


CerCo

INFORMATION AND COMMUNICATION
TECHNOLOGIES
(ICT)
PROGRAMME

Project FP7-ICT-2009-C-243881 CerCo

Report n. D6.3
Final Report on User Validation

Version 1.0

Main Authors:
Roberto M. Amadio, Gabriele Pulcini, Claudio Sacerdoti Coen

Project Acronym: CerCo
Project full title: Certified Complexity
Proposal/Contract no.: FP7-ICT-2009-C-243881 CerCo

Abstract We review the techniques experimented in CerCo for cost annotation exploitment at the user level. We also report on recent work towards precise time analysis at the source level for modern hardware architectures whose instructions cost is a function of the internal hardware state (pipelines, caches, branch prediction units, etc.).

Contents

1 Task

The Grant Agreement describes deliverable D6.3 as follows:

> **Final Report on User Validation**: An articulated analysis and critics of the user validation experiences. In particular we will review the effectiveness of the techniques for cost annotation exploitment that have been employed in the project and that have been validated on simple and non trivial examples. We will also identify additional techniques that could be exploited in the middle and long term to bring the CerCo compiler to its full potentialities.

2 Review of cost synthesis techniques

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 binary code.

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

This principle entails that:

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

Of course, *automation* is desirable whenever possible. Within this framework, automation means writing programs that give hints to the general purpose tool. These hints may take the form, say, of loop invariants/variants, of predicates describing the structure of the heap, or of types in a light logic. If these hints are correct and sufficiently precise the general purpose tool will produce a proof automatically, otherwise, user interaction is required. What follows is a summary of work described in more detail in deliverables D5.1 and D5.3. The cost synthesis techniques we outline are at varying degree of maturity ranging from a complete experimental validation to preliminary thought experiments.

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

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

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

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

Soundness The soundness of the whole framework depends on the cost annotations added by the *CerCo* compiler, the synthetic costs produced by the Cost plug-in, the verification conditions (VCs) generated by Jessie, and the external provers discharging the VCs. The synthetic costs being in ACSL format, Jessie can be used to verify them. Thus, even if the added synthetic costs are incorrect (relatively to the cost annotations), the process in its globality is still correct: indeed, Jessie will not validate incorrect costs and no conclusion can

be made about the WCET of the program in this case. In other terms, the soundness does not really depend on the action of the Cost plug-in, which can in principle produce *any* synthetic cost. However, in order to be able to actually prove a WCET of a C function, we need to add correct annotations in a way that Jessie and subsequent automatic provers have enough information to deduce their validity. In practice this is not straightforward even for very simple programs composed of branching and assignments (no loops and no recursion) because a fine analysis of the VCs associated with branching may lead to a complexity blow up.

Experience with Lustre Lustre is a data-flow language to program synchronous systems and the language comes with a compiler to C. We designed a wrapper for supporting Lustre files. The C function produced by the compiler implements the *step function* of the synchronous system and computing the WCET of the function amounts to obtain a bound on the reaction time of the system. We tested the Cost plug-in and the Lustre wrapper on the C programs generated by the Lustre compiler. For programs consisting of a few hundreds loc, the Cost plug-in computes a WCET and Alt − Ergo is able to discharge all VCs automatically.

2.2 Handling C programs with simple loops

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

- No recursive functions.
- Every loop must be annotated with a *variant*. The variants of 'for' loops are automatically inferred.

The plugin proceeds as follows.

- First the callgraph of the program is computed. If the function f calls the function g then the function g is processed before the function f.
- The computation of the cost of the function is performed by traversing its control flow graph. The cost at a node is the maximum of the costs of the successors.
- In the case of a loop with a body that has a constant cost for every step of the loop, the cost is the product of the cost of the body and of the variant taken at the start of the loop.
- In the case of a loop with a body whose cost depends on the values of some free variables, a fresh logic function f is introduced to represent the cost of the loop in the logic assertions. This logic function takes the variant as a first parameter. The other parameters of f are the free variables of the body of the loop. An axiom is added to account the fact that the cost is accumulated at each step of the loop:

$$f(v, \vec{x}) = \text{if } v < 0 \text{ then } 0 \text{ else } (f(v-1, \phi(\vec{x})) + \psi(\vec{x}))$$

where $\vec{x}$ are the free variables, v is the variant, ϕ computes the modification of the variable at each step of the loop, and ψ is the cost of the body of the loop.

- The cost of the function is directly added as post-condition of the function: $\_\_cost \leq \backslash old(\_\_cost) + t$ where t is the term computing the cost of the function, $\_\_cost$ is the time taken from the start of the program, $\backslash old(\_\_cost)$ is the same time but before the execution of the function.

The user can influence the annotation by different means:

- By using more precise variants.
- By annotating function with cost specification. The plugin will use this cost for the function instead of computing it.

2.3 C programs with pointers

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

In an attempt to conciliate both approaches, Bobot introduced the notion of separation predicates during his PhD thesis. The approach consists in reformulating some ideas from separation logic into a traditional verification framework where the specification language, the verification condition generator, and the theorem provers were not designed with separation logic in mind. Separation predicates are automatically derived from user-defined inductive predicates, on demand. Then they can be used in program annotations, exactly as other predicates, *i.e.*, without any constraint. Simply speaking, where one would write $P * Q$ in separation logic, one will here ask for the generation of a separation predicate *sep* and then use it as $P \wedge Q \wedge sep(P, Q)$. We have implemented separation predicates within the Jessie plug-in and tested it on a non-trivial case study (the composite pattern from the VACID-0 benchmark). In this case, we achieve a fully automatic proof using three existing SMT solver. We have also used the separation predicates to reason on the *cost* of executing simple heap manipulating programs such as an in-place list reversal.

2.4 The cost of higher-order functional programs

We have analysed a rather standard compilation chain from a higher-order functional languages to an abstract RTL language which corresponds directly to the source language of the back-end of the C compiler developed in the *CerCo* project. The compilation consists of four transformations: continuation passing-style, value naming, closure conversion, and hoisting.

We have shown that it is possible to extend the labelling approach described for the C language to a higher-order call-by-value functional language.

The first issue we have considered is that of designing a 'good labelling' function, *i.e.*, a function that inserts labels in the source code which correspond to 'basic blocks' of the

compiled code. To this end, we have introduced two labelling operators: a *pre-labelling* $\ell > M$ which emits the label ℓ before running M and a *post-labelling* $M > \ell$ which reduces M to a value and then emits the label ℓ. Roughly speaking, the 'good labelling' associates a pre-labelling to every function abstraction and a post-labelling to every application which is not immediately surrounded by an abstraction. In particular, the post-labelling takes care of the functions created by the CPS translation.

