Loop Quasi-Invariant Chunk Motion
by peeling with statement composition

Jean-Yves Moyen
Department of Computer Science
University of Copenhagen (DIKU)
moyen@lipn.univ-paris13.fr

Thomas Rubiano
Université Paris 13 - LIPN
Department of Computer Science
University of Copenhagen (DIKU)
rubiano@lipn.univ-paris13.fr

Thomas Seiller
Department of Computer Science
University of Copenhagen (DIKU)
seiller@di.ku.dk

Several techniques for analysis and transformations are used in compilers. Among them, the peeling of loops for hoisting quasi-invariants can be used to optimize generated code, or simply ease developers' lives. In this paper, we introduce a new concept of dependency analysis borrowed from the field of Implicit Computational Complexity (ICC), allowing to work with composed statements called "Chunks" to detect more quasi-invariants. Based on an optimization idea given on a `WHILE` language, we provide a transformation method - reusing ICC concepts and techniques [9, 10] - to compilers. This new analysis computes an invariance degree for each statement or chunks of statements by building a new kind of dependency graph, finds the "maximum" or "worst" dependency graph for loops, and recognizes if an entire block is Quasi-Invariant or not. This block could be an inner loop, and in that case the computational complexity of the overall program can be decreased. We already implemented a proof of concept on a toy C parser[1] analysing and transforming the AST representation.

In this paper, we introduce the theory around this concept and present a prototype analysis pass implemented on LLVM. In a very near future, we will implement the corresponding transformation and provide benchmarks comparisons.

1 Introduction

A command inside a loop is an *invariant* if its execution has no effect after the first iteration of the loop. Typically, an assignment `x:=0` in a loop is invariant (provided `x` is not modified elsewhere). Loop invariants can safely be moved out of loops (*hoisted*) in order to make the program faster.

A command inside a loop is *quasi-invariant* if its execution has no effect after a finite number of iterations of the loop. Typically, if a loop contains the sequence `x:=y, y:=0`, then `y:=0` is invariant. However, `x:=y` is **not** invariant. The first time the loop is executed, `x` will be assigned the old value of `y`, and only from the second time onward will `x` be assigned the value `0`. Hence, this command is *quasi-invariant*. It can still be hoisted out of the loop, but to do so requires to *peel* the loop first, that is execute its body once (by copying it before the loop).

The number of times a loop must be executed before a quasi-invariant can be hoisted is called here the *degree* of the invariant.

[1] `https://github.com/ThomasRuby/LQICM_On_C_Toy_Parser`

G. Bonfante, G. Moser (Eds.): 8th Workshop on Developments in Implicit Computational complExity and 5th Workshop on Foundational and Practical Aspects of Resource Analysis (DICE-FOPARA 2017)
EPTCS 248, 2017, pp. 47–59, doi:10.4204/EPTCS.248.9

An obvious way to detect quasi-invariants is to first detect invariants (that is, quasi-invariants of degree 1) and hoist them; and iterate the process to find quasi-invariant of degree 2, and so on. This is, however, not very efficient since it may require a large number of iterations to find some invariance degrees.

We provide here an analysis able to directly detect the invariance degree of any statements in the loop. Moreover, our analysis is able to assign an invariance degree non only to individual statements but also to groups of statements (called *chuncks*). That way it is possible, for example, to detect that a whole inner loop is invariant and hoist it, thus decreasing the asymptotic complexity of the program.

Loop optimization techniques based on quasi-invariance are well-known in the compilers community. The transformation idea is to peel loops a finite number of time and hoist invariants until there are no more quasi-invariants. As far as we know, this technique is called "peeling" and it was introduced by Song *et al.* [13].

The present paper offers a new point of view on this work. From an optimization on a `WHILE` language by Lars Kristiansen [9], we provide a redefinition of peeling and another transformation method based on techniques developed in the field of Implicit Computational Complexity.

Implicit Computational Complexity (ICC) studies computational complexity in terms of restrictions of languages and computational principles, providing results that do not depend on specific machine models. Based on static analysis, it helps predict and control resources consumed by programs, and can offer reusable and tunable ideas and techniques for compilers. ICC mainly focuses on syntactic [4, 3], type [6, 2] and Data Flow [11, 7, 8, 12] restrictions to provide bounds on programs' complexity. The present work was mainly inspired by the way ICC community uses different concepts to perform Data Flow Analysis, e.g. "Size-change Graphs" [11] or "Resource Control Graphs"[12] which track data values' behavior and use a matrix notation inspired by [1], or "mwp-polynomials" [8] to provide bounds on data size.

For our analysis, we focus on dependencies between variables to detect invariance. Dependency graphs [10] can have different types of arcs representing different kind of dependencies. Here we will use a kind of Dependence Graph Abstraction [5] that can be used to find local and global quasi-invariants. Based on these techniques, we developed an analysis pass and we will implement the corresponding transformation in LLVM.

