PROJECT FINAL REPORT

Grant Agreement number: 243381

Project acronym: CERCO

Project title: Certified Complexity

Funding Scheme: STREP

Period covered: from 01 February 2010 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

4.1 Final publishable summary report

4.1.1 Executive summary

Problem statement: Computer programs can be specified with both functional constraints (what a program must do) and non-functional constraints (e.g. what resources – time, space, etc – the program may use). In the current state of the art, functional properties are verified for high-level source code by combining user annotations (e.g. preconditions and invariants) with a multitude of automated analyses (invariant generators, type systems, abstract interpretation, theorem proving, etc.). By contrast, non-functional properties are generally checked on low-level object code, but also demand information about high-level functional behavior that must somehow be recreated. This situation presents several problems: 1) it can be hard to infer this high-level structure in the presence of compiler optimizations; 2) techniques working on object code are not useful in early development; yet problems detected later are more expensive to tackle; 3) parametric cost analysis is very hard: how can we reflect a cost that depends on the execution state (e.g. the value of a register or a carry bit) to a cost that the user can understand looking at source code?; 4) functional analysis performed only on object code leaves out any contribution from the programmer, giving results less precise than those from source code and reducing the precision of the cost estimates computed.

CerCo vision and approach: We propose a reconciliation of functional and non-functional analysis: to share information and perform both at the same time on source code. What has previously prevented this approach is lack of a uniform and precise cost model for high-level code: 1) each statement occurrence is compiled differently and optimizations may change control flow; 2) the cost of an object code instruction may depend on the runtime state of hardware components like pipelines and caches, which is not visible in the source code. To solve the issue, we envision a new generation of compilers able to keep track of program structure through compilation and optimisation, and able to exploit that information to define a cost model for source code that is precise, non-uniform, and accounts for runtime state. With such a source-level cost model we can reduce non-functional verification to the functional case and exploit the state of the art in automated high-level verification. The techniques previously used by WCET analysers on the object code are still available, but can now be coupled with additional source-level analysis. Where the approach produces precise cost models too complex to reason about, safe approximations can be used to trade complexity with precision. Finally, analysis on source code can be performed even during early development stages, when components have been specified but not yet implemented: source code modularity means that it is enough to specify the non-functional behavior of unimplemented components.

Contributions: We have developed a technique, the *labelling approach*, to implement compilers that induce cost models on source programs by very lightweight tracking of code changes through compilation. We have studied how to formally prove the correctness of compilers implementing the technique. We have implemented such a compiler from C to object binaries for the 8051 microcontroller, and verified it in an interactive theorem prover. We have implemented a Frama-C plug-in that invokes the compiler on a source program and uses this to generate invariants on the high-level source that correctly model low-level costs. Finally, the plug-in certifies that the program respects these costs by calling automated theorem provers, a new and innovative technique in the field of cost analysis. As a case study, we show how the plug-in can automatically compute and certify the exact reaction time of *Lustre* dataflow programs compiled into C.

4.1.2 Project context and objectives

Formal methods for verification of functional properties of programs have reached a level of maturity and automation that is allowing a slow but increasing adoption rate in production environments. For safety critical code, it is getting usual to combine rigorous software engineering methodologies and testing with static analysis in order to benefits from the strong points of every approach and mitigate the weaknesses. Particularly interesting are open frameworks for the combination of different formal methods, where the programs can be progressively specified and are continuously enriched with new safety guarantees: every method contributes knowledge (e.g. new invariants) that becomes an assumption for later analysis.

The scenario for the verification of non functional properties (time spent, memory used, energy consumed) is more bleak and the future seems to be getting even worse. Most industries verify that real time systems meets their deadlines simply measuring the time spent in many runs of the systems, computing the maximum time and adding an empirical safety margin, claiming the result to be a bound for the Worst Case Execution Time of the program. Formal methods and software to statically analyse the WCET of programs exist, but they often produce bounds that are too pessimistic to be useful. Recent advancements in hardware architectures is all focused on the improvement of the average case performance, not the predictability of the worst case. Execution time is getting more and more dependent from the execution history, that determines the internal state of hardware components like pipelines and caches. Multi-core processors and non uniform memory models are drastically reducing the possibility of performing static analysis in isolation, because programs are less and less time composable. Clock precise hardware models are necessary to static analysis, and getting them is becoming harder as a consequence of the increased hardware complexity.

Despite the latter scenario, the need for reliable real time systems and programs is increasing, and there is an increasing pressure from the research community towards the differentiation of hardware. The aim is the introduction of alternative hardware whose behavior would be more predictable and more suitable to be statically analysed, for example decoupling execution time from the execution history by introducing randomization.

In the CerCo project we do not try to address this problem, optimistically assuming that static analysis of non functional properties of programs will return to be feasible in the long term. The main objective of our work is instead to bring together static analysis of functional and non functional properties, which, according to the current state of the art, are completely independent activities with limited exchange of information: while the functional properties are verified on the source code of programs written in high level languages, the analysis of non functional properties is entirely performed on the object code to exploit clock precise hardware models.

There are two main reasons to currently perform the analysis on the object code. The first one is the obvious lack of a uniform, precise cost model for source code instructions (or even basic blocks). During compilation, high level instructions are torn apart and reassembled in context specific ways so that there is no way to identify a fragment of object code with a single high level instruction. Even the control flow of the object and source code can be very different as a result of optimizations. For instance, loop optimizations reduce the number or the order of the iterations of loops, and may assign different object code, and thus different

costs, to different iterations. Despite the lack of a uniform, compilation and program independent cost model on the source language, the literature on the analysis of non asymptotic execution time on high level languages that assumes such a model is growing and getting momentum. Its practical usefulness is doomed to be minimal, unless we can provide a replacement for such cost models. Some hope has been provided by the *EmBounded* project (FP6 FET-Open STReP, IST-510255), which compositionally compiles high level code to a byte code that is executed by an emulator with guarantees on the maximal execution time spent for each byte code instruction. The approach indeed provides a uniform model, at the price of loosing precision of the model (each cost is a pessimistic upper bound) and performance of the executed code (because the byte code is emulated compositionally instead of performing a fully non compositional compilation).

The second reason to perform the analysis on the object code is that bounding the worst case execution time of small code fragments in isolation (e.g. loop bodies) and then adding up the bounds yields very poor estimations because no knowledge on the hardware state can be assumed when executing the fragment. By analysing longer runs (e.g. by full unrolling loops) the bound obtained becomes more precise because the lack of knowledge on the initial state has less effects on longer computations.

In CerCo we propose a radically new approach to the problem: we reject the idea of a uniform cost model and we propose that the compiler, which knows how the code is translated, must return the cost model for basic blocks of high level instructions. It must do so by keeping track of the control flow modifications to reverse them and by interfacing with static analysers. By embracing compilation, instead of avoiding it like EmBounded did, a CerCo compiler can at the same time produce efficient code and return costs that are as precise as the static analysis can be. Moreover, we allow our costs to be parametric: the cost of a block can depend on actual program data or on a summary of the execution history or on an approximated representation of the hardware state. For example, loop optimizations assign to a loop body a cost that is a function of the number of iterations performed. For another example, the cost of a loop body may be a function of the vector of stalled pipeline states, which can be exposed in the source code and updated at each basic block exit. It is parametricity that allows to analyse small code fragments without loosing precision: in the analysis of the code fragment we do not have to be ignorant on the initial hardware state. On the contrary, we can assume to know exactly which state (or mode, as WCET literature calls it) we are in.

The cost of an execution is always the sum of the cost of basic blocks multiplied by the number of times they are executed, which is a functional property of the program. Therefore, in order to perform (parametric) time analysis of programs, it is necessary to combine a cost model with control and data flow analysis. Current state of the art WCET technology performs the analysis on the object code, where the logic of the program is harder to reconstruct and most information available on the source code (e.g. types) has been lost. Imprecision in the analysis leads to useless bounds. To augment precision, the tools ask the user to provide constraints on the object code control flow, usually in the form of bounds on the number of iterations of loops or linear inequalities on them. This requires the user to manually link the source and object code, translating his often wrong assumptions on the source code to object code constraints. The task is error prone and, in presence of complex optimizations, may be very hard if not impossible.

The CerCo approach has the potentiality to dramatically improve the state of the art. By performing control and data flow analysis on the source code, the error prone translation of invariants is completely avoided. It is in fact performed by the compiler itself when it induces the cost model on the source language. Moreover, any available technique for the verification of functional properties can be immediately reused and multiple

techniques can collaborate together to infer and certify cost invariants for the program. Parametric cost analysis becomes the default one, with non parametric bounds used as last resorts when trading the complexity of the analysis with its precision. A priori, no technique previously used in traditional WCET is lost (e.g. full unrolling for non parametric costs): they can just be applied on the source code.

Traditional techniques for WCET that work on object code are also affected by another problem: they cannot be applied before the generation of the object code. Therefore analysis of functional properties of programs already starts in early development stages, while when analysis of non functional properties becomes possible the cost of changes to the program architecture can already be very high. Our approach already works in early development stages by axiomatically attaching costs to components that are not implemented yet.

