Annales Univ. Sci. Budapest., Sect. Comp. 36 (2012) 179–199

Superoptimization in LLVM

Dávid Juhász and Tamás Kozsik

(Budapest, Hungary)

Communicated by Zoltán Horváth

(Received December 20, 2011; revised February 10, 2012;
accepted February 24, 2012)

Abstract. Superoptimization is a known technique to integrate the analyses and transformations of a number of separate optimizations in order to obtain an optimization that is more expressive than the sequential and iterative application of the original optimizations. This paper describes the elaboration of this technique within the Low Level Virtual Machine (LLVM) Compiler Infrastructure. A framework supporting the integration of modular optimizations into superoptimization is presented. Some LLVM-specific implementation considerations are also discussed. Finally, a brief introduction to the use of the framework is provided.

1. Introduction

Compilation is more than merely translating higher level programming languages into machine code. All of the practically used compilers are optimizing compilers, which means that besides generating native code the compiler transforms programs in different ways in order to make the generated code more efficient – either run faster or consume less memory.

Key words and phrases: Compiler, optimization, dataflow analysis, superoptimization.
2010 Mathematics Subject Classification: 68N20.
1998 CR Categories and Descriptors: D.3.4.
The research is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TÁMOP 4.2.1./B-09/1/KMR-2010-0003).
https://doi.org/10.71352/ac.36.179

Typically, compilers apply optimizations one after the other, which leads to an iterative optimization technique. Iteratively applied optimizations can observe changes performed by preceding transformations. However, this simple technique is known to be ineffective in some cases, namely when optimizations would benefit from *sharing information* about the program to be optimized. To address this problem, separate optimizations can be combined into more complex optimizations. Such a more complex optimization is sometimes called a superoptimization. A case study on handcrafted superoptimizations is given in [2].

Superoptimization is an improvement to iterative optimization in respect of the efficiency of the generated code. However, superoptimizations are large and monolithic, and so their complexity makes them unusable in practice. Can we combine the modularity and maintainability of iterative optimization and the effectiveness of superoptimization? A framework to compose optimizations and to share information among them can make this possible. The developers of the Vortex compiler [3] laid down the foundations for such a technique. In [8] a method to combine individual optimizations into a modularly built superoptimization is described, and the semantics of optimization of one instruction is formally given. Although according to the cited papers the technique was utilized within the Vortex compiler, the exact algorithm to solve the combined analysis for a whole program has not been published by the developers.

The main contributions of this paper are:

- an algorithm to optimize a whole program using the above method, and
- the implementation of a superoptimization framework for LLVM (available for downloading at `http://kp.elte.hu/superoptimization`).

The Low Level Virtual Machine (LLVM) Compiler Infrastructure [6] "is a collection of modular and reusable compiler and toolchain technologies". It supports, officially or through external projects, the static and dynamic compilation of many programming languages. LLVM provides a static single assignment (SSA) form based virtual instruction set (LLVM intermediate representation [16]), as well as a collection of core libraries. The aim of this compiler infrastructure is to facilitate the development of compilers that use LLVM as an optimizer and code generator.

The rest of the paper is structured as follows. Section 2 illustrates how superoptimization is more effective than iterative optimization. The two approaches are compared using a small example. Section 3 summarizes a superoptimization framework. Section 4 explains how the superoptimization framework can be implemented within the LLVM infrastructure. Section 5 gives a brief introduction on the use of the developed LLVM superoptimization framework. Section 6 discusses related work, and, finally, Section 7 concludes the paper.

2. Motivation

Typically, compilers perform optimizations one after the other in an iterative approach. In LLVM, for instance, optimizations are introduced in a modular way; an optimization manager is responsible for running the selected optimizations in some order, and for detecting whether an optimization has applied any transformations – hence, in the case of changes it is possible to re-run the (whole or just some sub-) sequence of optimizations.

The possible inefficiency of programs which are optimized by the iterative technique is demonstrated in this section on a small program snippet. Thereafter a trivial solution of information exchange between optimizations, resulting in a superoptimization, is shown. The lack of maintainability of this solution is pointed out as well.

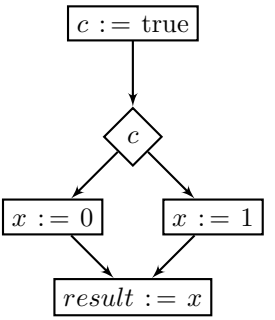
Before presenting the iterative optimization, the concept of *Control Flow Graph*s (*CFG*s) described in [10] is required, because we give the examples using this notation. In *Control Flow Graph*s, nodes represent instructions and directed edges represent possible execution paths between them. Diamond nodes are conditional branches, and rectangle nodes are instructions from which the control can go further only in one way. The control enters into a *Control Flow Graph* through its only instruction with no predecessors.

