Loop Quasi-Invariant Chunk Motion by peeling with statement composition

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
rubiano@lipn.univ-paris13.fr

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

Abstract
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 [8] - 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.

Categories and Subject Descriptors F.3.1 [Logics and Meaning of Programs]: Invariants

Keywords Static Analysis, Transformations, Optimization, Compilers, Loop Invariants, Complexity, Quasi-Invariants

1. Introduction
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 quasi-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 a proven optimization on a WHILE language by Lars Kristiansen [8], we provide a refinement 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 [3][4], type [2][6] and Data Flow [7][9][11][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 [11], or “mwp-polynomials” [9] 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 regarding to represent 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 on 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]

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

1. Add s to LI;

end

end

End

Algorithm 1: Basic invariants detection

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.
2. Rethinking the 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 size-change graph \([11]\).

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) \(X \ ::= \ X_1 | X_2 | X_3 | \ldots | X_n\)

(Expression) \(exp \ ::= \ X | op(exp, \ldots, exp)\)

(Command) \(com \ ::= \ Xexp | com; com | \text{skip} | \text{while} \ exp \ do \ com \ od | \text{if} \ exp \ then \ com \ fi | \text{use}(X_1, \ldots, X_n)\)

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.

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 type of arrow represents different types of dependencies.

\begin{align*}
C := & \ x = x + 1; \quad \text{dependence} \rightarrow x \\
y = y; \quad \text{propagation} \rightarrow y \\
z = 0; \quad \text{reinitialization} \rightarrow z
\end{align*}

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 from \(C\) since the value of \(y\) is still propagated.

More formally, a DFG of a command \(C\) is a triple \((V, R_{dep}, R_{prop})\) 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 definition in terms of matrices. While less intuitive, this formal definition allows for more natural definitions, 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.

\begin{align*}
\oplus & \ | \ 0 \ 1 \\
0 & \ | \ 0 \ 1 \\
1 & \ | \ 1 \ 1 \\
\otimes & \ | \ 0 \ 1 \\
0 & \ | \ 0 \ 0 \\
1 & \ | \ 0 \ 1
\end{align*}

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

\begin{align*}
\text{DEFINITION 1.} & \quad \text{A Data Flow Graph for a command } C \text{ is a } n \times n \text{ matrix over the semi-ring } \{0, 1\} \text{ where } n \text{ is the number of variables involved in } C. \\
& \quad \text{We write } M(C) \text{ the DFG of } C. \text{ At line } i, \text{ column } j, \text{ we have a } \emptyset \text{ if the output value of the } j\text{th variable does not depend on the input value of the } i\text{th}; \text{ a } 0 \text{ in case of propagation (unmodified variable); and a } 1 \text{ for any other kind of dependence.}
\end{align*}

\begin{align*}
\text{DEFINITION 2.} & \quad \text{Let } C \text{ be a command. We define } \text{In}(C) \text{ (resp. } \text{Out}(C)) \text{ as the set of variables used (resp. modified) by } C.
\end{align*}

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.

\begin{align*}
2.2.1 & \quad \text{Composition and Multipath} \\
& \quad \text{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.}
\end{align*}

\begin{align*}
\text{DEFINITION 3.} & \quad \text{Let } C \text{ be a sequence of commands } [C_1; C_2; \ldots; C_n]. \text{ Then } M(C) \text{ is defined as the matrix product } M(C_1)M(C_2) \ldots M(C_n).
\end{align*}

Following the usual product of matrices, the product of two matrices \(A, B\) is defined here as the matrix \(C\) with coefficients:

\[C_{i,j} = \sum_{k=1}^{n} A_{i,k} \otimes B_{k,j}\]

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

\begin{align*}
\text{Figure 4: DFG of Composition.} \\
& \quad \text{Here } C_1 := [w := w + x; z := y + 2]; \text{ and } C_2 := [x := y; z := z + 2];
\end{align*}

2.2.2 Condition
We now explain how to compute the DFG of a command \(C := \text{if } E \text{ then } C_1; \text{ from the DFG of the command } C_1.

