PROJECT PERIODIC REPORT

PROJECT OBJECTIVES, PROGRESS AND ACHIEVEMENTS FOR THE PERIOD

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

Table of contents

1 Project objectives for the period

Overview

The two main objectives for the first period of the project have been:
1) the development of an untrusted prototype of a compiler from a large subset of C to the object code of a simple microprocessor, of the kind used in embedded systems (first milestone of the project). The prototype also instruments the source code with “ghost code” to keep track of the total and exact execution time (in clock cycles) of the program. The cost of each basic block is determined by keeping track of blocks during compilation, associating to each C block the cost of the corresponding object code produced.
2) the formalization in the Matita interactive theorem prover of the two executable semantics (emulators) for both the C subset we consider and the targeted microprocessor.

The two main objectives for the second period, as included in Annex I to the Grant Agreement, have been:
1) the development of a proof-of-concept prototype that interfaces the untrusted prototype compiler with extant tools for the certification of extensional properties of C programs. The new prototype must allow the user to manually enrich the instrumented source code with cost assertions and it must generate proof obligations to verify the assertions. Automatic and interactive theorem provers can then be used to formally prove all obligations. Automatic inference of cost assertions by means of abstract interpretation of the source code must also be tested. The prototype represents the second milestone of the project and the first practical tool suitable for dissemination.
2) the formalization in the Matita interactive theorem prover of the executable semantics for all intermediate languages and the rewriting of the untrusted CerCo prototype in Matita. This is the last preliminary step required to start the certification of the whole compiler, which will also start during the second period and to be completed at the end of the project.

The two main objectives for the third period, as included in Annex I to the Grant Agreement, have been:
1) the formalization in the Matita interactive theorem prover of the whole compiler
2) the refinement and assessment of techniques to automatically infer cost assertions for the source code in such a way that automatic provers can solve the proof obligations generated for the cost invariants
3) the development of a major case study for 2) consisting in the fully automatic cost analysis of reaction time for Lustre programs

Follow-up of previous review

During the second project review the reviewers have made three main recommendations. We report here the recommendations and describe our actions taken.

1) “The consortium should concentrate on positioning the project with respect to the state of the art which requires a detailed literature survey. In particular, the consortium should clearly formulate what distinguishes the project from existing work and which results are clear improvements.”

2) “The consortium should as soon as possible start with the verification of the compiler.” “The reviewers are worried that the whole verification tasks cannot be successfully completed by the end of the third project period.” “The consortium should as soon as possible start with the verification of the compiler. Integration of the simulation proofs for adjacent intermediate layers requires special efforts which should not be underestimated.”

3) “The project consortium should position the project with respect to the state of the art. It should investigate which results achieved in the project advance and improve the state-of-the-art. This will help publishing the project’s results in appropriate venues. To achieve this positioning, the reviewers recommend carrying out a detailed study of the existing related work. In particular, the references already mentioned in the previous review report should be considered. A literature survey (with technical details and at least 5 pages) should be submitted as revised addendum to D2.1”

4) “Deliverable D6.2 does not contain a specific and clear dissemination strategy or plan till the end of the project. The reviewers request the resubmission of D6.2 detailing an event targeting at the promoting the results of the CerCo project. It should be planned at least as a half-day event solely presenting the CerCo project. In order to attract an audience from academia as well as from industry the event should try to focus on case studies for the dependent labelling approach in more modern processor architectures, e.g., as applied in the embedded domain.”

5) “The consortium should develop a code extractor for MATITA which allows obtaining executable code from MATITA specifications such that an executable version of the trusted CerCo compiler can be automatically generated. This extraction will be crucial to apply the CerCo approach to practical case studies and to present it to industry.”

6) “The consortium should concentrate on further developing and publishing the dependent labelling approach. In particular, the approach should be extended to further compiler optimizations, apart from loop peeling and hoisting, and be applied to more modern processor architectures, eventually aiming at multi-level caches and pipelines.”

XXXXXXXXXXXXXXXXXXXX OLD STUFF XXXXXXXXXXXXXXX

1) “In order to not limit the results that can be obtained in the project to processors with very old architectures … the project partners should explicitly consider how the assumptions on the hardware architecture influence the results obtained and in how far these assumptions could be a threat to the validity and generality of the results.''

The basic labelling approach developed during the first period was based on the assumption that there exists a function that associates to each basic block its execution cost (in cycles). The assumption is clearly violated in modern processors where executution cost is a function of the state of the system (internal processor state plus cache) and different executions of the same code require a different number of cycles.