All software used to verify properties of programs must be as bug free as possible. The trusted code base for verification is made by the code that needs to be trusted to believe that the property holds. The trusted code base of state-of-the-art WCET tools is very large: one needs to trust the control flow analyser and the linear programming libraries it uses and also the formal models of the hardware. In CerCo we are moving the control flow analysis to the source code and we are introducing a non standard compiler too. To reduce the trusted code base, we implemented a prototype and a static analyser in an interactive theorem prover, which was used to certify that the cost computed on the source code is indeed the one actually spent by the hardware. Formal models of the hardware and of the high level source languages were also implemented in the interactive theorem prover. Control flow analysis on the source code has been obtained using invariant generators, tools to produce proof obligations from generated invariants and automatic theorem provers to verify the obligations. If the automatic provers are able to generate proof traces that can be independently checked, the only remaining component that enters the trusted code base is an off-the-shelf invariant generator which, in turn, can be proved correct using an interactive theorem prover. Therefore we achieve the double objective of allowing to use more off-the-shelf components (e.g. provers and invariant generators) while reducing the trusted code base at the same time.

Summary of the CerCo objectives. To summarize, the goal of CerCo is to reconcile functional with non functional analysis by performing them together on the source code, sharing common knowledge about execution invariants. We want to achieve the goal implementing a new generation of compilers that induce a parametric, precise cost model for basic blocks on the source code. The compiler should be certified using an interactive theorem prover to minimize the trusted code base of the analysis. Once the cost model is induced, off-the-shelf tools and techniques can be combined together to infer and prove parametric cost bounds.

The long term benefits of the CerCo vision are expected to be:

1. the possibility to perform static analysis during early development stages
2. parametric bounds made easier
3. the application of off-the-shelf techniques currently unused for the analysis of non functional properties, like automated proving and type systems

4. simpler and safer interaction with the user, that is still asked for knowledge, but on the source code, with the additional possibility of actually verifying the provided knowledge

5. a reduced trusted code base

6. the increased accuracy of the bounds themselves.

The long term success of the project is hindered by the increased complexity of the static prediction of the non functional behavior of modern hardware. In the time frame of the European contribution we have focused on the general methodology and on the difficulties related to the development and certification of a cost model inducing compiler.

4.1.3 Main S&T results

We will now review the main S&T results achieved in the CerCo project. We will address them in the following order:

1. **The (basic) labelling approach.** It is the main technique that underlies all the developments in CerCo. It allows to track the evolution of basic blocks during compilation in order to propagate the cost model from the object code to the source code without loosing precision in the process.

2. **Dependent labelling.** The basic labelling approach assumes a bijective mapping between object code and source code O(1) blocks (called basic blocks). This assumption is violated by many program optimizations (e.g. loop peeling and loop unrolling). It also assumes the cost model computed on the object code to be non parametric: every block must be assigned a cost that does not depend on the state. This assumption is violated by stateful hardware like pipelines, caches, branch prediction units. The dependent labelling approach is an extension of the basic labelling approach that allows to handle parametric cost models. We showed how to the method allows to deal with loop optimizations and pipelines, and we speculated about its applications to caches.

3. **Techniques to exploit the induced cost model.** Every technique used for the analysis of functional properties of programs can be adapted to analyse the non functional properties of the source code instrumented by compilers that implement the labelling approach. In order to gain confidence in this claim, we showed how to implement a cost invariant generator combining abstract interpretation with separation logic ideas. We integrated everything in the Frama-C modular architecture, in order to automatically compute proof obligations from the functional and the cost invariants and to use automatic theorem provers to proof them. This is an example of a new technique that is not currently exploited in WCET analysis. It also shows how precise functional invariants benefits the non functional analysis too. Finally, we show how to fully automatically analyse the reaction time of Lustre nodes that are first compiled to C using a standard Lustre compiler and then processed by a C compiler that implements the labelling approach.

4. **The CerCo compiler.** This is a compiler from a large subset of the C program to 8051/8052 object code. The compiler implements the labelling approach and integrates a static analyser for 8051 executables. The latter can be implemented easily and does not require dependent costs because the 8051 microprocessor is a very simple processor whose instructions generally have a constant cost. It was picked to separate the issue of exact propagation of the cost model from the target to the source language from the orthogonal problem of the static analysis of object code that requires approximations or dependent costs. The compiler comes in several versions: some are prototypes implemented directly in OCaml, and they implement both the basic and dependent labelling approaches; the final version is extracted from a Matita certification and at the moment implements only the basic approach.

5. **A formal cost certification of the CerCo compiler.** We implemented the CerCo compiler in the interactive theorem prover Matita and have formally certified that the cost model induced on the source code correctly and precisely predicts the object code behavior. We actually induce two cost

models, one for time and one for stack space used. We show the correctness of the prediction only for those programs that do not exhaust the available machine stack space, a property that thanks to the stack cost model we can statically analyse on the source code using our Frama-C tool. The preservation of functional properties we take as an assumption, not itself formally proved in CerCo. Other projects have already certified the preservation of functional semantics in similar compilers, and we have not attempted to directly repeat that work. In order to complete the proof for non-functional properties, we have introduced a new semantics for programming languages based on a new kind of structured observables with the relative notions of forward similarity and the study of the intentional consequences of forward similarity. We have also introduced a unified representation for back-end intermediate languages that was exploited to provide a uniform proof of forward similarity.

6. **Additional theoretical investigations on computational models and the prediction of program complexity.**

    a) The certification of the CerCo compiler formally shows what is common knowledge among computer scientists: it is possible to write compilers/interpreters that preserve the asymptotic costs of programs. The property is supposed to hold also for abstract computational models like Turing machines, but there is currently no evidence for this: all known implementations of universal machines do not respect the property. More generally, most of computability theory is focused on extensional (or functional) properties only and little is known on the intensional one. In CerCo we provided the first formal certification of a universal Turing machine where we try to preserve the complexity, reducing the problem to the preservation of the complexity in the encoding/decoding of the machine to be executed. The problem is not settled yet.

    b) We studied extension Implicit Computational Complexity (ICC) techniques. Classical ICC results provide restrictions of purely functional languages that allow only programs that belong to a certain complexity class (e.g. elementary, exponential, log-space). They do not provide precise bounds and thus do not need precise cost models. In CerCo we studied how to extend one technique to cover imperative, concurrent languages, achieving elementary or exponential complexity. Then we started the study of a new semantic interpretation that should eventually allow to provide concrete, non exponential, bounds, using precise cost models as the ones that we can provide in CerCo.

4.1.3.1: The (basic) labelling approach.

Problem statement: given a source program *P,* we want to obtain an *instrumented* source program *P',* written in the same programming language, and the object code *O* such that: 1) *P'* is obtained by inserting into *P* some additional instructions to update global cost information like the amount of time spent during execution or the maximal stack space required; 2) *P* and *P'* must have the same functional behavior, i.e., they must produce that same output and intermediate observables; 3) *P* and *O* must have the same functional behavior; 4) after execution and in interesting points during execution, the cost information computed by *P'* must be an upper bound of the one spent by *O* to perform the corresponding operations (*soundness*

property); 5) the difference between the costs computed by *P'* and the execution costs of *O* must be bounded by a program dependent constant (*precision property*).

The labeling software components: we solve the problem in four stages, implemented by four software components that are used in sequence.

1. The first component *labels* the source program *P* by injecting *label emission statements* in the program in appropriate positions. The set of labels with their positions is called labelling. The syntax and semantics of the source programming language is augmented with label emission statements. The statement "EMIT *l*" behaves like a NOP instruction that does not affect the program state or control flow, but it changes the semantics by making the label *l* observable. Therefore the observables of a run of a program becomes a stream of label emissions: *l*1 … *l*n, called the program trace. We clarify the conditions that the labelling must respect later.

2. The second component is a *labelling preserving compiler*. It can be obtained from an existing compiler by adding label emission statements to every intermediate language and by propagating label emission statements during compilation. The compiler is correct if it preserves both the functional behavior of the program and the generated traces. We may also ask that the function that erases the cost emission statements commute with compilation. This optional property grants that the labelling does not interfere with the original compiler behavior. A further set of requirements will be added later.

3. The third component is a *static object code analyser*. It takes in input the object code augmented with label emission statements and it computes, for every such statement, its *scope*. The scope of a label emission statement is the tree of instructions that may be executed after the statement and before a new label emission statement is found. It is a tree and not a sequence because the scope may contain a branching statement. In order to grant that such a finite tree exists, the object code must not contain any loop that is not broken by a label emission statement. This is the first requirement of a *sound labelling*. The analyser fails if the labelling is unsound. For each scope, the analyser computes an upper bound of the execution time required by the scope, using the maximum of the costs of the two branches in case of a conditional statement. Finally, the analyser computes the cost of a label by taking the maximum of the costs of the scopes of every statement that emits that label.

4. The fourth and last component takes in input the cost model computed at step 3 and the labelled code computed at step 1. It outputs a source program obtained by replacing each label emission statement with a statement that increments the global cost variable with the cost associated to the label by the cost model. The obtained source code is the *instrumented source code*.

