Indexed Labels for Loop Iteration Dependent Costs

Paolo Tranquilli*
DISI (Dipartimento di Informatica – Scienza e Ingegneria)
Università di Bologna Alma Mater
tranquil@cs.unibo.it

We present an extension to the labelling approach, a technique for lifting resource consumption information from compiled to source code. This approach, which is at the core of the annotating compiler from a large fragment of C to 8051 assembly of the CerCo project, looses preciseness when differences arise as to the cost of the same portion of code, whether due to code transformation such as loop optimisations or advanced architecture features (*e.g.* cache). We propose to addresses this weakness by formally indexing cost labels with the iterations of the containing loops they occur in. These indexes can be transformed during the compilation, and when lifted back to source code they produce dependent costs.

The proposed changes have been implemented in CerCo's untrusted prototype compiler from a large fragment of C to 8051 assembly [2].

1 Introduction

Recent years have seen impressive advancements in the field of formal description and certification of software components. In the fields of compilers a well-documented example is CompCert, a project which has spawned the proof of correctness of a compiler from a large fragment of C to assembly [9]. The success of this endeavour is also supported by a comparison with other compilers as to the number of bugs found with testing tools [14].

The CerCo project [2] strives to add a significant aspect to the picture: certified resource consumption. More precisely our aim is to build a certified C compiler targeting embedded systems that produces, apart from object code functionally equivalent to the input, an *annotation* of the source code which is a sound and precise description of the execution cost of the compiled code. Time and stack are the immediate resources on which the method can be applied.

The current state of the art in commercial products that analyse reaction time or memory usage of programs installed in embedded systems (*e.g.* Scade [8] or AbsInt [1]) is that the estimate is based upon an abstract interpretation of the object code that may require explicit and untrusted annotations of the binaries stating how many times loops are iterated (see *e.g.* [13]). Our aim, on the other hand, is to lift cost information of small fragments of object code, so that these bits of information may be compositionally combined at the source level, abstracting away the specifics of the architecture and only having to reason about standard C semantics the programmer will be familiar with. This information can be used to decide complexity assertions either with pencil and paper or with a tool for automated and formal reasoning about C programs such as Frama-C [3].

The theoretical basis of the CerCo compiler has been outlined by Amadio *et al* [6]. Summarising, the proposal consists in 'decorating' the source code by inserting labels at key points. These labels are preserved as compilation progresses, from one intermediate language to another. Once the final object

*This work is funded by the CerCo FET-Open EU Project.

Submitted to:
QAPL 2013

code is produced, such labels should correspond to the parts of the compiled code that have a constant cost. This cost can then be assigned to blocks of source code.

Two properties must hold of any cost estimate given to blocks of code. The first property, paramount to the correctness of the method, is *soundness*—the actual execution cost must be bounded by the estimate. In the labelling approach, this is guaranteed if every loop in the control flow of the compiled code passes through at least one cost label. Were it not the case, the cost of the loop would be taken in charge by a label external to it, so that any constant cost assignment would be invalidated by enough iterations of the loop. The second property, optional but desirable, is *preciseness*—the estimate *is* the actual cost. This is of particular importance for embedded real-time systems, where in particular situations we may care that a code runs for *at least* some clock cycles. In the labelling approach, this is true if, for every label, every possible execution of the compiled code starting from such a label yields the same cost before hitting another one. In simple architectures such as the 8051 micro-controller which is targeted by the current stage of the CerCo project, this can be guaranteed by placing labels at the start of any branch in the control flow, and by ensuring that no labels are duplicated.

The reader should note that the above mentioned requirements state properties that must hold for the code obtained *at the end* of the compilation chain. Even if one is careful about injecting the labels at suitable places in the source code, the requirements might still fail because of two main obstacles.

- The compilation process might introduce important changes in the control flow, inserting loops or branches. This might happen for example when replacing operations that are unavailable in the target architecture, such as generic shift and multi-byte division in the 8051 architecture[1].
- Even when the compiled code *does*—as far as the the syntactic control flow graph is concerned—respect the conditions for soundness and preciseness, the cost of blocks of instructions might not be independent of context and thus not compositional, so that different passes through a label might have different costs. This becomes a concern if one wishes to apply the approach to more complex architectures, for example one with caching or pipelining.

Even if we solved the problem outlined in the first point for our current compilation chain, the point remains a weakness of the current labelling approach when it comes to some common code transformations. In particular, most *loop optimisations* change the control flow graph duplicating code and adding or changing the branches. An example optimisation of this kind is *loop peeling*, where a first iteration of the loop is hoisted out of and before its body. This optimisation is employed by compilers in order to trigger other optimisations, such as dead code elimination or invariant code motion. Here, the hoisted iteration might possibly be assigned a different cost than later iterations.

The second point above highlights another weakness. Different tools allow to predict up to a certain extent the behaviour of cache. For example, the aiT tool [1] allows the user to estimate the worst-case execution time taking into account advanced features of the target architecture. While such a tool is not fit for a compositional approach which is central to CerCo's project[2], aiT's ability to produce tight estimates of execution costs would still enhance the effectiveness of the CerCo compiler, *e.g.* by integrating such techniques in its development. A typical case where cache analysis yields a difference in the execution cost of a block is in loops: the first iteration will usually stumble upon more cache misses than subsequent iterations.

If one looks closely, the source of the two weaknesses of the regular labelling approach of [6] outlined above is common: the inability to state different costs for different occurrences of labels in the execution

[1] The reader might see the work outlined in [5] to get a grasp of how we tackle this problem in CerCo's compiler.

[2] aiT assumes the cache is empty at the start of computation, and treats each procedure call separately, unrolling a great part of the control flow.

$$x, y, \ldots \quad \text{(identifiers)} \qquad e, f, \ldots \quad \text{(expressions)}$$
$$P, S, T, \ldots ::= \mathsf{skip} \mid s;t \mid \mathsf{if}\ e\ \mathsf{then}\ S\ \mathsf{else}\ T \mid \mathsf{while}\ e\ \mathsf{do}\ s \mid x := e \quad \text{(statements)}$$

Figure 1: The syntax of Imp.

trace. The difference in cost might be originated by labels being duplicated along the compilation, or by the costs being sensitive to the current state of execution.

The work we present here addresses this weakness by introducing cost labels that are dependent on which iteration of its containing loops it occurs in. This is achieved by means of *indexed labels*; all cost labels are decorated with formal indexes coming from the loops containing such labels. These indexes allow us to rebuild, even after multiple loop transformations, which iterations of the original loops in the source code a particular label occurrence belongs to. During the annotating stage, this information is presented to the user by means of *dependent costs*.

Here we concentrate on integrating the labelling approach with two loop transformations—*loop peeling* and *loop unrolling*. They will be presented for a toy language in section 2, For general information on compiler optimisations (and loop optimisations in particular) we refer the reader to the vast literature on the subject (*e.g.* [12, 11]).

The proposed changes have been implemented in CerCo's untrusted prototype compiler available on CerCo's homepage[3]. For lack of space the present work will not delve into the details of the implementation.

Whilst we cover only two loop optimisations in this paper, we argue that the work presented herein poses a good foundation for extending the labelling approach, in order to cover more and more common optimisations, as well as gaining insight into how to integrate advanced cost estimation techniques, such as cache analysis, into the CerCo compiler. Moreover loop peeling itself has the fortuitous property of enhancing and enabling other optimisations. Experimentation with CerCo's untrusted prototype compiler, which implements constant propagation and partial redundancy elimination [10, 12], show how loop peeling enhances those other optimisations.

Outline. We will present our approach on a minimal 'toy' imperative language, Imp with gotos, which we present in section 2 along with formal definitions of the loop transformations. This language already presents most of the difficulties encountered when dealing with C, so we stick to it for the sake of this presentation. In section 3 we summarize the labelling approach as presented in [6]. Section 4 presents *indexed labels*, our proposal for dependent labels which are able to describe precise costs even in the presence of the various loop transformations we consider, together with a more detailed example (subsection 4.5). Finally section 5 speculates on further work on the subject.