Let us assume the existence of a technique that associates different cost estimates to different executions. For instance, such a technique can be found in WCET tools that take cache effects into consideration: loops are virtually peeled or fully unrolled and each iteration is assigned a different cost via abstract interpretation. From the point of view of the basic labelling approach, the situation is identical to the one obtained via actual loop unrolling: the cost label for the loop body is duplicated many times and different copies are assigned different costs. The basic labelling approach can still compute a correct bound by picking the maximum of the different costs, but this is extremely imprecise since it amounts to ignoring the analysis and always assuming the worst cache configuration (all misses). The bounds obtained, then, are still valid, but so large as to be useless.

In order to solve this problem, we need to relax our assumptions and allow functions that associate to every cost label functions from an approximation of the program (processor) state to cost bounds. In particular we are interested in associating to bodies of loops a cost that is dependent on the number of loop iterations performed. This change surely permits us to accommodate loop optimizations performed by the compiler, and so represents a major improvement to the basic labelling approach. Moreover, we conjecture that it should be sufficient to accommodate also modern architectures with caches. During the second period we have worked on this idea producing: a) a new technique that extends the basic labelling approach that we call the *dependent labelling approach* since it yields dependent cost annotations; b) a pen-and-paper certification of a toy compiler for an IMP language extended with gotos and loop optimizations; c) a new version of the CerCo untrusted compiler that has dependent labels and loop optimizations; d) the integration of the new compiler in the larger CerCo prototype. In the next period we will try to apply the same methodology to consider caches by either simulating a cache at the software level or picking an MCS-51 variant with a cache.

“... we recommend to adapt the labelling approach such that basic complexity annotations can be obtained by program pieces of larger granularity than is done now. This will allow … the use of WCET tools to infer time bounds...”

The solution recommended by the reviewers is to take extant WCET tools as black boxes and use them to assign costs to larger program pieces, i.e. to program slices that contain loops. This is basically the approach used by the EmBounded EU Project where the compiler is responsible for informing the WCET component about high-level constraints on the control flow of the program, like bounds to loops or recursive function invocations. Internally, the WCET tool does a fine grained analysis where it computes upper bounds for the various loop executions under additional assumptions about the control flow, either provided by the user or by an external tool (the compiler in the EmBounded case). The result returned by the tool is less fine grained, usually being just a number (and not a function of the program state), and is not "trusted" in our sense as the tool is neither certified nor because the user provided assumptions are also not certified.

We would like to follow a different approach. We conjecture that, using our dependent labelling approach, it should be possible to export from a WCET tool and expose the details of the fine grained analysis performed internally. We are quite confident that this is the case for WCET tools for simple processors that have a cache, but no other modern features like speculative branching. To support this opinion, in the next period we will try to apply the same methodology to caches by either simulating a cache at software level or picking an MCS-51 variant with a cache.

We would also like to stress what we believe to be a significant disadvantage in using WCET as black box oracles for accurately estimating the cost of program pieces (w.r.t. the standard application to whole programs). State of the art WCET tools for complex microprocessor architectures, like AbsInt or Chronos,
do not permit the user to make analyses of code assuming a particular processor state. Rather, all analyses run from a fixed initial assumed state, like an empty cache or pipeline, that evolves as the analysis progresses. When the processor state is not the assumed one this leads to unsound predictions (underestimates) because of the "timing anomalies" phenomenon. One possible solution is to insert code that sets the processor state to the expected one before the code to be analyzed. However this approach affects the global performance of the code.

2 Work progress and achievements during the period

Progress overview and contribution to the research field

As clearly visible in the Pert diagram in Annex 1, the workplan for CerCo has been designed to maximize parallelism between two important activities. The first activity, performed in WP2 and WP5, consists of the development of a framework for the analysis of intensional properties of programs written in C. The important landmarks for this activity are:

1) the development of the untrusted cost annotating O'Caml compiler, to be later replaced with a trusted version that has the same interface;
2) the integration of the compiler into a larger framework that allows one to manage the machine-provided cost annotations and the human-provided cost invariants;
3) the integration within the framework of procedures to automate the trivial proof cases commonly found in proofs of complexity obligations;
4) the study of some use-cases

Landmark 1), which is also the first project milestone, was successfully attained during the first period of the project; Landmark 2) was successfully attained during the second period of the project; Landmarks 3) and 4) were successfully attained during the third period.