The second issue relates to the instrumentation of the program. To this end, we have relied on a *cost monad* which associates to each program a pair consisting of its denotation and the cost of reducing the program to a value. In this way, the instrumented program can still be regarded as a higher-order functional program.

The third issue concerns the method to *reason on the instrumented (functional) program.* We have built on a higher-order Hoare logic and a related tool that generates automatically the proof obligations. These proof obligations can either be discharged automatically or interactively using the Coq proof assistant and its tactics. Some simple experiments are described in the LamCost software.

2.5 The cost of memory management

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

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

2.6 Feasible bounds by light typing

In our experience, the cost analysis of higher-order programs requires human intervention both at the level of the specification and of the proofs. One path to automation consists in devising programming disciplines that entail feasible bounds (polynomial time). The most interesting approaches to this problem build on *light* versions of linear logic. Our main contribution is to devise a type system that guarantees feasible bounds for a higher-order call-by-value functional language with references and threads. The first proof of this result relies on a kind of standardisation theorem and it is of a combinatorial nature. More recently, we have shown that a proof of a similar result can be obtained by semantic means building on the so called *quantitative realizability models* proposed by Dal Lago and Hofmann. We believe this semantic setting is particularly appropriate because it allows to reason both on typed and

untyped programs. Thus one can imagine a framework where some programs are feasible 'by typing' while others are feasible as a result of an 'interactive proof' of the obligations generated by quantitative realizability. Beyond building such a framework, an interesting issue concerns the certification of *concrete bounds* at the level of the *compiled code*. This has to be contrasted with the current state of the art in implicit computational complexity where most bounds are *asymptotic* and are stated at the level of the *source code*.

3 Middle and Long Term Improvements

The future improvements that will affect the user experience falls into two categories:

1. **Improvements to invariant generators** The invariant generator that we implemented in the plug-in allows to compute the parametric worst case execution time for all Lustre programs and for almost all the C tests that we targeted. Nevertheless, at the moment the generator does not degrade gracefully: if the source code does not respects the syntactic requirements of the generator, no cost invariants are generated at all. This behaviour is consistent with the traditional use in proving functional properties, but for non functional ones we are interested in always providing a worst case bound, possibly by dropping the dependencies and computing a very rough one. That is the behaviour of standard WCET analyzers (that, most of the time, are not parametric anyway).

   Other future improvements consist in enlarging the class of recognized program shapes by integrating more advanced techniques or interacting with existing tools.

   Both kind of improvements can be performed in the middle term.

2. **Improvements to cost annotation exploitment** One benefit of CerCo w.r.t. traditional WCET is that the user does not need to trust the bound provided by the tool, but it can at least partially verify it manually or using automated techniques.

   The combinations of techniques described in the previous section allowed to automatically certify the parametric worst case execution time for all Lustre programs and for the majority of simple C tests we had at our disposal. Nevertheless, we expect automation to fail more frequently on real world, industrial examples. In the middle term we should experiment with more complex code and enlarge the set of techniques according to the observed results. In particular, we should implement at the source level at least all those that are used on the object code in standard WCET tools. It may well be the case that we identify a set of recurrent proof obligations that are not solved by the existing theorem provers, but that admit a solution by means of a uniform strategy. In any case, the failure to automatically prove sound a cost invariant does not invalidate the invariant itself, assuming that the invariant generator is correct.

3. **Applications to time analysis for modern hardware** At the moment, the main drawback of the CerCo Prototype is that it cannot be ported to modern architectures whose instruction cost depend on the internal state of hardware components like pipelines, caches or branch predictors. The major long term improvement to the CerCo Prototype will be the study of how to accommodate in the labelling approach these kind of cost models. We attach to this document the technical report "*Dependent labelling applied to stateful hardware components*" which describes what seems to be at the moment the most promising approach to the problem. Unsurprisingly, the solution uses

dependent labels, that allow to associate a different cost to multiple executions of the same block. Dependent labels were developed in CerCo to allow loop optimizations, where the dependency was over the number of iterations of the loops. In the case of modern hardware, the dependency is on approximations of the internal hardware state, that needs to be made manifest in the source code too.

The strategy described in the technical report is assumed to work on pipelines, while additional research is expected for caches. Moreover, in the middle term we need to be implement the solution for pipelines to be able to perform experiments. In particular, we need to understand the behaviour on automated provers on the more complex generated cost invariants, and we need to understand to which extent the cost invariants can work with the more precise cost models before introducing severe approximations.

Dependent labelling applied to stateful hardware components

G. Pulcini*, C. Sacerdoti Coen*

{pulcini,sacerdot}@cs.unibo.it

Dipartimento di Informatica - Scienza ed Ingegneria
Università di Bologna

03/04/2013

Abstract

The Certified Complexity (CerCo) EU Project [1] aims at integrating and unifying the functional and non functional analyses of safety and time critical software by performing both of them together on the source code. It is based on the labelling approach [2] that allows to write compilers that induce on the source code a sound and precise cost model for basic blocks. The cost model is computed on object code blocks and then transferred to the source code by reversing the optimizations to the control flow [3].

In this technical report we address the important issue of stateful hardware whose instructions cost is a function of the internal state. Typical examples are pipelines and caches. In order to avoid loss of precision, the cost model must be parametric on the state, which has no correspondent one at the source level. We show how to enrich the source code with the minimal amount of information that allows to compute costs precisely. We also briefly argue in favour of probabilistic static time analysis and we examine how it is already supported by the labelling approach.

1 Introduction

The labelling approach [2, 3] is a new technique to implement compilers that induce on the source code a sound and precise cost model for basic blocks. The cost model is computed on object code blocks and then transferred to the source code by reversing the optimizations to the control flow [3]. In the rest of this technical report we assume reader's knowledge on the labelling approach.

At the end of the CerCo project, we are convinced that the labelling approach can support most optimizations currently supported by real world compilers. Moreover, we have formally certified the code of a compiler that transfers cost models according to the labelling approach, greatly reducing the trusted code base of time analysis. The main issue that is left is to identify the classes of cost models that can be expressed at the source level and that we can automatically reason on. In particular, we know that cost models for modern hardware are complex objects that assign to each instruction a cost which also depends on the internal state of stateful hardware components that are designed to

*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

optimize the average case. Pipelines, caches, branch predictors are typical examples of components that have no impact on the functional behavior of the code, but that greatly augment the code performance.

