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 πr^2)
[4] Universitè Paris Diderot

Abstract. We provide an overview of the FET-Open Project CerCo ('Certified Complexity'). Our main achievement is the development of a technique for analysing non-functional properties of programs (time, space) at the source level with little or no loss of accuracy and a small trusted code base. The core component is a C compiler, verified in Matita, that produces an instrumented copy of the source code in addition to generating object code. This instrumentation exposes, and tracks precisely, the actual (non-asymptotic) computational cost of the input program at the source level. Untrusted invariant generators and trusted theorem provers may then be used to compute and certify the parametric execution time of the code.

1 Introduction

Programs can be specified with both functional constraints (what the program must do) and non-functional constraints (what time, space or other resources the program may use). In the current state of the art, functional properties are verified by combining user annotations—preconditions, invariants, and so on—with a multitude of automated analyses—invariant generators, type systems, abstract interpretation, theorem proving, and so on—on the program's high-level source code. By contrast, many non-functional properties are verified using analyses on low-level object code, but these analyses may then need information about the high-level functional behaviour of the program that must then be reconstructed. This analysis on low-level object code has several problems:

– It can be hard to deduce the high-level structure of the program after compiler optimisations. The object code produced by an optimising compiler may have radically different control flow to the original source code program.
– Techniques that operate on object code are not useful early in the development process of a program, yet problems with a program's design or implementation are cheaper to resolve earlier in the process, rather than later.

* 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

- Parametric cost analysis is very hard: how can we reflect a cost that depends on the execution state, for example the value of a register or a carry bit, to a cost that the user can understand looking at the source code?
- Performing functional analyses on object code makes it hard for the programmer to provide information about the program and its expected execution, leading to a loss of precision in the resulting analyses.

Vision and approach. We want to reconcile functional and non-functional analyses: to share information and perform both at the same time on high-level source code. What has previously prevented this approach is the lack of a uniform and precise cost model for high-level code as each statement occurrence is compiled differently, optimisations may change control flow, and 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.

We envision a new generation of compilers that track program structure through compilation and optimisation and exploit this information to define a precise, non-uniform cost model for source code that accounts for runtime state. With such a cost model we can reduce non-functional verification to the functional case and exploit the state of the art in automated high-level verification [18]. The techniques currently used by the Worst Case Execution Time (WCET) community, who perform analyses on object code, are still available but can be coupled with additional source-level analyses. Where our approach produces overly complex cost models, safe approximations can be used to trade complexity with precision. Finally, source code analysis can be used early in the development process, when components have been specified but not implemented, as modularity means that it is enough to specify the non-functional behaviour of missing components.

Contributions. We have developed *the labelling approach* [5], 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, and have implemented such a compiler from C to object binaries for the 8051 microcontroller for predicting execution time and stack space usage, verifying it in an interactive theorem prover. As we are targeting an embedded microcontroller we do not consider dynamic memory allocation.

To demonstrate source-level verification of costs we have implemented a Frama-C plugin [10] that invokes the compiler on a source program and uses it to generate invariants on the high-level source that correctly model low-level costs. 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. Finally, we have conducted several case studies, including showing that the plugin can automatically compute and certify the exact reaction time of Lustre [7] data flow programs compiled into C.

2 Project context and approach

Formal methods for verifying functional properties of programs have now reached a level of maturity and automation that their adoption is slowly increasing in production environments. For safety critical code, it is becoming commonplace to combine rigorous software engineering methodologies and testing with static analyses, taking the strengths of each 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 enriched with new safety guarantees: every method contributes knowledge (e.g. new invariants) that becomes an assumption for later analysis.

The outlook for verifying non-functional properties of programs (time spent, memory used, energy consumed) is bleaker. 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 architecture have been 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 due to the increased sophistication of hardware design.

Despite these problems, the need for reliable real time systems and programs is increasing, and there is pressure from the research community for the introduction of hardware with more predictable behaviour, which would be more suitable for static analysis. One example, being investigated by the Proartis project [8], is to decouple execution time from execution history by introducing randomisation.