Landmark 1), which is also the project milestone M1, already is a significant achievement in the domain of compiler design. Indeed, this constitutes the first example of a compiler that is able to induce absolutely precise cost annotations on the source language. As far as we know, in the compiler construction literature there is no comparable work for comparative assessment.

With our approach we finally obtain a fully certified compiler that preserves both the extensional and intensional semantics of the program. However, at the end of the first period we paid the price of not being able to exploit all the optimizations of existing compilers. In particular, loop optimizations were prevented. During the second period we have dramatically enhanced the basic labelling approach so that it now accomodates some loop optimizations and we are now confident that the method scales to most optimizations in the literature. In the third period we have make a theoretical study on how the same technique used for loop optimizations can accommodate some modern processor features like pipelines and caches. The results are promizing for pipelines, while for caches we need further investigation to be performed in future work. Finally, 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.

Practical applications of CerCo are enabled by Landmark 2) that allows users to perform *parametric WCET*, i.e. to prove worst case execution times on functions as a function of the parameters of the function. This gives our tool a distinct flavour respect to standard WCET techniques that focus on producing a number representing the WCET of the function on every possible parameter. Compared to experimental tools for parametric WCET, the benefit of performing the analysis on the source language is that of potentially achieving more precision in the bound by means of more precise control flow analysis obtained from better knowledge about the functional invariants/variants of the high level code.

Moreover, the results of the plugin have a very high level of trust. Firstly, because the cost annotations added by CerCo will be granted to be correct when the certification of the compiler will be completed. Secondly, because verification of the proof obligations, which is performed comfortably on source code, is deductive and proof obligations can be discharged with various provers. The more provers that discharge an obligation, the more reliable we may view the result as being.

When automatic provers fail to discharge an obligation, the user can still try to verify them manually, with an interactive theorem prover such as Coq or Matita. This possibility is enabled by another peculiarity of the CerCo methodology: while other WCET tools act as black boxes, the cost plugin provides the user with as much information as it possibly can. When a WCET tool fails, the user generally has few hopes, if any, of understanding and resolving the issue in order to obtain a result. Contrarily, when the cost plugin fails to add an annotation, the user can still try to complete it. Further, since the output of CerCo is valid C code, it is also much easier to understand the behavior of the annotations.

The work completed on Landmark 4) yields a trusted and completely automatic WCET analyser for Lustre programs. It contributes evidence for the feasibility of our approach. We provided other evidence by using the CerCo plug-in to certify parametric time bounds of simple C programs taken from benchmarks that use nested loops and manipulate data structures via pointers. To deal with pointers, in the third period we have implemented a lightweight version of separation logic in the CerCo plug-in. In the second period we have also shown how the CerCo approach can be applied equally well to functional programming languages that have no implicit management of the memory. In the third period we have continued the study to also include implicit memory management. The solution we found is not based on garbage collectors, which are rather unpredictable, but on the insertion in the target language of predictable explicit dispose operations proved safe by a types-and-effect systems.

The second activity, performed in WP3 and WP4, consists of the formalisation of the core component of the framework, the cost annotating compiler itself. The important landmarks for this activity are:

1) the formalisation of the source language and the target architecture;
2) the formalisation of the intermediate languages used during compilation;
3) the rewriting of the untrusted compiler in Matita;
4) the certification of the rewritten compiler.

Landmark 1) has been successfully attained during the first period of the project and Landmarks 2) and 3) during the second period. Significant progress towards Landmark 4) has been achieved during the third period, but the formal proof has not been completed.

The part of the proof that we completed deals with the peculiarities of the CerCo compiler, which is the preservation of the cost model induced by the compilation itself. The first step to achieve this result has been the definition of a new semantics for programming languages that makes observable enough program structure to allow the inference of the cost model and the backward propagation of the model along the compilation chain. The first major result of the proof is that a forward simulation that respects all the observable structure allows to transfer any model from the target to the source language. The second major result is the correctness of the algorithm that compute the cost model on the object code: the cost prediction is correct only for those runs that respect the structure imposed by our new structured semantics. The third major result is that every run of a C program respects the structure. Therefore the cost prediction for C programs obtained using the model computed on object code produced by our compiler is always correct. The fourth major result is a reduction of the complexity of forward simulation for our new semantics: we prove with non trivial arguments that a traditional proof of forward simulation based on a standard semantics can be lifted to a forward simulation proof for our new semantics just proving a minimal amount of simple additional obligations. Therefore the only part of the proof that still needs to be completed is rather standard and expected to give no new major insights.

