Certified Complexity (CerCo)[⋆]

R. M. Amadio[4], N. Ayache[3,4], F. Bobot[3,4], J. P. Boender[1], B. Campbell[2], I. Garnier[2], A. Madet[4], J. McKinna[2], D. P. Mulligan[1], M. Piccolo[1], R. Pollack[2], Y. Régis-Gianas[3,4], C. Sacerdoti Coen[1], I. Stark[2], and P. Tranquilli[1]

[1] Dipartimento di Informatica - Scienza e Ingegneria, Universitá di Bologna
[2] LFCS, School of Informatics, University of Edinburgh
[3] INRIA (Team πr2)
[4] Universitè Paris Diderot

Abstract. This paper provides an overview of the FET-Open Project CerCo ('Certified Complexity'). The project's main achievement is the development of a technique for performing static analyses of non-functional properties of programs (time, space) at the source level, without loss of accuracy and with a small, trusted code base. The main software component developed is a certified compiler producing cost annotations. The compiler translates source code to object code and produces an instrumented copy of the source code. This instrumentation exposes at the source level—and tracks precisely—the actual (non-asymptotic) computational cost of the input program. Untrusted invariant generators and trusted theorem provers are then used to compute and certify the parametric execution time of the code.

1 Introduction

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, and so on). By contrast, non-functional properties are generally checked on low-level object code, but also demand information about high-level functional behaviour 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 optimisations; 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)

[⋆] The project CerCo acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number: 243881 functional analysis performed only on object code makes any contribution from the programmer hard, leading to less precision in the estimates.

Vision and approach. We want to reconcile functional and non-functional analyses: to share information and perform both at the same time on source code. What has previously prevented this approach is the lack of a uniform and precise cost model for high-level code: 1) each statement occurrence is compiled differently and optimisations 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, all of which are 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 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 [15]. The techniques currently used by the Worst Case Execution Time (WCET) community, which performs its analysis on object code , are still available but can now be coupled with an 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, source code analysis can be made during early development stages, when components have been specified but not implemented: source code modularity means that it is enough to specify the non-functional behaviour of unimplemented components.

Contributions. We have developed what we refer to as *the labeling approach* [4], a technique 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 this technique. We have implemented such a compiler from C to object binaries for the 8051 micro-controller, and verified it in an interactive theorem prover. We have implemented a Frama-C plugin [7] 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 plugin 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 plugin can automatically compute and certify the exact reaction time of Lustre [5] data flow programs compiled into C.

2 Project context and objectives

Formal methods for verification of functional properties of programs have now reached a level of maturity and automation that is facilitating a slow but increasing adoption in production environments. For safety critical code, it is becoming commonplace to combine rigorous software engineering methodologies and testing with static analysis, taking the strong points of each approach and mitigating their weaknesses. Of particular interest 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 bleaker and the future seems to be getting even worse. Most industries verify that real time systems meet their deadlines by simply performing many runs of the system and timing their execution, computing the maximum time and adding an empirical safety margin, claiming the result to be a bound for the WCET 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 are all focused on the improvement of the average case performance, not the predictability of the worst case. Execution time is becoming increasingly dependent on execution history and 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 for static analysis, and obtaining them is becoming harder as a consequence of the increased sophistication of hardware design.

Despite the aforementioned problems, the need for reliable real time systems and programs is increasing, and there is increasing pressure from the research community towards the differentiation of hardware. The aim is to introduce alternative hardware with more predictable behaviour and hence more suitability for static analyses, for example, the decoupling of execution time from execution history by introducing randomisation [6].

In the CerCo project [1] we do not try to address this problem, optimistically assuming that static analysis of non-functional properties of programs will become feasible in the longer 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, the analysis of non-functional properties is entirely performed on the object code to exploit clock-precise hardware models.