2.1. Iterative optimization

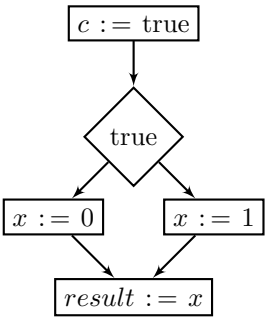
Iterative optimization is illustrated by a small example in Figure 1. We start with a simple conditional branch (Figure 1a). Note that the conditional construct can be eliminated and the value to be stored in the variable *result* can be determined in static time (Figure 1d). The application of two well-known optimizations, constant propagation and reachability analysis, can achieve this goal.

Constant propagation is an optimization that substitutes occurrences of variables with their values, if the value of such an occurrence is known in compile time. In SSA all occurrences of a given variable can be substituted with the (single) value, if that value is statically known. For instance, the transition from Figure 1a to 1b is the propagation of value "true" into the occurrences of variable *c*. Reachability analysis can be applied to find dead code, which can be eliminated. The transition from Figure 1b to 1c illustrates how this works: the "else" branch of the conditional has been found needless and so has been eliminated.

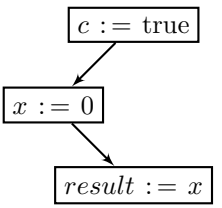
It is obvious that reachability analysis cannot yet transform the code in Figure 1a because of the variable *c* appearing in the condition. However, a preceding constant propagation can substitute the variable with the stored



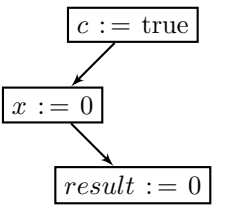
(a) Original *CFG*



(b) Propagate constants



(c) Analyse reachability



(d) Propagate constants again

Figure 1: Iterative optimization of a program snippet

value "true". Having performed constant propagation, reachability analysis can proceed. As a result, the value of x becomes unambiguous, hence another pass of constant propagation can modify the last assignment and output the code in Figure 1d. The iterative execution of optimizations is effective in the example given in Figure 1 with performing one reachability analysis between two constant propagations.

The example shows that dependencies may exist among optimizations, i.e. the result of an optimization is dependent on that of another. Moreover, there may be mutual dependencies between constant propagation and reachability analysis: one execution of constant propagation reveals new opportunities for reachability analysis and vice versa. The mutual dependencies of optimizations could be resolved by iterative execution for the *CFG* of Figure 1, but there are graphs on which the iterative optimization is unavoidably ineffective.

2.2. Superoptimization

The next example is the *CFG* of the Euclidean algorithm for non-negative integers, depicted in Figure 2. In the case when the value of parameter b is 0, the condition of the loop will be false right at the first time, and the result of function "gcd" will be the value of parameter a. Now consider the program

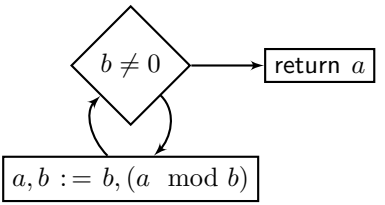
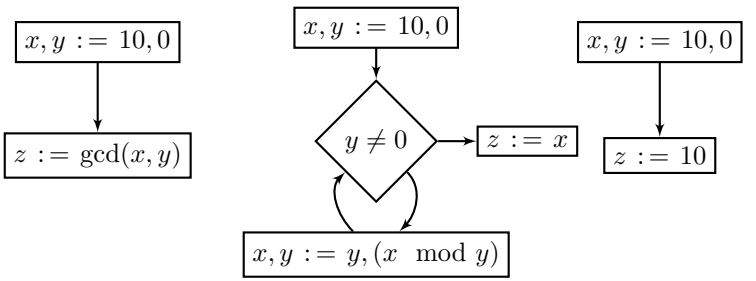


Figure 2: *CFG* of function $\gcd(a, b)$



(a) Original program (b) After inlining (c) Desirable form

Figure 3: Simple example with Euclidean algorithm

snippet in Figure 3a. Inlining function "gcd" produces the *CFG* in Figure 3b. The conditional will be false when the execution first reaches it. Thus z will store the initial value of x in the end of the execution. The optimized form of the original program snippet is shown in Figure 3c.