Ignorance on the internal state of these components during a WCET analysis forces to assume the worst case, leading to useless time bounds. Therefore, standard WCET tools tend to statically execute code fragments as much as possible, to minimize the ignorance on the state. For loops, for example, are totally unrolled. In CerCo we could do the same, departing from the labelling approach. However, what is more natural is to stay closer to the labelling approach and avoid loosing precision by computing parametric exact costs. This is the topic of this research report.

In Section 2 we briefly review the premises and goals of the CerCo approach in order to compare the CerCo approach to the traditional WCET one, which is able to deal with modern hardware. In Section 3 we compare the control flow analyses performed in CerCo and in traditional WCET, arguing that our approach is winning. In Section 4 we compare the static analyses performed in CerCo and in traditional WCET. Only the latter can deal with modern hardware. Section 5 contains the new material. We revise our static analysis and the labelling approach in general to accommodate modern hardware. Temporary conclusions are found in Section 6.

2 CerCo: premises and goals

We briefly review here the premises and goals of the CerCo approach to resource analysis.

- There is a lot of recent and renewed activity in the formal method community to accommodate resource analysis using techniques derived from functional analysis (type systems, logics, abstract interpretation, amortized analysis applied to data structures, etc.)
- Most of this work, which currently remains at theoretical level, is focused on high level languages and it assumes the existence of correct and compositional resource cost models.
- High level languages are compiled to object code by compilers that should respect the functional properties of the program. However, because of optimizations and the inherently non compositional nature of compilation, compilers do not respect compositional cost models that are imposed a priori on the source language. By controlling the compiler and coupling it with a WCET analyser, it is actually possible to choose the cost model in such a way that the cost bounds are high enough to bound the cost of every produced code. This was attempted for instance in the EMBounded project [4] with good success. However, we believe that bounds obtained in this way have few possibilities of being tight.
- Therefore our approach consists in having the compiler generate the cost model for the user by combining tracking of basic blocks during code transformations with a static resource analysis on the object code for basic blocks. We formally prove the compiler to respect the cost model that is induced on the source level based on a very few assumptions: basically the cost of a sequence of instructions should be associative and commutative and it should not depend on the machine status, except its program counter. Commutativity can be relaxed at the price of introducing more cost updates in the instrumented source code.
- The cost model for basic blocks induced on the source language must then be exploited to derive cost invariants and to prove them automatically. In CerCo we have shown how even simple invariant generations techniques are sufficient to enable the fully automatic proving of parametric WCET bounds for simple C programs and for Lustre programs of arbitrary complexity.

Compared to traditional WCET techniques, our approach currently has many similarities, some advantages and some limitations. Both techniques need to perform data flow analysis on the control flow graph of the program and both techniques need to estimate the cost of control blocks of instructions.

3 Control flow analysis

The first main difference is in the control flow analysis. Traditional WCET starts from object code and reconstructs the control flow graph from it. Moreover, abstract interpretation is heavily employed to bound the number of executions of cycles. In order to improve the accuracy of estimation, control flow constraints are provided by the user, usually as systems of (linear) inequalities. In order to do this, the user, helped by the system, needs to relate the object code control flow graph with the source one, because it is on the latter that the bounds can be figured out and be understood. This operations is untrusted and potentially error prone for complex optimizations (like aggressive loop optimizations). Efficient tools from linear algebra are then used to solve the systems of inequations obtained by the abstract interpreter and from the user constraints.

In CerCo, instead, we assume full control on the compiler that is able to relate, even in non trivial ways, the object code control flow graph onto the source code control flow graph. A clear disadvantage is the impossibility of applying the tool on the object code produced by third party compilers. On the other hand, we get rid of the possibility of errors in the reconstruction of the control flow graph and in the translation of high level constraints into low level constraints. The second potentially important advantage is that, once we are dealing with the source language, we can augment the precision of our dataflow analysis by combining together functional and non functional invariants. This is what we attempted with the CerCo Cost Annotating Frama-C Plug-In. The Frama-C architecture allows several plug-ins to perform all kind of static analysis on the source code, reusing results from other plug-ins and augmenting the source code with their results. The techniques are absolutely not limited to linear algebra and abstract interpretation, and the most important plug-ins call domain specific and general purpose automated theorem provers to close proof obligations of arbitrary shape and complexity.

In principle, the extended flexibility of the analysis should allow for a major advantage of our technique in terms of precision, also considering that all analysis used in traditional WCET can still be implemented as plug-ins. In particular, the target we have in mind are systems that are both (hard) real time and safety critical. Being safety critical, we can already expect them to be fully or partially specified at the functional level. Therefore we expect that the additional functional invariants should allow to augment the precision of the cost bounds, up to the point where the parametric cost bound is fully precise. In practice, we have not had the time to perform extensive comparisons on the kind of code used by industry in production systems. The first middle term improvement of CerCo would then consist in this kind of analysis, to support or disprove our expectations. It seems that the newborn TACLe Cost Action (Timing Analysis on Code Level) would be the best framework to achieve this improvement. In the case where our technique remains promising, the next long term improvement would consist in integrating in the Frama-C plug-in ad-hoc analysis and combinations of analysis that would augment the coverage of the efficiency of the cost estimation techniques.

4 Static analysis of costs of basic blocks

At the beginning of the project we have deliberately decided to focus our attention on the control flow preservation, the cost model propagation and the exploitation of the cost model induced on the high level code. For this reason we have devoted almost no attention

to the static analysis of basic blocks. This was achieved by picking a very simple hardware architecture (the 8051 microprocessor family) whose cost model is fully predictable and compositional: the cost of every instruction — except those that deal with I/O — is constant, i.e. it does not depend on the machine status. We do not regret this choice because, with the limited amount of man power available in the project, it would have been difficult to also consider this aspect. However, without showing if the approach can scale to most complex architectures, our methodology remains of limited interest for the industry. Therefore, the next important middle term improvement will be the extension of our methodology to cover pipelines and simple caches. We will now present our ideas on how we intend to address the problem. The obvious long term improvement would be to take in consideration multicores system and complex memory architectures like the ones currently in use in networks on chips. The problem of execution time analysis for these systems is currently considered extremely hard or even unfeasible and at the moments it seems unlikely that our methodology could contribute to the solution of the problem.