2 The minimal imperative language Imp

We briefly outline the toy language, the minimalist imperative language Imp. Its syntax is presented in 1. We may omit the else clause of a conditional if it leads to a skip statement. The precise grammar for expressions is not particularly relevant so we do not give one in full. We will use the notation $(S, K, s) \to (S', K', s')$ for Imp's small-step semantics of which we skip the unsurprising definition. S is

[3] http://cerco.cs.unibo.it/

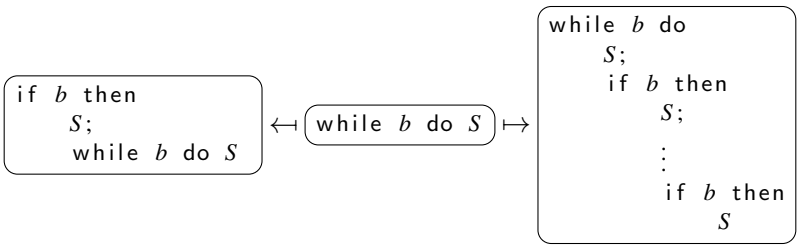

Figure 2: Loop peeling (left) and loop unrolling (right).

the statement being executed, K is a continuation (*i.e.* a stack of statements to be executed after S) and s is the store (*i.e.* a map from variables to integers).

Further down the compilation chain. We abstract over the rest of the compilation chain. We posit the existence, for every language L further down the compilation chain, of a suitable notion of 'sequential instructions', wherein each instruction has a single natural successor. To these sequential instructions to which we can add our own.

Loop transformations. We present the loop transformations we deal with in Figure 2. These transformations are local, *i.e.* they target a single loop and transform it. Which loops are targeted may be decided by some *ad hoc* heuristic. However, the precise details of which loops are targeted and how is not important here.

As already mentioned in the introduction, loop peeling consists in preceding the loop with a copy of its body, appropriately guarded. This is usually done to trigger further optimisations. Integrating this transformation into the labelling approach would also allow, in the future, the integration of a common case of cache analysis, as predicting cache hits and misses benefits from a form of *virtual* loop peeling [7].

Loop unrolling consists of the repetition of several copies of the body of the loop inside the loop itself (inserting appropriate guards, or avoiding them altogether if enough information about the loop's guard is available at compile time). This can limit the number of (conditional or unconditional) jumps executed by the code and trigger further optimisations dealing with pipelining, if appropriate for the architecture. Notice that we present unrolling in a wilfully *naïve* version. On the one hand usually less general loops and more well-behaving loops are targeted; on the other hand, conditionals are seldom used to cut up the body of the unrolled loop. However we are mainly interested in the changes to the control flow the transformation does. The problem this transformation poses to CerCo's labelling approach are independent of the sophistication of the actual transformation.

We decided to apply transformations in the front-end in order to only target loops explicitly written by the programmer. This is because we need to output source code annotations that are meaningful to the user, and in order to do so we only transform loops that where explicitly written as so.

Example 1. In Figure 3 we show a program (a wilfully inefficient computation of of the sum of the first n factorials) and a possible combination of transformations applied to it (again for the sake of presentation rather than efficiency).

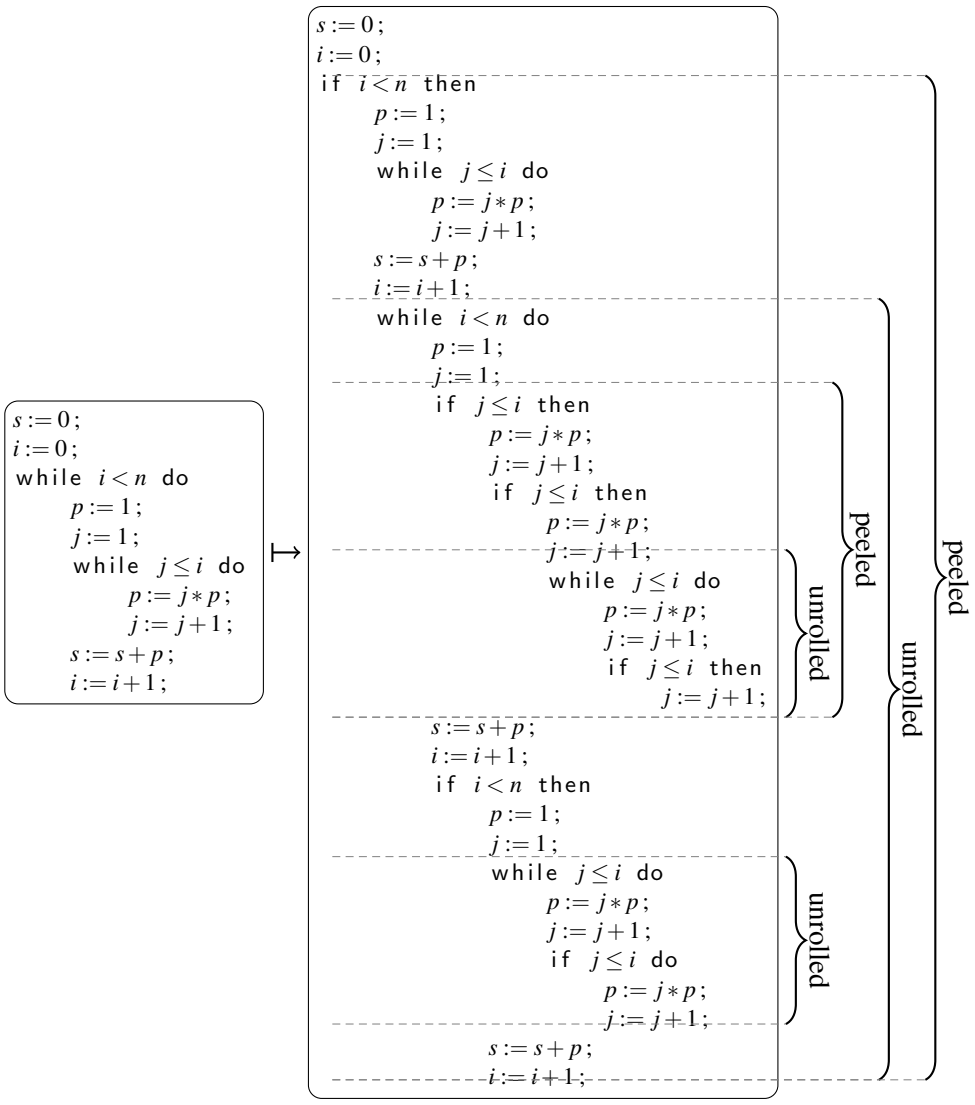

Figure 3: An example of loop transformations. Blocks are delimited by indentation.

3 Labelling: a quick sketch of the previous approach

Plainly labelled Imp is obtained by adding to the code *cost labels* (with metavariables $\alpha, \beta, \ldots$), and cost-labelled statements:

$$S, T ::= \cdots \mid \alpha : S$$

Cost labels allow us to track some program points along the compilation chain. For further details we refer to [6].

The small step semantics turns into a labelled transition system and a natural notion of trace (*i.e.* lists of labels) arises. The small-step rules of Imp remain as unlabelled steps, while adding the rule

$$(\alpha : S, K, s) \xrightarrow{\alpha} (S, K, s)$$

Cost labels are thus emitted by cost-labelled statements only[4]. We then write $\xrightarrow{\lambda}{}^*$ for the transitive closure of the small step semantics which produces by concatenation the trace λ.

