexity is generally regarded as difficult to estimate.

The second route has consisted of the formalization of the prototype using the interactive theorem prover Matita, followed by the setup of infrastructure for the certification of the prototype itself. The aim is to prove in Matita both the preservation of the extensional semantics and the correctness of propagation of cost information. The formalization has also provided feedback to the untrusted compiler in terms of bugs discovered, algorithms simpler for proving correct and more efficient generated code. We currently have formal executable semantics for C, every intermediate language, an MCS-51 assembly language and MCS-51 object code. The proof of correctness for the optimising assembler, and the correspondence between statically predicted and the actual program costs for object code, is almost completed. The latter, however, only holds for programs that are well-behaved at run time. To characterize them we have developed a notion of 'structured traces' which preserves important structural information from the earlier stages of compilation necessary for the proofs about costs. We have finally ported to CompCert the proof of equivalence of our executable semantics for C, developed during the first project period, with the non-executable one provided by CompCert. Bugs in CompCert semantics were found and one of them hid also a bug in the CompCert compiler.

The last route has consisted of integrating the compiler with Frama-C. Frama-C is an open source and well-established platform for reasoning on C programs.

The proof obligations generated from Hoare-style assertions on C programs are passed to provers that try to discharge them automatically.

Frama-C has been designed to be extensible using plugins. We have provided a CerCo plugin that takes the following actions: (1) it receives as input a C program, (2) it applies the CerCo compiler that annotates the program with cost annotations, (3) it applies heuristics, based on abstract interpretation, to produce a tentative bound on the cost of executing a C function as a function of its parameters, (4) it calls the provers to discharge the proof obligations.

In order to assess the practical applicability of the plugin we have interfaced it with a standard compiler for the Lustre synchronous language. A Lustre program (a net of communicating nodes) is fed to the compiler and turned into C code that is passed to our plugin. Every node becomes a non-recursive, loop-free function that describes the activities of the node during each cycle. The worst-case cost certified by the plugin for the node functions is therefore the reaction time of the node, the most valuable non-functional property of a synchronous program. Several programs from the Lustre distribution have been certified completely automatically using our plugin.

Third period In the third period the development of CerCo has followed two parallel routes.

In the first route we have continued our studies on the dependent labelling approach. The final aim was to understand if it is possible to deal with modern architectures whose cost model is dependent on the state of hardware components (pipelines, caches, branch prediction units, etc.). The idea suggested by the labelling approach is that the cost of labelled blocks can be made parametric on some abstraction of the hardware state. The abstraction has to be reflected on the source code: the source code is instrumented with ad-hoc instructions that manipulate the hardware state, without affecting any functional property of the compiler. What we discovered during this period are certain conditions on the hardware that allow us to actually expose a limited amount of knowledge about the hardware state, without loosing precision in the process. In particular, pipelines seem to satisfy these conditions, while caches do not. In order to deal with caches, it seems that we could switch from deterministic to Statis Probabilistic Time Analysis (SPTA), as defined in the EU Project PROARTIS. Further research is required to understand to what extent we can still automatically work at the source level on the instrumented code without introducing major approximations. In any case, the methodologies currently adopted in standard WCET remain available at the source level.

In the second route we have started the certification in Matita of the prototype formalized during the second period. The proof of correctness turned out to be more complex than the one for the small imperative language that was done in the first period, and also partially certified using a proof assistant. The reason for the additional complexity is the presence of function calls that introduces two new critical aspects that we dealt with with just one solution.

The first is stack consumption: functions that nest function calls too deeply can run out of stack, breaking the simulation proof. Not running out of space is a property associated to program runs and not to static code. Therefore we needed to single out source level runs that are well behaved with respec to the object code stack usage model. To infer a cost model for stack usage on the source code from the object code model, we re-used our own technology --- the labelling approach --- by associating implicit labels to function calls and returns.

The second is proper nesting of function calls and returns: runs of low level languages need not respect the invariant that every terminating function call returns just after the call point. This property is necessary during static analysis of object code to associate costs to basic blocks that contain function calls.