Analysis currently takes place on object code for two main reasons. First, there is a 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 identifying a fragment of object code and a single high level instruction is infeasible. Even the control flow of the object and source code can be very different as a result of optimisations, for example aggressive loop optimisations may completely transform source level loops. 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 gaining momentum. However, unless we can provide a replacement for such cost models, this literature's future practical impact looks to be minimal. Some hope has been provided by the EmBounded project [8], 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 provides a uniform model at the price of the model's precision (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 estimates because no knowledge on the hardware state can be assumed when executing the fragment. By analysing longer runs the bound obtained becomes more precise because the lack of knowledge on the initial state has less of an effect 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 may assign to a loop body a cost that is a function of the number of iterations performed. As another example, the cost of a block 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 one to analyse small code fragments without losing precision: in the analysis of the code fragment we do not have to ignore the initial hardware state. On the contrary, we can assume that we know exactly which state (or mode, as the WCET literature calls it) we are in.

The cost of an execution is 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 [14] performs the analysis on the object code, where the logic of the program is harder to reconstruct and most information available at the source code level 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 assumptions on the source code (which may be wrong) to object code constraints.

The task is error prone and hard, especially in the presence of complex compiler optimisations.

The CerCo approach has the potential to dramatically improve the state of the art. By performing control and data flow analyses 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: they can just be applied at the source code level.

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. Functional properties can be analysed in early development stages, while analysis of non-functional properties may come too late to avoid expensive changes to the program architecture. Our approach already works in early development stages by axiomatically attaching costs to unimplemented components.

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 the use more off-the-shelf components (e.g. provers and invariant generators) whilst reducing the trusted code base at the same time.

3 The typical CerCo workflow

We illustrate the workflow we envisage (on the right of Figure 1) on an example program (on the left of Figure 1). The user writes the program and feeds it to the CerCo compiler, which outputs an instrumented version of the same program

```
char a[] = {3, 2, 7, -4};
char treshold = 4;

int main() {
  char j;
  char *p = a;
  int found = 0;
  for (j=0; j < 4; j++) {
    if (*p <= treshold)
      { found++; }
    p++;
  }
  return found;
}
```

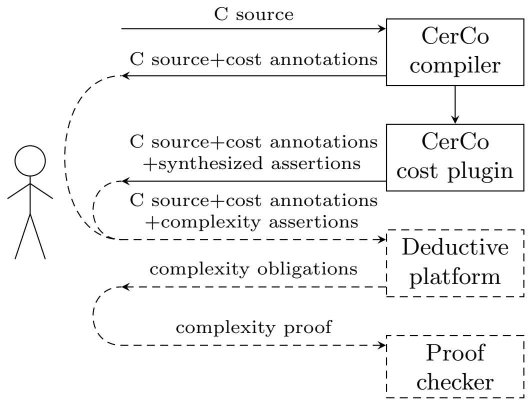
C source
C source+cost annotations
CerCo compiler
C source+cost annotations +synthesized assertions
CerCo cost plugin
C source+cost annotations +complexity assertions
Deductive platform
complexity obligations
complexity proof
Proof checker

Fig. 1. On the left: code to count the array's elements that are greater or equal to the treshold. On the right: CerCo's interaction diagram. CerCo's components are drawn solid.

that updates global variables that record the elapsed execution time and the stack space usage. The red lines in Figure 2 are the instrumentation introduced by the compiler. The annotated program can then be enriched with complexity assertions in the style of Hoare logic, that are passed to a deductive platform (in our case Frama-C). We provide as a Frama-C cost plugin a simple automatic synthesiser for complexity assertions (the blue lines in Figure 2), which can be overridden by the user to increase or decrease accuracy. From the assertions, a general purpose deductive platform produces proof obligations which in turn can be closed by automatic or interactive provers, ending in a proof certificate. Nine proof obligations are generated from Figure 2 (to prove that the loop invariant holds after one execution if it holds before, to prove that the whole program execution takes at most 1228 cycles, etc.). The CVC3 prover closes all obligations in a few seconds on routine commodity hardware.

4 Main scientific and technical results

We will now review the main results that the CerCo project has achieved.

4.1 The (basic) labeling approach

The labeling approach is the foundational insight that underlies all the developments in CerCo. It allows the tracking of the evolution of basic blocks during the compilation process in order to propagate the cost model from the object code to the source code without losing precision in the process.