In the literature there already exists several formalisations of assemblers and compilers. We diverge from the existing literature in two ways. First, as far as we know we will formalize the first compiler that infers and preserve intensional properties of high level programs parameterized on the cost model. The closest existing work is represented by the Piton project, which formalized in ACL2 a series of compilers for high-level languages to an assembly language. In order to formalize the forward simulation in ACL2, the authors needed to define a *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. Clock functions are necessary in ACL2 to even state the forward simulation theorem and they are not meant to be presented to the user for high level reasoning on the code. In particular, the code of the functions, if presented to the user, could be hardly understandable in presence of significant optimizations. Moreover, proofs done using clock functions are menageable with a proof assistant, but not on paper, while our methodology is. Moreover, clock functions measure time only and extending the methodology to deal with other costs, like space or energy consumption, does not amount to a change of a single function, like in our approach.

The second reason for divergence from the literature is the heavy exploitation of dependent types and executable semantics in the formalization. Small examples of compilers implemented using dependent types already exist, but ours is the first large-scale formalization to employ them. Executable semantics are heavily used in ACL2 to prove compiler correctness, but not in combination with dependent types. The combination of both techniques at once yields a totally new proving style (“Russell-style”) where the user simply writes 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 Matita at the refiner level and is therefore much more flexible.

As expected, during the first formalization steps we have already faced some weaknesses in Matita's support for this style of development, and we have modified Matita accordingly. A better understanding of the methodology, and the requisite improvements to the interactive theorem prover are valuable side-effects of CerCo. A final comparison between the proof style adopted and the traditional one used, for instance, in CompCert is an interesting by-product of the project that will be possible only after completion of the whole formalization.

In addition to the two activities described in the workplan, we have started in the second period a third activity to generalize the methodology of CerCo to cover more complex scenarios and get closer to industrial applications. In particular, we have been interested into optimizations that do not preserve the control flow of the program and into modern hardware components that make the cost of execution sensitive to the state of the components. An example of such optimizations are loop optimizations and an example of such components are pipelines, caches and branch predicting units. In both cases the effect is that it is no longer possible to assign a constant cost to a block in the high level language. In the case of pipelines, for example, the cost of a block depends on the state of the pipeline at block entry, which in turn depends on the execution history. In the case of a loop unfolding, for example, the cost of even and odds loop executions is different, because there is no longer a bijection between the object code executed and the corresponding source code. During the second and third period we have shown how it is possible to replace constant costs of blocks with costs that are parametric on a *view* of the state of the high level program. The proposed solution requires minimal changes to the compiler and to its proof of correctness: all the complexity is confined to the static analysis performed on the object code and to the instrumentation phase that injects at the beginning of every source code block additional code to update the view. The solution is fully satisfactory for loop optimizations and we also provided an implementation. In the case of execution history dependent hardware components, the situation is less clear. We have a full solution that works in theory for pipelines, but we have not done an implementation yet to verify if working automatically with the parametric costs generated is practical. For caches, we know that we need to combine our technique with Static Probabilistic Time Analysis in order to obtain manageable views and we have proved that the move to probabilistic analysis has again no impact on the compiler and its certification. Further studies are necessary to understand if the obtained solution is satisfactory or if we need something stronger to further reduce the complexity of views management and cost invariants for caches.

Work packages progress

WP2: Untrusted compiler prototype

The goal of this Work Package was to implement a proof-of-concept prototype for the cost annotating compiler. The untrusted prototype compiler has driven the design and implementation of the trusted version, and at the same time has allowed us to experiment with the management of cost annotations, the declaration of complexity assertions, the generation of complexity obligations and their interactive solution (tasks covered by WP5).

During the third period no tasks have been active for WP2. Bug fixing and minor code improvements have been nevertheless performed throughout the project.

We do not deviate from Annex 1 for WP2.

WP3: Verified Compiler – Front End

The goal of this Work Package is to build the trusted version of the compiler front-end, from some abstract syntax tree representation of (a large subset of) the C language to three-address like intermediate code.

During the third period only Task 3.4 has been active.

Task 3.4 is the certification of the compiler front-end.

The first difficulty has been to understand the exact statement we were interested in proving. Real time programs are often reactive programs that loop forever responding to events (inputs) by performing some computation followed by some action (output) and the return to the initial state. For looping programs the overall execution time does not make sense. The same happens for reactive programs that spend an unpredictable amount of time in I/O. What is interesting is the *reaction time* that measure the time spent between I/O events. Moreover, we are interested in ruling out programs that crash running out of space on a certain input.