We propose a tool which is notably able to give enough information to easily peel and hoist an inner loop, thus decreasing the complexity of a program from n^2 to n.

1.1 State of the art on Quasi-Invariant detection in loop

Invariants are basically detected using algorithm 1.

A *dependency graph* around variables is needed to provide relations between statements. For quasi-invariance, we need to couple dependence and *dominance* informations. In [13], the authors define a *variable dependency graph* (VDG) and detect a loop quasi-invariant variable `x` if, among all paths ending at `x`, no path contain a node included in a circular path. Then they deduce an *invariant length* which corresponds to the length of the longest path ending in `x`. In the present paper, this *length* is called *invariance degree*.

1.2 Contributions

To the authors' knowledge, this is the first application of ICC techniques on a mainstream compiler. One interest is that our tool potentially applies to programs written in any programming language managed

by LLVM. Moreover, this work should be considered as a first step of a larger project that will make ICC techniques more accessible to programmers.

On a more technical side, our tool aims at improving on currently implemented loop invariant detection and optimization techniques. The main LLVM tool for this purpose, Loop Invariant Code Motion (LICM), does not detect quasi-invariant of degree more than 3 (and not all of those of degree 2). More importantly, LICM will not detect quasi-invariant blocks of code (what we call *chunk*), such as whole loops. Our tool, on the other hand, detects quasi-invariants of arbitrary degree and is able to deal with chunks. For instance the optimization shown in Figure 9 is not performed by LLVM nor in GCC even at their maximum optimization level.

2 In theory

In this section, we redefine our own types of relations between variables to build a new dependency graph and apply a composition inspired by the graph composition of *Size-Change Termination* [11].

2.1 Relations and Data Flow Graph

We work with a simple imperative `WHILE`-language (the grammar is shown in Figure 1), with semantics similar to `C`.

(Variables)	X	::=	$X_1 \mid X_2 \mid X_3 \mid \ldots \mid X_n$
(Expression)	*exp*	::=	$X \mid$ `op(`*exp*`,...,`*exp*`)`
(Command)	*com*	::=	X=*exp* | *com;com* | `skip` | `while` *exp* `do` *com* `od` | `if` *exp* `then` *com* `fi` | `use(`X_1`,...,`X_n`)`

Figure 1: Grammar

A `WHILE` program is thus a sequence of statements, each statement being either an *assignment*, a *conditional*, a *while* loop, a *function call* or a *skip*. The *use* command represents any command which does not modify its variables but use them and should not be moved around carelessly (typically, a `printf`). *Statements* are abstracted into *commands*. A *command* can be a statement or a sequence of commands. We also call a sequence of commands a *chunk*.

Data: List of Statements in the Loop
Result: List of Loop-invariants `LI`
Initialization;
while *search until there is no new invariant...* **do**
  for *each statement* `s` **do**
    if *each variable in* `s`
    has no definition in the loop ***or***
    has exactly one loop-invariant definition ***or***
    is constant **then**
      Add `s` to `LI`;
    end
  end
end

Algorithm 1: Basic invariants detection

We start by giving an informal but intuitive definition of the notion of *Data Flow Graph* (DFG). A DFG represents dependencies between variables as a bipartite graph as in Figure 2. Each different types of arrow represents different types of dependencies.

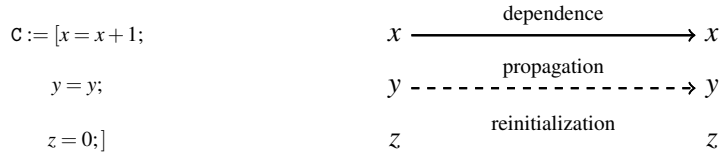


Figure 2: Types of dependence

Each variable is shown twice: the occurrence on the left represents the variable before the execution of the command while the occurrence on the right represents the variable after the execution. Dependencies are then represented by two types of arrows from variables on the left to variables on the right: plain arrows for *direct dependency*, dashed arrows for *propagation*. *Reinitialisation* of a variable z then corresponds to the absence of arrows ending on the right occurrence of z. Figure 2 illustrates these types of dependencies; let us stress here that the DFG would be the same if the assignment $y = y;$ were to be removed[2] from C since the value of y is still propagated.

More formally, a DFG of a command C is a triple $(V, \mathscr{R}_{\text{dep}}, \mathscr{R}_{\text{prop}})$ with V the variables involved in the command C and a pair of two relations on the set of variables. These two relations express how the values of the involved variables *after* the execution of the command depend on their values *before* the execution. There is a *direct* dependence between variables appearing in an expression and the variable on the left-hand side of the assignment. For instance x directly depends on y and z in the statement $x = y + z;$. When variables are unchanged by the command we call it *propagation*. Propagation only happens when a variable is not affected by the command, not when it is copied from another variable. If the variable is set to a constant, we call this a *reinitialization*.