5 Static analysis of costs of basic blocks revisited

We will now describe what currently seems to be the most interesting technique for the static analysis of the cost of basic blocks in presence of complex hardware architectures that do not have non compositional cost models.

We start presenting an idealized model of the execution of a generic microprocessor (with caches) that has all interrupts disabled and no I/O instructions. We then classify the models according to some properties of their cost model. Then we show how to extend the labelling approach of CerCo to cover models that are classified in a certain way.

The microprocessor model Let $\sigma, \sigma_1, \ldots$ range over Σ, the set of the fragments of the microprocessor states that hold the program counter, the program status word and all the data manipulated by the object code program, i.e. registers and memory cells. We call these fragments the *data states*.

Let $\delta, \delta_1, \ldots$ range over Δ, the set of the fragments of the microprocessor state that holds the *internal state* of the microprocessor (e.g. the content of the pipeline and caches, the status of the branch prediction unit, etc.). The internal state of the microprocessor influences the execution cost of the next instruction, but it has no effect on the functional behaviour of the processor. The whole state of the processor is represented by a pair (σ, δ).

Let $I, I_1, \ldots$ range over $\mathcal{I}$, the the set of *instructions* of the processor and let $\gamma, \gamma_1, \ldots$ range over Γ, the set of *operands* of instructions after the fetching and decoding passes. Thus a pair (I, γ) represents a *decoded instruction* and already contains the data required for execution. Execution needs to access the data state only to write the result.

Let $fetch : \Sigma \to \mathcal{I} \times \Gamma$ be the function that performs the fetching and execution phases, returning the decoded instruction ready for execution. This is not meant to be the real fetch-decode function, that exploits the internal state too to speed up execution (e.g. by retrieving the instruction arguments from caches) and that, in case of pipelines, works in several stages. However, such a function exists and it is observationally equivalent to the real fetch-decode.

We capture the semantics of the microprocessor with the following set of functions:

- The *functional transition* function $\longrightarrow: \Sigma \to \Sigma$ over data states. This is the only part of the semantics that is relevant to functional analysis.
- The *internal state transition* function $\Longrightarrow: \Sigma \times \Delta \to \Delta$ that updates the internal state.
- The *cost function* $K : \mathcal{I} \times \Gamma \times \Delta \to \mathbb{N}$ that assigns a cost to transitions. Since decoded instructions hold the data they act on, the cost of an instruction may depend both on the data being manipulated and on the internal state.

Given a processor state (σ, δ), the processor evolves in the new state (σ', δ') in n cost units if $\sigma \longrightarrow \sigma'$ and $(\sigma, \delta) \Longrightarrow \delta'$ and $fetch(\sigma) = (I, \gamma)$ and $K(I, \gamma, \delta) = n$.

An *execution history* is a stream of states and transitions $\sigma_0 \longrightarrow \sigma_1 \longrightarrow \sigma_2 \ldots$ that can be either finite or infinite. Given an execution history, the corresponding *execution path* is the stream of program counters obtained from the execution history by forgetting all the remaining information. The execution path of the history $\sigma_0 \longrightarrow \sigma_1 \longrightarrow \sigma_2 \ldots$ is $pc_0, pc_1, \ldots$ where each pc_i is the program counter of σ_i. We denote the set of finite execution paths with EP.

We claim this simple model to be generic enough to cover real world architectures.

Classification of cost models A cost function is *exact* if it assigns to transitions the real cost incurred. It is *approximated* if it returns an upper bound of the real cost.

A cost function is *operand insensitive* if it does not depend on the operands of the instructions to be executed. Formally, K is operand insensitive if there exists a $K' : \mathcal{I} \times \Delta \to \mathbb{N}$ such that $K(I, \gamma, \delta) = K'(I, \delta)$. In this case, with an abuse of terminology, we will identify K with K'.

The cost functions of simple hardware architectures, in particular RISC ones, are naturally operand insensitive. In the other cases an exact operand sensitive cost function can always be turned into an approximated operand insensitive one by taking $K'(I, \delta) = \max\{K(I, \gamma, \delta) \mid \gamma \in \Gamma\}$. The question when one performs these approximation is how severe the approximation is. A measure is given by the *jitter*, which is defined as the difference between the best and worst cases. In our case, the jitter of the approximation K' would be $\max\{K(I, \gamma, \delta) \mid \gamma \in \Gamma\} - \min\{K(I, \gamma, \delta) \mid \gamma \in \Gamma\}$. According to experts of WCET analysis, the jitters relative to operand sensitivity in modern architectures are small enough to make WCET estimations still useful. Therefore, in the sequel we will focus on operand insensitive cost models only.

Note that cost model that are operand insensitive may still have significant dependencies over the data manipulated by the instructions, because of the dependency over internal states. For example, an instruction that reads data from the memory may change the state of the cache and thus greatly affect the execution time of successive instructions.

Nevertheless, operand insensitivity is an important property for the labelling approach. In [3] we introduced *dependent labels* and *dependent costs*, which are the possibility of assigning costs to basic blocks of instructions which are also dependent on the state of the high level program at the beginning of the block. The idea we will now try to pursue is to exploit dependent costs to capture cost models that are sensitive to the internal states of the microprocessor. Operand sensitivity, however, is a major issue in presence of dependent labels: to lift a data sensitive cost model from the object code to the source language, we need a function that maps high level states to the operands of the instructions to be executed, and we need these functions to be simple enough to allow reasoning over them. Complex optimizations performed by the compiler, however, make the mappings extremely cumbersome and history dependent. Moreover, keeping track of status transformations during compilation would be a significant departure from classical compilation models which we are not willing to undertake. By assuming or removing operand sensitivity, we get rid of part of the problem: we only need to make our costs dependent on the internal state of the microprocessor. The latter, however, is not at all visible in the high level code. Our next step is to make it visible.

In general, the value of the internal state at a certain point in the program history is affected by all the preceding history. For instance, the pipeline stages either hold operands of instructions in execution or bubbles [1]. The execution history contains data states that

[1] A bubble is formed in the pipeline stage n when an instruction is stuck in the pipeline stage $n-1$, waiting for some data which is not available yet.

in turn contain the object code data which we do not know how to relate simply to the source code data. We therefore introduce a new classification.

A *view* over internal states is a pair $(\mathcal{V}, |.|)$ where $\mathcal{V}$ is a finite non empty set and $|.| : \Delta \to \mathcal{V}$ is a forgetful function over internal states.