The solution we found is based on the notion of *measurability*. We say that a run has a measurable sub-run when both the prefix of the sub-run and the sub-run do not exhaust the stack space, the number of function calls and returns in the sub-run is the same, and the sub-run starts with a label emission statement and ends with a return or another label emission statements. The stack usage is estimated using the *stack usage model* that is computed by the compiler.

What we prove is that for each C run with a measurable sub-run there exists an object code run with a sub-run such that the observables of the pairs of prefixes and sub-runs are the same and the time spent by the object code in the sub-run is the same as the one predicted on the source code using the *time cost model* generated by the compiler.

The main idea of the proof consists in tracing the evolution of measurable sub-runs during compilation. The proof for a While language developed in WP2 during the first period did not scale to a language with function calls. In particular, we found no way to statically predict the cost of basic blocks of generic object code programs that contain function calls. The solution found at the end of the second period consisted in proposing a new style for describing semantics of programs. We call it the *structured traces semantics*. A structured traces semantics maps programs to structured traces, that are streams of observables where additional structure is imposed on the stream. For example, we impose that every converging function call should return immediately after the call itself, which is not generally true for every run of an object code program. For example, programs that accidentally corrupt the stack do not satisfy the property.

The front-end correctness proof is made of two parts. The first one proves that the the source code instrumented with label emission statements in proper places simulates the original program and respects some syntactic invariants. This part of the proof has been completed in the third period.

The second part proves that every measurable sub-run of a C program admits an RTLabs structured trace (RTLabs is the last language of the front-end and the first one of the back-end). Moreover, the sub-run and the structure trace have the same observables and the same happens for their prefixes. The proof consists in a standard forward simulation argument from C to RTLabs, paired with a new proof of propagation of syntactic invariants from the C to the RTLabs program. Finally, there is a proof that every measurable sub-run of an RTLabs program that satisfy the invariants admits a structured trace.

The existence of structured traces, that is peculiar of our approach, has been completed in the third period. The proof of propagation of invariants has been replaced with a certified procedure that checks if the invariants are respected on the RTLabs code, aborting compilation if this is not the case. Only the forward simulation proof, which is not novel and pretty standard, has not been completed yet.

An additional result, not planned in the description of work, consisted of showing the equivalence of our executable semantics for C with the non-executable one developed in CompCert and ported to Matita. The aim of this additional proof was to raise our confidence in the C semantics: bugs in the semantics may hide bugs in the compiler. A very surprising discovery had been bugs in the CompCert semantics, one of which actually masking a bug in the CompCert compiler itself. This represents evidence of the benefits of an executable semantics that can be tested by running it on actual programs. Contrary to CompCert, all the semantics in use in CerCo are executable.

We deviated from Annex 1 for WP3 in non completing the only part of the proof that is standard.

WP4: Verified Compiler – Back End

The goal of this Work Package is to build the trusted version of the compiler back-end, from intermediate three address code to object code language.

During the second period only Task 4.4 has been active.

Task 4.4, is the certification of the compiler back-end.

The compiler back-end proof is made of several independent results: 1) there exists a notion of similarity among structured traces that induces a notion of formal simulation: a program S is simulated by a program T when every structured trace of S is simulated by a similar structured trace of T; 2) for each compiler pass there exists a forward simulation (in the previous sense) between the source code and the target code of the pass; 3) a proof of forward simulation implies that the source and target runs that admit a structured trace emit the same labelled trace (sequences of label emissions or function calls/returns); 4) the proof that execution cost of a run associated to a structured trace executes with a certain cost iff that cost is the sum of the costs statically associated to each label occurring in the labelled trace.

Because of 4), the time cost model computed by the back-end is correct for the object code. Because of 2) combined with 3) and the front-end proof, the same cost model is also correct for the C source code. The proof of 2) also establishes correctness of the stack cost model computed by the back-end.

The proofs 1), 3) and 4) are peculiar of our approach and they have been completed.

The forward simulations proof of 2) are non standard since they involve the consumption and construction of structured traces, mixing together intensional and extensional reasoning. We have provided and proved correct two proof layers that try to disentangle the two reasonings.

The first proof layer reduces a forward simulation between structured traces to a series of local conditions that ask, for each execution step in the source code, to show the existence of structured traces fragments that have some properties. Proving the local conditions is much simpler than proving the existence of the entire trace, and the proof layer is so generic to be reused to save code for every simulation argument.