More technically, we will work with an alternative representation in terms of matrices. While less intuitive, this allows for more natural compositions, based on standard linear algebra operations. Before providing the formal definition, let us introduce the semi-ring $\{\emptyset, 0, 1\}$: the addition $\oplus$ and multiplication $\otimes$ are defined in Figure 3. Let us remark that, identifying $\emptyset$ as $-\infty$, this is a sub-semi-ring of the standard *tropical semi-ring*, with $\oplus$ and $\otimes$ interpreted as max and $+$ respectively.

$\oplus$	$\emptyset$	0	1
$\emptyset$	$\emptyset$	0	1
0	0	0	1
1	1	1	1

$\otimes$	$\emptyset$	0	1
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
0	$\emptyset$	0	1
1	$\emptyset$	1	1

Figure 3: Addition and Multiplication in the semi-ring $\{\emptyset, 0, 1\}$.

Definition 1 *A* Data Flow Graph *for a command* C *is a $n \times n$ matrix over the semi-ring $\{\emptyset, 0, 1\}$ where n is the number of variables involved in* C.

We write $\mathrm{M}(\mathtt{C})$ *the DFG of* C. *At line i, column j, we have a $\emptyset$ if the output value of the jth variable does not depend on the input value of the ith; a 0 in case of propagation (unmodified variable); and a 1 for any other kind of dependence.*

[2]Note that $y = y;$ does not create a direct dependence

Definition 2 *Let* C *be a command. We define* In(C) *(resp.* Out(C)*) as the set of variables* used *(resp.* modified*) by* C*.*

Note that In(C) and Out(C) are exactly the set of variables that are at either ends of the "dependence" arrows.

2.2 Constructing DFGs

We now describe how the DFG of a command can be computed by induction on the structure of the command. Base cases (skip, use and assignment) are done in the obvious way, generalising slightly the definitions of DFGs shown in Figure 2.

2.2.1 Composition and Multipath

We now turn to the definition of the DFG for a (sequential) *composition* of commands. This abstraction allows us to see a block of statements as one command with its own DFG.

Definition 3 *Let* C *be a sequence of commands* $[\mathtt{C}_1;\mathtt{C}_2;\ldots;\mathtt{C}_n]$*. Then* M(C) *is defined as the matrix product* $\mathrm{M}(\mathtt{C}_1)\mathrm{M}(\mathtt{C}_2)\ldots\mathrm{M}(\mathtt{C}_n)$.

Following the usual product of matrices, the product of two matrices A, B is defined here as the matrix C with coefficients: $C_{i,j} = \bigoplus_{k=1}^{n}(A_{i,k} \otimes B_{k,j})$.

This operation of matrix multiplication corresponds to the computation of *multipaths* [11] in the graph representation of DFGs. We illustrate this intuitive construction on an example in Figure 4.

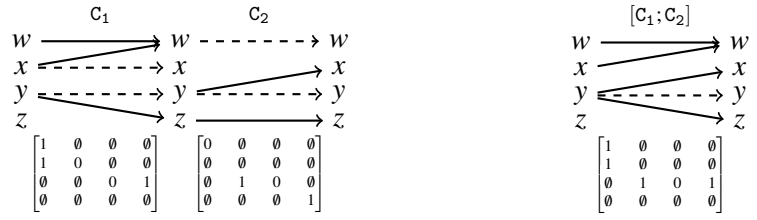


Figure 4: DFG of Composition.

Here $\mathtt{C}_1 := [w = w + x; z = y + 2;]$ and $\mathtt{C}_2 := [x = y; z = z * 2;]$

2.2.2 Condition

We now explain how to compute the DFG of a command $\mathtt{C} := \mathtt{if\ E\ then\ C}_1;$, from the DFG of the command $\mathtt{C}_1$.

Firstly, we notice that in C, all modified variables in $\mathtt{C}_1$, i.e. in Out($\mathtt{C}_1$), will depend on the variables used in E. Let us denote by $\mathrm{M}(\mathtt{C}_1)^{[\mathtt{E}]}$ the corresponding DFG, i.e. the matrix $\mathrm{M}(\mathtt{C}_1) \oplus (E^t O)$, where E (resp. O) is the vector representing variables in[3] Var(E) (resp. in Out(C_1)), and $(\cdot)^t$ denotes the transpose.

Secondly, we need to take into account that the command $\mathtt{C}_1$ may be skipped. In that case, the overall command C should act as an empty command, i.e. be represented by the identity matrix Id (diagonal elements are equal to 0, all other are equal to $\emptyset$).