Note that the iterative optimization using constant propagation and reachability analysis is unable to transform the *CFG* in Figure 3b. The reachability analysis cannot eliminate the conditional construct, because its condition contains a variable; furthermore, constant propagation can substitute neither x (by ten) nor y (by zero) because of the assignment to x and y inside the loop. However, it is apparent that a combination of these two optimizations could achieve the desired state shown in Figure 3c.

The ineffectiveness of the iterative optimization here is due to the lack of proper communication between the individual optimizations. They could only exchange information via transformations, and they cannot share knowledge about the current state of the code to optimize. One has to define a new individual optimization which utilizes the combined knowledge of the dependent optimizations to ensure the combined effect of the optimizations. The complex optimizations, which combine the logic of some simpler optimizations, are called superoptimizations. A superoptimization combining constant propaga-

tion and reachability analysis can convert the *CFG* in Figure 3b into the one in Figure 3c, as it can neglect the back edge inside the loop, because it is aware of the value of the condition when the execution hits it for the first time, and therefore it can eliminate the "then" branch from the code.

The watchful Reader might notice that in this very example, adding one more optimization, namely Loop Unrolling, to the set of applied optimizations, the iterative approach can also transform the *CFG* in Figure 3b into the one in Figure 3c. Still, it is clear that a superoptimization constructed from a set of optimizations is more powerful than the iterative application of the same set of optimizations.

Unfortunately, a monolithic superoptimization, which is constructed manually, is rather unmaintainable compared to the simple component optimizations. This is the reason why superoptimizations are practically unusable without a technique to construct them modularly.

One might think that an alternative to superoptimizations could be a technology to construct optimizations from transformations which use common analyses. This technology would be supported by current practice, because the analyses and transformations are executed separately in many compilers. However, this approach only moves the complexity problem to another level, since in this case the analyses have to be aware of all the possible transformations to compute the most precise information for the current program. For this reason a better solution should be found – such a solution is proposed in the forthcoming sections.

3. Superoptimization framework

This section describes a method to combine individual optimizations automatically so that the result of execution is equivalent to the superoptimization determined by the separate optimizations. The concept of Integrated Analyses and that of Composed Analysis was suggested in [8]. Integrated Analyses are a generalization of Data Flow Analyses. Composed Analysis is obtained by combining Integrated Analyses into a single analysis. The soundness and termination of Composed Analysis was also proven in [8].

A Data Flow Analysis collects information about a program. This information can be used to direct transformations applied on that program. To compute the necessary information for every single instruction of a program, a system of data flow equations must be solved. Data Flow Analyses is a well studied topic – there are several methods to solve data flow equations. Summaries of Data Flow Analyses and solver algorithms can be found e.g. in [10, 11].

3.1. Integrated Analyses

The main difference between Data Flow Analyses and their generalization, Integrated Analyses, is that the former only provides some information about the individual instructions of a program, while the latter may provide a replacement graph for an instruction as well. Integrated Analyses let us express a transformation together with the necessary analyses as a single entity.

Individual Integrated Analyses are combined into one special Integrated Analysis by the Composed Analysis, which recursively optimizes the replacements provided by component analyses, and returns the most efficient replacement according to an arbitrarily chosen metrics. The information sharing between analyses is achieved by merging all the information collected during both the analysis of the *CFG* and the recursive optimization of replacements.

According to the original definition given in [8], an Integrated Analysis results in either some information *or* a replacement, but never both, for an instruction. In our framework, however, an Integrated Analysis is permitted to return some information *and* a replacement at the same time. This minor generalization allows us to express certain optimizations more compactly and efficiently. In the case of the Inliner optimization, when inlining the body of a function at a call site, the original definition of Integrated Analyses does not allow the propagation of any information to successor instructions, it only allows a replacement graph to be returned. However, information about the inlined function is very useful to be propagated in order to avoid re-analysing the function at its next call site. Due to the generalization of the original definition, Integrated Analyses in our framework support this beneficial behavior.

At first glance it might be strange that with Integrated Analyses the analysis of an instruction is allowed to yield a replacement and no information at all. If no information is returned from the analysis of an instruction, what can be used as input for the analysis of the successor instruction(s)? The answer is that the Composed Analysis collects information during the optimization of the replacement graph, and this information will be propagated as input to the analysis of every successor instruction.

3.2. Composed Analysis

A Composed Analysis, as defined originally in [8], combines Integrated Analyses only for analysing a single instruction. In order to implement a superoptimization framework, it was necessary to work out an algorithm to optimize a whole program utilizing Composed Analysis. This algorithm is one of the contributions of this paper.