In CerCo [2] we do not address this problem, optimistically assuming that improvements in low-level timing analysis or architecture will make verification feasible in the longer term. Instead, the main objective of our work is to bring together the static analysis of functional and non-functional properties, which in the current state of the art are independent activities with limited exchange of information: while the functional properties are verified on the source code, the analysis of non-functional properties is performed on object code to exploit clock-precise hardware models.

2.1 Current object-code methods

Analysis currently takes place on object code for two main reasons. First, there cannot be a uniform, precise cost model for source code instructions (or even basic blocks). During compilation, high level instructions are broken up 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 assuming such a model is growing and gaining momentum. However, unless we 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 [11], which compositionally compiles high-level code to a byte code that is executed by an interpreter with guarantees on the maximal execution time spent for each byte code instruction. This provides a uniform model at the expense of the model's precision (each cost is a pessimistic upper bound) and the performance of the executed code (because the byte code is interpreted 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 as no knowledge of the hardware state prior to executing the fragment can be assumed. By analysing longer runs the bound obtained becomes more precise because the lack of information about the initial state has a relatively small impact.

To calculate the cost of an execution, value and control flow analyses are required to bound the number of times each basic block is executed. Currently, state of the art WCET analysis tools, such as AbsInt's aiT toolset [1], perform these analyses on object code, where the logic of the program is harder to reconstruct and most information available at the source code level has been lost; see [17] for a survey. Imprecision in the analysis can lead 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.

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.

2.2 CerCo's approach

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 processor timing analysis.

By embracing compilation, instead of avoiding it like EmBounded did, a CerCo compiler can both produce efficient code and return costs that are as precise as the processor timing analysis can be. Moreover, our costs can be parametric: the cost of a block can depend on actual program data, on a summary of the execution history, or on an approximated representation of the hardware state. For example, loop optimisations may assign a cost to a loop body 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, rather, we may assume that we know exactly which state (or mode, as the WCET literature calls it) we are in.

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. Instead, this work is done at the source level using tools of the user's choice. 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. There are no limitations on the types of loops or data structures involved. Parametric cost analysis becomes the default one, with non-parametric bounds used as a last resort when the user decides to trade the complexity of the analysis with its precision. *A priori*, no technique previously used in traditional WCET is lost: processor timing analyses can be used by the compiler on the object code, and the rest can be applied at the source code level. Our approach can also work in the early stages of development by axiomatically attaching costs to unimplemented components.

Software used to verify properties of programs must be as bug free as possible. The trusted code base for verification consists of 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, the linear programming libraries used, and also the formal models of the hardware under analysis, for example. 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 costs added to the source code are indeed those incurred 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 these tools 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

```
char a[] = {3, 2, 7, 14};
char threshold = 4;

int count(char *p, int len) {
  char j;
  int found = 0;
  for (j=0; j < len; j++) {
    if (*p <= threshold)
      found++;
    p++;
    }
  return found;
}

int main() {
  return count(a,4);
}
```

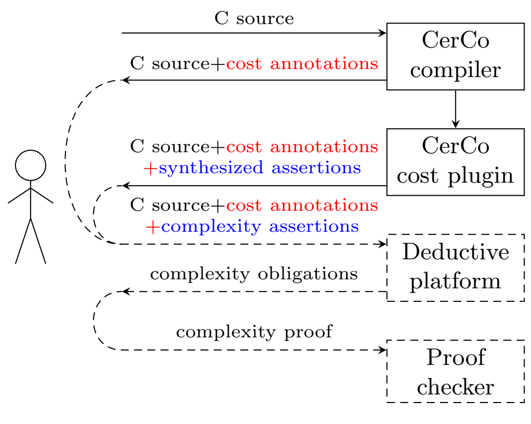
C source
CerCo compiler
C source+cost annotations
CerCo cost plugin
C source+cost annotations +synthesized assertions
C source+cost annotations +complexity assertions
Deductive platform
complexity obligations
complexity proof
Proof checker

Fig. 1. On the left: C code to count the number of elements in an array that are less than or equal to a given threshold. On the right: CerCo's interaction diagram. Components provided by CerCo are drawn with a solid border.

of 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 that updates global variables that record the elapsed execution time and the stack space usage. The red lines in Figure 2 introducing variables, functions and function calls starting with `__cost` and `__stack` are the instrumentation introduced by the compiler. For example, the two calls at the start of `count` say that 4 bytes of stack are required, and that it takes 111 cycles to reach the next cost annotation (in the loop body). The compiler measures these on the labelled object code that it generates.