Correctness: Requirements 1, 2 and 3 of the program statement obviously hold, with 2 and 3 being a consequence of the definition of a correct labelling preserving compiler. It is also obvious that the value of the global cost variable of an instrumented source code is at any time equal to the sum of the costs of the labels emitted by the corresponding labelled code. Moreover, because the compiler preserves all traces, the sum of the costs of the labels emitted in the source and target labelled code are the same. Therefore, to satisfy the 4th requirement, we need to grant that the time taken to execute the object code is equal to the

sum of the costs of the labels emitted by the object code. We collect all the necessary conditions for this to happen in the definition of a *sound labelling*: a) all loops must be broken by a cost emission statement; b) all program instructions must be in the scope of some cost emission statement. To satisfy also the 5th requirement, additional requirements must be imposed on the object code labelling to avoid all uses of the maximum in the cost computation: the labelling is *precise* if every label is emitted at most once and both branches of a conditional jump start with a label emission statement.

The correctness and precision of the labelling approach only rely on the correctness and precision of the object code labelling. The simplest, but not necessary, way to achieve them is to impose correctness and precision requirements also on the initial labelling produced by the first software component, and to ask the labelling preserving compiler to preserve these properties too. The latter requirement imposes serious limitations on the compilation strategy and optimizations: the compiler may not duplicate any code that contains label emission statements, like loop bodies. Therefore several loop optimizations like peeling or unrolling are prevented. Moreover, precision of the object code labelling is not sufficient per se to obtain global precision: we also implicitly assumed the static analysis to be able to associate a precise constant cost to every instruction. This is not possible in presence of stateful hardware whose state influences the cost of operations, like pipelines and caches. In the next section we will see an extension of the basic labelling approach to cover this situation.

The labelling approach described in this section can be applied equally well and with minor modifications to imperative and functional languages. In the CerCo project, we have tested it on a simple imperative language without functions (a While language), to a subset of C and to two compilation chains for a purely functional higher order language. The two main changes required to deal with functional languages are: 1) because global variables and updates are not available, the instrumentation phase produces monadic code to "update" the global costs; 2) the requirements for a sound and precise labelling of the source code must be changed when the compilation is based on CPS translations. In particular, we need to introduce both labels emitted before a statement is executed and labels emitted after a statement is executed. The latter capture code that is inserted by the CPS translation and that would escape all label scopes.

Phases 1, 2 and 3 can be applied as well to logic languages (e.g. Prolog). However, the instrumentation phase cannot: in standard Prolog there is no notion of (global) variable whose state is not retracted during backtracking. Therefore, the cost of executing computations that are later backtracked would not be correctly counted in. Any extension of logic languages with non-backtrackable state should support the labelling approach.

4.1.3.2: Dependent labelling.

The core idea of the basic labelling approach is to establish a tight connection between basic blocks executed in the source and target languages. Once the connection is established, any cost model computed on the object code can be transferred to the source code, *without affecting the code of the compiler or its proof*. In particular, it is immediate that we can also transport cost models that associate to each label functions from hardware state to natural numbers. However, a problem arises during the instrumentation phase that replaces cost emission statements with increments of global cost variables. The global cost variable must be incremented with the result of applying the function associated to the label to the hardware state at the time of execution of the block.

The hardware state comprises both the “functional” state that affects the computation (the value of the registers and memory) and the “non functional” state that does not (the pipeline and caches content for example). The former is in correspondence with the source code state, but reconstructing the correspondence may be hard and lifting the cost model to work on the source code state is likely to produce cost expressions that are too hard to reason on. Luckily enough, in all modern architectures the cost of executing single instructions is either independent of the functional state or the jitter --- the difference between the worst and best case execution times --- is small enough to be bounded without loosing too much precision. Therefore we can concentrate on dependencies over the “non functional” parts of the state only.

The non functional state has no correspondence in the high level state and does not influence the functional properties. What can be done is to expose the non functional state in the source code. We just present here the basic intuition in a simplified form: the technical details that allow to handle the general case are more complex and can be found in the CerCo deliverables. We add to the source code an additional global variable that represents the non functional state and another one that remembers the last labels emitted. The state variable must be updated at every label emission statement, using an update function which is computed during static analysis too. The update function associates to each label a function from the recently emitted labels and old state to the new state. It is computed composing the semantics of every instruction in a basic block and restricting it to the non functional part of the state.

Not all the details of the non functional state needs to be exposed, and the technique works better when the part of state that is required can be summarized in a simple data structure. For example, to handle simple but realistic pipelines it is sufficient to remember a short integer that encodes the position of bubbles (stuck instructions) in the pipeline. In any case, the user does not need to understand the meaning of the state to reason over the properties of the program. Moreover, at any moment the user or the invariant generator tools that analyse the instrumented source code produced by the compiler can decide to trade precision of the analysis with simplicity by approximating the parametric cost with safe non parametric bounds. Interestingly, the functional analysis of the code can determine which blocks are executed more frequently in order to approximate more aggressively the ones that are executed less.

The idea of dependent labelling can also be applied to allow the compiler to duplicate blocks that contain labels (e.g. to allow loop optimizations). The effect of duplication is to assign a different cost to the different occurrences of a duplicated label. For example, loop peeling turns a loop into the concatenation of a copy the loop body (that executes the first iteration) with the conditional execution of the loop (for the successive iterations). Because of further optimizations, the two copies of the loop will be compiled differently, with the first body usually taking more time. By introducing a variable that keep tracks of the iteration number, we can associate to the label a cost that is a function of the iteration number. The same technique works for loop unrolling without modifications: the function will assign a cost to the even iterations and another cost to the odd ones. The actual work to be done consists in introducing in the source code and for each loop a variable that counts the number of iterations. The loop optimization code that duplicate the loop bodies must also modify the code to propagate correctly the update of the iteration numbers. The technical details are more complex and can be found in the CerCo reports and publications. The implementation, however, is quite simple and the changes to a loop optimizing compiler are minimal.

4.1.3.3: Techniques to exploit the induced cost model.

We review the cost synthesis techniques developed in the project.

The starting hypothesis is that we have a certified methodology to annotate blocks in the source code with constants which provide a sound and possibly precise upper bound on the cost of executing the blocks after compilation to object code.

The principle that we have followed in designing the cost synthesis tools is that the synthetic bounds should be expressed and proved within a general purpose tool built to reason on the source code. In particular, we rely on the Frama − C tool to reason on C code and on the Coq proof-assistant to reason on higher-order functional programs.

This principle entails that:

- The inferred synthetic bounds are indeed correct as long as the general purpose tool is.
- There is no limitation on the class of programs that can be handled as long as the user

is willing to carry on an interactive proof.

Of course, automation is desirable whenever possible. Within this framework, automation means writing programs that give hints to the general purpose tool. These hints may take the form, say, of loop invariants/variants, of predicates describing the structure of the heap, or of types in a light logic. If these hints are correct and sufficiently precise the general purpose tool will produce a proof automatically, otherwise, user interaction is required.

The Cost plug-in and its application to the Lustre compiler

Frama − C is a set of analysers for C programs with a specification language called ACSL. New analyses can be dynamically added through a plug-in system. For instance, the Jessie plug-in allows deductive verification of C programs with respect to their specification in ACSL, with various provers as back-end tools.

We developed the CerCo Cost plug-in for the Frama − C platform as a proof of concept of an automatic environment exploiting the cost annotations produced by the CerCo compiler. It consists of an OCaml program which in first approximation takes the following actions: (1) it receives as input a C program, (2) it applies the CerCo compiler to produce a related C program with cost annotations, (3) it applies some heuristics to produce a tentative bound on the cost of executing the C functions of the program as a function of the value of their parameters, (4) the user can then call the Jessie tool to discharge the related proof obligations.

In the following we elaborate on the soundness of the framework and the experiments we performed with the Cost tool on the C programs produced by a Lustre compiler.

Soundness The soundness of the whole framework depends on the cost annotations added by the CerCo compiler, the synthetic costs produced by the Cost plug-in, the verification conditions (VCs) generated by Jessie, and the external provers discharging the VCs. The synthetic costs being in ACSL format, Jessie can be used to verify them. Thus, even if the added synthetic costs are incorrect (relatively to the cost annotations), the process as a whole is still correct: indeed, Jessie will not validate incorrect costs and no conclusion can be made about the WCET of the program in this case. In other terms, the soundness does not really depend on the action of the Cost plug-in, which can in principle produce any synthetic cost. However, in order to be able to actually prove a WCET of a C function, we need to add correct annotations in a way that Jessie and subsequent automatic provers have enough information to deduce their validity. In practice this is not straightforward even for very simple programs composed of branching and assignments (no loops and no recursion) because a fine analysis of the VCs associated with branching may lead to a complexity blow up.

Experience with Lustre Lustre is a data-flow language to program synchronous systems and the language comes with a compiler to C. We designed a wrapper for supporting Lustre files.

The C function produced by the compiler implements the step function of the synchronous system and computing the WCET of the function amounts to obtain a bound on the reaction time of the system. We tested the Cost plug-in and the Lustre wrapper on the C programs generated by the Lustre compiler. For programs consisting of a few hundreds loc, the Cost plug-in computes a WCET and Alt − Ergo is able to discharge all VCs automatically.

Handling C programs with simple loops

The cost annotations added by the CerCo compiler take the form of C instructions that update by a constant a fresh global variable called the cost variable. Synthesizing a WCET of a C function thus consists in statically resolving an upper bound of the difference between the value of the cost variable before and after the execution of the function, i.e. find in the function the instructions that update the cost variable and establish the number of times they are passed through during the flow of execution. In order to do the analysis the plugin makes the following assumptions on the programs:

• No recursive functions.