Labelling. Given an Imp program P its *labelling* in ℓImp is defined by $\alpha : \mathscr{L}(P)$, putting cost labels after every branching statement, at the start of both branches, and a cost label at the beginning of the program. The relevant recursive cases for the definition of $\mathscr{L}(P)$ are

$$\mathscr{L}(\mathsf{if}\ e\ \mathsf{then}\ S\ \mathsf{else}\ T) = \mathsf{if}\ e\ \mathsf{then}\ \alpha : \mathscr{L}(S)\ \mathsf{else}\ \beta : \mathscr{L}(T)$$
$$\mathscr{L}(\mathsf{while}\ e\ \mathsf{do}\ S) = (\mathsf{while}\ e\ \mathsf{do}\ \alpha : \mathscr{L}(S)); \beta : \mathsf{skip}$$

where α, β are fresh cost labels. In all other cases the definition just passes to substatements.

Labels in the rest of the compilation chain. All languages further down the chain get a new sequential statement emit α whose effect is to be consumed in a labelled transition while keeping the same state. All other instructions guard their operational semantics and do not emit cost labels.

Preservation of semantics throughout the compilation process is restated, in rough terms, as:

$$\text{starting state of } P \xrightarrow{\lambda}{}^* \text{ halting state} \iff \text{starting state of } \mathscr{C}(P) \xrightarrow{\lambda}{}^* \text{ halting state} \tag{1}$$

Here P is a program of a language along the compilation chain, starting and halting states depend on the language, and $\mathscr{C}$ is any of the compilation passes[5]. This must in particular be true for any optimisation pass the compilation undergoes.

Instrumentations. Let $\mathscr{C}$ be the whole compilation from ℓImp to the labelled version of some low-level language L. Supposing such compilation has not introduced any new loop or branching, we have that:

- Every loop contains at least a cost label (*soundness condition*)
- Every branching has different labels for the two branches (*preciseness condition*).

With these two conditions, we have that each and every cost label in $\mathscr{C}(P)$ for any P corresponds to a block of sequential instructions, to which we can assign a constant *cost*[6] We therefore may assume the existence of a *cost mapping* κ_P from cost labels to natural numbers, assigning to each cost label α the cost of the block containing the single occurrence of α.

Given any cost mapping κ, we can enrich a labelled program so that a particular fresh variable (the *cost variable* c) keeps track of the summation of costs during the execution. We call this procedure *instrumentation* of the program, and it is defined recursively by:

$$\mathscr{I}(\alpha : S) = c := c + \kappa(\alpha); \mathscr{I}(S)$$

In all other cases the definition passes to substatements. One can then reason on the instrumented version of the code like he would on any program, asserting statements about complexity by inspecting c.

[4] In the general case, because of the ternary operator, any evaluation of expressions can emit cost labels too (see B).

[5] The case of divergent computations needs to be addressed too. Also, the requirement can be weakened by demanding a weaker form of equivalence of the traces than equality. Both of these issues are beyond the scope of this presentation.

[6] This in fact requires the machine architecture to be 'simple enough', or for some form of execution analysis to take place.

The problem with loop optimisations. Let us take loop peeling, and apply it to the labelling of a program without any prior adjustment:

$$(\mathsf{while}\ e\ \mathsf{do}\ \alpha : S); \beta : \mathsf{skip} \mapsto (\mathsf{if}\ b\ \mathsf{then}\ \alpha : S; \mathsf{while}\ b\ \mathsf{do}\ \alpha : S); \beta : \mathsf{skip}$$

What happens is that the cost label α is duplicated with two distinct occurrences. If these two occurrences correspond to different costs in the compiled code, the best the cost mapping can do is to take the maximum of the two, preserving soundness (*i.e.* the cost estimate still bounds the actual one) but losing preciseness (*i.e.* the actual cost could be strictly less than its estimate).

4 Indexed labels

This section presents the core of the new approach. In brief points it amounts to the following:

4.1. Enrich cost labels with formal indexes stating, for each loop containing the label in the source code, what iteration it occurs in.

4.2. Each time a loop transformation is applied and a cost labels is split in different occurrences, each of these will be reindexed so that every time they are emitted their position in the original loop will be reconstructed.

4.3. Along the compilation chain, alongside the emit instruction we add other instructions updating the indexes, so that iterations of the original loops can be rebuilt at the operational semantics level even when the original structure of loops is lost.

4.4. The machinery computing the cost mapping will still work, but assigning costs to indexed cost labels, rather than to cost labels as we wish. However, *dependent costs* can be calculated, where dependency is on which iteration of the containing loops we are in.

4.1 Indexing the cost labels

Formal indexes and $\iota\ell$Imp. Let $i_0, i_1, \ldots$ be a sequence of distinguished fresh identifiers that will be used as loop indexes. A *simple expression* is an affine arithmetical expression in one of these indexes, that is $a * i_k + b$ with $a, b, k \in \mathbb{N}$. Simple expressions $e_1 = a_1 * i_k + b_1$ and $e_2 = a_2 * i_k + b_2$ in the same index can be composed, yielding $e_1 \circ e_2 := (a_1 a_2) * i_k + (a_1 b_2 + b_1)$, and this operation has an identity element $1 * i_k + 0$, which we may denote simply by i_k by abuse of notation. Constants can be expressed as simple expressions, so that we identify a natural c with $0 * i_k + c$.

An *indexing* (with metavariables I, J, ...) is a list of transformations of successive formal indexes dictated by simple expressions, that is a mapping[7]

$$i_0 \mapsto a_0 * i_0 + b_0, \ldots, i_{k-1} \mapsto a_{k-1} * i_{k-1} + b_{k-1}$$

An *indexed cost label* (metavariables A, B, ...) is the combination of a cost label α and an indexing I, written $\alpha\langle I \rangle$. The cost label underlying an indexed one is called its *atom*.

Imp with indexed labels (from now on $\iota\ell$Imp) is defined by having loops with a formal index attached to them and by allowing statements to by labelled by indexed labels:

$$S, T, \ldots ::= \cdots \mid i_k : \mathsf{while}\ e\ \mathsf{do}\ S \mid A : S$$

[7] Here we restrict each mapping to be one from an index to a simple expression *on the same index*. This might not be the case if more loop optimisations are accounted for (for example, interchanging two nested loops could give rise to an indexing like $i_0 \mapsto i_1, i_1 \mapsto i_0$).

Notice that unindexed loops may still exist in the language: though it does not concern this simple toy exampl, they would correspond to multi-entry loops which are ignored by indexing and optimisations in a scenario with gotos.

We will discuss $\iota\ell$Imp's semantics later, in subsection 4.3.

Indexed labelling. In order to compute the *indexed labelling* $\mathscr{L}^\iota$ of a program, we need to keep track of the nesting of indexed loops as we visit the program abstract syntax tree.

Let Id_k be the indexing of length k made from identity simple expressions, *i.e.* the sequence $i_0 \mapsto i_0, \ldots, i_{k-1} \mapsto i_{k-1}$. We define the tiered indexed labelling $\mathscr{L}^\iota(S,k)$ by recursion setting:

$$\mathscr{L}^\iota(\mathsf{while}\ b\ \mathsf{do}\ T, k) := i_k : \mathsf{while}\ b\ \mathsf{do}\ \alpha\langle Id_{k+1}\rangle : \mathscr{L}^\iota_P(T, k+1)); \beta\langle Id_k\rangle : \mathsf{skip}$$
$$\mathscr{L}^\iota(\mathsf{if}\ b\ \mathsf{then}\ T_1\ \mathsf{else}\ T_2, k) := \mathsf{if}\ b\ \mathsf{then}\ \alpha\langle Id_k\rangle : \mathscr{L}^\iota_P(T_1, k)\ \mathsf{else}\ \beta\langle Id_k\rangle : \mathscr{L}^\iota_P(T_2, k)$$

Here, as usual, α and β are fresh cost labels, and other cases just keep making the recursive calls on the substatements. The *indexed labelling* of a program P is then defined as $\alpha\langle\rangle : \mathscr{L}^\iota(P, 0)$, *i.e.* a further fresh unindexed cost label is added at the start, and we start from level 0.