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 which can be overridden by the user to increase or decrease accuracy. These are the blue comments starting with `/*@` in Figure 2, written in Frama-C's specification language, ACSL. 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.

Twelve 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 1358 cycles, and so on). Note that the synthesised time bound for `count`, $178 + 214 * (1 + \texttt{len})$ cycles, is parametric in the length of the array. The CVC3 prover closes all obligations within half a minute on routine commodity hardware. A simpler non-parametric version can be solved in a few seconds.

4 Main scientific and technical results

First we describe the basic labelling approach and our compiler implementations that use it. This is suitable for basic architectures with simple cost models. Then we will discuss the dependent labelling extension which is suitable for more advanced processor architectures and compiler optimisations. At the end of this section we will demonstrate automated high level reasoning about the source level costs provided by the compilers.

4.1 The (basic) labelling approach

The labelling approach is the foundational insight that underlies all the developments in CerCo. It allows the evolution of basic blocks to be tracked throughout 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 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 labelling software components. We solve the problem in four stages [5], 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. These are the positions where the cost instrumentation will appear in the final output. 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. For the example in Section 3 this is just the original C code with "EMIT" instructions added at every point a `__cost_incr` call appears in the final code.

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

char a[4] = {3, 2, 7, 14};  char threshold = 4;

/*@ behavior stack_cost:
      ensures __stack_max <= __max(\old(__stack_max), 4+\old(__stack));
      ensures __stack == \old(__stack);
    behavior time_cost:
      ensures __cost <= \old(__cost)+(178+214*__max(1+\at(len,Pre), 0));
*/
int count(char *p, int len) {
  char j;  int found = 0;
  __stack_incr(4);  __cost_incr(111);
  __l: /* internal */
  /*@ for time_cost: loop invariant
        __cost <= \at(__cost,__l)+
                  214*(__max(\at((len-j)+1,__l), 0)-__max(1+(len-j), 0));
      for stack_cost: loop invariant
        __stack_max == \at(__stack_max,__l);
      for stack_cost: loop invariant
        __stack == \at(__stack,__l);
      loop variant len-j;
  */
  for (j = 0; j < len; j++) {
    __cost_incr(78);
    if (*p <= threshold) { __cost_incr(136); found ++; }
    else { __cost_incr(114); }
    p ++;
  }
  __cost_incr(67);  __stack_incr(-4);
  return found;
}