Problem statement. Given a source program P, we want to obtain an instrumented source program P', written in the same programming language, and the

```
int __cost = 33; int __stack = 5, __stack_max = 5;
void __cost_incr(int incr) { __cost = __cost + incr; }
void __stack_incr(int incr) {
  __stack = __stack + incr;
  __stack_max = __stack_max < __stack ? __stack : __stack_max;
}

char a[4] = { 3, 2, 7, 252, };
char treshold = 4;

/*@ behaviour stack_cost:
      ensures __stack_max <=
              __max(\old(__stack_max), \old(__stack));
      ensures __stack == \old(__stack);

    behaviour time_cost:
      ensures __cost <= \old(__cost)+1228; */
int main(void)
{
  char j;
  char *p;
  int found;
  __stack_incr(0); __cost_incr(91);
  p = a;
  found = 0;
  __l: /* internal */
  /*@ for time_cost: loop invariant
        __cost <=
        \at(__cost,__l)+220*(__max(\at(5-j,__l), 0)
                        -__max (5-j, 0));
      for stack_cost: loop invariant
        __stack_max == \at(__stack_max,__l);
      for stack_cost: loop invariant
        __stack == \at(__stack,__l);
      loop variant 4-j; */
  for (j = 0; j < 4; j++) {
    __cost_incr(76);
    if (*p <= treshold) {
      __cost_incr(144);
      found++;
    } else {
      __cost_incr(122);
    }
    p++;
  }
  __cost_incr(37); __stack_incr(-0);
  return found;
}
```

Fig. 2. The instrumented version of the program in Figure 1, with instrumentation added by the CerCo compiler in red and cost invariants added by the CerCo Frama-C plugin in blue. The `__cost`, `__stack` and `__stack_max` variables hold the elapsed time in clock cycles and the current and maximum stack usage. Their initial values hold the clock cycles spent in initialising the global data before calling `main` and the space required by global data (and thus unavailable for the stack).

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 behaviour, i.e. they must produce that same output and intermediate observables; 3) P and O must have the same functional behaviour; 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 [4], implemented by four software components that are used in sequence.

The first component labels the source program P by injecting label emission statements in appropriate positions to mark the beginning of basic blocks. 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 its execution is observable.

The second component is a labeling 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 behaviour of the program and the traces of observables.

The third component is a static object code analyser. It takes in input a labeled object code and it computes the scope of each of its label emission statements, i.e. the tree of instructions that may be executed after the statement and before a new label emission is encountered. 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 labeling. The analyser fails if the labeling 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.

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 problem statement obviously hold, with 2 and 3 being a consequence of the definition of a correct labeling 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 fourth 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 labeling: 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 fifth requirement, additional requirements must be imposed on the object code labeling to avoid all uses of the maximum in the cost computation: the labeling 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 labeling approach only rely on the correctness and precision of the object code labeling. The simplest way to achieve them is to impose correctness and precision requirements also on the initial labeling produced by the first software component, and to ask the 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 optimisations like peeling or unrolling are prevented. Moreover, precision of the object code labeling 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 the presence of stateful hardware whose state influences the cost of operations, like pipelines and caches. In the next subsection we will see an extension of the basic labeling approach to cover this situation.

The labeling approach described in this section can be applied equally well and with minor modifications to imperative and functional languages [2]. We have tested it on a simple imperative language without functions (a ‘while’ language), on a subset of C and on 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 labeling of the source code must be changed when the compilation is based on CPS translations. In particular, we need to introduce labels emitted before a statement is executed and also 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 could support our labeling approach.

4.2 Dependent labeling

The core idea of the basic labeling 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 cache contents 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 complex to understand and reason about. 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 losing 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 present here the basic intuition in a simplified form: the technical details that allow us to handle the general case are more complex and can be found in [13]. 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.

Dependent labeling can also be applied to allow the compiler to duplicate blocks that contain labels (e.g. in loop optimisations) [13]. The effect 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 of the loop body (that executes the first iteration) with the conditional execution of the loop (for the successive iterations). Because of further optimisations, 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 of introducing within the source code, and for each loop, a variable that counts the number of iterations. The loop optimisation 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 optimising compiler are minimal.