In plainer words: each loop is indexed by i_k where k is the number of other loops containing this one, and all cost labels under the scope of a loop indexed by i_k are indexed by all indexes $i_0, \ldots, i_k$, without any transformation.

4.2 Indexed labels and loop transformations

We define the *reindexing* $\alpha\langle I\rangle \circ (i_k \mapsto f)$ as an operator on indexed labels by setting[8]:

$$\alpha\langle i_0 \mapsto e_0, \ldots, i_k \mapsto e_k, \ldots, i_n \mapsto e_n\rangle \circ (i_k \mapsto f) := \alpha\langle i_0 \mapsto e_0, \ldots, i_k \mapsto e_k \circ f, \ldots, i_n \mapsto e_n\rangle.$$

We extend this definition to statements in $\iota\ell$Imp by applying the above transformation to all indexed labels contained in a statement.

We can now finally redefine loop peeling and loop unrolling, taking into account indexed labels. The attentive reader will notice that no assumptions will be made as to the labelling of the statements that are involved. This ensures that the transformation can be repeated and composed at will. Also, notice that after erasing all labelling information (*i.e.* indexed cost labels and loop indexes) we recover exactly the same transformations presented in 2. The transformations are presented in Figure 4.

As can be expected, in loop peeling the peeled iteration of the loop gets reindexed with 0, as it always correspond to the first iteration of the loop. The iterations of the remaining loop are shifted by 1. Notice that this transformation can lower the actual depth of some loops, however their index is left untouched. In loop unrolling each copy of the unrolled body has its indexes remapped so that when they are executed, the original iteration of the loop to which they correspond can be recovered.

Fact 2. Loop peeling and unrolling preserve the following invariant, which we call *non-overlap of indexed labels*: for all labels $\alpha\langle I\rangle$ and $\alpha\langle J\rangle$ such that $I \neq J$, the first different simple expressions of the two are disjoint, *i.e.* they always evaluate to different constants. Moreover for every loop $i_k : \mathsf{while}\ e\ \mathsf{do}\ S$ and label $\alpha\langle I\rangle$ in S, no label outside the loop with the same atom can share the same prefix up to i_k.

[8] If mappings are not restricted to only depend on the index being mapped, reindexing should be substituted in each occurrence of i_k.

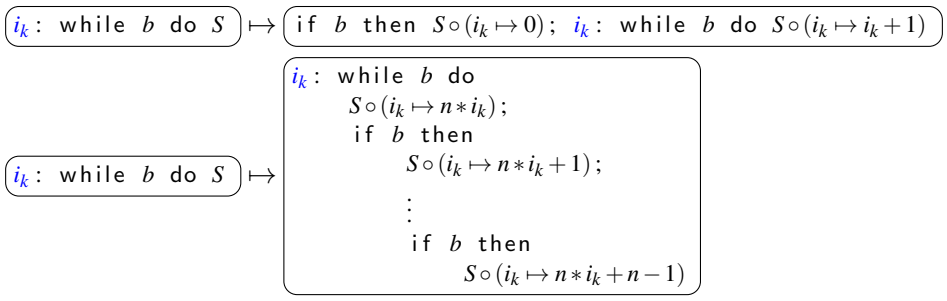

Figure 4: Loop peeling and loop unrolling in the presence of indexed labels. In loop unrolling n is the times the loop is unrolled.

4.3 Semantics and compilation of indexed labels

In order to make sense of loop indexes, one must keep track of their values in the state. A *constant indexing* (metavariables $C, \ldots$) is an indexing which employs only constant simple expressions. The evaluation of an indexed label A in a constant indexing C, noted $A|_C$, is defined by:

$$A|_{i_0 \mapsto c_0, \ldots, i_{k-1} \mapsto c_{k-1}} := A \circ (i_0 \mapsto c_0) \circ \cdots \circ (i_{k-1} \mapsto c_{k-1})$$

Here, we are using the definition of $- \circ -$ given in subsection 4.1 at page 7. We consider the above defined only if the the resulting indexing of the label is constant too[9].

Constant indexings will be used to keep track of the exact iterations of the original code that the emitted labels belong to. We thus define two basic actions to update constant indexings: $C[i_k\uparrow]$ increments the value of i_k by one, and $C[i_k\downarrow 0]$ resets it to 0.

We are ready to explain how the operational semantics of indexed labelled Imp updates the one of plain ℓImp. The emitted cost labels will now be ones indexed by constant indexings. We add to continuations a special indexed loop constructor i_k : while b do S then K.

The difference between the regular stack concatenation i_k : while b do $S \cdot K$ and the new constructor is that the latter indicates the loop is the active one in which we already are, while the former is a loop that still needs to be started[10].

The state will now be a 4-tuple (S, K, s, C) which adds a constant indexing to the triple of the regular semantics. The small-step rules for all but cost-labelled and indexed loop statements remain the same, without touching the C parameter. The new cases are:

$$(A : S, K, s, C) \xrightarrow{A|_C} (S, K, s, C)$$

$$(i_k : \text{while } b \text{ do } S, K, C) \to \begin{cases} (S, i_k : \text{while } b \text{ do } S \text{ then } K, s, C[i_k\downarrow 0]) & \text{if } (b,s) \Downarrow v \neq 0, \\ (\text{skip}, K, s, C) & \text{otherwise,} \end{cases}$$

$$(\text{skip}, i_k : \text{while } b \text{ do } S \text{ then } K, C) \to \begin{cases} (S, i_k : \text{while } b \text{ do } S \text{ then } K, s, C[i_k\uparrow]) & \text{if } (b,s) \Downarrow v \neq 0, \\ (\text{skip}, K, s, C) & \text{otherwise.} \end{cases}$$

Some explanations are in order. We can see that emitting a label always instantiates it with the current indexing, and that hitting an indexed loop the first time initializes the corresponding index to 0. Continuing the same loop increments the index as expected.

[9] For example $(i_0 \mapsto 2 * i_0, i_1 \mapsto i_1 + 1)|_{i_0 \mapsto 2}$ is undefined, but $(i_0 \mapsto 2 * i_0, i_1 \mapsto 0)|_{i_0 \mapsto 2} = i_0 \mapsto 4, i_1 \mapsto 2$, is indeed a constant indexing, even if the domain of the original indexing is not covered by the constant one.

[10] In the presence of continue and break statements active loops need to be kept track of in any case.

The starting state with store s for a program P is $(P, \varepsilon, s, (i_0 \mapsto 0, \ldots, i_{n-1} \mapsto 0))$ where ε is the empty stack and $i_0, \ldots, i_{n-1}$ cover all loop indexes of P[11].

Compilation. Further down the compilation chain the loop structure is usually partially or completely lost. We cannot rely on it any more to keep track of the original source code iterations. We therefore add, alongside the emit instruction, two other sequential instructions ind_reset k and ind_inc k whose only effect is to reset to 0 (resp. increment by 1) the loop index i_k. These instructions will keep track of points in the code corresponding to loop entrances and continuations respectively.

The first step of compilation from $\iota\ell$Imp consists of prefixing the translation of an indexed loop i_k : while b do S with ind_reset k and postfixing the translation of its body S with ind_inc k. Later in the compilation chain we must propagate the instructions dealing with cost labels.

We would like to stress the fact that this machinery is only needed to give a suitable semantics of observables on which preservation proofs can be done. By no means are the added instructions and the constant indexing in the state meant to change the actual (let us say denotational) semantics of the programs. In this regard the two new instruction have a similar role as the emit one. A forgetful mapping of everything (syntax, states, operational semantics rules) can be defined erasing all occurrences of cost labels and loop indexes, and the result will always be a regular version of the language considered.

Stating the preservation of semantics. In fact, the statement of preservation of semantics does not change at all, if not for considering traces of evaluated indexed cost labels rather than traces of plain ones. So every pass will still need to enjoy property (1).