/*@ behavior stack_cost:
      ensures __stack_max <= __max(\old(__stack_max), 6+\old(__stack));
      ensures __stack == \old(__stack);
    behavior time_cost:
      ensures __cost <= \old(__cost)+1358;
*/
int main(void) {
  int t;
  __stack_incr(2);  __cost_incr(110);
  t = count(a,4);
  __stack_incr(-2);
  return t;
}
```

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).

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 behaviour of the program and the traces of observables, including the labels 'emitted'.

The third component analyses the labelled object code to compute the scope of each of its label emission statements, i.e. the instructions that may be executed after the statement and before a new label emission is encountered, and then computes the maximum cost of each. Note that we only have enough information at this point to compute the cost of loop-free portions of code. We will consider how to ensure that every loop is broken by a cost label shortly.

The fourth and final component replaces the labels in the labelled version of the source code produced at the start with the costs computed for each label's scope. This yields the instrumented source code. For the example, this is the code in Figure 2, except for the specifications in comments, which we consider in Section 4.5.

Correctness. Requirements 1 and 2 hold because of the non-invasive labelling procedure. Requirement 3 can be satisfied by implementing compilation correctly. It is obvious that the value of the global cost variable of the instrumented source code is always 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 soundness requirement, we need to ensure 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. This ensures that every label's scope is a tree of instructions, with the cost being the most expensive path. To satisfy also the precision requirement, we must make the scopes flat sequences of instructions. We require a *precise* labelling where every label is emitted at most once and both branches of each 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 way to achieve that is to impose correctness and precision requirements on the source code labelling produced at the start, and to demand that the compiler preserves these properties too. The latter requirement imposes serious limitations on the compilation strategy and optimisations: the compiler may not duplicate any code that contains label emission statements, like loop bodies. Therefore various loop optimisations like peeling or unrolling are prevented. Moreover, precision of the object code labelling is not sufficient *per se* to obtain global precision: we implicitly assumed that a precise constant cost can be assigned 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 Section 4.4 we will see an extension of the basic labelling approach which tackles these problems.

In CerCo we have developed several 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 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). We describe the C compilers in detail in the following section.

Two compilation chains were implemented for a purely functional higher-order language [3]. 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 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.

4.2 The CerCo C compilers

We implemented two C compilers, one implemented directly in OCaml and the other implemented in Matita, an interactive theorem prover [4]. The first acted as a prototype for the second, but also supported MIPS and acted as a testbed for more advanced features such as the dependent labelling approach in Section 4.4.

The second C compiler is the *Trusted CerCo Compiler*, whose cost predictions are formally verified. The executable code is OCaml code extracted from the Matita implementation. The Trusted CerCo Compiler only targets the C language and the 8051/8052 family, and does not yet implement any advanced optimisations. Its user interface, however, is the same as the other version for interoperability purposes. In particular, the Frama-C CerCo plugin descibed in Section 4.5 can work without recompilation with both of our C compilers.

The 8051 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 low-level timing analysis of object code that may require approximation 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 tunnelling. The two latter optimisations are performed by our optimising assembler [14]. 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 [13], a recently developed certified compiler from C to the PowerPC, ARM and x86 microprocessors. In contrast to CompCert, both the CerCo code and its certification are fully 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 École Polytechnique. The main differences in the CerCo compiler are:

– All the intermediate languages include label emitting instructions to implement the labelling approach, and the compiler preserves execution traces.
– Instead of targeting an assembly language with additional macro-instructions which are expanded before assembly, we directly produce object code in order to perform the timing analysis, using an integrated optimising assembler.
– In order to avoid the additional work of implementing a linker and a loader, we do not support separate compilation and external calls. Adding them is orthogonal to the labelling approach and should not introduce extra problems.
– We target an 8-bit processor, in contrast to CompCert's 32-bit targets. This requires many changes and more compiler code, but it is not fundamentally more complex. The proof of correctness, however, becomes much harder.
– 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.

4.3 Formal certification of the CerCo compiler

We have formally certified in Matita that the cost models induced on the source code by the Trusted CerCo Compiler correctly and precisely predict the object code behaviour. There are two cost models, one for execution time and one for stack space consumption. We show the correctness of the prediction only for those programs that do not exhaust the available stack space, a property that—thanks to the stack cost model—we can statically analyse on the source code in sharp contrast to other certified compilers. Other projects have already certified the preservation of functional semantics in similar compilers, so we have not attempted to directly repeat that work and assume functional correctness for most passes. In order to complete the proof for non-functional properties, we have introduced a new, structured, form of execution trace, with the related notions for forward similarity and the intensional 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 found in the CerCo deliverables and papers [9]. In this section we will only hint at the correctness statement, which turned out to be more complex than expected.