The operand insensitive cost function K is *dependent on the view* $\mathcal{V}$ if there exists a $K' : \mathcal{I} \times \mathcal{V} \to \mathbb{N}$ such that $K(I, \delta) = K'(I, |\delta|)$. In this case, with an abuse of terminology, we will identify K with K'.

Among the possible views, the ones that we will easily be able to work with in the labelling approach are the *execution history dependent views*. A view $(\mathcal{V}, |.|)$ is execution history dependent with a lookahead of length n when there exists a transition function $\hookrightarrow$: $PC^n \times \mathcal{V} \to \mathcal{V}$ such that for all (σ, δ) and $pc_1, \ldots, pc_n$ such that every pc_i is the program counter of σ_i defined by $\sigma \longrightarrow^i \sigma_i$, we have $(\sigma, \delta) \Longrightarrow \delta'$ iff $((pc_1, \ldots, pc_n), |\delta|) \hookrightarrow |\delta'|$.

Less formally, a view is dependent on the execution history if the evolution of the views is fully determined by the evolution of the program counters. To better understand the definition, consider the case where the next instruction to be executed is a conditional jump. Without knowing the values of the registers, it is impossible to determine if the true or false branches will be taken. Therefore it is likely to be impossible to determine the value of the view the follows the current one. On the other hand, knowing the program counter that will be reached executing the conditional branch, we also know which branch will be taken and this may be sufficient to compute the new view. Lookaheads longer then 1 will be used in case of pipelines: when executing one instruction in a system with a pipeline of length n, the internal state of the pipeline already holds information on the next n instructions to be executed.

The reference to execution histories in the names is due to the following fact: every execution history dependent transition function $\hookrightarrow$ can be lifted to the type $EP \times \mathcal{V} \to \mathcal{V}$ by folding the definition over the path trace: $((pc_0, \ldots, pc_m), v_0) \hookrightarrow v_n$ iff for all $i \leq m-n$, $((pc_i, \ldots, pc_{i+n}), v_i) \hookrightarrow v_{i+1}$. Moreover, the folding is clearly associative: $(\tau_1 @ \tau_2, v) \hookrightarrow v''$ iff $(\tau_1, v) \hookrightarrow v'$ and $(\tau_2, v') \hookrightarrow v''$.

As a final definition, we say that a cost function K is *data independent* if it is dependent on a view that is execution path dependent. In two paragraphs we will show how we can extend the labelling approach to deal with data independent cost models.

Before that, we show that the class of data independent cost functions is not too restricted to be interesting. In particular, at least simple pipeline models admit data independent cost functions.

A data independent cost function for simple pipelines We consider here a simple model for a pipeline with n stages without branch prediction and hazards. We also assume that the actual value of the operands of the instruction that is being read have no influence on stalls (i.e. the creation of bubbles) nor on the execution cost. The type of operands, however, can. For example, reading the value 4 from a register may stall a pipeline if the register has not been written yet, while reading 4 from a different register may not stall the pipeline.

The internal states Δ of the pipeline are n-tuples of decoded instructions or bubbles: $\Delta = (\mathcal{I} \times \Gamma \cup \mathbb{1})^n$. This representation is not meant to describe the real data structures used in the pipeline: in the implementation the operands are not present in every stage of the pipeline, but are progressively fetched. A state $(x_1, x_2, \ldots, (I, \gamma))$ represents the state of the pipeline just before the completion of instruction (I, γ). The first $n-1$ instructions that follow I may already be stored in the pipeline, unless bubbles have delayed one or more of them.

We introduce the following view over internal states: $(\{0,1\}^n, |.|)$ where $\mathbb{N}_n = 0, \ldots, 2^n - 1$ and $|(x_1, \ldots, x_n)| = (y_1, \ldots, y_n)$ where y_i is 1 iff x_i is a bubble. Thus the view only remembers which stages of the pipelines are stuck. The view holds enough information to reconstruct the internal state given the current data state: from the data state we can fetch

the program counter of the current and the next $n-1$ instructions and, by simulating at most n execution steps and by knowing where the bubbles were, we can fill up the internal state of the pipeline.

The assumptions on the lack of influence of operands values on stalls and execution times ensures the existence of the data independent cost function $K : PC \times \{0,1\}^n \to \mathbb{N}$. The transition function for a pipeline with n stages may require n lookaheads: $\hookrightarrow$: $PC^n \times \{0,1\}^n \to \{0,1\}^n$.

While the model is a bit simplicist, it can nevertheless be used to describe existing pipelines. It is also simple to be convinced that the same model also captures static branch prediction: speculative execution of conditional jumps is performed by always taking the same branch which does not depend on the execution history. In order to take in account jump predictions based on the execution history, we just need to incorporate in the status and the view statistical informations on the last executions of the branch.

The labelling approach for data independent cost models We now describe how the labelling approach can be slightly modified to deal with a data independent cost model $(\hookrightarrow, K)$ built over $(\mathcal{V}, |.|)$.

In the labelling approach, every basic block in the object code is identified with a unique label L which is also associated to the corresponding basic block in the source level. Let us assume that labels are also inserted after every function call and that this property is preserved during compilation. Adding labels after calls makes the instrumented code heavier to read and it generates more proof obligations on the instrumented code, but it does not create any additional problems. The preservation during compilation creates some significant technical complications in the proof of correctness of the compiler, but those can be solved.

The static analysis performed in the last step of the basic labelling approach analyses the object code in order to assign a cost to every label or, equivalently, to every basic block. The cost is simply the sum of the cost of very instruction in the basic block.

In our scenario, instructions no longer have a cost, because the cost function K takes in input a program counter but also a view v. Therefore we replace the static analysis with the computation, for every basic block and every $v \in \mathcal{V}$, of the sum of the costs of the instructions in the block, starting in the initial view v. Formally, let the sequence of the program counters of the basic block form the execution path $pc_0, \ldots, pc_n$. The cost $K(v_0, L)$ associated to the block labelled with L and the initial view v_0 is $K(pc_0, v_0) + K(pc_1, v_1) + \ldots + K(pc_n, v_n)$ where for every $i < n$, $((pc_i, \ldots, pc_{i+l}), v_i) \hookrightarrow v_{k+1}$ where l is the lookahead required. When the lookahead requires program counters outside the block under analysis, we are free to use dummy ones because the parts of the view that deal with future behaviour have no impact on the cost of the previous operations by assumption.

