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 **x** 3rd ☐ 4th ☐
Period covered: from 01 February 2011 to 31 January 2012

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.

Follow-up of previous review

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

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 execution 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) “The authors should quickly outline a paper-and-pencil correctness proof for each of the seven stages … in order to allow for an estimation of the complexity and time required”

We have completed the suggestion above by sketching the proof at a high level of detail, and we have also used data in existing compiler certification papers to estimate the number of man-months required to complete the project as a function of the lines of code to be certified and their estimated complexity. The exercise has given us some insights into the proof plan. In particular, we spotted a major problem (described below in the description of WP4) with the proof that relates the static cost prediction on object code with the actual execution cost. The solution has had an impact on the whole proof plan that has been updated accordingly.

3) “Based on the outline of the paper-and-pencil proofs, the authors should estimate the effort for two possible scenarios: a) proceed as planned in the Matita system (this involves porting all proofs of the CompCert project to Matita); b) modify the existing proofs in Coq... If the project partners stick to the usage of Matita they should argue convincingly why this is a suitable choice and in how far Matita is superior to Coq...”

We have argued in favour of the use of Matita in the Commitment Letter sent to the project officer and reviewers. The effort estimation has not changed the opinion expressed there since it did not demonstrate the infeasibility of the Project Plan presented in Annex I of the Contract Agreement. All the other motivations expressed in the letter still hold. More details are given in the conclusion part of the document that describes the compiler proof outline and estimates the required effort.

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 an untrusted 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;
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, and work on landmark 4) has been moved from the third period to the second one. Landmark 3) and 4) will be fully attained at the end of the project.

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. The same technique used for loop optimizations should also accommodate some modern processor features like caches. This needs further assessment in the third period of the project. 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 one 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.

Moreover, the results of the plugin have a very high level of trust. Firstly, because the cost annotations added by CerCo will be proved correct. 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.

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.

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.

Work packages progress

WP2: Untrusted compiler prototype

The goal of this Work Package is to implement a proof-of-concept prototype for the cost annotating compiler. The compiler will be untrusted, meaning that no proof will be given that the machine code and the cost annotations returned by the compiler are correct. It will be written in a high-level, ergonomic programming language particularly tailored to compiler construction (O'Caml).

The untrusted prototype compiler will drive the design and implementation of the trusted version, and at the same time will allow us to start experimenting 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 second period one task have been active: Task 2.4.

Task 2.4, not active in the first and third periods of the project, was for integration, validation and testing of the compiler developed in Deliverable D2.2. In particular, D2.2 has been modified to reflect changes to the code suggested during the formalisation. The task has also been exploited for the implementation of a realistic cost inference engine for MCS-51 and the implementation of tail recursive calls and calls via function pointers. These features were expected for D2.2, but not fully achieved during the first period. The integration of MCS-51 specific extensions, not originally planned in Annex 1, has been accomplished in a fork of the code. These extensions have been partially propagated to the Matita code. Their full integration and certification will be given a low priority to avoid delays in our schedule.

Finally, in WP2 and WP5, a fork of the code has been produced that also implements loop optimizations, extending the labelling approach to dependent labelling. Architectural changes and code improvements have been back-ported from the fork to the main branch, leading to improved code generation.

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 second period the active tasks have been: Task 3.2, Task 3.3, Task 3.4.

Task 3.2, active only in the second period, consisted of the formal rewriting of the (untrusted) compiler front-end in the Calculus of (Co)Inductive Constructions, using the Matita interactive theorem prover. A library of datastructures and generic algorithms to be used in common with the back-end formalization has also been designed. The code is directly executable inside Matita.

Task 3.3, also active only in the second period, consisted in the formalization in Matita of a directly executable semantics for the intermediate languages used in the front-end. This has allowed us to obtain a first validation of the formalized compiler by emulating some C programs after each compilation step, verifying that the semantics are preserved.

Task 3.4 is the certification of the compiler front-end. The two basic results to be proved are: 1) for each compiler pass, the existence of a forward simulation between the source code and the one obtained by applying the pass; the simulation must also show preservation of labelled traces; 2) the proof that the instrumented code and the source code behave the same way (up to the update of cost variables) and that, for converging programs, the final value of the cost variable is equal to the cost associated to the trace. An additional result, not planned in the description of work, consists 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 is to raise our confidence in the C semantics: bugs in the semantics may hide bugs in the compiler.

The task started on time at month 18 and focused first on the equivalence of the two semantics and on the set-up of the infrastructure for the simulation proof. A very surprising discovery has 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 will be executable.

A major problem with the operational semantics of the back-end will be discussed in the section devoted to WP5. It has led to the introduction of "structured traces", used to capture the execution runs of well-behaved back-end programs. Structured traces do not seem to be applicable to the front-end languages that lack a good notion of program counter. Since the certification of the back-end will be based on structured traces, an additional amount of unplanned work completed in Task 3.4 has been the correlation of the labelled semantics used in the front-end, based on flat traces, with the one used in the back-end, based on structured traces. Flat traces are correlated to structured traces during the RTLabs to RTL translation, that is the first pass of the back-end.

We do not deviate from Annex 1 for WP3.

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 assembly language.

During the second period the active tasks have been: Task 4.2, Task 4.3, Task 4.4.

Task 4.2, active only in the second period, consisted in the formal rewriting of the (untrusted) compiler back-end in the Calculus of (Co)Inductive Constructions, using the Matita interactive theorem prover. The back-end of the compiler is also comprised of a non-trivial assembler performing branch displacement optimizations. According to a now well-established approach, we have not formalized a few functions that are very difficult to prove correct, but whose results can be easily tested. The main drawback of this approach is that the code is no longer directly executable inside Matita, being based on a sort of oracle to be realized during code extraction. However, instruction selection, the first pass of the back-end, augments the size of the code so much that realistic programs are too large to enable direct execution inside Matita anyway. Therefore, the code has been validated only on artificial examples.