Algorithm 1 to optimize a program with Composed Analysis

```
put entry point(s) into working queue Q
while Q is not empty do
    get next instruction from Q into i
    if i is virtually dead then
        process i as a virtually dead instruction
        continue with next instruction
    else if i is a loop header then
        if information about i's loop is in fixed point then
            process 1-step-outside exiting edges
            continue with next instruction
        else
            prepare for re-analysing i's loop
        end if
    else
        update i's ingoing information
    end if
    analyse i with the computed ingoing information
    store replacement for i
    propagate outgoing information to i's successors using Algorithm 2
end while
perform stored replacements
```

The solver algorithm elaborated here traverses a *CFG* either in forward or backward direction, depending on the traversal order of the analysis to be performed. As Integrated Analyses are a generalization of Data Flow Analyses, the algorithm has been developed from a well-known Data Flow Algorithm. The applied modifications make it possible to handle replacements and related problems. However, unlike the algorithm which can be found in [11] Section 6.3, our algorithm decides dynamically in which order the instructions are to be visited. The decision is made according to the availability of information on the edges of the *Control Flow Graph*. The principles of our algorithm are also similar to those of the structural analysis algorithm given in [10] Section 8.7.

Algorithm 1 uses a working queue to store instructions to be analysed, i.e. an instruction can only be put in the queue when all its predecessors are already analysed. Initially, only instructions with no predecessors are inserted in the queue. Until the queue becomes empty, the algorithm processes the next instruction from the queue, which means that the order the instructions are to be visited is decided dynamically, while processing instructions. However, a preliminary traversal is needed by a typical efficient Data Flow Algorithm, such as the one described in [11], in order to make up a plan about the order in which instructions are to be visited. Delaying the analysis of an instruction until its predecessors have been analysed, and analysing instructions insistently in an order that matches the structure of the program leads to a very efficient

traversal algorithm – in the sense of how many times instructions must be visited before reaching a fixed point of information. This idea is also utilized in the structural analysis algorithm which can be found in [10].

A crucial part of analysing a program is the act of deciding whether the analysis has reached a fixed point during the collection of information about a loop construct. As usual with Data Flow Analyses, there are three sorts of *CFG* edges to consider when analysing loops: entering into a loop, jumping back to the header of a loop and exiting from a loop. Also, a given *CFG* edge may belong to more than one category at the same time. In order to ensure that the information about a loop construct is in fixed point, loop headers are also to be considered carefully. When the traversal reaches a loop of which the source instruction of every entering edge is already analysed, the header of the loop has to be inserted in the queue (lines 6–7 in Algorithm 2). When the traversal reaches an instruction with exiting edges, information to each successor outside of the loop is to be held up until information about the outermost enclosing loop the edge exits from is proven to be in fixed point. In Algorithm 2, outgoing information is always stored (line 3), but checking of readiness is skipped if the particular edge is exiting from the loop (lines 4–5). When the traversal reaches a loop header from inside of its loop by meeting condition of line 7 in Algorithm 1, there are two possible cases. In the first case (lines 8–10) information is in fixed point, and the traversal is free to propagate information to the outside of the loop, so checking readiness of instructions reached through an exiting edge is to be performed here similarly to the way in Algorithm 2. After processing all the exiting edges, traversing can continue normally. In the second case (lines 11–12), when information is not yet in fixed point, the algorithm must re-traverse and re-analyse the loop.

Algorithm 2 to propagate information to successors of an instruction i

```
1: for all successors of i do
2:    let the successor be called j and the edge leading to j be called e
3:    store outgoing information for j
4:    if e is an exiting edge then
5:       continue with next successor
6:    else if e is an entering edge and j's loop is ready then
7:       put the header of j's loop into the queue
8:    else if e is not an entering edge and j is ready then
9:       put j into the queue
10:   end if
11: end for
```

Our contribution to the Data Flow Algorithms mentioned before is the handling of replacements. Replacements for each instruction are stored (line 18 in Algorithm 1), when the Composed Analysis returns them. Eventually, after

analysing the whole program, the stored replacements are performed (line 21). However, the handling of replacements really complicates things. On the one hand, a replacement provided by the analysis can only be performed when the information of the outermost enclosing loop is in fixed point. That is why replacements are stored, and not performed immediately. On the other hand, not yet performed replacements must be taken into account when checking whether the analysis has reached a fixed point. This makes it necessary to introduce the concept of the *virtual state of the CFG*, and *virtually dead instructions*.