4.4 Dependent costs in the source code

The task of producing dependent costs from constant costs induced by indexed labels is quite technical. Before presenting it here, we would like to point out that the annotations produced by the procedure described in this subsection, even if correct, can be enormous and unreadable. The prototype compiler employs simplifications that will not be documented here to mitigate this problem.

Upon compiling the indexed labelling $\mathscr{L}^{\iota}(P)$ of an Imp program P, we may still apply the machinery described in [6] and sketched in section 3 and get a statically computed cost mapping from *indexed* labels to naturals.

As we need to annotate the source code, we want a way to express and compute the costs of cost labels. In order to do so, we have to group the costs of single indexed labels with the same atom. In order to do so we introduce *dependent costs*.

Let us suppose that for the sole purpose of annotation, we have available in the language ternary expressions of the form

$$e \,?\, f_1 : f_2,$$

and that we have access to common operators on integers such as equality, order and modulus.

[11] For a program which is the indexed labelling of an Imp one this corresponds to the maximum nesting of single-entry loops. We can also avoid computing this value in advance if we define $C[i{\downarrow}0]$ to extend C's domain as needed, so that the starting constant indexing can be the empty one.

Simple conditions. First, we need to shift from *transformations* of loop indexes to *conditions* on them. We identify a set of conditions on natural numbers which are able to express the image of any composition of simple expressions. *Simple conditions* are of three possible forms:

$$p ::= i_k = n | i_k \geq n | i_k \bmod a = b \wedge i_k \geq n$$

Given a simple condition p and a constant indexing C we can easily define when p holds for C (written $p|_C$): it suffices to substitute the formal indexes with their value in C. A *dependent cost expression* is an expression built solely out of integer constants and ternary expressions with simple conditions at their head, *i.e.* $K ::= n \mid p\ ?\ K_1 : K_2$. Given a dependent cost expression K where all of the loop indexes appearing in it are in the domain of a constant indexing C, we can easily define the value $K|_C \in \mathbb{N}$ by evaluating the heads of all ternary expressions in C.

Every simple expression e corresponds to a simple condition $p(e)$ which expresses the set of values that e can take. Following is the definition of such a relation[12]:

$$\begin{aligned} p(0 * i_k + b) &:= (i_k = b) & p(1 * i_k + b) &:= (i_k \geq b) \\ p(a * i_k + b) &:= (i_k \bmod a = b' \wedge i_k \geq b) & &\text{otherwise, where } b' = b \bmod a. \end{aligned}$$

The fact that this mapping has sense is stated by the following fact.

Fact 3. For every expression e on i_k, $p(e)|_{(i_k \mapsto c)}$ iff there is a constant d such that $e|_{(i_k \mapsto d)} = c$.

From indexed costs to dependent ones. Suppose we are given a mapping κ from indexed labels to natural numbers. We must transform it to a mapping (identified, by abuse of notation, with the same symbol κ) from atoms to dependent expressions. The reader uninterested in the technical details explained below can get a grasp of how this is done by going through the example in subsection 4.5.

We will allow indexings to start from other index variables than i_0. Let $\mathbb{S}$ be the set of sets of indexings with fixed domain. Formally:

$$\mathbb{S} := \{\, S \mid S \subseteq \{\, i_h \mapsto e_h, \ldots, i_k \mapsto e_k \,\} \text{for some } h \leq k \text{ and } e_i\text{'s} \,\},$$

For every set $S \in \mathbb{S}$, we are in one of the following three mutually exclusive cases:

- $S = \emptyset$.
- $S = \{\varepsilon\}$, *i.e.* a singleton of the empty indexing.
- There is $i_h \mapsto e$ such that S can be decomposed in $(i_h \mapsto e)S' + S''$, with $S' \neq \emptyset$ and none of the sequences in S'' start with e. Here $(i_h \mapsto e)S'$ denotes prepending $i_h \mapsto e$ to all elements of S', while $+$ is disjoint union.

The above classification can serve as the basis of a definition by recursion on $n + \sharp S$ where n is the size of indexings in S and $\sharp S$ is its cardinality. Indeed in the third case in S' the size of indexings decreases strictly (and cardinality does not increase) while for S'' the size of tuples remains the same but cardinality strictly decreases. The expression e of the third case can be chosen as minimal for some total order[13].

[12] We recall that in this development, loop indexes are always mapped to simple expressions over the same index. If it was not the case, the condition obtained from an expression should be on the mapped index, not the indeterminate of the simple expression. We leave all generalisations of what we present here for further work

[13] The specific order used does not change the correctness of the procedure, but different orders can give more or less readable results. An empirically "good" order is the lexicographic one, with $a * i_k + b \leq a' * i_k + b'$ if $a < a'$ or $a = a'$ and $b \leq b'$.

We first define of the auxiliary function κ_I^α, parametrized by atoms and 0-based indexings, and going from $\mathbb{S}$ to dependent expressions, using the previous classification of elements in $\mathbb{S}$.

$$\kappa_L^\alpha(\emptyset) := 0 \qquad \kappa_L^\alpha(\{\varepsilon\}) := \kappa(\alpha\langle L\rangle) \qquad \kappa_L^\alpha((i_h \mapsto e)S' + S'') := p(e)\ ?\ \kappa^\alpha_{L(i_k \mapsto e)}(S') : \kappa_L^\alpha(S'')$$

Finally the wanted dependent cost mapping is defined by

$$\kappa(\alpha) := \kappa_\varepsilon^\alpha(\{\, L \mid \alpha\langle L\rangle \text{ appears in the compiled code}\,\}) \tag{2}$$

where one must notice that the set of indexings of an atom appearing in the code inhabits $\mathbb{S}$ because the domain of all indexings is fixed by the number of nested loops in the source code.

The correctness of the above formula, which is a consequence of Fact 3, can be stated as the following.

Fact 4. If there is no overlap (see Fact 2), and $\alpha\langle I\rangle|_C = \alpha\langle D\rangle$ for $\alpha\langle I\rangle$ occurring in the compiled code, then $\kappa(\alpha)|_D = \kappa(\alpha\langle I\rangle)$.

The no overlap hypothesis ensures that if we are in the third case $\kappa_L^\alpha((i_h \mapsto e)S' + S'')$ of the formula above and $I = L, J$ with $J \in S''$, then $p(e)|_D$ does not hold.

Indexed instrumentation. The *indexed instrumentation* generalises the instrumentation as presented in [6] and sketched in section 3. We described above how cost atoms can be mapped to dependent costs. The indexed instrumentation $\mathscr{I}^\iota$ must also insert code dealing with loop indexes. As instrumentation is done on the code produced by the labelling phase, all cost labels are indexed by identity indexings. The relevant cases of the recursive definition (supposing c is the cost variable) are then:

$$\mathscr{I}^\iota(\alpha\langle Id_k\rangle : S) = c := c + \kappa(\alpha); \mathscr{I}^\iota(S)$$
$$\mathscr{I}^\iota(i_k : \mathsf{while}\ b\ \mathsf{do}\ S) = i_k := 0; \mathsf{while}\ b\ \mathsf{do}\ (\mathscr{I}^\iota(S); i_k := i_k + 1)$$

This means that instrumentation internalises an index state C as the actual values of variables $i_0, \ldots$, and when a cost must be registered it adds to the global cost variable the value $\kappa(\alpha)|_C$ using the current index state.

Suppose we guarantee the semantic correctness of the compilation and the fact that we never produce overlapping indexed labels (Fact 2 for loop transformations, trivial for other passes). The correctness of the instrumentation then follows from Fact 4. Indeed if the source code emits $\alpha\langle C\rangle$, by semantic correctness we have the corresponding point in the execution of the compiled code emitting the same, which means that we have encountered $\alpha\langle I\rangle$ under index state D such that $\alpha\langle I\rangle|_D = \alpha\langle C\rangle$. Moreover the index state in the labelled source is C, as all indexings are identities. It follows that when evaluating the instrumentation $c := c + \kappa(\alpha)$, we add to the cost variable the amount $\kappa(\alpha)|_C = \kappa(\alpha\langle I\rangle)$, which is correct if the static analysis correctly analysed the cost.