• Every loop must be annotated with a variant. The variants of ‘for’ loops are automatically inferred.

The plugin proceeds as follows.

• First the call graph of the program is computed. If the function f calls the function g

then the function g is processed before the function f .

• The computation of the cost of the function is performed by traversing its control flow graph. The cost at a node is the maximum of the costs of the successors.

• In the case of a loop with a body that has a constant cost for every step of the loop, the cost is the product of the cost of the body and of the variant taken at the start of the loop.

• In the case of a loop with a body whose cost depends on the values of some free variables, a fresh logic function f is introduced to represent the cost of the loop in the logic assertions. This logic function takes the variant as a first parameter. The other parameters of f are the free variables of the body of the loop. An axiom is added to account the fact that the cost is accumulated at each step of the loop.

• The cost of the function is directly added as post-condition of the function

The user can influence the annotation by different means:

• By using more precise variants.

• By annotating function with cost specification. The plugin will use this cost for the function instead of computing it.

C programs with pointers

When it comes to verifying programs involving pointer-based data structures, such as linked lists, trees, or graphs, the use of traditional first-order logic to specify, and of SMT solvers to verify, shows some limitations. Separation logic is then an elegant alternative. Designed at the turn of the century, it is a program logic with a new notion of conjunction to express spatial heap separation. Separation logic has been implemented in dedicated theorem provers such as Smallfoot or VeriFast. One drawback of such provers, however, is to either limit the expressiveness of formulas (e.g. to the so-called symbolic heaps), or to require some user-guidance (e.g. open/close commands in VeriFast).

In an attempt to conciliate both approaches, we introduced the notion of separation predicates. The approach consists in reformulating some ideas from separation logic into a traditional verification framework where the specification language, the verification condition generator, and the theorem provers were not designed with separation logic in mind. Separation predicates are automatically derived from user-defined inductive predicates, on demand. Then they can be used in program annotations, exactly as other predicates, i.e., without any constraint. Simply speaking, where one would write P ⍰ Q in separation logic, one will here ask for the generation of a separation predicate sep and then use it as P ⍰ Q ⍰ sep(P, Q). We have implemented separation predicates within the Jessie plug-in and tested it on non-trivial case studies (e.g. the composite pattern from the VACID-0 benchmark). In these cases, we achieve a fully automatic proof using three existing SMT solver.

We have also used the separation predicates to reason on the cost of executing simple heap manipulating programs such as an in-place list reversal.

The cost of memory management for functional programs

Modern high level programming languages implicitly handle memory. In particular, all functional programming languages do that. We already said how we were able to simply modify the labelling approach to deal with an higher order functional language without any implicit form of memory management. We address here how to cope with the latter.

In a realistic implementation of a functional programming language, the runtime environment

usually includes a garbage collector. In spite of considerable progress in real-time garbage collectors it seems to us that such collectors do not offer yet a viable path to a certified and usable WCET prediction of the running time of functional programs. As far as we know, the cost predictions concern the amortized case rather than the worst case and are supported more by experimental evaluations than by formal proofs.

The approach we have adopted instead, following the seminal work of Tofte et al., is to enrich the last calculus of the compilation chain : (1) with a notion of memory region, (2) with operations to allocate and dispose memory regions, and (3) with a type and effect system that guarantees the safety of the dispose operation. This allows to further extend the compilation chain mentioned above and then to include the cost of safe memory management in our analysis. Actually, because effects are intertwined with types, what we have actually done, following the work of Morrisett et al., is to extend a typed version of the compilation chain. An experimental validation of the approach is left for future work and it would require the integration of region-inference algorithms such as those developed by Aiken et al. in the compilation chain.

4.1.3.4 The CerCo Compiler

In CerCo we have developed a certain number of cost preserving compilers based on the labelling approach. Excluding an initial certified compiler for a While language, all remaining compilers target realistic source languages --- a pure higher order functional language and a large subset of C with pointers, gotos and all data structures --- and real world target processors --- MIPS and the Intel 8051/8052 processor family. Moreover, they achieve a level of optimization that ranges from moderate (comparable to gcc level 1) to intermediate (including loop peeling and unrolling, hoisting and late constant propagation). The so called Trusted CerCo Compiler is the only one that was implemented in the interactive theorem prover Matita and its costs certified. The code distributed is obtained extracting OCaml code from the Matita implementation. In the rest of this section we will only focus on the Trusted CerCo Compiler, that only targets the C language and the 8051/8052 family, and that does not implement the advanced optimizations yet. Its user interface, however, is the same as the one of the other versions, in order to trade safety with additional performances. In particular, the Frama-C CerCo plug-in can work without recompilation with all compilers.

The (trusted) CerCo compiler implements the following optimizations: cast simplification, constant propagation in expressions, liveness analysis driven spilling of registers, dead code elimination, branch displacement, tunneling. The two latter optimizations are performed by the optimizing assembler which is part of the compiler. The back-end of the compiler works on three addresses instructions, preferred to static single assignment code for the simplicity of the formal certification.

The CerCo compiler is loosely based on the CompCert compiler, a recently developed certified compiler from C to the PowerPC, ARM and x86 microprocessors. Contrarily to CompCert, both the CerCo code and its certification are open source. Some data structures and language definitions for the front-end are directly taken from CompCert, while the back-end is a redesign and reimplementation of a didactic compiler from Pascal to MIPS used by Francois Pottier for a course at the Ecole Polytechnique.

Following CompCert tradition, the compiler is organised in an unusually large number of intermediate passes, all responsible for just one change in the semantics of the source and target languages. Introducing multiple passes has minor performance implications on modern hardware and it allows to simplify the simulation proofs. The first three intermediate languages for the front-end. They are syntactically and semantically quite different between each other. For example, in the first language we find the traditional

looping structures of C, in the second all loops are infinite loops (built with GOTOs) interrupted using BREAKs and in the third one the code is organised as a graph of statements where loops become loops in the graph. The four back-end languages, instead, have a more similar syntax.

Departing from CompCert, we do not provide a stand alone syntax and semantics for every back-end language. Instead, we developed a generic representation of back-end languages as a parametric data type that can be instantiated to the wanted language. The generic representation allows to multiply the number of passes without increasing too much the code size. For example, we also provide a single generic semantics for the generic representation, parameterized over pass specific details.

Other departures from CompCert are:

1. all of our intermediate languages include label emitting instructions to implement the labelling approach, and the compiler preserves execution traces.

2. the target language of CompCert is an assembly language with additional macro-instructions to be expanded before assembly; for CerCo we need to go all the way down to object code in order to perform the static analysis. Therefore we developed also an optimizing assembler and a static analyser, all integrated in the compiler.

3. to avoid implementing a linker and a loader, we do not support separate compilation and external calls. Adding a linker and a loader is a transparent process to the labelling approach and should create no unknown problem

4. we target an 8-bit processor. Targeting an 8 bit processor requires several changes and increased code size, but it is not fundamentally more complex. The proof of correctness, however, becomes much harder.

5. we target a microprocessor that has a non uniform memory model, which is still often the case for microprocessors used in embedded systems and that is becoming common again in multi-core processors. Therefore the compiler has to keep track of the position of data and it must move data between memory regions in the proper way. Also the size of pointers to different regions is not uniform. In our case, an additional difficulty was that the space available for the stack in internal memory in the 8051 is tiny, allowing only a minor number of nested calls. To support full recursion in order to test the CerCo tools also on recursive programs, the compiler manually manages a stack in external memory.

6. while there is a rough correspondence between CompCert and CerCo back-end passes, the order of the passes is permuted. In the future we want to explore how to exploit our generic back-end language representation to try to freely compose and permute passes.

4.1.3.5 A formal certification of the CerCo compiler

The Trusted CerCo Compiler has been implemented and certified using the interactive theorem prover Matita. In this section we briefly hint at the exact correctness statement and at the main ingredients of the proof. Details on the proof techniques employed and further information can be collected from the CerCo deliverables and papers.

The statement

The most natural statement of correctness for our complexity preserving compiler is that the time spent for execution by a terminating object code program should be the time predicted on the source code by adding up the cost of every label emission statement. This statement, however, is too naïve to be useful for real world real time programs like those used in embedded systems.

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 predicting and ruling out programs that crash running out of space on a certain input.

Therefore we need to look for a more complex statement that talks about sub-runs of a program. The most natural statement is that the time predicted on the source code and spent on the object code by two corresponding sub-runs are the same. The problem to solve to make this statement formal is how to identify the corresponding sub-runs and how to single out those that are meaningful.

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, the sub-run does not perform any I/O 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.

The statement that we want to formally 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.*

We briefly discuss the constraints for measurability. Not exhausting the stack space is a clear requirement of meaningfulness of a run, because source programs do not crash for lack of space while object code ones do. The balancing of function calls/returns is a requirement for precision: the labelling approach allows the scope of label emission statements to extend after function calls to minimize the number of labels. Therefore a label pays for all the instructions in a block, excluding those executed in nested function calls. If the number of calls/returns is unbalanced, it means that there is a call we have not returned to that could be followed by additional instructions whose cost has already been taken in account. To make the statement true (but less precise) in this situation, we could only say that the cost predicted on the source code is a safe bound, not that it is exact. The last condition on the entry/exit points of a run is used to identify sub-runs whose code correspond to a sequence of blocks that we can measure precisely. Any other choice would start/end the run in the middle of a block and we would be forced again to weaken the statement taking as a bound the cost obtained counting in all the instructions that precede the starting one in the block/follow the final one in the block.