The virtual state of the *CFG* consists of the edges which would be present after performing the currently stored replacements. Replacements can delete existing edges from the *CFG* – although they are not allowed to introduce new ones. The analysis does not provide outgoing information for the edges which the replacement deletes. Storing outgoing information (line 3 in Algorithm 2) in such a case means that no information is actually stored, only a flag is set. Therefore, the computation of ingoing information for an instruction i can be based on merging information about only those instructions which would be the predecessors of i after performing the replacements.

While maintaining the virtual state of the *CFG*, an instruction may be left behind with only *virtually removed* ingoing edges. Since such instructions cannot be reached by executing the program represented by the virtual state of the *CFG*, they are virtually dead instructions. There is no ingoing information for such an instruction, hence those instructions cannot be analysed. They have to be handled specially: the only information to be propagated about those in the virtual state of the CFG is that their outgoing edges are virtually removed. Algorithm 1 checks for virtually dead instructions and processes them between lines 4 and 6.

Further details about this algorithm together with some examples can be found in the technical report [5]. The presentation is general enough to enable the implementation of the algorithm in different environments. However, it is worth to take the specialities of those environments into account in order to arrive at an efficient implementation. A few of our implementation considerations for the Low Level Virtual Machine are discussed in the forthcoming section.

4. Superoptimizing in LLVM

The Low Level Virtual Machine (LLVM) Compiler Infrastructure is a modularly built compilation framework. A new compiler can be easily constructed by composing the existing modules of LLVM. Moreover, the functionalities of such a compiler can be extended by implementing new modules in the framework.

Due to this extensible, modular design, developers of high-level programming languages often use LLVM for building compilers.

A short summary of the optimization module of LLVM is given in Section 4.1. Section 4.2 discusses some characteristics of LLVM which our framework is affected by. In Section 4.3 some implementation considerations of the framework are briefly revealed.

4.1. LLVM support for optimization

LLVM offers a simple way to implement separate analyses and transformations as optimization passes. The passes are activated by an optimization manager which applies the scheduled passes in an efficient order. A transformation can gain information from analyses if those analyses have been applied previously. Since the optimization manager has to determine the activation order of the passes, it requires each pass to declare

- which other passes compute information needed by the current pass, and
- which are the passes that need not be re-run after this pass (i.e. the result of which passes are not affected by the execution of the current pass).

According to this information, the manager is able to share the results of analyses among transformations, and keep the number of analysis executions at the minimum. A transformation can let the manager know whether it performed any modifications. Based on this information, the execution of the optimization manager can be organized into iteration.

4.2. LLVM and the superoptimization framework

LLVM was designed to ease the implementation of optimizations and to that end many known compiler techniques have been utilized. One of those techniques is the application of the *Static Single Assignment Form* (*SSA*), which is supposed to simplify Data Flow Analyses. *SSA* means that every variable can occur at the right hand side of at most one assignment in the source code. To select a value from different execution paths the so called ϕ function is used. A simple example is given in Figure 4. In order to fully exploit the benefits of the *SSA* form, one has to work with DU-chains. Both DU-chains and UD-chains are available in LLVM. DU-chains let us access all uses of a definition at the place of the definition, and UD-chains let us access the definition of a variable wherever the variable is used.

Unfortunately, the use of DU-chains in transformations causes non-local modifications of the analyzed program, and this is forbidden in superoptimization [8]. Thus, the Integrated Analyses are unable to profit from DU-chains and, consequently, the *SSA* form is also useless for our framework. In fact, the