4.5 A detailed example

Take the program in Figure 3. Its initial labelling is shown in Figure 5a. Supposing for example, $n = 3$ the trace of the program will be

$$\alpha\langle\rangle\,\beta\langle 0\rangle\,\delta\langle 0\rangle\,\beta\langle 1\rangle\,\gamma\langle 1,0\rangle\,\delta\langle 1\rangle\,\beta\langle 2\rangle\,\gamma\langle 2,0\rangle\,\gamma\langle 2,1\rangle\,\delta\langle 2\rangle\,\varepsilon\langle\rangle$$

```
α⟨⟩ : s := 0;
i := 0;
i0 : while i < n do
     β⟨i0⟩ : p := 1;
     j := 1;
     i1 : while j ≤ i do
          γ⟨i0, i1⟩ : p := j ∗ p;
          j := j + 1;
     δ⟨i0⟩ : s := s + p;
     i := i + 1;
ε⟨⟩ : skip
```

(a)

```
α⟨⟩ : s := 0;
i := 0;
if i < n then
     β⟨0⟩ : p := 1;
     j := 1;
     i1 : while j ≤ i do
          γ⟨0, i1⟩ : p := j ∗ p;
          j := j + 1;
     δ⟨0⟩ : s := s + p;
     i := i + 1;
     i0 : while i < n do
          β⟨2 ∗ i0 + 1⟩ : p := 1;
          j := 1;
          if j ≤ i then
               γ⟨2 ∗ i0 + 1, 0⟩ : p := j ∗ p;
               j := j + 1;
               if j ≤ i then
                    γ⟨2 ∗ i0 + 1, 1⟩ : p := j ∗ p;
                    j := j + 1;
                    i1 : while j ≤ i do
                         γ⟨2 ∗ i0 + 1, 2 ∗ i1 + 2⟩ : p := j ∗ p;
                         j := j + 1;
                         if j ≤ i then
                              γ⟨2 ∗ i0 + 1, 2 ∗ i1 + 3⟩ : p := j ∗ p;
                              j := j + 1;
          δ⟨2 ∗ i0 + 1⟩ : s := s + p;
          i := i + 1;
          if i < n then
               β⟨2 ∗ i0 + 2⟩ : p := 1;
               j := 1;
               i1 : while j ≤ i do
                    γ⟨2 ∗ i0 + 2, 2 ∗ i1⟩ : p := j ∗ p;
                    j := j + 1;
                    if j ≤ i do
                         γ⟨2 ∗ i0 + 2, 2 ∗ i1 + 1⟩ : p := j ∗ p;
                         j := j + 1;
               δ⟨2 ∗ i0 + 2⟩ : s := s + p;
               i := i + 1;
ε⟨⟩ : skip
```

(b)

Figure 5: The result of indexed labeling and reindexing loop transformations on the program in Figure 3. A single skip after the δ label has been suppressed, and we are writing $\alpha\langle e_0,\dots,e_k\rangle$ for $\alpha\langle i_0 \mapsto e_0,\dots,i_k \mapsto e_k\rangle$.

Now let as apply the transformations of Figure 3 with the additional information detailed in Figure 4. The result is shown in Figure 5b. One can check that the transformed code leaves the same trace when executed.

Let us compute the dependent cost of γ, supposing no other loop transformations are done. Ordering its indexings we have the list in Figure 6a. If we denote with $a,b,\dots,g$ the integer costs statically computed from the compiled code for each of the indexed occurrences of γ in the compiled code in Figure 5b, we obtain, using equation (2) and the order of indexings in Figure 6a, the depedent cost in

$0, i_1$

$2 * i_0 + 1, 0$

$2 * i_0 + 1, 1$

$2 * i_0 + 1, 2 * i_1 + 2$

$2 * i_0 + 1, 2 * i_1 + 3$

$2 * i_0 + 2, 2 * i_1$

$2 * i_0 + 2, 2 * i_1 + 1$

(a) The indexings of γ in Figure 5b.

$$
\begin{array}{l}
(i_0 = 0)\ ? \\
\quad (i_1 \geq 0)\ ?\ a : 0 : \\
\quad (i_0 \bmod 2 = 1 \wedge i_0 \geq 1)\ ? \\
\quad\quad (i_1 = 0)\ ? \\
\quad\quad\quad b : \\
\quad\quad\quad (i_1 = 1)\ ? \\
\quad\quad\quad\quad c : \\
\quad\quad\quad\quad (i_1 \bmod 2 = 0 \wedge i_1 \geq 2)\ ? \\
\quad\quad\quad\quad\quad d : \\
\quad\quad\quad\quad\quad (i_1 \bmod 2 = 1 \wedge i_1 \geq 3)\ ?\ e : 0 : \\
\quad (i_0 \bmod 2 = 0 \wedge i_0 \geq 2)\ ? \\
\quad\quad (i_1 \bmod 2 = 0 \wedge i_1 \geq 0)\ ? \\
\quad\quad\quad f : \\
\quad\quad\quad (i_1 \bmod 2 = 1 \wedge i_1 \geq 1)\ ?\ g : 0 : \\
\quad\quad 0
\end{array}
$$

(b) The dependent cost of γ as given by equation (2).

$$
\begin{array}{l}
(i_0 = 0)\ ? \\
\quad a : \\
\quad (i_0 \bmod 2 = 1)\ ? \\
\quad\quad (i_1 = 0)\ ? \\
\quad\quad\quad b : \\
\quad\quad\quad (i_1 = 1)\ ? \\
\quad\quad\quad\quad c : \\
\quad\quad\quad\quad (i_1 \bmod 2 = 0)\ ?\ d : e : \\
\quad\quad (i_1 \bmod 2 = 0)\ ?\ f : g
\end{array}
$$

(c) The dependent cost of γ as simplified by a procedure not described in this work but implemented in CerCo's compiler. Further simplifications would be possible if any of the constants turn out to be equal.

Figure 6: The dependent cost of γ in the program of Figure 7, as transformed in Figure 5b.

Figure 6b. Applying some simplifications that are not documented here but that are implemented in CerCo's untrusted prototype, we obtain the equivalent dependent cost in Figure 6c.

One should keep in mind that the example was wilfully complicated, in practice the cost expressions produced have rarely more clauses than the number of nested loops containing the annotation.

5 Future work

For the time being, indexed labels are only implemented in the untrusted Ocaml compiler, while they are not present yet in the code on which the computer assisted proof can be carried out (in case of CerCo's project, the tool used is Matita [4]). Porting them should pose no significant problem. Once ported, the task of proving properties about them in Matita can begin.

Because most of the executable operational semantics of the languages across the front end and the back end are oblivious to cost labels, it should be expected that the bulk of the semantic preservation proofs that still needs to be done will not get any harder because of indexed labels. The only trickier point that we foresee would be in the translation of Clight to Cminor, where we pass from structured indexed loops to atomic instructions on loop indexes.

An invariant which should probably be proved and provably preserved along the compilation chain is the non-overlap of indexings for the same atom. Then, supposing cost correctness for the unindexed approach, the indexed one will just need to amend the proof by stating

$$\forall C \text{ constant indexing}. \forall \alpha\langle I \rangle \text{ appearing in the compiled code}. \kappa(\alpha)|_{I_C} = \kappa(\alpha\langle I \rangle).$$

Here, C represents a snapshot of loop indexes in the compiled code, while $I \circ C$ is the corresponding snapshot in the source code. Semantics preservation will ensure that when, with snapshot C, we emit $\alpha\langle I \rangle$ (that is, we have $\alpha\langle I \circ C \rangle$ in the trace), α must also be emitted in the source code with indexing $I \circ C$, so the cost $\kappa(\alpha) \circ (I \circ C)$ applies.