After submission of D4.2, we started the re-writing of some of the passes using improved sets of generic iterators over graphs, along with generic functions to translate a graph by expanding single nodes into linear chains of nodes. Simpler versions of the generic functions were used in D4.2. Moreover, during the re-writing we are also applying the patches, already applied to the untrusted prototype, that are required to accommodate dependent labels and loop optimizations. Loop optimizations will not be formalized in Matita due to time constraints, but we will try to formalize the architecture to support dependent labels.

Task 4.3, also active only in the second period, consisted of the formalization in Matita of a directly executable semantics for the intermediate languages used in the back-end. In the formalization we decided to depart significantly from the untrusted compiler by introducing an abstract definition of back-end language that can be instantiated to capture all the back-end languages used in CerCo. We discuss the advantages of this novel approach in deliverable D4.3. Amongst them: 1) it greatly simplifies the insertion of new intermediate back-end passes, in case one pass is too complex to be certified without splitting; 2) it permits some passes to be written in such a way that they can be freely permuted; 3) it should allow in the future to retarget the compiler by simply parameterizing the generic grammar to directly embed the target processor instructions. This would make all back-end passes but the first (instruction selection) generic in the target architecture while, at the moment in CompCert, this is not the case.

After submission of D4.3 some of the generic definitions have been re-organized to better accommodate the changes applied to D4.2 after submission. For example, the LTL to LIN pass of the compiler is now a fully generic pass from generic graph languages to a corresponding linear language. When applied to LTL it yields LIN, but it can now also be applied to previous languages (e.g. RTL) to get a linearized syntax for the programs. We are thus now free to commute between the two representations in case one makes the certification of the compiler easier.

Task 4.4, is the certification of the compiler back-end. The two basic results to be proved are: 1) for each compiler pass, the existence of a forward simulation between the source code and the one obtained by applying the pass; the simulation must also show preservation of labelled traces; 2) the proof that the labelled object code executes with a certain cost iff that cost is the sum of the costs statically associated to each label occurring in the execution trace. 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.

Task 4.4 has started at month 18. We focused first on the last part of the back-end: the certification of the optimizing assembler, and the proof that relates to the object code the static cost prediction with the dynamic one. The certification of the optimizing assembler is split into two separate proofs: 1) the correctness of the algorithm that determines a strategy for branch displacement optimization; 2) the correctness of the assembler that is parameterized over a correct strategy. As far as we know, the two proofs, not yet completed, will constitute the first formal certification of branch displacement algorithms, and are known in the literature as being particularly hard to prove correct (displacement algorithms are heuristics to approximate the solution of a problem that is NP-complete for many assembly languages).

Proofs about the object code, like the one about cost prediction, are extremely hard and time consuming. The main reasons are the following: 1) we reason on a realistic formalization of the microprocessor, where memory is limited and fetching an instruction may overflow the program counter even when execution of the instruction is correct and unaffected by the overflow (e.g. unconditional jumps); 2) in CISC architectures, different opcodes are represented in memory using a different number of bytes; therefore not every value of the program counter represents a valid opcode; misbehaving programs can even jump back in the code, re-decoding the code memory starting from a different address, so that each byte in memory is read as part of two distinct opcodes; 3) "function" calls in assembly, even if terminating, are not guaranteed to return to the instruction immediately following the call.

In order to address the difficulties above we have introduced the notions of well formed program and that of structured trace. Well formedness is used to statically recognize a subset of the object code programs that is also a superset of the image of the compilation and that are not affected by problems 1) and 2) listed above. Structured traces are used to describe the dynamic behaviour of programs that do not suffer from 3). A structured trace also grants termination of every function call. Termination plays an important role since, in the labelling approach, a label covers the cost of the instructions that follow the label up to the next label, even when the instructions are interrupted by a function call. Since the cost associated to a label is immediately paid when the label is encountered, we are effectively paying in advance also the cost of the instructions that follow function calls. If the function call is diverging or does not return control just after the calling point, then the paid cost is over-estimated. Precision can thus be obtained only in presence of terminating function calls. Using structured traces we have been able to prove that the dynamic and static cost predictions are perfectly related (are correct and precise) for well formed programs whose execution yields a structured trace.
Structured traces also incorporate cost labels in their definition, so that preservation of the structure of a structured trace during compiler passes implies preservation of cost traces as well. The forward simulation proof for the back-end will thus consist in demonstrating the preservation of structured traces.
Finally, we have also provided a coinductive notion of infinite structured traces for diverging programs. The structure of these traces clearly shows that it is still possible to provide precise bounds for the execution cost of converging structured subtraces, like bodies of diverging loops. An example application is the computation of precise cost bounds for the response time of a server.

We do not deviate from Annex 1 for WP4.

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 will be:

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, user validation and exploitability, was not active in the first period. Task 6.2 is about the contribution to portfolio and concertation activities at FET-Open level.

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 Design	1.0	2	UPD	R	PU		16/05/2011	Submitted	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	and Intermediate Languages										after the first 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
D3.1	Executable Formal Semantics of C	1.0	3	UEDIN	P	PU	10	16/12/2010	Submitted	Yes	
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	1.0	1	UNIBO	R	CO	12	31/03/2011	Submitted	Yes	

	Statements										
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	
D4.3	Executable Formal Semantics of back-end intermediate languages	1.0	4	UNIBO	P	PU	18	23/10/2011	Submitted	Yes	
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			Yes	This document
D5.3	Case study: analysis of syncronous code	1.0	5	UPD	P	PU	36	16/02/2012	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	