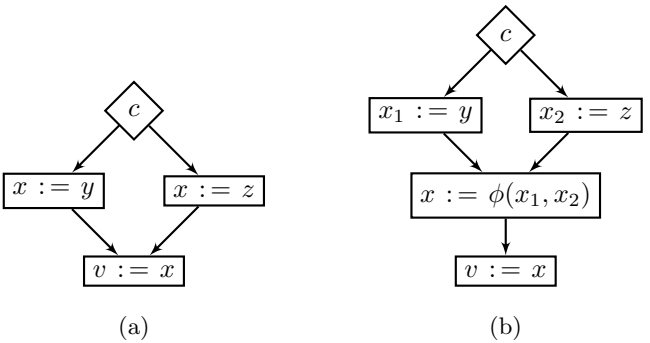


Figure 4: The conventional (a) and *SSA* (b) form of a simple branch

SSA form of LLVM programs forced us to handle the ϕ instructions in a rather special way. This is because replacements inside loops can only be performed if the information is in fixed point, and to check whether this is the case the virtual state of the *CFG* must be considered; therefore, virtual ϕ instructions must be used. For instance, Figure 5 depicts the *SSA* form of the inlined "gcd" function given in Figure 3b. In order to realize the optimized form (which can be seen in Figure 3c), the Integrated Analyses have to operate with modified ϕ instructions, otherwise the unknown values of x_1 and y_1 interfere with the elimination of the conditional branch. So when the second assignment is being analysed for the first time, the Integrated Analyses have to be given a modified assignment, namely: $x, y := \phi(x_0), \phi(y_0)$. Then the suggested replacement, which eventually is the elimination of the branch, causes neglecting the back edge and reaching a fixed point of information, and hence the replacement can be performed.

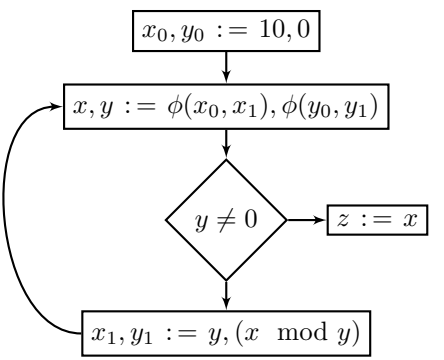


Figure 5: *SSA* form of inlined "gcd" in Figure 3b

In contrast to the *SSA* form, there are advantageous features as well in LLVM with respect to the superoptimization framework. To mention but a few, the structure of LLVM programs makes the control flow explicit. Moreover, iterators are available to access predecessor and successor instructions according to the *CFG* of the program. Finally, the existing `LoopInfo` analysis and `SimplifyCFGPass` transformation have proven to be very useful. The former can identify loops and various types of edges in the program, and the latter can eliminate possible block chains left behind by the framework.

4.3. Further implementation considerations

In this section some particularities of our approach to implement a superoptimization framework for LLVM are presented. These can be regarded as "the lessons learned" from this research, and are useful for understanding the details of the framework.

LLVM programs are explicitly separated into "basic blocks". A basic block is a sequence of instructions without a branch. The control can enter into a basic block only through its first instruction. Therefore, it is enough for the analyses to store the ingoing information of only the first instruction of each basic block – for all other instructions it can be computed easily by demand.

It is very realistic to assume that the functions to merge information arriving at a *CFG* node from different edges are associative and commutative operations, thus the ingoing information of a basic block can be merged incrementally. Merging is performed immediately when information is being propagated from a predecessor, so there is no need to store outgoing information permanently. However, information which has been propagated through an exiting edge of a loop is held up until it is proven to be in fixed point.

As the merging of ingoing information is made incrementally, so does the checking of fixed point condition for loops. For each loop the framework maintains whether information belonging to that loop is in fixed point – maintenance is due when ingoing information is merged on back edges.

Typical analyses work in an iterative manner: they propagate information through back edges and seek for a fixed point of information about loop constructs. There are, however, some special analyses which visit each instruction exactly once. In contrast to the previously mentioned iterative gathering of information, such analyses maintain a so called "non-iterative information", which is never propagated through back edges and is always treated as being in fixed point. For example, an analysis that counts instructions meeting some condition is a typical analysis with non-iterative information. On the other hand, there are analyses which collect information iteratively, and gain runtime benefit from being aware of propagating through back edges. For example, an interval analysis, applied in the case of incrementing a variable inside

a loop, can reach a fixed point very fast, if it sets the upper bound of the variable to the maximal representable value in one step, rather than incrementing the boundary value step-by-step. In order to support these two use cases, the framework supports abstractions for both the iterative and non-iterative gathering of analysis information. In the iterative case a special merge function is provided for back edges – as a default implementation it merely calls the regular merge function.

The result of the analysis of an instruction is either some information or a replacement for that instruction or both. Replacements are performed as soon as possible. This means that replacements for instructions outside of loops are performed at the end of the containing basic block, and other replacements are performed only when the information of their outermost enclosing loop is proven to be in fixed point.

The final note about replacements is that their introduction required a slight modification to the code of the LLVM infrastructure. One of the C++ classes that constitute the implementation of LLVM, `BasicBlock`, had to be made suitable for subclassing by introducing a protected constructor and turning one of its operations, `eraseFromParent`, virtual. The reason for this was to enable the creation of an immutable proxy class as a subclass of `BasicBlock`. The proxy class is used to represent successors of instructions within replacements. This is necessary because replacements are represented with `Function` objects (same as functions are represented in LLVM), which has a positive and a negative consequence. The positive one is that recursive optimization of replacements can be implemented in a straightforward way. The negative consequence is that since basic blocks of a function can only refer to each other, the introduction of proxies to implement references to successors in other functions was unavoidable.