The second layer only works for those compiler passes whose source and target language are graph languages. All complex passes (parameter passing implementation, register allocation, etc.) have this shape. The layer reduces the local proof obligations to simpler ones that essentially ask the same local conditions of a traditional forward simulation proof, plus a very few and simple syntactic conditions on the locally generated code (e.g. the translation of a step that does not perform a call should not perform a call either). This complex second layer, that alone saves about 3000 proof lines per pass, was made possible by our uniform representation of back-end languages. Contrarily to standard practice, in Task 4.2 we defined a common parameterized syntax and semantics for all back-end languages (but assembly and object code) and a common parametric translations between any two back-end language instances. The common representation allows to experiment more quickly with additional passes and to factorize more code reducing debugging and proving time. It also allows to make some passes more generic and potentially more easily swappable. For example, the LTL to LIN pass of the untrusted prototype has been replaced by a generic pass that takes any graph based intermediate language and generates a linear representation for it serializing the graph and introducing explicit GOTO statements to represent backward graph links.

The two proof layers peculiar of our approach have been certified in Matita. They reduce the simulation proofs on structured traces to pretty standard simulation proofs. The standard proofs have not been completed, with the exception of the linearization proof (to assess the advantages of our generic representation) and of the proof of correctness of our optimizing assembly. In particular, this is the first formal proof of an assembler that implements branch displacement, which is known to be an error prone optimization.

Note that, comparing with the proof methodology presented in D2.1, we are dropping the proof that shows that compilation commutes with erasure of labels. That proof only provides more information on the actual behaviour of the compiler by showing that insertion of cost labels has no ultimate effect on the object code. At the C level it is already possible to easily prove that labels do not affect the extensional properties and, as far as intensional properties are concerned, the additional insight gained is not worth the effort of a formal proof. Moreover, it also rejects some potentially useful compiler behaviours, like the possibility of compiling the same code twice, with and without enabling an optimization, to compare the cost associated to cost labels and decide if the optimization actually improves the execution time

We deviated from Annex 1 for WP4 in non completing the only parts of the proof that are standard.

WP5: Interfaces and Interactive components

The aim of WP5 is to develop a proof of concept prototype, by interfacing with extant tools, to show how the annotations produced by the compiler can be exploited in order to draw complexity assertions on the execution time of the program.

During the second period the only active task should have been T5.1. However, we moved Task T6.1 forward from the third to the second period.

Task 5.1 is devoted to the management of the cost annotations (produced by the compiler) and the complexity assertions (added by the user or synthesized automatically by an abstract interpretation algorithm) in order to produce the right complexity obligations, that is the goals to be proved in order to check the correctness of the assertions.

The most significant result, that is also the second CerCo milestone, has been the development of a plugin for the Frama-C open source platform to reason on C programs using Hoare logic. The plugin, to a 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 an abstract interpretation analysis to produce a tentative bound on the cost of executing a C function as a function of the value of its parameters;

(4) it calls the provers embedded in the Frama−C tool to discharge the related proof obligations.

Like most WCET tools, the aim of the plugin is to provide a bound for the worst case execution time of a function. The novelty, compared to standard WCET tools, are: 1) the automatic generation of the bound of a function is a function of the value of the parameters and not simply a number; 2) the bound is formally proved to be correct. The trusted code base for point 2) is constituted by: a) the CerCo compiler that is untrusted at the moment but will be replaced by the certified one at the end of the project; b) the theorem provers called by Frama-C; however it is possible to obtain two independent validations from two different provers; c) the Frama-C plugin that generates the proof obligations from the complexity assertions. The plugin is described in D5.1.

Task 5.3 is aimed at delimiting the practical applicability of the plugin developed in T5.1. To this end, the tool has been applied to the C code generated by the Lustre compiler and to some other simple C programs.

Lustre is a synchronous language where reactive systems are described by the flow of values. It is provided with a compiler that transforms a Lustre node (any part of or the whole system) into a C step function that represents one synchronous cycle of the node. A WCET for the step function is thus the worst case reaction time for the component, the most valuable non-functional property of a synchronous language. The generated C step function neither contains loops nor is recursive, which makes it particularly well suited for the use case: the complexity proof obligations generated by Frama-C are simple enough to be automatically solved by the theorem provers integrated in Frama-C without any human intervention.

The most significant result for T5.3 has been the development of another Frama-C plugin that, given a Lustre program, compiles it to C using the standard Lustre compiler and then interfaces with the previous cost plugin to fully automatically certify worst case reaction times for Lustre programs. The plugin has been successfully tested on some test programs from the Lustre distribution. The plugin code together with a description and some benchmarks has been released as deliverable D5.3.