I/O operations can be performed in the prefix of the run, but not in the measurable sub-run. Therefore we prove that we can predict reaction times, but not I/O times, as it should be.

Dealing with function calls

For obvious complexity reasons, the proof of correctness must be modular in the compilation passes. We are looking for independent proofs of correctness for the single passes that, composed together, should yield the global proof. Achieving modularity for a language that does not have function calls and function pointers is quite simple. The most difficult part of the proof is a forward simulation lemma that says that for each pass and each (sub)-run in the source program, there is a corresponding (sub)-run in the target program that emits the same observables.

The previous lemma holds also when function calls and function pointers are introduced, but make the lemma too weak. As we already explained, the labelling approach assumes that the cost of a function call will be paid by the labels of the function body, while the instructions after the call are paid by the label before the call. In order to predict correctly the costs, we therefore need to assume that: 1) every function body starts with a label; 2) every terminating function returns just after the call. The problem with function pointers is that property 1 is no longer a syntactic property that can be checked at compile time and that can be propagated by compilation: a function pointer allows to jump everywhere in the code. The problem with function calls in general is that property 2 is no longer valid and cannot be checked at compile time for every back-end language where the RET statement passes control to the address on top of the stack, which is not necessarily the one after the last function call. It is even simple to design compilers that intentionally change the order of instructions and introduce code to manipulate the stack so that function do not return to the call point, but the compiler preserves all functional properties anyway.

In order to strengthen the statement of the lemma, we need identify those runs that satisfy the assumption above and prove the statement for those runs only (both in the source and target language). Note that both properties are automatically guaranteed by the semantics of the high level source language, which is structured and more compositional. Therefore what we are doing is to try to identify the low-level runs that keep part of the structure of the original program (a clearly intensional or non functional property).

The solution we found to single out those runs consisted in proposing a new style for describing the 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 and that every call must be followed by a cost emission statement. We also integrate in the structure additional requirements that capture on traces all the syntactic requirements that soundness and precision of labelling impose on the code. For example, we drop the precision requirement that every branch in the code must be followed by a cost emission statement and we put add it to the structure of the trace: a conditional branch statement in a structured trace must be followed by a cost emission statement. The aim is to drop the proof that the compiler preserves precision and correctness, incorporating (and generalizing) it in the proof of forward simulation of structured traces. The generalization is actually genuine because, for example, with the new approach dead code does not need to be soundly labelled since it produces no runs.

The proof sketch

The formal proof, based on the structured traces just described, is achieved combining several stand alone results. Those starred are standard proofs that show that functional properties are preserved. Not all the starred proofs have been completed during the project, while we completed the ones that are issued by the labelling methodology and deal with non functional properties:

1 Correctness of the instrumentation algorithm: the source code augmented with label emission statements in

proper places simulates the original program and the labelling obtained is sound and precise

2 * Existence of a standard functional forward simulation between the source code and the last code of the front-end.

3 Correctness of an algorithm that checks if the program generated by the front-end is soundly and precisely labelled. Compilation is aborted otherwise. The algorithm can be dropped replacing the proof with the longer and harder proof of preservation of soundness and precision of labelling by the front-end.

4 Existence of structured traces: we prove that every measurable sub-run of a soundly and precisely labelled program written in the last front-end language admits a corresponding structured trace such that the sub-run and the trace have the same observables and their prefixes have the same property too.

5 The definition of a forward simulation relation between structured trace.

6 The proof that similar structured traces produce that same set of observables.

7 The existence of a forward simulation (in the sense of 5) for every back-end pass. The proof relies on the hypothesis that the stack usage predicted for a structured trace does not exceed the available machine stack. The proofs have been simplified by factorizing out some common results:

7.1 A complex proof that reduces the existence of a forward simulation (which is a global condition) to a set of local conditions to be checked for every source step. For example, in a structured trace all RETs must return control just after the corresponding CALL statement (the global condition). This can only happen if the CALL pushes the right address on the stack and if all instructions in the function body manipulate the stack in the expected way (the local conditions). The local conditions combine together standard functional forward simulation conditions with additional ones that still mention structured traces.

7.2 Thanks to our uniform and generic representation of back-end languages, we were able to provide a generic representation of back-end passes that covers most of them. A second complex but generic proof allows to reduce the local proof obligations generated by 7.1 to a new set of local proof obligations that are exactly the functional ones with additional very simple ones that do not mention structured traces any longer.

7.3 * Finally, one needs to inhabit all the proof obligations generated by 7.2, i.e. it must show the preservation of the functional properties.

8 Correctness of static analysis: the actual execution cost of a structured object code trace is the sum of the statically computed cost of each label in the trace.

The correctness statement is then proved as follow: by 1,3 and 4 there exists an intermediate structure trace (and prefix) that have the same observables of the source program and the same functional behavior; by 6 and 7 there exists a structured object code

(and prefix) that have the same observables of the source program and the same functional behavior. Therefore any cost model that is correct for the object code can be used to predict the cost on the source code. Finally, by 8 the cost model computed by the compiler is correct for the object code and thus any cost analysis performed on the source code is also correct.

Additional work

The executable semantics of C and of the 8051 object code are part of the trusted code base of CerCo. Errors in the semantics may hide errors in the compiler. Therefore it is of maximal importance that the semantics are as accurate and error free as possible.

Executability is an important key to the minimization of errors because it allows to run in parallel tests on the semantics and on the machine (or a C emulator) and compare the executions. Debugging the two semantics took a considerable time in the project.

For the C semantics we had another way to raise confidence: we have formally proved in Matita the equivalence of our executable semantics for C with the non-executable one developed in CompCert and ported to Matita. Non executable semantics are often picked in formal verifications because they allow to simplify the simulation proof. This was also the case for CompCert.

A very surprising discovery had been bugs in the CompCert semantics, one of which actually masking a bug in the CompCert compiler itself.

The discover represents further evidence of the benefits of an executable semantics that can be actually tested.

Contrary to CompCert, all the semantics in use in CerCo are executable.

4.1.3.6 Additional theoretical investigations on computational models and the prediction of program complexity

In the introduction of Section 4.1.3 we have already hinted at additional theoretical investigations that have been performed in the CerCo project and that are loosely connected to the labelling approach. Detailed information are provided in the CerCo papers. We only want here to recall the motivations for the investigation.

Complexity preserving universal Turing machines

Most computational models that are studied theoretically are considered to be realistic up to the "minor" detail of allowing an infinite amount of resources. For example, Turing machines and lambda calculi allow infinite memory, process algebras allow infinite channels, RAM allows for unbounded integers to be stored in registers, etc. These unrealistic assumptions make the problems interesting. For example, termination is trivially decidable when program states are bounded. Moreover, often in practice a large number of resources only do not make practical any analysis that is rooted in the finiteness of the resources. Continuing the example, the number of different states of a real world computer is no huge that it is unfeasible to build the transition matrix between all pair of states to determine termination this way.

Functional properties are, almost by definition, insensitive to resources. Therefore most computability theory, that deals only with functional properties, continues to hold when applied to realistic models that have limited resources. Curiously, almost nothing is known theoretically for non functional properties. The object of study of CerCo is the]preservation of the exact computational cost during compilation. As expected, we have formally certified that such preservation is possible when working with real world languages, hardware and compilers. The natural question is if the same holds for the usual theoretical computational models with unbounded resources. An even simpler problem, which is the one we started to address in the timeframe of CerCo, is the existence of universal interpreters that preserve the computational costs.

Feasible bounds by light typing

In our experience, the cost analysis of higher-order programs requires human intervention both at the level of the specification and of the proofs. One path to automation consists in devising programming disciplines that entail feasible bounds (polynomial time). The most interesting approaches to this problem, even if maybe not the most practical ones at the moment, build on light versions of linear logic. Our main contribution is to devise a type system that guarantees feasible bounds for a higher-order call-by-value functional language with references and threads.

The first proof of this result relies on a kind of standardisation theorem and it is of a combinatorial nature. More recently, we have shown that a proof of a similar result can be obtained by semantic means building on the so called quantitative realizability models proposed by Dal Lago and Hofmann. We believe this semantic setting is particularly appropriate because it allows to reason both on typed and untyped programs. Thus one can imagine a framework where some programs are feasible 'by typing' while others are feasible as a result of an 'interactive proof' of the obligations generated by quantitative realizability. Beyond building such a framework, an interesting issue concerns the certification of concrete bounds at the level of the compiled code. This has to be contrasted with the current state of the art in implicit computational complexity where most bounds are asymptotic and are stated at the level of the source code.