The static analysis can be performed in linear time in the size of the program because the cardinality of the sets of labels (i.e. the number of basic blocks) is bounded by the size of the program and because the set V is finite. In the case of the pipelines of the previous paragraph, the static analysis is 2^n times more expensive than the one of the basic labelling approach, where n is the number of pipeline stages.

The first important theorem in the labelling approach is the correctness of the static analysis: if the (dependent) cost associated to a label L is k, then executing a program from the beginning of the basic block to the end of the basic block should take exactly k cost units. The proof only relies on associativity and commutativity of the composition of costs. Commutativity is only required if basic blocks can be nested, i.e. if a basic block does not terminate when it reaches a call, but it continues after the called function returns. By assuming to have a cost label after each block, we do not need commutativity any longer, which does not hold for the definition of K we just gave. The reason is that, if pc_i is a function call executed in the view (state) v_i, it is not true that, after return, the state will be $v_i + 1$ defined by $(pc_i, pc_{i+1}, v_i) \hookrightarrow v_{i+1}$ (assuming a lookahead of 1, which is

already problematic). Indeed pc_{i+1} is the program counter of the instruction that follows the call, whereas the next program counter to be reached is the one of the body of the call. Moreover, even if the computation would make sense, v_{i+1} would be the state at the beginning of the execution of the body of the call, while we should know the state after the function returns. The latter cannot be statically predicted. That's why we had to impose labels after calls. Associativity, on the other hand, trivially holds. Therefore the proof of correctness of the static analysis can be reused without any change.

So far, we have computed the dependent cost $K : \mathcal{V} \times \mathcal{L} \to \mathbb{N}$ that associates a cost to basic blocks and views. The second step consists in statically computing the transition function $\hookrightarrow$: $\mathcal{L} \times \mathcal{L} \times \mathcal{V} \to \mathcal{V}$ that associates to each pair of consecutively executed basic blocks and input view the view obtained at the end of the execution of the first block.

The definition is the following: $(L, L', v) \hookrightarrow v'$ iff $((pc_0, \ldots, pc_n, pc'_0, \ldots, pc'_m), v) \hookrightarrow v'$ where $(pc_0, \ldots, pc_n)$ are the program counters of the block labelled by L and $(pc'_0, \ldots, pc'_m)$ are those of the block labelled with L'. We assume here that m is always longer or equal to the lookahead required by the transition function $\hookrightarrow$ over execution paths. When this is not the case we could make the new transition function take in input a longer lookahead of labels. Or we may assume to introduce enough `NOP`s at the beginning of the block L' to enforce the property. In the rest of the paragraph we assume to have followed the second approach to simplify the presentation.

The extended transition function over labels is not present in the basic labelling approach. Actually, the basic labelling approach can be understood as the generalized approach where the view $\mathcal{V} = \mathbb{1}$. The computation of the extended $\hookrightarrow$ transition function is again linear in the size of the program.

Both the versions of K and $\hookrightarrow$ defined over labels can be lifted to work over traces by folding them over the list of labels in the trace: for K we have $K((L_1, \ldots, L_n), v) = K(L_1, v) + K((L_2, \ldots, L_n), v')$ where $(L_1, L_2, v) \hookrightarrow v'$; for $\hookrightarrow$ we have $((L_1, \ldots, L_n), v) \hookrightarrow v''$ iff $(L_1, L_2, v) \hookrightarrow v'$ and $((L_2, \ldots, L_n), v') \hookrightarrow v''$. The two definitions are also clearly associative.

The second main theorem of the labelling approach is trace preservation: the trace produced by the object code is the same as the trace produced by the source code. Without any need to change the proof, we immediately obtain as a corollary that for every view v, the cost $K(\tau, v)$ computed from the source code trace τ is the same than the cost $K(\tau, v)$ computed on the object code trace, which is again τ.

The final step of the labelling approach is source code instrumentation. In the basic labelling approach it consists in adding a global variable `__cost`, initialized with 0, which is incremented at the beginning of every basic block with the cost of the label of the basic block. Here we just need a more complex instrumentation that keeps track of the values of the views during execution:

- We define three global variables `__cost`, initialized at 0, `__label`, initialized with `NULL`, and `__view`, uninitialized.
- At the beginning of every basic block labelled by L we add the following code fragment:

  ```
  __view = __next(__label,L,__view);
  __cost += K(__view, L);
  __label = L;
  ```

  where `__next`$(L_1, L_2, v) = v'$ iff $(L_1, L_2, v) \hookrightarrow v'$ unless L_1 is `NULL`. In that case `(__next(NULL,`L`)` $= v_0$ where $v_0 = |\delta_0|$ and δ_0 is the initial value of the internal state at the beginning of program execution.

  The first line of the code fragment computes the view at the beginning of the execution of the block from the view at the end of the previous block. Then we update the cost function with the cost of the block. Finally we remember the current block to use it for the computation of the next view at the beginning of the next block.

```
int fact (int n) {
  int i, res = 1;
  for (i = 1 ; i <= n ; i++) res *= i;
  return res;
}

int main () {
  return (fact(10));
}
```

Figure 1: A simple program that computes the factorial of 10.

An example of instrumentation in presence of a pipeline In Figure 2 we show how the instrumentation of a program that computes the factorial of 10 would look like in presence of a pipeline. The instrumentation has been manually produced. The `__next` function says that the body of the internal loop of the `fact` function can be executed in two different internal states, summarized by the views 2 and 3. The view 2 holds at the beginning of even iterations, while the view 3 holds at the beginning of odd ones. Therefore even and odd iterations are assigned a different cost. Also the code after the loop can be executed in two different states, depending on the oddness of the last loop iteration.

The definitions of `__next` and `__K` are just examples. For instance, it is possible as well that each one of the 10 iterations is executed in a different internal state.

Considerations on the instrumentation The example of instrumentation in the previous paragraph shows that the approach we are proposing exposes at the source level a certain amount of information about the machine behavior. Syntactically, the additional details, are almost entirely confined into the `__next` and `__K` functions and they do not affect at all the functional behaviour of the program. In particular, all invariants, proof obligations and proofs that deal with the functional behavior only are preserved.

The interesting question, then, is what is the impact of the additional details on non functional (intensional) invariants and proof obligations. At the moment, without a working implementation to perform some large scale tests, it is difficult to understand the level of automation that can be achieved and the techniques that are likely to work better without introducing major approximations. In any case, the preliminary considerations of the project remain valid:

- The task of computing and proving invariants can be simplified, even automatically, trading correctness with precision. For example, the most aggressive approximation simply replaces the cost function `__K` with the function that ignores the view and returns for each label the maximum cost over all possible views. Correspondingly, the function `__next` can be dropped since it returns views that are later ignored.