Aside from carrying over the proofs, we would like to extend the approach to more loop transformations. Important examples are loop inversion (where a for loop is reversed, usually to make iterations

appear to be truly independent) or loop interchange (where two nested loops are swapped, usually to have more loop invariants or to enhance strength reduction). This introduces interesting changes to the approach, where we would have indexings such as:

$$i_0 \mapsto n - i_0 \quad \text{or} \quad i_0 \mapsto i_1, i_1 \mapsto i_0.$$

In particular dependency over actual variables of the code would enter the frame, as indexings would depend on the number of iterations of a well-behaving guarded loop (the n in the first example).

Finally, as stated in the introduction, the approach should allow some integration of techniques for cache analysis, a possibility that for now has been put aside as the standard 8051 target architecture for the CerCo project lacks a cache. Two possible developments for this line of work present themselves:

1. One could extend the development to some 8051 variants, of which some have been produced with a cache.
2. One could make the compiler implement its own cache: this cannot apply to RAM accesses of the standard 8051 architecture, as the difference in cost of accessing the two types of RAM is only one clock cycle, which makes any implementation of cache counterproductive. So for this proposal, we could either artificially change the accessing cost of RAM of the model just for the sake of possible future adaptations to other architectures, or otherwise model access to an external memory by means of the serial port of the microcontroller.

References

[1] *AbsInt Angewandte Informatik*. Available at `http://www.absint.com/`.

[2] *Certified Complexity (CerCo), FET-Open EU Project*. Available at `http://cerco.cs.unibo.it/`.

[3] *Frama-C software analyzers*. Available at `http://frama-c.com/`.

[4] *Matita*. Available at `http://matita.cs.unibo.it/`.

[5] Roberto M. Amadio, Nicolas Ayache, Yann Régis-Gianas & Ronan Saillard (2010): *Prototype implementation*. Deliverable 2.2 of Project FP7-ICT-2009-C-243881 CerCo. Available at `http://cerco.cs.unibo.it/`.

[6] Nicholas Ayache, Roberto M. Amadio & Yann Régis-Gianas (2012): *Certifying and Reasoning on Cost Annotations in C Programs*. In Mariëlle Stoelinga & Ralf Pinger, editors: *FMICS*, *Lecture Notes in Computer Science* 7437, Springer, pp. 32–46, doi:10.1007/978-3-642-32469-7_3.

[7] Christian Ferdinand & Reinhard Wilhelm (1999): *Efficient and Precise Cache Behavior Prediction for Real-TimeSystems*. *Real-Time Syst.* 17, pp. 131–181, doi:10.1023/A:1008186323068.

[8] Xavier Fornari: *Understanding how SCADE suite KCG generates safe C code*. White paper, Esterel Technologies. Available at `http://www.esterel-technologies.com/technology/WhitePapers/`.

[9] Xavier Leroy (2009): *Formal verification of a realistic compiler*. *Commun. ACM* 52(7), pp. 107–115, doi:10.1145/1538788.1538814.

[10] E. Morel & C. Renvoise (1979): *Global optimization by suppression of partial redundancies*. *Commun. ACM* 22, pp. 96–103, doi:10.1145/359060.359069.

[11] Robert Morgan (1998): *Building an Optimizing Compiler*. Digital Press.

[12] Steven S. Muchnick (1997): *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

[13] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat & Per Stenström (2008): *The worst-case execution-time problem - overview of methods and survey of tools*. *ACM Trans. Embedded Comput. Syst.* 7(3), doi:10.1145/1347375.1347389.

[14] Xuejun Yang, Yang Chen, Eric Eide & John Regehr (2011): *Finding and understanding bugs in C compilers*. In Mary W. Hall & David A. Padua, editors: *PLDI*, ACM, pp. 283–294, doi:10.1145/1993498.1993532.

A Detailed proofs of the facts

Fact 2. Loop peeling and unrolling preserve the following invariant, which we call *non-overlap of indexed labels*: for all labels $\alpha\langle I\rangle$ and $\alpha\langle J\rangle$ such that $I \neq J$, the first different simple expressions of the two are disjoint, *i.e.* they always evaluate to different constants. Moreover for every loop i_k : while e do S and label $\alpha\langle I\rangle$ in S, no label outside the loop with the same atom can share the same prefix up to i_k.

Proof. The loop transformations can be taken one by one, as each loop transformation of the whole program, independently on the heuristics used, can be factored into the single peeling or unrolling steps.

For peeling, suppose the invariant holds before it. Take two labels $\alpha\langle I_1\rangle$ and $\alpha\langle I_2\rangle$ occuring in the transformed code, with $I_1 \neq J_2$, and let i_k be the index of the peeled loop. If i_k does not appear in I_1 (and therefore not in I_2 either), we are done. So suppose $I_1 = J_1, i_k \mapsto e_1 \circ f_1, J_1'$ and $I_2 = J_2, i_k \mapsto e_2 \circ f_2, J_2'$, where $J_1, i_k \mapsto e_1, J_1'$ and $J_2, i_k \mapsto e_2, J_2'$ are the indexings in the pre-peeling code from which the two labels come. f_1 and f_2 can be either the identity (the label was outside the loop), $i_k \mapsto 0$ or $i_k \mapsto i_k + 1$. Suppose $J_1 = J_2$, otherwise the invariant on the pre-peeled code kicks in and we are done. If $e_1 \neq e_2$ we are also done, as disjointness of e_1 and e_2 implies the same for any of their compositions. Supposing $e_1 = e_2$, also when $e_1 \circ f_1 = e_1 \circ f_2$ we are done, as it means the first difference is between J_1' and J_2' and the pre-peeled code hypothesis applies.

So in fact we are left with the case where f_1 and f_2 are different transformations that take $e_1 = e_2$ to different expressions. This means that one is $i_k \mapsto 0$ and the other $i_k \mapsto i_k + 1$, and that e_1 actually uses i_k, *i.e.* it does not have a constant for it. Then it is immediate that the two are disjoint, because all non-constant simple expressions are injective.

As for preserving the second invariant, suppose that a label inside the surviving loop shares a prefix up to i_k with one outside. This is trivially impossible for one in the peeled iteration. Suppose a label outside does: however this means that in the pre-peeled code the two labels would overlap. For the same reason none of the labels transformed in the peeling process can share a prefix with a label with the same atom inside a loop not interested by the transformation.

A very similar reasoning can be applied to unrolling. □

Fact 3. For every expression e on i_k, $p(e)|_{(i_k \mapsto c)}$ iff there is a constant d such that $e|_{(i_k \mapsto d)} = c$.

Proof. The only if part is pretty straightforward. For the if part, if e is constant this is trivial. If e is "rigid", *i.e.* $e = 1 * i_k + b$, then if $i_k \geq b$ holds for $i_k = c$ it means we can write $1 \cdot (c - b) + b = c$. If $e = a * i_k + b$ with $a > 1$ and $c \bmod a = b \bmod a$, we have q_1 and q_2 such that $c = q_1 a + (c \bmod a)$ and $b = q_2 a + (b \bmod a)$. If moreover $c \geq b$, we have $q_1 \geq q_2$ and we can write

$$a(q_1 - q_2) + b = aq_1 - aq_2 + aq_2 + (b \bmod a) = aq_1 + (c \bmod a) = c.$$

□

Fact 4. If there is no overlap (see Fact 2), and $\alpha\langle I\rangle|_C = \alpha\langle D\rangle$ for $\alpha\langle I\rangle$ occurring in the compiled code, then $\kappa(\alpha)|_D = \kappa(\alpha\langle I\rangle)$.