5. Using the LLVM superoptimization framework

In this section a brief introduction to the use of the framework is provided. Section 5.1 discusses how Integrated Analyses can be implemented. Section 5.2 reveals two ways an LLVM program can be optimized with the framework. Finally, Section 5.3 revisits the motivational example in LLVM.

5.1. Developing Integrated Analyses

The superoptimization framework, as well as LLVM itself, is implemented in C++. To implement an Integrated Analysis, one has to design a C++ class to represent the information which is used by the Integrated Analysis, and the class implementing the analysis as well. Integrated Analyses are derived from the abstract class `IntegratedAnalysis`, which introduces two abstract

operations: `initInfo` and `process`. The former is a constant function which should return the initial information of the optimization implemented by the specific Integrated Analysis. The `process` operation should define the analysis of a single instruction.

As mentioned earlier, there are two abstractions for analysis information: `AnalysisInfo` and `IterativeAnalysisInfo`, where the latter is a specialization of the former, and supports information propagation on back edges. Every class representing analysis information has to be capable of making a copy of itself (`copy` operation), and merge the same type of information into itself (`merge` operation). A class representing iterative analysis information has to provide the `asGeneralAs` operation, and it may optionally provide the `generalizingMerge` operation. The obligatory `asGeneralAs` is utilized to check if the information is in a fixed point. The optional `generalizingMerge` is used by the framework to merge information through back edges; a default implementation is provided for this operation, which simply calls the `merge` function.

The result type of the `process` function is the `AnalysisResult` class. Its specialization for information propagation is the `ContinueResult` class, which has two further subclasses: `TerminatorContinueResult` maps information to be propagated for multiple successors of an instruction, while `InBlockContinueResult` holds only one information instance for propagating to the next instruction of a sequence. Another subclass of `AnalysisResult` is `ReplaceResult`, which supports the creation of replacements. A `ReplaceResult` object makes the proxy objects (see the end of Section 4.3) to the successors accessible, and creates the basic blocks of a replacement on demand. Finally, the `CombinedResult` class, which is again a subclass of `AnalysisResult`, supports *both* information propagation *and* replacements at the same time. This class aggregates a `ContinueResult` instance and a `ReplaceResult` instance.

So far five Integrated Analyses have been implemented in the LLVM superoptimization framework: four forward analyses (Constant Propagation, Common Subexpression Elimination, Inliner and Reachability Analysis) and one backward analysis (Dead Assignment Elimination). It turned out that the implementation of an optimization as an Integrated Analysis needs 1.5 times more lines of code than the optimization pass written to standard LLVM. This is the overhead one has to pay in order to define an optimization pass that can be organized automatically into a superoptimization with the help of our framework.

5.2. Activation of the framework

The LLVM superoptimization framework with the implemented five Integrated Analyses has been implemented as a function level optimization pass, as it is written in [17]. Therefore, it can be used via the `opt` command line tool with option flag `-so`. The prepared optimization applies the already implemented four forward analyses and then the one backward analysis. Because the forward analyses are not dependent on Dead Assignment Elimination, there is no need to iterate the forward and backward analyses.

Activating the framework in source code is also quite simple. There are two classes which handle the Integrated Analyses: `ForwardAnalysis` for the forward and `BackwardAnalysis` for the backward ones. An Integrated Analysis can be added to an optimization with the operation `addAnalysis`. An LLVM function can be optimized by the actual added analyses with operation `traverse`, which returns a logical value indicating if any replacements were performed during traversal. An example of the use of Constant Propagation and Reachability Analysis to optimize an LLVM function is given in Listing 1.

Listing 1 Optimize a function with two Integrated Analyses.

```
bool runOnFunction(Function &F) {
    ForwardAnalysis FA;
    FA.addAnalysis(createConstantPropagator());
    FA.addAnalysis(createReachabilityAnalysis());
    return FA.traverse(F);
}
```

5.3. The motivational example in LLVM

Now the example from Section 2 is revisited in LLVM. The LLVM function of the Euclidean Algorithm for non-negative integers with *CFG* in Figure 2 can be seen in Listing 2. The LLVM code of the problematic example from Figure 3a is in Listing 3, and the state of the program after inlining "gcd" is shown in Listing 4.