4.3 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: 1) The inferred synthetic bounds are indeed correct as long as the general purpose tool is; 2) 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 plugin and its application to the Lustre compiler. Frama-C [7] is a set of analysers for C programs with a specification language called ACSL. New analyses can be dynamically added via a plugin system. For instance, the Jessie plugin 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 plugin 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 plugin 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 plugin, 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 plugin, 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 for programming synchronous systems, with the language coming 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 plugin and the Lustre wrapper on the C programs generated by the Lustre compiler. For programs consisting of a few hundred lines of code, the cost plugin 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: 1) there are no recursive functions; 2) 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. Then the computation of the cost of the function is performed by traversing its control flow graph. If the function f calls the function g then the function g is processed before the function f. 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 two different means: 1) by using more precise variants; 2) by annotating functions with cost specifications. 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 [11] is an elegant alternative. It is a program logic with a new notion of conjunction to express spatial heap separation. Bobot has recently introduced automatically generated separation predicates to simulate separation logic reasoning in the Jessie plugin where the specification language, the verification condition generator, and the theorem provers were not designed with separation logic in mind. CerCo’s plugin can exploit the separation predicates to automatically reason on the cost of execution of simple heap manipulation programs such as an in-place list reversal.

4.4 The CerCo compiler

In CerCo we have developed a certain number of cost preserving compilers based on the labeling 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 optimisation 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 [3] and its costs certified. The code distributed is extracted 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 any advanced optimisations 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 plugin can work without recompilation with all compilers.

The 8051/8052 microprocessor is a very simple one, with constant-cost instructions. It was chosen to separate the issue of exact propagation of the cost model from the orthogonal problem of the static analysis of object code that may require approximations or dependent costs.

The (trusted) CerCo compiler implements the following optimisations: cast simplification, constant propagation in expressions, liveness analysis driven spilling of registers, dead code elimination, branch displacement, and tunneling. The two latter optimisations are performed by our optimising assembler [10]. The back-end of the compiler works on three address instructions, preferred to static single assignment code for the simplicity of the formal certification.

The CerCo compiler is loosely based on the CompCert compiler [9], a recently developed certified compiler from C to the PowerPC, ARM and x86 microprocessors. Contrary 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 of a compiler from Pascal to MIPS used by François Pottier for a course at the Ecole Polytechnique.

The main peculiarities of the CerCo compiler are the following.

1. All the intermediate languages include label emitting instructions to implement the labeling approach, and the compiler preserves execution traces.
2. Traditionally compilers target 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 integrated also an optimising assembler and a static analyser.
3. In order to avoid implementing a linker and a loader, we do not support separate compilation and external calls. Adding them is a transparent process to the labeling 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 data and it must move data between memory regions in the proper way. Moreover the size of pointers to different regions is not uniform. 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 implements a stack in external memory.

4.5 Formal 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 behaviour. We actually induce two cost models, one for time and one for stack space consumption. 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.

The details on the proof techniques employed and the proof sketch can be collected from the CerCo deliverables and papers. In this section we will only hint at the correctness statement, which turned out to be more complex than what we expected at the beginning.

The statement. 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 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 formally proved 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 and returns is a requirement for precision: the labeling 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 or 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, or 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.

5 Future work

All the CerCo software and deliverables can be found on the CerCo homepage at `http://cerco.cs.unibo.it`.

The results obtained so far are encouraging and provide evidence that it is possible to perform static time and space analysis at the source level without losing accuracy, reducing the trusted code base and reconciling the study of functional and non-functional properties of programs. The techniques introduced seem to be scalable, cover both imperative and functional languages and are compatible with every compiler optimisation considered by us so far.

To prove that compilers can keep track of optimisations and induce a precise cost model on the source code, we targeted a simple architecture that admits a cost model that is execution history independent. The most important future work is dealing with hardware architectures characterized by history dependent stateful components, like caches and pipelines. The main issue consists in assigning a parametric, dependent cost to basic blocks that can be later transferred by the labeling approach to the source code and represented in a meaningful way to the user. The dependent labeling approach that we have studied seems a promising tool to achieve this goal, but the cost model generated for a realistic processor could be too large and complex to be exposed in the source code. Further study is required to evaluate the technique on a realistic processor and to introduce early approximations of the cost model to make the technique feasible.