Finally, the DFG of a conditional will be computed by summing these two possibilities, as in Figure 5.

Definition 4 *Let* C *be a command of the form* $\mathtt{if\ E\ then\ C}_1;$*. Then* $\mathrm{M}(\mathtt{C}) = \mathrm{M}(\mathtt{C}_1)^{[\mathtt{E}]} \oplus \mathrm{Id}$.

[3] I.e. the vector with a coefficient equal to 1 for the variables in Var(E), and $\emptyset$ for all others variables.

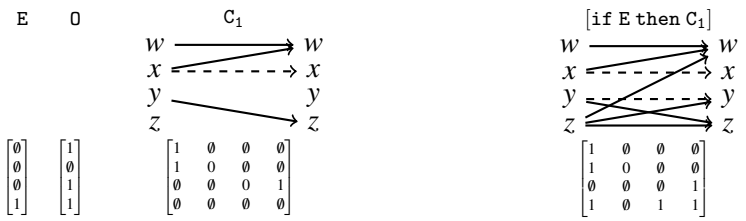


Figure 5: DFG of Conditional.

Here $\mathtt{E} := z \geq 0$ and $\mathtt{C_1} := [w = w + x; z = y + 2; y = 0;]$;

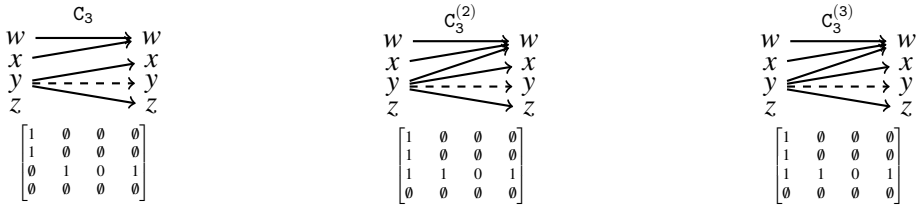


Figure 6: Finding fix point of dependence (simple example)

Here $\mathtt{C_3} := [w = w + x; z = y + 2; x = y; z = z * 2]$;

2.2.3 While Loop

Finally, let us define the DFG of a command $\mathtt{C}$ of the form $\mathtt{C} := \mathtt{while\ E\ do\ C_1;}$. This definition splits into two steps. First, we define a matrix $\mathrm{M}(\mathtt{C_1^*})$ representing iterations of the command $\mathtt{C_1}$; then we deal with the condition of the loop in the same way we interpreted the conditional above.

When considering iterations of $\mathtt{C_1}$, the first occurrence of $\mathtt{C_1}$ will influence the second one and so on. Computing the DFG of $\mathtt{C_1^n}$, the n-th iteration of $\mathtt{C_1}$, is just computing the power of the corresponding matrix, i.e. $\mathrm{M}(\mathtt{C_1^n}) = \mathrm{M}(\mathtt{C_1})^n$. But since the number of iteration cannot be decided *a priori*, we need to add all possible values of n. The following expression then expresses the DFG of the (informal) command $\mathtt{C_1^*}$ corresponding to "iterating $\mathtt{C_1}$ a finite (but arbitrary) number of times":

$$\mathrm{M}(\mathtt{C_1^*}) = \mathrm{limit}_{k\to\infty} \bigoplus_{i=1}^{k} \mathrm{M}(\mathtt{C_1})^i$$

To ease notations, we note $\mathrm{M}(\mathtt{C_1^{(k)}})$ the partial summations $\sum_{i=1}^{k} \mathrm{M}(\mathtt{C_1})^i$.

Since the set of all relations is finite and the sequence $(\mathrm{M}(\mathtt{C_1^{(k)}}))_{k\geqslant 0}$ is monotonous, this sequence is eventually constant. I.e., there exists a natural number N such that $\mathrm{M}(\mathtt{C_1^{(k)}}) = \mathrm{M}(\mathtt{C_1^{(N)}})$ for all $k \geqslant N$. One can obtain the following bound on the value of N.

Lemma 1 *Consider a command* $\mathtt{C}$ *and define* $K = \min(i, o)$*, where* i *(resp.* o*) denotes the number of variables in* $\mathrm{In}(\mathtt{C})$ *(resp.* $\mathrm{Out}(\mathtt{C})$*). Then, the sequence* $(\mathrm{M}(\mathtt{C^{(k)}}))_{k\geqslant K}$ *is constant.*

Figure 6 illustrates the computation of $\cdot^*$. The second step then consists in dealing with the loop condition, using the same constructions as for conditionals.

Definition 5 *Let* $\mathtt{C}$ *be a command of the form* $\mathtt{while\ E\ do\ C_1;}$*. Then* $\mathrm{M}(\mathtt{C}) = \mathrm{M}(\mathtt{C_1^*})^{[\mathtt{E}]}$.