The correctness 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 continuing as before. For these programs the overall execution time does not make sense. The same is true for reactive programs that spend an unpredictable amount of time in I/O. Instead, what is interesting is the reaction time — the time spent between I/O events. Moreover, we are interested in predicting and ruling out crashes due to running out of space on certain inputs. Therefore we need a statement that talks about sub-runs of a program. A natural candidate is that the time predicted on the source code and spent on the object code by two corresponding sub-runs are the same. To make this statement formal we must identify the corresponding sub-runs and how to single out those that are meaningful. We introduce the notion of a *measurable* sub-run of a run which does not exhaust the available stack before or during the sub-run, 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 statement. The stack usage is bounded using the stack usage model that is computed by the compiler.

The statement that we formally proved is: for each C run with a measurable sub-run, there exists an object code run with a sub-run, with the same execution trace for both the prefix of the run and the sub-run itself, and where the time spent by the object code in the sub-run is the same as the time 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 necessary for a run to be meaningful, because the source semantics has no notion of running out of memory. Balancing function calls and returns is a requirement for precision: the labelling approach allows the scope of a label to extend after function calls to minimize the number of labels. (The scope excludes the called function's execution.) 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. The last condition on the start and end points of a run is also required to make the bound precise. With these restrictions and the 8051's simple timing model we obtain *exact* predictions. If we relax these conditions then we obtain a corollary with an upper bound on the cost. Finally, 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 desired.

4.4 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, we can also transport cost models that associate to each label a *function* from the hardware state to a natural number. However, a problem arises during the instrumentation phase that replaces label emission statements with increments of global cost variables. They are incremented by the result of applying the label's cost function to the hardware state at the time of execution of the block. However, 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).

We can find corresponding information for the former in the source code state, but constructing 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. Fortunately, in 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 only consider dependencies on the 'non-functional' parts of the state.

The non-functional state is not directly related to the high level state and does not influence the functional properties. What can be done is to expose key aspects of 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 [16]. We add to the source code an additional global variable that represents the non-functional state and another one that remembers the last few labels emitted. The state variable must be updated at every label emission statement, using an update function which is computed during the processor timing analysis. This update function assigns to each label a function from the recently emitted labels and old state to the new state. It is computed by composing the semantics of every instruction in a basic block restricted 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 summarised 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, it is not necessary for the user to understand the meaning of the state to reason over the properties of the program. Moreover, 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 for simplicity by approximating the cost by safe bounds that do not depend on the processor state. Interestingly, the functional analysis of the code could determine which blocks are executed more frequently in order to use more aggressive approximations for those that are executed least.

Dependent labelling can also be applied to allow the compiler to duplicate blocks that contain labels (e.g. in loop optimisations) [16]. 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 for the first iteration and the conditional execution of the loop for successive iterations. Further optimisations will compile the two copies of the loop differently, with the first body usually taking more time.

By introducing a variable that keeps track 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 modification: the function will assign one cost to the even iterations and another cost to the odd ones. The optimisation code that duplicates the loop bodies must also modify the code to correctly propagate the update of the iteration numbers. The technical details are more complicated and can be found in the CerCo reports and publications. The implementation, however, is quite simple (and forms part of our OCaml version of the compiler) and the changes to a loop optimising compiler are minimal.

4.5 Techniques to exploit the induced cost model

We now turn our attention to synthesising high-level costs, such as the reaction time of a real-time program. We consider as our starting point source level costs provided by basic labelling, in other words annotations on the source code which are constants that provide a sound and sufficiently 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 synthesised 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, and that there is no limitation on the class of programs that can be handled, for example by resorting to 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 [10] is a set of analysers for C programs with a specification language, ACSL. New analyses can be added dynamically via plugins. For instance, the Jessie plugin [12] 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 essentially uses the CerCo compiler to produce a related C program with cost annotations, and 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. 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 C programs, including some produced by a Lustre compiler.