Firstly, we notice that in \(C\), all modified variables in \(C_1\), i.e. in \(\text{Out}(C_1)\), will depend on the variables used in \(E\). Let us denote by \(M(C)\oplus(E')\), the corresponding DFG, i.e. the matrix \(M(C) \oplus (E')\).
where $E$ (resp. $O$) is the vector representing variables in $\text{Var}(E)$ (resp. in $\text{Out}(C)$), and $\cdot^t$ denotes the transpose.

Secondly, we need to take into account that the command $C_1$ may be skipped. In that case, the overall command $C$ should act as an empty command, i.e. be represented by the identity matrix $\text{Id}$ (diagonal elements are equal to 0, all other are equal to 0).

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 if $E$ then $C_1$: Then $M(C) = M(C_1)^{[E]} \oplus \text{Id}$.**

Figure 5: DFG of Conditional.
Here $E := z \geq 0$ and $C_1 := \{w = w + x; z = y + 2; y = 0\}$.

### 2.2.3 While Loop

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

When considering iterations of $C_1$, the first occurrence of $C_1$ will influence the second one and so on. Computing the DFG of $C_1^n$, the $n$-th iteration of $C_1$, is just computing the power of the corresponding matrix, i.e. $M(C_1^n) = M(C_1)^n$. 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 $C_1$ corresponding to “iterating $C_1$ a finite (but arbitrary) number of times”:

$$M(C_1) = \lim_{k \to \infty} \sum_{i=1}^{k} M(C_1)^i$$

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

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

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

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

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

2 I.e. the vector with a coefficient equal to 1 for the variables in $\text{Var}(E)$, and 0 for all others variables.
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. We’ve seen that the transformation is possible from and to a C syntax, now we are dealing with a Single Static Assignment form: the LLVM Intermediate Representation.

3.1 Preliminaries

First, we want to visit all loops using a bottom-up strategy (the inner loop first). Then, as for the 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 optimized, 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.

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 in front of 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.

This algorithm is fast because it 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 \((\text{canSinkOrHoist, isGuaranteedToExecute etc...})\) to figure out if an instruction is movable or not. These methods provide the anchors instructions for the current loop.

3.3 Peeling loop idea

The transformation will consists in creating as many preheaders basic blocks before the loop as needed to remove all 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 \( \infty \).

4. Conclusion and Future work

Developers expect that compilers provide certain more or less “obvious” optimizations. When peeling is possible, that often means:
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\footnote{https://github.com/ThomasRuby/lqicm}. 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 [8] that initiated the present work. Jean-Yves Moyen is supported by the European Commission’s Marie Skłodowska-Curie Individual Fellowship (H2020-MSCA-IF-2014) 655222 - Walgo; Thomas Rubiano is supported by the ANR project “Elica” ANR-14-CES-0005; Thomas Seiller is supported by the European Commission’s Marie Skłodowska-Curie Individual Fellowship (H2020-MSCA-IF-2014) 659920 - ReACT.

References


Figure 11: Invariance Degree

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure11.png}
\caption{Invariance Degree}
\end{figure}

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 -ftime=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 computes n times the inner loop so the time computation is increasing quadratically. For the example shown in Figure 9, our pass computes the degrees shown in Figure 11.

To each instruction printed corresponds an invariance degree. The assignment instructions are listed by loops, the inner loop (starting with \texttt{while\_cond}) and the outer loop (starting with \texttt{while\_cond}). The inner loop has its own invariance degree equal to 1 (line 9). -1 means that the instruction or chunk was initialised to 0. Remarque that we do consider the phi initialization instructions of an inner loop. Here \%fact and \%i.1 are reinitialized in the inner loop condition block. So phi instructions are analysed

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure9.png}
\caption{Hoisting inner loop}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure10.png}
\caption{LLVM Intermediate Representation}
\end{figure}