4.1.4 Potential impact, main dissemination activities and exploitation of results

Potential impact

Business- or safety-critical systems increasingly permeate the society. Rigorous software engineering processes and testing are not sufficient to grant safety, and failures can cause massive loss of money or even endanger human lives. The formal method community provides tool to statically analyse software systems dramatically increasing our confidence. Manual and interactive theorem proving can further increase confidence by reducing the trusted code base of the software analysers and the other tools used for deployment (compilers, linkers, operating systems and middleware software). The great challenge for the formal method community is to reduce the costs of formalization and to balance the tradeoff between automation and precision of the analysis towards the latter. The long term aim is to enlarge the percentage of software that is currently certified to be safe and to foster adoption of these technologies by software industries ultimately decreasing the development time and costs.

Safety properties involve both the functional behavior of a program --- what the program does --- and the non functional behavior --- how it does it and what resources it needs. The non functional behavior that is harder to establish is execution in a bounded amount of time. Embedded systems increasingly permeate our daily lives and most of them are soft or hard real time systems: when late, they can cause major failures and disruptions in physical systems. Nowadays most hard real time components are not rigorously granted to execute in the expected amount of time. The standard industrial practice just consists in observing and measuring the time spent in a certain number of executions, adding a safety margin to the largest observed execution time and claiming that the program is time safe if the obtained estimation is in the bound. Occasional failures are known to have happened, with disastrous effects. Therefore code-level timing analysis, the branch of static program analysis that deals with execution time, is an indispensable technique for ascertaining whether or not the timing requirements are systematically met (or, alternatively, what is the probability of not meeting them).

Code-level time analysis is currently performed on the object code, and it starts by reconstructing the functional behavior of the program on the object code. In order to do so, user interaction is provided. This requires the user to be able to provide knowledge and invariants on the control flow of the object code, which may be arbitrarily different from the one on the source code. The CerCo methodology allows to perform time analysis on the source code instead, exploiting all the techniques, methodologies, tools and user knowledge used to formally verify the functional properties of the program, reducing the trusted code base at the same time. Potentially the CerCo methodology allows to increase the precision of the analysis (thus reducing the cost of hardware) and to dramatically increase the confidence on the result.

A second benefit of the CerCo methodology is that it allows to remove the main obstacle to the adoption of high level programming methodologies for the development of time critical software. High level methodologies and tools are slowly being adopted for the production of non time critical industrial software in order to greatly decrease development time while increasing confidence in the code. Among them,

domain specific languages, code generators, advanced programming and specification methods and higher level languages in general. By using these methodologies the user can program more comfortably because the language used to express the solution gets closer to the one the problem is expressed in. The main drawback is that the gap with the object code is enlarged and it becomes harder if not impossible to predict and control the amount of resources (time, memory) that will be spent. Adopting the CerCo approach, maybe in combination with the work previously done in EmBounded, it is now possible to pick any high level language and still be able to perform time analysis on the source code, with the compiler responsibility to induce on the high level code a sound and precise cost model that reflects the generated code.

An application of CerCo technology that will be likely to deliver immediately exploitable progress in the field is to early development phases. WCET analysis has traditionally been used in the verification phase of a system, after all components have been built. Since redesigning a software system is very costly, designers usually choose to over-specify the hardware initially and then just verify that it is indeed sufficiently powerful. However, as systems' complexity rises, these initial safety margins can prove to be very expensive. Undertaking lightweight (but less precise) analysis in the early stages of the design process has the potential to drastically reduce total hardware costs. Obviously current WCET technology that depends on the generation of the object code is unsuitable for this task. Embracing the CerCo approach, instead, we can already reason at early stages by axiomatizing the cost of unimplemented components or by reasoning parametrically over those costs. The CerCo plug-in can already combine together actual costs computed on the generated object code and axiomatized costs, and it smoothly supports progressive program refinement.

A final application of the CerCo technology is to portability of software. When the early-stage analysis is performed, it is clearly targeted at selecting a platform which is cost-efficient for the problem at hand. It would be highly beneficial to perform the WCET analysis in such a way that parts of the analysis can be redeployed on other hardware platforms, in order to avoid ballooning costs in the analysis. The CerCo approach can perform the analysis on the object code, keeping the time invariants parametric on the cost model. Only the latter changes when the program is recompiled for a different architecture.

Main dissemination activities

The main dissemination activities have consisted in the publication of peer-reviewed papers on international journals or proceedings of international conferences and workshops, and in presentation of the project results to international conferences, workshops and symposia. Several seminars have been given to inviting institutions. Finally, we have organised two CerCo specific events in the form of technical days to present to industrial and academics all the developments of the project. The first event have been held as a satellite event of HiPEAC 2013, and the second one as a satellite event of ETAPS 2013.

List of papers published in international journals or proceedings of international conferences:

1. **A Canonical Locally Named Representation of Binding,** R. Pollack, M. Sato and W. Ricciotti, *Journal of Automated Reasoning*, DOI 10.1007/s10817-011-9229-y.

2. **Certifying and reasoning on cost annotations of functional programs,** R.M. Amadio, Y. Regis-Gianas, In proceedings of Foundations and Practical Aspects of Resource Analysis (FOPARA 2011), LNCS, 71277:72-89, DOI 10.1007/978-3-642-32495-6_5.

3. **An Elementary Affine λ-Calculus with Multithreading and Side Effects**, A. Madet, R.M. Amadio., In Proceedings Typed Lambda Calculi and Applications (TLCA 2011), Springer LNCS 6690:138-152, 2011.

4. **Certified Complexity**, R. Amadio, A. Asperti, N. Ayache, B. Campbell, D. Mulligan, R. Pollack, Y. Regis-Gianas, C. Sacerdoti Coen, I. Stark, in Procedia Computer Science, Volume 7, 2011, Proceedings of the 2nd European Future Technologies Conference and Exhibition 2011 (FET 11), 175-177.

5. **The Matita Interactive Theorem Prover**, A. Asperti, W. Ricciotti, C. Sacerdoti Coen, E. Tassi, In Automated Deduction – CADE-23, LECTURE NOTES IN COMPUTER SCIENCE, ISBN:978-3-642-22437-9, Pages 64-69, Volume 6803, Year 2011 64-69

6. **Certifying and Reasoning on Cost Annotations in C Programs,** N. Ayache, R.M. Amadio, Y. Régis-Gianas, in Proc. FMICS, Springer LNCS 7437: 32-46, 2012, DOI:10.1007/978-3-642-32469-7_3.

7. **Rating Disambiguation Errors,** A. Asperti, W. Ricciotti, In Proceedings of the 2nd International Conference on Certified Programs and Proofs (CPP 2012), LNCS, Volume 7679, Pages 240--255, Springer, 2012, DOI:10.1007/978-3-642-35308-6_19.

8. **Certifying and reasoning on cost annotations of functional programs**, R.M. Amadio, Y. Régis-Gianas, in *Higher-order and Symbolic Computation,* to appear, 2012.

9. **Separation Predicates: A Taste of Separation Logic in First-Order Logic**, F. Bobot, J.-C. Filliatre, in Proc. IFCEM, Springer LNCS 7635:167-181, 2012, DOI:10.1007/978-3-642-34281-3_14.

10. **A Web Interface for Matita**, A. Asperti, W. Ricciotti, In Proceedings of the Conference on Intelligent Computer Mathematics (CICM 2012), LNAI, Volume 7362/2012, Pages 417-421, DOI:10.1007/978-3-642-31374-5_28, ISBN 978-3-642-31373-8.

11. **Indexed Realizability for Bounded-Time Programming with References and Type Fixpoints**, A. Brunel, A. Madet, In Proc. APLAS, Springer LNCS 7705:264-279 , 2012, DOI:10.1007/978-3-642-35182-2_19.

12. **A polynomial time λ-calculus with multithreading and side effects**, A. Madet in Proc. ACM-PPDP, pages 55-66, 2012, DOI:10.1145/2370776.2370785.

13. **An executable semantics for CompCert C,** B. Campbell, in Proceedings of Certified Proofs and Programs 2012, LNCS 7679:60-75, 2012, DOI:10.1007/978-3-642-35308-6_8.

14. **On the Correctness of an Optimising Assembler for the Intel MCS-51 Microprocessor,** D. Mulligan, C. Sacerdoti Coen, in Proceedings of Certified Proofs and Programs 2012, LNCS 7679:43-59, 2012, DOI:10.1007/978-3-642-35308-6_7.

15. ***Indexed Labels for Loop Iteration Dependent Costs,*** P. Tranquilli, in Proceedings of the 11th International Workshop on Quantitative Aspects of Programming Languages and Systems (QAPL 2013), Rome, 23rd-24th March 2013, Electronic Proceedings in Theoretical Computer Science, to appear in 2013.

Exploitation of results

The foreground of CerCo that can lead to exploitable results in industry is:

1. the development of a methodology to transfer to the source code a cost model computed on the object code, without loss of precision in the process. To assess the methodology we developed a cost-certified prototypical compiler, which also shows how to greatly reduce the trusted code base for time analysis.

2. some experiments on the combination of invariant generator, abstract interpretation, separation logic and theorem provers to perform parametric time analysis of code according to the cost model induced by the CerCo prototypical compiler.

3. a preliminary study on the extensions of the CerCo methodology to transfer parametric cost models that can take in account modern hardware whose instruction cost depends on the internal hardware state (pipelines, caches, etc.)

The software developed constitutes a good platform for further experiments and comparative testing with existing techniques. Immediate exploitability in an industrial setting however is not possible yet. We briefly review here the steps that need to be taken toward industrial exploitability.