2.3 Independence

Our purpose is to move commands around: exchange them, but more importantly to pull them out of loops when possible. We allow these moves only when semantics are preserved: to ensure this is the case, we describe a notion of independence.

Definition 6 *If* $\mathrm{Out}(\mathtt{C}_1) \cap \mathrm{In}(\mathtt{C}_2) = \emptyset$ *then* $\mathtt{C}_2$ *is* independent *from* $\mathtt{C}_1$. *This is denoted* $\mathtt{C}_1 \prec \mathtt{C}_2$.

It is important to notice that this notion is not symmetric. As an example, let us consider Figure 7: Here, $\mathtt{C}_2$ is independent from $\mathtt{C}_1$ but the inverse is not true.

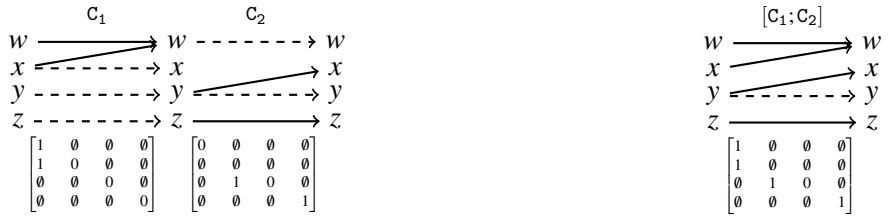


Figure 7: Composition of independent chunks of commands

Here $\mathtt{C}_1 := [w = w + x;]$ and $\mathtt{C}_2 := [x = y; z = z * 2];$

A particular case is *self-independence*, i.e. independence of a command w.r.t. itself. In that case, we can find non-trivial program transformations preserving the semantics. We denote by $[\![\mathtt{C}]\!] \equiv [\![\mathtt{D}]\!]$ the relation "C and D have the same semantics".

Lemma 2 (Specialization for `while`) *If* $\mathtt{C}_1$ *is self-independent and* $\mathrm{Var}(\mathtt{E}) \cap \mathrm{Out}(C_1) = \emptyset$*:*

$$[\![\mathtt{while\ E\ do\ C_1}]\!] \equiv [\![\mathtt{if\ E\ then\ C_1; While\ E\ do\ skip}]\!]$$

Remark that we need to keep the loop `While` with a skip statement inside because we need to consider an infinite loop if E is always true to keep the semantic equivalent.

In general, we will consider mutual independence.

Definition 7 *If* $\mathtt{C}_2 \prec \mathtt{C}_1$ *and* $\mathtt{C}_1 \prec \mathtt{C}_2$*, we say that* $\mathtt{C}_2$ *and* $\mathtt{C}_1$ *are mutually independents, and write* $\mathtt{C}_1 \asymp \mathtt{C}_2$.

While independence in one direction only, such as in the example above, does not imply that $\mathtt{C}_1;\mathtt{C}_2$ and $\mathtt{C}_2;\mathtt{C}_1$ have the same semantics, mutual independence allows to perform program transformation that do not impact the semantics.

Lemma 3 (Swapping commands) *If* $\mathtt{C}_1 \asymp \mathtt{C}_2$*, then* $[\![\mathtt{C}_1;\mathtt{C}_2]\!] \equiv [\![\mathtt{C}_2;\mathtt{C}_1]\!]$

Lemma 4 (Moving out of `while` loops) *If* $\mathtt{C}_1$ *is self-independent (i.e.* $\mathtt{C}_1 \asymp \mathtt{C}_1$*), and if* $\mathtt{C}_1 \asymp \mathtt{C}_2$*, then:*

$$[\![\mathtt{while\ E\ do\ [C_1;C_2]}]\!] \equiv [\![\mathtt{if\ E\ then\ C_1; while\ E\ do\ C_2}]\!]$$

Based on those lemmas, we can decide that an entire block of statement is invariant or quasi-invariant in a loop by computing the DFGs. The quasi-invariance comes with an *invariance degree* wich is the number of time the loop needs to be peeled to be able to hoist the corresponding invariant. We can then implement program transformations that reduce the overall complexity while preserving the semantics.

3 In practice

This section explains how we implemented the pass which computes the *invariance degree* and gives the main idea of how the transformation can be performed. In the previous Section, we have seen that the transformation is possible from and to a WHILE language; and from a previous implementation[4], we have shown it can be done on `C` Abstract Syntax Trees.

Compilers, and especially LLVM on which we are working, use an *Intermediate Representation* to handle programs. This is an assembly-like language that is used during all the stages of the compilation. Programs (in various different languages) are first translated into the IR, then several optimisations are performed (implemented in so-called *passes*), and finally the resulting IR is translated again in actual assembly language depending on the machine it will run on. Using a common IR allows to do the same optimisations on several different source languages and for several different target architectures.