  A more refined possibility consists in approximating the output only on those labels whose jitter is small or for those that mark basic blocks that are executed only a small number of times. By simplifying the `__next` function accordingly, it is possible to considerably reduce the search space for automated provers.

- The situation is not worse than what happens with time analysis on the object code (the current state of the art). There it is common practice to analyse larger chunks of execution to minimize the effect of later approximations. For example, if it is known that a loop can be executed at most 10 times, computing the cost of 10 iterations yields a better bound than multiplying by 10 the worst case of a single interaction.

  We clearly can do the same on the source level. More generally, every algorithm that works in standard WCET tools on the object code is likely to have a counterpart on the source code. We also expect to be able to do better working on the source code. The reason is that we assume to know more functional properties of the program and

```
int __cost = 8;
int __label = 0;
int __view;

void __cost_incr(int incr) {
  __cost = __cost + incr;
}

int __next(int label1, int label2, int view) {
       if (label1 == 0)                              return 0;
  else if (label1 == 0 && label2 == 1)               return 1;
  else if (label1 == 1 && label2 == 2)               return 2;
  else if (label1 == 2 && label2 == 2 && view == 2) return 3;
  else if (label1 == 2 && label2 == 2 && view == 3) return 2;
  else if (label1 == 2 && label2 == 3 && view == 2) return 1;
  else if (label1 == 2 && label2 == 3 && view == 3) return 0;
  else if (label1 == 3 && label2 == 4 && view == 0) return 0;
  else if (label1 == 3 && label2 == 4 && view == 1) return 0;
}

int __K(int view, int label) {
       if (view == 0 && label == 0) return 3;
  else if (view == 1 && label == 1) return 14;
  else if (view == 2 && label == 2) return 35;
  else if (view == 3 && label == 2) return 26;
  else if (view == 0 && label == 3) return 6;
  else if (view == 1 && label == 3) return 8;
  else if (view == 0 && label == 4) return 6;
}

int fact(int n)
{
  int i;
  int res;
  __view = __next(__label,1,__view); __cost_incr(_K(__view,1)); __label = 1;
  res = 1;
  for (i = 1; i <= n; i = i + 1) {
     __view = __next(__label,2,__view); __cost_incr(__K(__view,2)); __label = 2;
    res = res * i;
  }
  __view = __next(__label,3,__view); __cost_incr(K(__view,3)); __label = 3;
  return res;
}

int main(void)
{
  int t;
  __view = __next(__label,0,__view); __cost_incr(__K(__view,0)); __label = 0;
  t = fact(10);
  __view = __next(__label,4,__view); __cost_incr(__K(__view,4)); __label = 4;
  return t;
}
```

Figure 2: The instrumented version of the program in Figure 1.

more high level invariants, and to have more techniques and tools at our disposal. Even if at the moment we have no evidence to support our claims, we think that this approach is worth pursuing in the long term.

The problem with caches Cost models for pipelines — at least simple ones — are data independent, i.e. they are dependent on a view that is execution path dependent. In other words, the knowledge about the sequence of executed instructions is sufficient to predict the cost of future instructions.

The same property does not hold for caches. The cost of accessing a memory cell strongly depends on the addresses of the memory cells that have been read in the past. In turn, the accessed addresses are a function of the low level data state, that cannot be correlated to the source program state.

The strong correlation between the internal state of caches and the data accessed in the past is one of the two main responsibles for the lack of precision of static analysis in modern uni-core architectures. The other one is the lack of precise knowledge on the real behaviour of modern hardware systems. In order to overcome both problems, that Cazorla&alt. [5] call the "*Timing Analysis Walls*", the PROARTIS European Project has proposed to design "*a hardware/software architecture whose execution timing behaviour*

eradicates dependence on execution history" ([5], Section 1.2). The statement is obviously too strong. What is concretely proposed by PROARTIS is the design of a hardware/software architecture whose execution timing is *execution path dependent* (our terminology).

We have already seen that we are able to accommodate in the labelling approach cost functions that are dependent on views that are execution path dependent. Before fully embracing the PROARTIS vision, we need to check if there are other aspects of the PROARTIS proposal that can be problematic for CerCo.

Static Probabilistic Time Analysis The approach of PROARTIS to achieve execution path dependent cost models consists in turning the hard-to-analyze deterministic hardware components (e.g. the cache) into probabilistic hardware components. Intuitively, algorithms that took decision based on the program history now throw a dice. The typical example which has been thoroughly studied in PROARTIS [6] is that of caches. There the proposal is to replace the commonly used deterministic placement and replacement algorithms (e.g. LRU) with fully probabilistic choices: when the cache needs to evict a page, the page to be evicted is randomly selected according to the uniform distribution.

The expectation is that probabilistic hardware will have worse performances in the average case, but it will exhibit the worst case performance only with negligible probability. Therefore, it becomes no longer interesting to estimate the actual worst case bound. What becomes interesting is to plot the probability that the execution time will exceed a certain threshold. For all practical purposes, a program that misses its deadline with a negligible probability (e.g. 10^-9 per hour of operation) will be perfectly acceptable when deployed on an hardware system (e.g. a car or an airplane) that is already specified in such a way.

In order to plot the probability distribution of execution times, PROARTIS proposes two methodologies: Static Probabilistic Time Analysis (SPTA) and Measurement Based Probabilistic Time Analysis (MBPTA). The first one is similar to traditional static analysis, but it operates on probabilistic hardware. It is the one that we would like to embrace. The second one is based on measurements and it is justified by the following assumption: if the probabilities associated to every hardware operation are all independent and identically distributed, then measuring the time spent on full runs of sub-systems components yields a probabilistic estimate that remains valid when the sub-system is plugged in a larger one. Moreover, the probabilistic distribution of past runs must be equal to the one of future runs.

We understand that MBPTA is useful to analyze closed (sub)-systems whose functional behavior is deterministic. It does not seem to have immediate applications to parametric time analysis, which we are interested in. Therefore we focus on SPTA.

According to [5], "*in SPTA, execution time probability distributions for individual operations ... are determined statically from a model of the processor. The design principles of PROARTIS will ensure ... that the probabilities for the execution time of each instruction are independent. ... SPTA is performed by calculating the convolution ($\oplus$) of the discrete probability distributions which describe the execution time for each instruction on a CPU; this provides a probability distribution ... representing the timing behaviour of the entire sequence of instructions.*"

We will now analyze to what extend we can embrace SPTA in CerCo.