1. The compiler needs to be retargeted to a microprocessor that is actually used in software production (which is simple), floating point support needs to be added and additional optimizations need to be implemented to produce object code whose performance is comparable to the one actually in use in industrial setting. All these objectives may be reached in a short time, but the formal certification of the correctness of the new extensions, which is not required, if done is likely to require a long time.

2. The methodology to compute and transfer parametric cost models that deal with modern hardware needs to be refined and tested. Testing requires first to obtain a clock precise model of the microprocessor or, in alternative, to interface with existent WCET tools like AbsInt to compute the parametric cost model to be transferred using the methodology to be tested. This objective may be reached in the medium-long term.

3. The mix of techniques used to perform the parametric time analysis on the source code needs to be consolidated and made more robust, introducing a degradation of precision in case of failure of the analysis as a tradeoff towards completion of the analysis itself. This objective may be reached in the medium term, but we expect the combination of techniques to yield superior results with respect to current WCET techniques only in the long medium-long term.

4. Industrial developers use Integrated Development Environments (IDEs) that yield a number of benefits over the simple compilation, and they are not willing to abandon the IDE or even to re-train to a different one. Therefore the CerCo compiler as well as the reasoning tools need to be integrated into the IDE. The integration is likely to be time consuming.

A final issue that is quickly becoming critical for all the static analysis approaches is the adoption of multi-core architecture and systems-on-chip. The multi-core architectures are born to optimise the average case, but they make prediction of the worse case extremely ineffective. The reason is that the time behavior of a program is affected by the behavior of other programs that run concurrently with it and that are unknown

during the analysis. Ignorance during the analysis introduce very pessimistic cost models because in case of ignorance the worst scenario must be assumed. In turn, the analysis yields useless bounds that can be order of magnitudes higher than the worst observed cases. The CerCo methodology is affected by the problem, and does not contribute a solution to it. Other European projects and initiatives are putting pressure on the hardware developers to deliver also alternative systems that would be easier to analyse at the price of reducing the average performance. Hardware randomization seems to be an important ingredient towards that aim, at the price of abandoning exact time analysis in favor of probabilistic time analysis.

In CerCo we have already speculatively considered probabilistic time analysis, coming to the conclusion that the labelling methodology to transfer the cost model works equally well with probabilistic models. Further research is needed to understand which techniques can then be used to certify probabilistic time bounds on the source code once the probabilistic cost model has been induced.

Academic exploitation of CerCo results is instead already possible and CerCo directly contributes to the exploitation of several academic results that show how to perform resource analysis on the source code, but that used to assume uniform, unrealistic cost models.

4.1.5. Further information

Further information can be obtained from the project Web site:

http://cerco.cs.unibo.it

or contacting the project coordinator:

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

4.2 Use and dissemination of foreground

Section A (public)

TEMPLATE A1: LIST OF SCIENTIFIC (PEER REVIEWED) PUBLICATIONS, STARTING WITH THE MOST IMPORTANT ONES										
NO.	Title	Main author	Title of the periodical or the series	Number, date or frequency	Publisher		Year of publication	Relevant pages	Permanent identifiers[1] (if available)	Is/Will open access[2] provided to this publication?
1	**Certifying and reasoning on cost annotations of functional programs**	R. M. Amadio	Higher Order and Symbolic Computation	In press	Springer		2012			Yes
2	**Certified Complexity**	R. Amadio	Procedia Computer Science	Volume 7	Elsevier		2011	175-177	DOI:10.1016/j.procs.2011.09.054	Yes
3	**A Canonical Locally Named Representation of Binding**	R. Pollack	Journal of Automated Reasoning	Volume 49(2)	Springer		2012	185-207	DOI:10.1007/s10817-011-9229-y	No

[1] A permanent identifier should be a persistent link to the published version full text if open access or abstract if article is pay per view) or to the final manuscript accepted for publication (link to article in repository).

[2] Open Access is defined as free of charge access for anyone via Internet. Please answer "yes" if the open access to the publication is already established and also if the embargo period for open access is not yet over but you intend to establish open access afterwards.

4	**Certifying and Reasoning on Cost Annotations in C Programs**	N. Ayache	LNCS	Volume 7437	Springer		2012	32-46	DOI:10.1007/978-3-642-32469-7_3	No
5	**Certifying and reasoning on cost annotations of functional programs**	R. M. Amadio	LNCS	Volume 7177	Springer		2012	72-89	DOI:10.1007/978-3-642-32495-6_5	No
6	**Separation Predicates: A Taste of Separation Logic in First-Order Logic**	F. Bobot	LNCS	Volume 7635	Springer		2012	167-181	DOI:10.1007/978-3-642-34281-3_14	No
7	**An executable semantics for CompCert C**	B. Campbell	LNCS	Volume 7679	Springer		2012	60-75	DOI:10.1007/978-3-642-35308-6_8	No
8	**On the Correctness of an Optimising Assembler for the Intel MCS-51 Microprocessor**	D. Mulligan	LNCS	Volume 7679	Springer		2012	43-59	DOI:10.1007/978-3-642-35308-6_7	No
9	**An Elementary Affine λ-Calculus with Multithreading and Side Effects**	A. Madet	LNCS	Volume 6690	Springer		2011	138-152	DOI:10.1007/978-3-642-21691-6_13	No
10	**The Matita Interactive Theorem Prover**	A. Asperti	LNCS	Volume 6803	Springer		2011	64-69	DOI:10.1007/978-3-642-22438-6_7	No
11	**Rating Disambiguation Errors**	A. Asperti	LNCS	Volume 7679	Springer		2012	240-255	DOI:10.1007/978-3-642-35308-6_19	No
12	**A Web Interface for Matita**	A. Asperti	LNCS	Volume 7362	Springer		2012	417-421	DOI:10.1007/978-3-642-31374-5_28	No
13	**Indexed Realizability for Bounded-Time Programming with References and Type Fixpoints**	A. Brunel	LNCS	Volume 7705	Springer		2012	264-279	DOI:10.1007/978-3-642-35182-2_19	No
14	**A polynomial time λ-calculus with multithreading and side effects**	A. Madet	ACM Digital Library	Proceedings of the 14th symposium on Principles and practice of declarative programming	ACM		2012	55-66	DOI:10.1145/2370776.2370785	No
15	**Indexed Labels for Loop Iteration Dependent Costs**	P. Tranquilli	Electronic Proceedings in Theoretical Computer Science	To appear			2013			

TEMPLATE A2: LIST OF DISSEMINATION ACTIVITIES								
NO.	Type of activities[3]	Main leader	Title	Date/Period	Place	Type of audience[4]	Size of audience	Countries addressed
1	Workshop		**Certifying Costs in a Certified Compiler: CerCo**	23 January 2013	Satellite event of HiPeac 2013, Berlin	Scientific Community, Industry	10	
2	Workshop		**Technical Day on Innovative Techniques on Timing Analysis**	23 March 2013	Satellite event of ETAPS 2013, Rome	Scientific Community	15 (35 during joint sessions with QAPL)	
3	Presentation	C. Sacerdoti Coen	**Certified Complexity**	15 October 2010	Types workshop, Warsaw	Scientific Community	50	
4	Presentation	M. Gaboardi	**Realizability Models for Cost-Preserving Compiler Correctness**	27 March 2010	DICE workshop, Cyprus	Scientific Community	30	
5	Presentation	W. Ricciotti	**A canonical locally nameless formalisation of an algebraic logical framework**	13 October 2010	Types workshop, Warsaw	Scientific Community	50	
6	Presentation	A. Madet	**An Elementary Affine λ-Calculus with Multithreading and Side Effects**	2 June 2011	TLCA conference, Novi Sad	Scientific Community	35	
7	Presentation	C. Sacerdoti Coen	**A type system for positivity**	10 September 2011	Types workshop, Bergen	Scientific Community	50	
8	Presentation	W. Ricciotti	**The Matita Interactive Theorem Prover**	3 August 2011	CADE	Scientific	100	

[3] A drop down list allows choosing the dissemination activity: publications, conferences, workshops, web, press releases, flyers, articles published in the popular press, videos, media briefings, presentations, exhibitions, thesis, interviews, films, TV clips, posters, Other.