Examples of further future work consist in improving the cost invariant generator algorithms and the coverage of compiler optimizations, in combining the labeling approach with the type and effect discipline of [12] to handle languages with implicit memory management, and in experimenting with our tools in early development phases. Some larger case study is also necessary to evaluate the CerCo's prototype on realistic, industrial-scale programs.

References

1. Amadio, R., Asperti, A., Ayache, N., Campbell, B., Mulligan, D., Pollack, R., Régis-Gianas, Y., Coen, C.S., Stark, I.: Certified complexity. Procedia Computer Science 7, 175–177 (2011), proceedings of the 2nd European Future Technologies Conference and Exhibition 2011 (FET 11)
2. Amadio, R., Régis-Gianas, Y.: Certifying and reasoning on cost annotations of functional programs. In: Peña, R., Eekelen, M., Shkaravska, O. (eds.) Foundational and Practical Aspects of Resource Analysis, LNCS, vol. 7177, pp. 72–89. Springer Berlin Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-32495-6_5, extended version to appear in Higher Order and Symbolic Computation, 2013
3. Asperti, A., Ricciotti, W., Coen, C.S., Tassi, E.: The matita interactive theorem prover. In: CADE. LNCS, vol. 6803, pp. 64–69. Springer (2011)
4. Ayache, N., Amadio, R., Régis-Gianas, Y.: Certifying and reasoning on cost annotations in C programs. In: Stoelinga, M., Pinger, R. (eds.) Formal Methods for Industrial Critical Systems, LNCS, vol. 7437, pp. 32–46. Springer Berlin Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-32469-7_3
5. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: Lustre: A declarative language for programming synchronous systems. In: POPL. pp. 178–188. ACM Press (1987)
6. Cazorla, F., Quiñones, E., Vardanega, T., Cucu, L., Triquet, B., Bernat, G., Berger, E., Abella, J., Wartel, F., Houston, M., Santinelli, L., Kosmidis, L., Lo, C., Maxim, D.: Proartis: Probabilistically analysable real-time systems. Transactions on Embedded Computing Systems (2012)
7. Correnson, L., Cuoq, P., Kirchner, F., Prevosto, V., Puccetti, A., Signoles, J., Yakobowski, B.: Frama-C user manual. CEA-LIST, Software Safety Laboratory, Saclay, F-91191, http://frama-c.com/
8. Hammond, K., Dyckhoff, R., Ferdinand, C., Heckmann, R., Hofmann, M., Jost, S., Loidl, H.W., Michaelson, G., Pointon, R.F., Scaife, N., Sérot, J., Wallace, A.: The Embounded Project (Project Start Paper). In: TFP. Trends in Functional Programming, vol. 6, pp. 195–210. Intellect, UK/The University of Chicago Press, USA (2005)
9. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009), http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf
10. Mulligan, D.P., Sacerdoti Coen, C.: On the correctness of an optimising assembler for the intel mcs-51 microprocessor. In: Hawblitzel, C., Miller, D. (eds.) CPP. LNCS, vol. 7679, pp. 43–59. Springer (2012)
11. Reynolds, J.C.: Intuitionistic reasoning about shared mutable data structure. In: Millennial Perspectives in Computer Science. pp. 303–321. Palgrave (2000)
12. Talpin, J.P., Jouvelot, P.: The type and effect discipline. Inf. Comput. 111(2), 245–296 (1994)
13. Tranquilli, P.: Indexed labels for loop iteration dependent costs (Jan 2013), http://www.cs.unibo.it/~tranquil/content/docs/indlabels.pdf, to appear in EPTCS, proceedings of QAPL 2013
14. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D.B., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P.P., Staschulat, J., Stenström, P.: The worst-case execution-time problem - overview of methods and survey of tools. ACM Trans. Embedded Comput. Syst. 7(3) (2008)
15. Wögerer, W.: A survey of static program analysis techniques. Tech. rep., Technische Universität Wien (2005)