The labelling approach for Static Probabilistic Time Analysis To summarize, the main practical differences between standard static time analysis and SPTA are:

- The cost functions for single instructions or sequences of instructions no longer return a natural numbers (number of cost units) but integral random variables.
- Cost functions are extended from single instructions to sequences of instructions by using the associative convolution operator $\oplus$

By reviewing the papers that described the labelling approach, it is easy to get convinced that the codomain of the cost analysis can be lifted from that of natural numbers to any group. Moreover, by imposing labels after every function call, commutativity can be dropped and the approach works on every monoid (usually called *cost monoids* in the literature). Because random variables and convolutions form a monoid, we immediately have that the labelling approach extends itself to SPTA. The instrumented code produced by an SPTA-CerCo compiler will then have random variables (on a finite domain) as costs and convolutions in place of the {__cost_incr} function.

What is left to be understood is the way to state and compute the probabilistic invariants to do *parametric SPTA*. Indeed, it seems that PROARTIS only got interested into non parametric PTA. For example, it is well known that actually computing the convolutions results in an exponential growth of the memory required to represent the result of the convolutions. Therefore, the analysis should work symbolically until the moment where we are interested into extracting information from the convolution.

Moreover, assuming that the problem of computing invariants is solved, the actual behavior of automated theorem provers on probabilistic invariants is to be understood. It is likely that a good amount of domain specific knowledge about probability theory must be exploited and incorporated into automatic provers to achieve concrete results.

Parametric SPTA using the methodology developed in CerCo is a future research direction that we believe to be worth exploring in the middle and long term.

Static Probabilistic Time Analysis for Caches in CerCo As a final remark, we note that the analysis in CerCo of systems that implement probabilistic caches requires a combination of SPTA and data independent cost models. The need for a probabilistic analysis is obvious but, as we saw in the previous paragraph, it requires no modification of the labelling approach.

In order to understand the need for dependent labelling (to work on data independent cost functions), we need to review the behaviour of probabilistic caches as proposed by PROARTIS. The interested reader can consult [6] for further informations.

In a randomized cache, the probability of evicting a given line on every access is $1/N$ where N stands for the number of cache entries. Therefore the hit probability of a specific access to such a cache is $P(hit) = (\frac{N-1}{N})^K$ where K is the number of cache misses between two consecutive accesses to the same cache entry. For the purposes of our analysis, we must assume that every cache access could cause an eviction. Therefore, we define K (the *reuse distance*) to be the number of memory accesses between two consecutive accesses to the same cache entry, including the access for which we are computing K. In order to compute K for every code memory address, we need to know the execution path (in our terminology). In other words, we need a view that records for each cache entry the number of memory accesses that has occurred since the last access.

For pipelines with n stages, the number of possible views is limited to 2^n: a view can usually just be represented by a word. This is not the case for the views on caches, which are in principle very large. Therefore, the dependent labelling approach for data independent cost functions that we have presented here could still be unpractical for caches. If that turns out to be the case, a possible strategy is the use of abstract interpretations techniques on the object code to reduce the size of views exposed at the source level, at the price of an early loss of precision in the analysis.

More research work must be performed at the current stage to understand if caches can be analyzed, even probabilistically, using the CerCo technology. This is left for future work and it will be postponed after the work on pipelines.

6 Conclusions

At the current state of the art functional properties of programs are better proved high level languages, but the non functional ones are proved on the corresponding object code. The non functional analysis, however, depends on functional invariants, e.g. to bound or correlate the number of executions of cycles.

The aim of the CerCo project is to reconcile the two analysis by performing non functional analysis on the source code. This requires computing a cost model on the object code and reflecting the cost model on the source code. We achieve this in CerCo by designing a certified Cost Annotating Compiler that keeps tracks of transformations of basic blocks, in order to create a correspondence (not necessarily bijection) between the basic blocks of the source and target language. We then prove that the sequence of basic blocks that are met in the source and target executions is correlated. Then, we perform a static analysis of the cost of basic blocks on the target language and we use it to compute the cost model for the source language basic blocks. Finally, we compute cost invariants on the source code from the inferred cost model and from the functional program invariants; we generate proof obligations for the invariants; we use automatic provers to try to close the proof obligations.

The cost of single instructions on modern architectures depend on the internal state of various hardware components (pipelines, caches, branch predicting units, etc.). The internal states are determined by the previous execution history. Therefore the cost of basic blocks is parametric on the execution history, which means both the instructions executed and the data manipulated by the instructions. The CerCo approach is able to correlate the sequence of blocks of source instructions with the sequence of blocks of target instructions. It does not correlate the high level and the low level data. Therefore we are not able in the general case to lift a cost model parametric on the execution history on the source code.

To overcome the problem, we have identified a particular class of cost models that are not dependent on the data manipulated. We argue that the CerCo approach can cover this scenario by exposing in the source program a finite data type of views over internal machine states. The costs of basic blocks is parametric on these views, and the current view is updated at basic block entry according to some abstraction of the machine hardware that does not need to be understood. Further studies are needed to understand how invariant generators and automatic provers can cope with these updates and parametric costs.

We have argued how pipelines, at least simple ones, are captured by the previous scenario and can in principle be manipulated using CerCo tools. The same is not true for caches, whose behaviour deeply depends on the data manipulated. By embracing the PROARTIS proposal of turning caches into probabilistic components, we can break the data dependency. Nevertheless, cache analysis remains more problematic because of the size of the views. Further studies need to be focused on caches to understand if the problem of size of the views can be tamed in practice without ruining the whole approach.

References

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

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

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

[4] **The EmBounded project (project paper)**, K. Hammond, R. Dyckhoff, C. Ferdinand, R. Heckmann, M. Hofmann, H. Loidl, G. Michaelson, J. Serot, A. Wallace, in Trends in Functional Programming, Volume 6, Intellect Press, 2006.

[5] **PROARTIS: Probabilistically Analysable Real-Time Systems**, F.J. Cazorla, E. Qui˜nones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, et al., in ACM Transactions on Embedded Computing Systems, 2012.

[6] **A Cache Design for Probabilistic Real-Time Systems**, L. Kosmidis, J. Abella, E. Quinones, and F. Cazorla, in Design, Automation, and Test in Europe (DATE), Grenoble, France, 03/2013.