One important feature of the LLVM IR is the *Single Static Assignment* form (SSA). A program is in SSA form if each variable is assigned at most once. In other words, setting a program in SSA form require a massive α-conversion of all the variables to ensure uniqueness of names. The advantages are obvious since this removes any name-aliasing problem and ease analysis and transformation.

The main drawback of SSA comes when several different paths in the Control Flow reach the same point (typically, after a conditional). Then, the values used after this point may come from any branch and this cannot be statically decided. For example, if the original program is `if (y) then x:=0 else x:=1;C`, it is relatively easy to turn it into a pseudo-SSA form by α-converting the `x`: `if (y) then` x_0`:=0 else` x_1`:=1;C` but we do not know in `C` which of x_0 or x_1 should be used.

SSA solves this problem by using *φ-functions* that, at runtime, can choose the correct value depending on the path just taken. That is, the correct SSA form will be `if (y) then` x_0`:=0 else` x_1`:=1; X:=`φ`(`x_0`, `x_1`); C`.

While the SSA itself eases the analysis, we do have to take into account the φ functions and handle them correctly.

3.1 Preliminaries

First, we want to visit all loops using a bottom-up strategy (the inner loop first). Then, as for the *Loop Invariant Code Motion* (LICM) pass, our pass is derived from the basic `LoopPass`. Which means that each time a loop is encountered, our analysis is performed.

At this point, the purpose is to gather the relations of all instructions in the loop to compose them and provide the final relation for the current loop. We decided to define a `Relation` object by three `SmallPtrSet` of `Value*`, listing the *variables*, the *propagations* and the *initializations*. Furthermore we represent the *dependencies* by a `DenseMap` of `Value*` to `SmallPtrSet<Value*>`. This way of representing our data is not fixed, it's certainly optimizable, but we think it's sufficient for our prototype analysis and examples. We will discuss the cost of this analysis later.

Then a `Relation` is generated for each command using a top-down strategy following the dominance tree. The *SSA* form helps us to gather dependence information on instructions. By visiting operands of each assignment, it's easy to build our map of `Relation`. With all the current loop's relations gathered, we compute the compositions, condition corrections and the maximums relations possible as described previously. Obviously this method can be enhanced by an analysis on bounds around conditional and number of iterations for a loop. Finally, with those composed relations we compute an invariance degree for each statement in the loop.

[4] `https://github.com/ThomasRuby/LQICM_On_C_Toy_Parser`

The only chunks considered in the current implementation are the one consisting of `while` or `if-then-else` statements.

3.2 Invariance degree computation

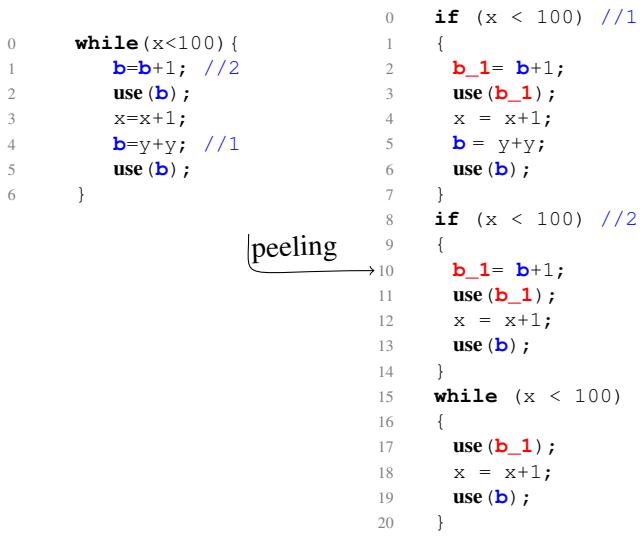
In this part, we will describe an algorithm – using the previous concepts – to compute the invariance degree of each quasi-invariant in a loop. After that, we will be able to peel the loop at once instead of doing it iteratively. To simplify and as a recall, Figure 8 shows a basic example of peeled loop.

The invariance degrees are given as comment after each Quasi-Invariant statements. So `b=y+y` is invariant of degree equal to one because `y` is invariant, that means it could be hoisted directly in the *preheader* of the loop. But `b` is used before, in `b=b+1`, so it's not the same `b` at the first iteration. We need to isolate this case by peeling one time the entire loop to use the first `b` computed by the initial `b`. If `b=y+y` is successfully hoisted, then `b` is now invariant. So we can remove `b=b+1` but we need to do it at least one time after the first iteration to set `b` to the new and invariant value. This is why the loop is peeled two times. The first time, all the statements are executed. The second time, the first degree invariants are removed. The main work is to compute the proper invariance degree for each statement and composed statements. This can be done statically using the dependency graph and dominance graph. Here is the algorithm. Let suppose we have computed the list of dependencies for all commands in a loop.

```
Data: Dependency Graph and Dominance Graph
Result: List of invariance degree for each statement
Initialize degrees of use to ∞ and others to 0;
for each statement s do
    if the current degree cd ≠ 0 then
        skip
    else
        Initialize the current degree cd to ∞;
        if there is no dependence for the current chunk then
            cd = 1;
        else
            for each dependence compute the degree dd of the command do
                if cd ≤ dd and the current command dominates this dependence then
                    cd = dd + 1
                else
                    cd = dd
                end
            end
        end
    end
end
```

Algorithm 2: Invariance degree computation.