Follow-up work for the first review, unplanned in Annex 1, has been the development of the *dependent labelling approach* also described in D5.1. The dependent labelling approach is an extension of the basic labelling approach that allows the basic blocks marked by cost labels to be assigned different costs during the execution of the program. In particular, the cost of a basic block is now a function of the number of iterations performed in any loop that surrounds the basic block. For example, the first iteration of a loop or all even iterations can be assigned a different cost. The different costs for a single label yield dependent cost annotations in the instrumented code.

The dependent labelling approach addresses two significant limitations of the basic labelling approach: the impossibility of implementing loop optimizations, and the impossibility of applying the existing CerCo methodology to modern processors that sport caches and speculative branching. The second one was the most severe critique raised by the reviewers.

The follow up work on dependent costs has impacted both WP2 and WP5. The pen-and-paper certification of a compiler for the toy language IMP, given in D2.1, has been extended to cover gotos and loop optimizations. A fork of the untrusted compiler has been made to scale the dependent labelling approach to the realistic C compiler. As a side effect, code generation has been made more effective and the changes unrelated to dependent costs have been applied also to the main untrusted compiler. The forked compiler implements loop optimizations, a more aggressive constant propagation optimization and improved instruction selection. Finally, the plugin developed in D5.1 has also been forked to obtain a plugin for reasoning on dependent costs.

The follow up work is fully described in D5.1.

We deviate from Annex 1 for WP5 by moving Task 5.3 from the third to the second period and by implementing a consistent amount of follow-up work driven by the comments of the reviewers during the first Project Review.

WP6: Dissemination and exploitation

The overall objective of WP6 is to manage the knowledge generated by the project and IPRs, and to bring the technological advances developed within the CerCo project to the wider scientific community and potential users. The project will target not only the scientific and academic communities but also European industries potentially interested in applying formal verification techniques to embedded software design.

The specific objectives of WP6 have been:

(1) a tailored dissemination activity that will make use of specific dissemination mechanisms in order to reach the relevant communities;

(2) supervision of the entire project with regard to result applicability and the promotion of the exploitation.

Task 6.1, active in the third period, focused on user validation and exploitability. The CerCo prototypes were used in the validation of selected benchmarks to verify the degree of automation achieved in the certification of parametric time bounds for C program snippets. We report on the results in D6.3 (Final report on user validation), where we also present some speculations on the improvements of the CerCo technology in the middle and long terms. Task 6.2 was about the contribution to portfolio and concertation activities at FET-Open level.

During the third period, we also developed Debian packages and a Live CD (D6.3) for allowing people to easily download and experiment with the CerCo trusted and untrusted prototypes. Moreover, we organized two events respectively targeted to potential industrail stackeholders (D6.4) and to the scientific community (D6.5).

The dissemination activity performed in the second period is described in the Project management report.

3 Deliverables and milestones tables

Table 1. Deliverables

Del. No.	Deliverable name	Version	WP no.	Lead beneficiary	Nature	Dissemination level[1]	Delivery date from Annex I (proj month)	Actual / Forecast delivery date	Status	Contractual	Comments
D6.1	Project Web Site and Software Repository	1.0	6	UNIBO	P	PU	3	15/06/2010	Submitted	Yes	
D2.1	Compiler Design and Intermediate Languages	1.0	2	UPD	R	PU	6	10/09/2010	Submitted	Yes	
Addendum	Compiler	1.0	2	UPD	R	PU		16/05/201	Submitte	Yes	Required

1

PU = Public
PP = Restricted to other programme participants (including the Commission Services).
RE = Restricted to a group specified by the consortium (including the Commission Services).
CO = Confidential, only for members of the consortium (including the Commission Services).
EU restricted = Classified with the mention of the classification level restricted "EU Restricted"
EU confidential = Classified with the mention of the classification level confidential " EU Confidential "
EU secret = Classified with the mention of the classification level secret "EU Secret "

to D2.1	Design and Intermediate Languages							1	d		after the first project review
Addendum to D2.1	Compiler Design and Intermediate Languages	2.0	2	UPD	R	PU		25/05/2012	Submitted	Yes	Required after the second project review
D6.2	Plan for the use and dissemination of foreground	1.0	6	UNIBO	R	CO	6	10/09/2010	Submitted	Yes	
Addendum to D6.2	Plan for the use and dissemination of foreground	1.0	6	UNIBO	R	CO		16/05/2011	Submitted	Yes	Required after the first project review
Addendum to D6.2	Plan for the use and dissemination of foreground	1.0	6	UNIBO	R	CO		25/05/2012	Submitted	Yes	Required after second project review
D3.1	Executable Formal Semantics of	1.0	3	UEDIN	P	PU	10	16/12/2010	Submitted	Yes	