Proof. Let's generalise the result to the fact that if LS occurs in the code as indexings of α (L is a prefix, S is a set of postfixes), $I \in S$ and $\alpha\langle LI\rangle = \alpha\langle D\rangle$ then $\kappa_L^\alpha(S)|_D = \kappa(\alpha\langle LI\rangle)$.

We reason by cases on the decomposition of S. $S = \emptyset$ is impossible as $I \in S$, and $S = \{\varepsilon\}$ so that $I = \varepsilon$ yields the result directly.

Suppose $S = (i_k \mapsto e)S' + S''$ and that $I = (i_k \mapsto e)I'$, so that $I' \in S'$. We have by the if direction of Fact 3 that $p(e)$ holds for D which is obtained by evaluation of I. So

$$\kappa_L^\alpha(S)|_D = \kappa^\alpha_{L(i_k \mapsto e)}(S')|_D = \kappa(L(i_k \mapsto e)I')$$

by inductive hypothesis and we are done.

Suppose on the other hand that $S = (i_k \mapsto e)S' + S''$ and that $I \in S''$. I must still be of the form $I = (i_k \mapsto f)I'$, with $f \neq e$. As there is a $J \in (i_k \mapsto e)S'$ with both LJ and LI occurring in the code, by non-overlap e and f must also be disjoint. This ensures that $p(e)$ does not hold for any evaluation of LI: if it were the case by the only if direction of Fact 3 we would have found a constant that equates e and f. So we can conclude as above. □

B Notes on the implementation

Implementing the indexed label approach in CerCo's untrusted Ocaml prototype does not introduce many new challenges beyond what has already been presented for the toy language. Clight, the C fragment source language of CerCo's compilation chain [6], has several more features, but few demand changes in the indexed labelled approach.

The source code of the prototype together with instructions for compiling are available on the project's homepage [2].

Goto, break and continue statements. Explicit goto statements introduce some complications as to loop optimizations. Indeed with gotos we lose the guarantee that a loop is entered from one point only in the control flow graph. Anyway loop optimizations usually loose soundness or efficiency when applied to multi-entry loops, so these are typically detected and excluded from optimizations.

In CerCo's prototype, multi-entry loops are detected by checking for exterior gotos pointing inside the body of the loop. These loops are left unindexed and do not contribute to raising the tier in the definition of indexed labelling.

Clight's loop flow control statements for breaking and continuing a loop are equivalent to appropriate goto statements. The only difference is that we are assured that they cannot cause loops to be multi-entry, and that when a transformation such as loop peeling is complete, they need to be replaced by actual gotos (which happens further down the compilation chain anyway).

Indexed loops *vs*. index update instructions. In our presentation we have indexed loops in $\iota\ell$lmp, while we hinted that later languages in the compilation chain would have specific index update instructions. In CerCo's actual compilation chain from Clight to 8051 assembly, indexed loops are only in Clight, while from Cminor onward all languages have the same three cost-involving instructions: label emitting, index resetting and index incrementing.

Loop transformations in the front end. We decided to implement the two loop transformations in the front end, namely in Clight. This decision is due to user readability concerns: if costs are to be presented to the programmer, they should depend on structures written by the programmer himself. If loop transformation were performed later it would be harder to create a correspondence between loops in the control flow graph and actual loops written in the source code. However, another solution would be to index loops in the source code and then use these indexes later in the compilation chain to pinpoint

explicit loops of the source code: loop indexes can be used to preserve such information, just like cost labels.

Function calls. Every internal function definition has its own space of loop indexes. Executable semantics must thus take into account saving and resetting the constant indexing of current loops upon hitting a function call, and restoring it upon return of control. A peculiarity is that this cannot be attached to actions that save and restore frames: namely in the case of tail calls the constant indexing needs to be saved whereas the frame does not.

Cost-labelled expressions. In labelled Clight, expressions also get cost labels, due to the presence of ternary conditional expressions (and lazy logical operators, which get translated to ternary expressions too). Adapting the indexed labelled approach to cost-labelled expressions does not pose any particular problems.

Simplification of dependent costs. As previously mentioned, the naïve application of the procedure described in 4.4 produces unwieldy cost annotations. In our implementation several transformations are used to simplify such complex dependent costs.

Disjunctions of simple conditions are closed under all logical operations, and it can be computed whether such a disjunction implies a simple condition or its negation.This can be used to eliminate useless branches of dependent costs, to merge branches that share the same value, and possibly to simplify the modulus case of simple condition skipping its inequality part. Examples of the three transformations are respectively:

- $i_0 = 0 \,?\, x : i_0 >= 1 \,?\, y : z \mapsto i_0 = 0 \,?\, x : y$,
- $c \,?\, x : d \,?\, x : y \mapsto c \vee d \,?\, x : y$,
- $i_0 = 0 \,?\, x : i_0 \bmod 2 = 0 \wedge i_0 \geq 2 \,?\, y : z \mapsto i_0 = 0 \,?\, x : i_0 \bmod 2 = 0 \,?\, y : z$.

The second transformation tends to accumulate disjunctions, to the detriment of readability. A further transformation swaps two branches of the ternary expression if the negation of the condition can be expressed with fewer clauses. For example:

$$i_0 \bmod 3 = 0 \vee i_0 \bmod 3 = 1 \,?\, x : y \mapsto i_0 \bmod 3 = 2 \,?\, y : x$$

An example with a C program. We will demonstrate the workings of the annotating compiler on a small C program. Figure 7 shows such a program. Running the CerCo annotating compiler on it with no optimization yields the annotated source code in Figure 8. Notice how the annotations give back to the programmer the actual cost in clock cycles of the object code. Finally, applying some loop transformations too, the source in Figure 9 is obtained, by turning on indiscriminate peeling and unrolling (by 2), constant propagation and partial redundancy elimination. We can see that, as can be expected of the 8051 architecture that lacks pipelining, unrolling does not make much difference between even and odd iterations.

```
int min (int tab[], int size) {
  int i, min_index, min_val;

  min_index = 0;
  min_val = tab[min_index];
  for (i = 1 ; i < size ; i++) {
    if (tab[i] < min_val) {
      min_index = i;
      min_val = tab[min_index];
    }
  }

  return min_index;
}
```

Figure 7: A small program computing the index of the minimal element in an array.

```
int _cost = 0;

void _cost_incr(int incr)
{
  _cost = _cost + incr;
}

int min(int *tab, int size)
{
  int _i0;
  int i;
  int min_index;
  int min_val;
  _cost4:
  _cost_incr(443);
  min_index = 0;
  min_val = tab[min_index];
  _i0 = 0;
  for (i = 1; i < size; i = i + 1) {
    _cost2_i0:
    _cost_incr(231);
    if (tab[i] < min_val) {
      _cost0_i0:
      _cost_incr(270);
      min_index = i;
      min_val = tab[min_index];
    } else {
      _cost1_i0:
      _cost_incr(77);
    }
    _i0 = _i0 + 1;
  }
  _cost3:
  _cost_incr(187);
  return min_index;
}
```

Figure 8: The result of running `acc -a` on the program of Figure 7.

```
int _cost = 0;

void _cost_incr(int incr)
{
  _cost = _cost + incr;
}

int min(int *tab, int size)
{
  int _i0;
  int i;
  int min_index;
  int min_val;
  _cost4:
  _cost_incr(267);
  min_index = 0;
  min_val = tab[min_index];
  _i0 = 0;
  for (i = 1; i < size; i = i + 1) {
    _cost2_i0:
    _cost_incr(126);
    if (tab[i] < min_val) {
      _cost0_i0:
      _cost_incr(_i0 == 0 ? 102 : (_i0 % 2 == 1 ? 150 : 153));
      min_index = i;
      min_val = tab[min_index];
    } else {
      _cost1_i0:
      _cost_incr(49);
    }
    _i0 = _i0 + 1;
  }
  _cost3:
  _cost_incr(187);
  return min_index;
}
```

Figure 9: The result of running `acc -a -peel -unroll -cst-prop -pre` on the program of Figure 7.