This algorithm is dynamic. It stores progressively each degree needed to compute the current one and reuse them. Note that, for the initialization part, we are using LLVM methods (`canSinkOrHoist`, `isGuaranteedToExecute` etc...) to figure out if an instruction is movable or not. These methods provide the anchors instructions for the current loop.




```
while(x<100){
    b=b+1; //2
    use(b);
    x=x+1;
    b=y+y; //1
    use(b);
}
```

```
if (x < 100) //1
{
  b_1= b+1;
  use(b_1);
  x = x+1;
  b = y+y;
  use(b);
}
if (x < 100) //2
{
  b_1= b+1;
  use(b_1);
  x = x+1;
  use(b);
}
while (x < 100)
{
  use(b_1);
  x = x+1;
  use(b);
}
```

Figure 8: Example: Hoisting twice.

3.3 Peeling loop idea

The transformation will consists in creating as many `preheaders` basic blocks before the loop as needed to remove all quasi-invariants out of the loop. Each `preheader` will have the same condition as the `.cond` block of the loop and will contain the incrementation of the iteration variable. The maximum invariance degree is the number of time we need to peel the loop. So we can create as many `preheaders` before the loop. For each block created, we include every commands with a higher or equal invariance degree. For instance, the first `preheader` block will contain every commands with an invariance degree higher or equal to 1, the second one, higher or equal to 2 etc... and the final loop will contain every commands with an invariance degree equal to ∞.

4 Conclusion and Future work

Developers expect that compilers provide certain more or less "obvious" optimizations. When peeling is possible, that often means: either the code was generated; or the developers prefer this form (for readability reasons) and expect that it will be optimized by the compiler; or the developers haven't seen the possible optimization (mainly because of the obfuscation level of a given code).

Our generic pass is able to provide a reusable abstract dependency graph and the quasi-invariance degrees for further loop optimization or analysis.

In this example (Figure 9), we compute the same factorial several times. We can detect it statically, so the compiler has to optimize it at least in `-O3`. Our tests showed that is done neither in LLVM nor in GCC (we also tried `-fpeel_loops` with profiling). The generated assembly shows the factorial computation in the inner loop.

Moreover, the computation time of this kind of algorithm compiled with `clang` in `-O3` still computes n times the inner loop so the computation time is quadratic, while hoisting it result in linear time. For the example shown in Figure 9, our pass computes the degrees shown in Figure 11 (where -1 represents a degree of ∞, that is an instruction that cannot be hoisted).

```
    srand(time(NULL));
    int n=rand()%10000;
    int j=0;
    while(j<n){
        fact=1;
        i=1;
        while (i<=n) {
            fact=fact*i;
            i=i+1;
        }
        j=j+1;
        use(fact);
    }
```

peeling →

```
srand(time(NULL));
int n = rand() % 10000;
int j = 0;
if (j < n)
{
  fact = 1;
  i = 1;
  while (i <= n)
  {
    fact = fact * i;
    i = i + 1;
  }
  j = j + 1;
  use(fact);
}
while (j < n)
{
  j = j + 1;
  use(fact);
}
```

Figure 9: Hoisting inner loop

```
...
while.cond:
  %i.0 = phi i32 [ 1, %entry ], [ %i.1.lcssa, %while.end ]
  %j.0 = phi i32 [ 0, %entry ], [ %add8, %while.end ]
  %exitcond = icmp ne i32 %j.0, 100
  br i1 %exitcond, label %while.body, label %while.end9

while.body:
  br label %while.cond3

while.cond3:
  %fact.0 = phi i32 [ 1, %while.body ], [ %mul, %while.body6 ]
  %i.1 = phi i32 [ 1, %while.body ], [ %add, %while.body6 ]
  %cmp4 = icmp slt i32 %i.1, %rem
  br i1 %cmp4, label %while.body6, label %while.end

while.body6:
  %mul = mul nsw i32 %fact.0, %i.1
  %add = add nuw nsw i32 %i.1, 1
  br label %while.cond3

while.end:
  %fact.0.lcssa = phi i32 [ %fact.0, %while.cond3 ]
  %i.1.lcssa = phi i32 [ %i.1, %while.cond3 ]
  %call7 = call ... i32 %i.1.lcssa, i32 %fact.0.lcssa)
  %add8 = add nuw nsw i32 %j.0, 1
  br label %while.cond
...
```