	C										
D4.1	Executable Formal Semantics of Machine Code	1.0	4	UNIBO	P	PU	10	16/12/2010	Submitted	Yes	
D2.2	Untrusted Cost-Annotating Ocaml compiler	1.0	2	UPD	P	PU	12	16/02/2011	Submitted	Yes	
D1.1	Periodic Activity Report and Financial Statements	1.0	1	UNIBO	R	CO	12	31/03/2011	Submitted	Yes	
D3.2	CIC encoding: Front-end	1.0	3	UEDIN	P	PU	18	23/10/2011	Submitted	Yes	
D3.3	Executable Formal Semantics of front-end intermediate languages	1.0	3	UEDIN	P	PU	18	23/10/2011	Submitted	Yes	
D4.2	CIC encoding Back-end	1.0	4	UNIBO	P	PU	18	23/10/2011	Submitted	Yes	
D4.2	CIC encoding	1.0	4	UNIBO	P	PU		25/05/201	Submitte	Yes	Required

	Back-end							2	d		after second project review
D4.3	Executable Formal Semantics of back-end intermediate languages	1.0	4	UNIBO	P	PU	18	23/10/2011	Submitted	Yes	
D4.3	Formal Semantics of back-end intermediate languages	1.0	4	UNIBO	P	PU		25/05/2013	Submitted	Yes	Required after second project review
D5.1	Untrusted CerCo Prototype	1.0	5	UPD	P	PU	24	16/02/2012	Submitted	Yes	
D1.2	Periodic Activity Report and Financial Statements	1.0	1	UNIBO	R	CO	24		Submitted	Yes	
D5.3	Case study: analysis of syncronous code	1.0	5	UPD	P	PU	29	30/04/2013	Submitted	Yes	

D6.6	Packages for Linux distributions and Live CD	1.0	6	UNIBO	P	PU	2	30/04/2013	Submitted	Yes	
D3.4	Front end Correctness Proofs	1.0	3	UEDIN	P	PU	41	30/04/2013	Submitted	Yes	
D4.4	Back-end Correctness Proofs	1.0	4	UNIBO	P	PU	25	30/04/2013	Submitted	Yes	
D5.2	Trusted CerCo Prototype	1.0	5	UPD	P	PU	7	30/04/2013	Submitted	Yes	
D6.3	Final Report on user validation	1.0	6	UNIBO	R	PU	2	30/04/2013	Submitted	Yes	
D6.4	Organization of an Event Targeted to Potential Industrial Stakeholders	1.0	6	UNIBO	O	PU	1	30/04/2013	Submitted	Yes	
D6.5	Organization of an Event Targeted to	1.0	6	UNIBO	O	PU	1	30/04/2013	Submitted	Yes	

	the Scientific Community										
D1.3	Periodic Activity Report and Financial Statements	1.0	1	UNIBO	R	CO	2	30/04/2013	Submitted	Yes	This document
D1.4	Final Report	1.0	1	UNIBO	R	CO	3	30/04/2013	Submitted	Yes	

PU = Public
PP = Restricted to other programme participants (including the Commission Services).
RE = Restricted to a group specified by the consortium (including the Commission Services).
CO = Confidential, only for members of the consortium (including the Commission Services).
EU restricted = Classified with the mention of the classification level restricted “EU Restricted”
EU confidential = Classified with the mention of the classification level confidential “ EU Confidential “
EU secret = Classified with the mention of the classification level secret “EU Secret “

Table 2. Milestones

Milestone no.	Milestone name	Work package no	Lead beneficiary	Delivery date from Annex	Achieved	Actual / Forecast achievement date	Comments
MS1	Untrusted Cost annotating compiler	2	UPD	31/01/2011	Yes	16/02/2011	
MS2	Untrusted CerCo Compiler	3,4,5	UPD	31/01/2012	Yes	16/02/2012	

MS3	Trusted CerCo Compiler	3,4,5	UPD	31/03/2013	Partially	30/04/2013	The software delivered is fully functional and tested. Its formal certification has not been completed, but we have tackled the part of the proofs that deals with the CerCo specific methodology. The missing parts are standard simulation proofs as already known in the literature.