Soundness. The soundness of the whole framework depends on the cost annotations added by the CerCo compiler, the verification conditions (VCs) generated by Jessie, and the external provers discharging the VCs. Jessie can be used to verify the synthesised bounds because our plugin generates them in ACSL format. 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 depend on the cost plugin, which can in principle produce any synthetic cost. However, in order to be able to actually prove a WCET bound for 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 [7] is a data-flow language for programming synchronous systems, with a compiler which targets C. We designed a wrapper for supporting Lustre files. The C function produced by the compiler is relatively simple loop-free code which implements the step function of the synchronous system and computing the WCET of the function amounts to obtaining 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 a fresh global variable called the cost variable by a constant. Synthesizing a WCET bound of a C function thus consists of 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. finding the instructions that update the cost variable and establish the number of times they are passed through during the flow of execution. To perform the analysis the plugin assumes that there are no recursive functions in the program, and that every loop is annotated with a variant. In the case of 'for' loops the variants are automatically inferred where a loop counter can be syntactically detected.

The plugin computes a call-graph and proceeds to calculate bounds for each function from the leaves up to the main function. The computation of the cost of each function is performed by traversing its control flow graph, where the cost of 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 for the fact that the cost is accumulated at each step of the loop. The cost of the function is added as post-condition of the function.

The user can also specify more precise variants and annotate functions with their own cost specifications. The plugin will use these instead of computing its own, allowing greater precision and the ability to analyse programs which the variant generator does not support.

In addition to the loop-free Lustre code, this method was successfully applied to a small range of cryptographic code. See [5] for more details. The example in Section 3 was also produced using the plug-in. The variant was calculated automatically by noticing that `j` is a loop counter with maximum value `len`. The most expensive path through the loop body ($78 + 136 = 214$) is then multiplied by the number of iterations to give the cost of the loop.

C programs with pointers. Using first-order logic and SMT solvers to specify and verify programs involving pointer-based data structures such as linked-lists or graphs shows some limitations. Separation logic, a program logic with a new notion of conjunction to express spatial heap separation, is an elegant alternative. 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 [6]. CerCo's plugin can exploit these predicates to automatically reason about the cost of execution of simple heap manipulation programs such as an in-place list reversal.

5 Conclusions and future work

All CerCo software and deliverables may be found on the project homepage [9].

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 characterised by history-dependent stateful components, like caches and pipelines. The main issue is to assign a parametric, dependent cost to basic blocks that can be later transferred by the labelling approach to the source code and represented in a meaningful way to the user. The dependent labelling approach that we have studied seems a promising tool to achieve this goal, but more work is required to provide good source level approximations of the relevant processor state.

Other examples of future work are to improve the cost invariant generator algorithms and the coverage of compiler optimisations, to combining the labelling approach with the type and effect discipline of [15] to handle languages with implicit memory management, and to experiment with our tools in the early phases of development. Larger case studies are also necessary to evaluate the CerCo's prototype on realistic, industrial-scale programs.

References

1. AbsInt: aiT WCET analysis tools, http://www.absint.com/ait/
2. Amadio, R., Asperti, A., Ayache, N., Campbell, B., Mulligan, D.P., 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)
3. Amadio, R., Régis-Gianas, Y.: Certifying and reasoning on cost annotations of functional programs. In: Foundational and Practical Aspects of Resource Analysis, LNCS, vol. 7177, pp. 72–89. Springer Berlin Heidelberg (2012), extended version to appear in Higher Order and Symbolic Computation
4. Asperti, A., Ricciotti, W., Coen, C.S., Tassi, E.: The Matita interactive theorem prover. In: CADE. LNCS, vol. 6803, pp. 64–69. Springer (2011)
5. Ayache, N., Amadio, R., Régis-Gianas, Y.: Certifying and reasoning on cost annotations in C programs. In: 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
6. Bobot, F., Filliâtre, J.C.: Separation predicates: A taste of separation logic in first-order logic. In: Formal Methods and Software Engineering. Lecture Notes in Computer Science, vol. 7635, pp. 167–181 (2012), http://dx.doi.org/10.1007/978-3-642-34281-3_14
7. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: Lustre: A declarative language for programming synchronous systems. In: POPL. pp. 178–188. ACM Press (1987)
8. 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)
9. The Certified Complexity (CerCo) project web site, http://cerco.cs.unibo.it
10. 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/
11. 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 (2005)
12. Jessie Frama-C plugin, http://krakatoa.lri.fr/
13. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009)
14. Mulligan, D.P., Sacerdoti Coen, C.: On the correctness of an optimising assembler for the Intel MCS-51 microprocessor. In: CPP. pp. 43–59 (2012)
15. Talpin, J.P., Jouvelot, P.: The type and effect discipline. Inf. Comput. 111(2), 245–296 (1994)
16. Tranquilli, P.: Indexed labels for loop iteration dependent costs. In: QAPL. EPTCS, vol. 117, pp. 19–23 (2013)
17. 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)
18. Wögerer, W.: A survey of static program analysis techniques. Tech. rep., Technische Universität Wien (2005)