Figure 10: LLVM Intermediate Representation

```
1  ---- MapDeg of while.cond3 ----
2  %mul = mul nsw i32 %fact.1, %i.1 = -1
3  %add = add nuw nsw i32 %i.1, 1 = -1
4  --------------------
5
6  ---- MapDeg of while.cond ----
7  %fact.0 = phi i32 [ 1, %while.body ], ... = 1
8  %i.1 = phi i32 [ 1, %while.body ], ... = 1
9  inner loop starting with while.cond3: = 1
10 %fact.0.lcssa = phi i32 [ %fact.0, %while.cond3 ] = -1
11 %i.1.lcssa = phi i32 [ %i.1, %while.cond3 ] = -1
12 %call7 = call ... i32 %i.1.lcssa, i32 %fact.0.lcssa) = -1
13 %add8 = add nuw nsw i32 %j.0, 1 = -1
14 --------------------
```

Figure 11: Invariance Degree

To each instruction printed corresponds an invariance degree. The assignment instructions are listed by loops, the inner loop (starting with `while.cond3`) and the outer loop (starting with `while.cond`). The inner loop has its own invariance degree equal to 1 (line 9). Remark that we do consider the `phi` initialization instructions of an inner loop. Here `%fact.0` and `%i.1` are reinitialized in the inner loop condition block. So `phi` instructions are analysed in two different cases: to compute the relation of the current loop or to give the initialization of a variable sent to an inner loop. Our analysis only takes the relevant operand regarding to the current case and do not consider others.

The code of this pass is available online[5]. To provide some real benchmarks on large programs we need to implement the transformation. We are currently implementing this second pass on LLVM.

Acknowledgments The authors wish to thank L. Kristiansen for communicating a manuscript [9] that initiated the present work. Jean-Yves Moyen is supported by the European Commision's Marie Skłodowska-Curie Individual Fellowship (H2020-MSCA-IF-2014) 655222 - Walgo; Thomas Rubiano is supported by the ANR project "Elica" ANR-14-CE25-0005; Thomas Seiller is supported by the European Commision's Marie Skłodowska-Curie Individual Fellowship (H2020-MSCA-IF-2014) 659920 - ReACT.

References

[1] A. Abel & T. Altenkirch (2002): *A Predicative Analysis of Structural Recursion*. *Journal of Functional Programming* 12(1), doi:10.1017/S0956796801004191.

[2] Patrick Baillot & Kazushige Terui (2009): *Light types for polynomial time computation in lambda calculus*. *Information and Computation* 207(1), pp. 41–62, doi:10.1016/j.ic.2008.08.005.

[3] S. Bellantoni & S. Cook (1992): *A new recursion-theoretic characterization of the poly-time functions*. *Computational Complexity* 2, doi:10.1007/BF01201998.

[4] A. Cobham (1962): *The intrinsic computational difficulty of functions*. In Y. Bar-Hillel, editor: *CLMPS*, doi:10.2307/2270886.

[5] John Cocke (1970): *Global Common Subexpression Elimination*. *SIGPLAN Not.* 5(7), doi:10.1145/390013.808480.

[6] J.-Y. Girard (1987): *Linear Logic*. *Th. Comp. Sci.* 50, doi:10.1016/0304-3975(87)90045-4.

[5] `https://github.com/ThomasRuby/lqicm_pass`

[7] M. Hofmann (1999): *Linear types and Non-Size Increasing polynomial time computation*. In: *LICS*, pp. 464–473, doi:10.1109/LICS.1999.782641.

[8] Neil D. Jones & Lars Kristiansen (2009): *A flow calculus of* mwp*-bounds for complexity analysis*. *Trans. Comp. Logic* 10(4), pp. 28:1–28:41, doi:10.1145/1555746.1555752.

[9] L. Kristiansen: *Notes on Code Motion*. Manuscript.

[10] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure & M. Wolfe (1981): *Dependence Graphs and Compiler Optimizations*. In: *POPL*, doi:10.1145/567532.567555.

[11] C. S. Lee, N. D. Jones & A. M. Ben-Amram (2001): *The Size-Change Principle for Program Termination*. In: *POPL*, doi:10.1145/360204.360210.

[12] Jean-Yves Moyen (2009): *Resource control graphs*. *ACM Trans. Computational Logic* 10, doi:10.1145/360204.360210.

[13] Litong Song, Yoshihiko Futurama, Robert Glück & Zhenjiang Hu (2000): *A Loop Optimization Technique Based on Quasi-Invariance*, pp. 80–90. doi:10.1.1.17.8939.