[4] A drop down list allows choosing the type of public: Scientific Community (higher education, Research), Industry, Civil Society, Policy makers, Medias, Other ('multiple choices' is possible).

					conference, Wroclaw	Community		
9	Presentation	W. Ricciotti	**A Web Interface for Matita**	10 July 2012	CICM conference, Bremen	Scientific Community	70	
10	Presentation	Y. Régis-Gianas	**Certifying and Reasoning on Cost Annotations in C Programs**	28 August 2012	Formal Methods for Industrial Critical Systems conference, Paris	Scientific Community, Industry	40	
11	Presentation	A. Madet	**A polynomial time lambda-calculus with multithreading and side effects**	20 September 2012	PPDP symposium, Leiven	Scientific Community	40	
12	Presentation	F. Bobot	**Separation Predicates: A Taste of Separation Logic in First-Order Logic**	15 November 2012	International Conference on Formal Engineering Methods, Kyoto	Scientific Community	60	
13	Presentation	C. Sacerdoti Coen	**On the correctness of an optimising assembler for the MCS-51 microcontroller**	13 December 2012	CPP conference, Kyoto	Scientific Community	60	
14	Presentation	W. Ricciotti	**Rating Disambiguation Errors**	15 December 2012	CPP conference, Kyoto	Scientific Community	60	
15	Presentation	B. Campbell	**An Executable Semantics for CompCert C**	13 December 2012	CPP conference, Kyoto	Scientific Community	60	
16	Presentation	P. Tranquilli	**Indexed Labels for Loop Iteration Dependent Costs**	23 March 2013	QAPL workshop, Rome	Scientific Community	35	
17	Presentation	C. Sacerdoti Coen	**Certified Complexity**	23 March 2013	QAPL workshop, Rome	Scientific Community	35	
18	Exhibition	D. Mulligan	**Certified Complexity**	4-6 May 2011	FET 2011, Prague	Scientific Community, Policy Makers,	200	

						Medias		
19	Presentation	N. Ayache	**Certifying cost annotations in compilers**	18 November 2010	University Paris 6, Paris	Scientific Community	20	
20	Presentation	R. Pollack	**A canonically nameless representation of binders**	11 June 2010	U. Penn, Philadelphia	Scientific Community	35	
21	Presentation	R. Pollack	**A canonically nameless representations of binders**	26 November 2010	INRIA-Rocquencourt, Paris	Scientific Community	10	
22	Presentation	R. Amadio	**Certifying cost annotations in compilers**	November 2010	Ecole Normale Superieure, Lyon	Scientific Community	35	
23	Presentation	C. Sacerdoti Coen	**Certified Complexity**	20 December 2011	University of Birmingham	Scientific Community	40	
24	Presentation	N. Ayache	**Certifying cost annotations in compilers**	13 December 2011	École Nationale Supérieure de Techniques Avancées	Scientific Community	25	
25	Presentation	R. Amadio	**Certifying and reasoning on cost annotations of functional programs**	February 2012	Université Paris XIII, Paris	Scientific Community	25	
26	Presentation	D. Mulligan	**Certified Complexity**	31 May 2012	University of Cambridge	Scientific Community	30	
27	Presentation	B. Campbell	**Operational Semantics in Specifications**	16 March 2010	University of Edinburgh	Scientific Community	20	
28	Presentation	B. Campbell	**An executable semantics for CompCert C**	11 November 2011	Scottish Programming Language Seminar, Heriot-Watt University	Scientific Community	35	
29	Presentation	I. Stark	**CerCo: Certified Complexity**	24 July 2012	University of Edinburgh	Scientific Community	25	

Section B (Confidential[5] or public: confidential information to be marked clearly)
Part B1

The project has not applied for any patent, trademark, etc.

TEMPLATE B1: LIST OF APPLICATIONS FOR PATENTS, TRADEMARKS, REGISTERED DESIGNS, ETC.					
Type of IP Rights[6]:	Confidential Click on YES/NO	Foreseen embargo date dd/mm/yyyy	Application reference(s) (e.g. EP123456)	Subject or title of application	Applicant (s) (as on the application)

[5] Note to be confused with the "EU CONFIDENTIAL" classification for some security research projects.

[6] A drop down list allows choosing the type of IP rights: Patents, Trademarks, Registered designs, Utility models, Others.

Part B2

Type of Exploitable Foreground[7]	Description of exploitable foreground	Confidential Click on YES/NO	Foreseen embargo date dd/mm/yyyy	Exploitable product(s) or measure(s)	Sector(s) of application[8]	Timetable, commercial or any other use	Patents or other IPR exploitation (licences)	Owner & Other Beneficiary(s) involved
General advancement of knowledge	Labelling approach	No		Compilers that induce cost models on the source code	J62.0.1 - Computer programming activities	2020		
General advancement of knowledge	Source code analysers	No		IDEs that perform both functional and non functional analysis on the source code	J62.0.1 - Computer programming activities	2025		

[7] [19] A drop down list allows choosing the type of foreground: General advancement of knowledge, Commercial exploitation of R&D results, Exploitation of R&D results via standards, exploitation of results through EU policies, exploitation of results through (social) innovation.

[8] A drop down list allows choosing the type sector (NACE nomenclature) : http://ec.europa.eu/competition/mergers/cases/index/nace_all.html

Labelling approach

The labelling approach is a new technique to transfer a cost model computed on the object code to the source code. The technique is implemented by modified compilers that keep track of basic block transformations and propagate suitable information to mark the beginning of the blocks. The markers and the knowledge on the transformations performed are then used in the propagation phase. The strong points of the technique are: 1) the backward propagation is lossless, i.e. it does not introduce any additional approximation in the cost model; 2) parametric and non parametric cost models can be propagated as well, allowing to perform parametric cost analysis on the source code and to capture dependencies of the cost on the execution history or the hardware state; 3) the propagation is parametric in the cost model: it can backward propagate at the same time space, energy and time costs; 4) the methodology has been optimised for simplicity of certification and comes with a formal proof that can reduce the trusted code base to the object code static analyser and the source and object code semantics.

The labelling approach can be exploited by software houses that develop compilers and assembler that are likely to be used for time critical software (e.g. compilers for embedded systems). Before that the methodology needs to be generalized to cover all the optimizations currently done in industrial compilers and we need to refine and test the methodology used to transfer the kind of parametric cost models used to capture time analysis for modern hardware processors. Moreover, it must be understood what techniques will be introduced to perform WCET analysis on multi-core hardware and which of these techniques are compatible with the labelling approach. Finally, the software houses will have interest in such compilers only if there will be a pressure from end users for performing time analysis on the source code. The latter will depend on successful acceptance of the wider CerCo methodology by the end user community. For all the previous reasons, we do not envision exploitation of the foreground in the short term, and at least a 7 years timescale is probably a non optimistic prevision.

Source code analysers

The most interesting promise of the CerCo project is the development of a new generation of software analysers that are able to perform at once functional and non functional analysis on high level languages. Working on the same language, the non functional analysis can exploit all the knowledge provided by the functional analysis that, in turn, obtains new constraints from the non functional one. For example, the non functional analysis may show that a certain recursive function cannot be called more than 100 times (to avoid running out of stack); in turn this limitation can show that the execution time of the function cannot exceed a certain threshold. Moreover, working on the source code, it is simpler for the user to provide invariants on the code (e.g. by means of assertions) and the range of techniques that can collaborate on automatically inferring information on the program behavior is very large.

Integrated source code analysers can be exploited by software houses that develop tools to perform static WCET analysis of programs. If the CerCo technique will actually deliver more precise execution bounds with easier user interaction and more parametricity, it may be the case that start-ups could be created to challenge the current domination by a couple of large software houses that monopolize the WCET market.

Before the foreground can be exploited, several research issues still need to be addressed and better understood.

1. The methodology to compute and transfer parametric cost models that deal with modern hardware needs to be refined and tested. Testing requires first to obtain a clock precise model of the microprocessor or, in alternative, to interface with existent WCET tools like AbsInt to compute the parametric cost model to be transferred using the methodology to be tested. This objective may be reached in the medium-long term.

2. The mix of techniques used to perform the parametric time analysis on the source code needs to be consolidated and made more robust, introducing a degradation of precision in case of failure of the analysis as a tradeoff towards completion of the analysis itself. This objective may be reached in the medium term, but we expect the combination of techniques to yield superior results with respect to current WCET techniques only in the long medium-long term.

3. Industrial developers use Integrated Development Environment (IDEs) that yield a number of benefits over the simple compilation, and they are not willing to abandon the IDE or even to re-train to a different one. Therefore the source code analyser and the cost preserving compiler need to be integrated into the IDE. The integration is likely to be time consuming.

4. Yet to be developed techniques to tame the complexity of static analysis of multi-core architectures need to be tested in combination with the CerCo approach. The multi-core architectures are born to optimise the average case, but they make prediction of the worse case extremely ineffective. The reason is that the time behavior of a program is affected by the behavior of other programs that run concurrently with it and that are unknown during the analysis. Ignorance during the analysis introduce very pessimistic cost models because in case of ignorance the worst scenario must be assumed. In turn, the analysis yields useless bounds that can be order of magnitudes higher than the worst observed cases. The CerCo methodology is affected by the problem, and does not contribute a solution to it. Other European projects and initiatives are putting pressure on the hardware developers to deliver also alternative systems that would be easier to analyse at the price of reducing the average performance. Hardware randomization seems to be an important ingredient towards that aim, at the price of abandoning exact time analysis in favor of probabilistic time analysis. In CerCo we have already speculatively considered probabilistic time analysis, coming to the conclusion that the labelling methodology to transfer the cost model works equally well with probabilistic models. Further research is needed to understand which techniques can then be used to certify probabilistic time bounds on the source code once the probabilistic cost model has been induced.

For the previous reasons we think that the industrial exploitation of the foreground is likely to be performed only in a medium-long timeframe. On the other hand, hardware development is currently undergoing a true revolution, with a swift move from single processors and uniform memory models to systems-on-chip that host multiple multi-core processors, networks on chips and extremely non uniform memory models. Whether in a medium-long timeframe these more complex solutions will be used for safety critical software is currently unknown and, if that's the case, whether static analysis will be still possible at all remains, at the moment, an open question.