To solve both issues, we introduced a new notion of semantics, based on structured traces of observables. Unlike normal traces of observables, a structured trace is well formed iff it captures a certain number of invariants like well nesting of function calls and bounded stack usage. We introduced a new notion of forward simulation for semantics based on structured traces and we proved the notion to be strong enough to imply the preservation of both functional and non functional properties. We also prove, using classical logic, that all traces produced by the front-end can be given the proper structure. Finally, we provided an infrastructure to take care of the additional details imposed by the structure in forward simulation proofs. The infrastructure applies to a specific compilation strategy that was actually used for most back-end passes. The proof obligations generated by the infrastructure are essentially the same that are met in a standard simulation proof that only considers functional properties.

To summarize, we have studied a new sets of concepts and techniques that allow to (almost) reduce correctness of a compiler that preserves both functional and non functional properties to the correctness of a compiler that only deals with the functional ones. Most results about this part have been fully certified using Matita.

What remains to be done to fully formalize the CerCo compiler is the completion of the proofs of functional simulation of several passes. Since the statements of these remaining proofs is now (almost) standard, the interest in these proofs is limited.

Expected final results

In spite of many recent, astonishing achievements in proof checking and program verification, the field of computer assisted reasoning, with its long term vision of a fully dependable, formally checked information society is still one of the most visionary fields of computer science. The impressive results obtained in this area are only partially due to the advancements in tools for assisted reasoning, but also to a growing confidence in their potential.

Compilers, filling the gap between humans and machines from the programming point of view, are one of the main tools at the base of information technologies. We cannot hope to build truly trustworthy, dependable and long-lasting systems on shaky foundations: the semantics of compilation must be better understood. There is some history of work in certified compilers, and a series of workshop meetings on the topic. Only very recently has end-to-end certification of a realistic compiler been attempted in the CompCert project, and several projects on this topic are starting to target more complex scenarios.

This previous work is focused on denotational aspects of computation. Our proposal is new in addressing intensional aspects, such as preserving space or time bounds of the source in compiled code. This is still perceived by the compiler community as a highly visionary goal, and would provide a major milestone in this area.

Our proposal contributes to the long-term community goal of formally checked reliable computer systems both directly, by our original work on a certified complexity preserving compiler, and more generally by large scale use of state-of-the-art tools and techniques for software certification. Our proposal takes such technologies past the proof-of-concept phase to a maturity level that could facilitate a substantial degree of technology transfer to industry.

The direct, long term, impact of the project on real-time systems (in particular reactive systems) will be that of dramatically increasing the trust of response times for the systems, while at the same time simplifying code by removing some of the checks for approaching deadlines. Moreover, we envision a future where compilers (certified or not) will give back to the user guarantees and information on the intensional behaviour of the produced binaries (like time, space and power consumption), to be exploited by semi-automated reasoning tools.

The certified compiler developed in the project is just a first step in this direction. Being the first example of a compiler that provides intensional guarantees, we do not expect to be able to immediately exploit cost annotations in an effective way. This requires an effective convergence of the compiler implementation, proof assistant and invariant generation communities. Our results in this directions are however promising. Moreover, further work is required to target processors for non-embedded systems (which implement many hardware optimisations). Together with the need for better understanding of the issues related to multiple backends for different architectures, the time-frame required to bring this technology into the mainstream can be estimated at around a decade.

It would certainly be interesting to address advanced features of general purpose processors such as cache memory. We are currently studying extensions of our approach that seem promising when combined with other current research trends, like the move to probabilistic analysis of complex systems. Another approach is the integration in certified compilers of the tools provided by the WCET community that, at the moment, are very powerful, but lack any sort of formal guarantee.

To summarize, the project responds to the increasing expectation for trustworthy, dependable and long-lasting systems by developing reliable compilers, not only from the point of view of behaviour but also of performance. Building certified compilers (combined with a machine checked proof of their semantics preservation) is an emerging topic whose relevance is destined to grow in coming years. It appears to be of high importance for the ICT industry to have methods and tools for producing certifiable systems, and automatic checking of complex invariants (such as preservation of complexity) is a critical step and a major scientific challenge.