LLVM has a Constant Propagation pass (`constantprop`) and a pass with an extended Reachability Analysis (`simplifycfg`). These two optimizations are unable to eliminate the unnecessary branch from the function in Listing 4. The Constant Propagation pass substitutes `%x` and `%y` in the `phi` instructions, but does nothing more, and those substitutions do not help the other pass.

The superoptimization framework using Constant Propagation and Reachability Analysis eliminates the conditional branch and the block, which becomes

unnecessary. Moreover, an optimization pass using also Dead Assignment Elimination produces the code in Listing 5, which consists of only one instruction.

Listing 2 LLVM code of function $\gcd(a, b)$.

```
define i32 @gcd(i32 %a, i32 %b) {
entry:
  br label %head

head:
  %a1 = phi i32 [%a, %entry], [%b1, %body]
  %b1 = phi i32 [%b, %entry], [%n, %body]
  %cond = icmp eq i32 %b1, 0
  br i1 %cond, label %exit, label %body

body:
  %n = srem i32 %a1, %b1
  br label %head

exit:
  ret i32 %a1
}
```

Listing 3 LLVM code of *CFG* in Figure 3a.

```
define i32 @gcd_of_10_and_0() {
entry:
  %x = add i32 10, 0
  %y = add i32 0, 0
  %z = call i32 @gcd(i32 %x, i32 %y)
  ret i32 %z
}
```

6. Related work

The topic of Data Flow Analyses has been researched for a long time, therefore good summaries are available, e.g. the related sections of [10, 11]. The Data Flow Algorithm which is the basis of our algorithm was used by I. Forgács [4], who modified it to solve interprocedural analyses in polynomial time. Another efficient solution for interprocedural Data Flow Analyses can be found in [13].

Listing 4 LLVM code of the inlined "gcd" example in Figure 3b.

```
define i32 @gcd_of_10_and_0() {
entry:
  %x = add i32 10, 0
  %y = add i32 0, 0
  br label %head

head:
  %a1 = phi i32 [%x, %entry], [%b1, %body]
  %b1 = phi i32 [%y, %entry], [%n, %body]
  %cond = icmp eq i32 %b1, 0
  br i1 %cond, label %exit, label %body

body:
  %n = srem i32 %a1, %b1
  br label %head

exit:
  ret i32 %a1
}
```

Listing 5 Result of the superoptimization.

```
define i32 @gcd_of_10_and_0() {
entry:
  ret i32 10
}
```

Incremental Data Flow Analysis, which means the selective re-analysis of a program after a modification, is also a well researched area. An algorithm for incremental analysis is provided in [14]. The methods to determine which parts of a program must be re-analyzed are known as impact analyses. A comparison of the major impact analysis methods is given in [15].

The resolution of the phase-ordering problem which occurs during iterative optimization is discussed in [2] through the combination of three analyses, however no general solution is revealed. Another possible solution for the phase-ordering problem is given in [12] where the iterative application of optimizations is kept, but the actual order the optimizations are applied in is determined dynamically with respect to the defined needs of the optimizations.

The basis of the superoptimization framework is described in the reports [1, 7] and the conference paper [8]. In those papers the critique of the iterative optimization technique and a method for combining individual optimizations automatically into a superoptimization are presented. According to the referenced papers, a superoptimization framework was developed for the Vortex compiler. A brief summary of this compiler is in [3].

The soundness of transformations performed by optimizations is usually proven by hand. A solution for automatic proving of correctness is given in [9] via a domain specific language the transformations have to be defined in. In that approach a set of proof obligations is generated from the definition, and the correctness of the transformation is decided by an automatic theorem prover. The correctness of a superoptimization created from correct transformations is investigated in [7, 8].

7. Conclusion

Optimization is a crucial step during compiling software and the practically used optimization technique is known to be ineffective in some cases. One way to address the problem is the creation of superoptimizations, but superoptimizations are proven to be too complex to create and maintain manually. A method was given in [8] to integrate special modular optimizations into a superoptimization.

In this paper the slightly modified version of that method was reviewed along with an algorithm which can utilize the method to optimize a whole program. Then some implementation considerations and an introduction to the use of the framework was provided, as we have implemented the framework for the Low Level Virtual Machine.

References

[1] **Chambers, C., J. Dean and D. Grove,** *Frameworks for Intra- and Interprocedural Dataflow Analysis*, Technical Report, University of Washington Computer Science & Engineering, 1996.

[2] **Click, C. and K.D. Cooper,** Combining analyses, combining optimizations, *ACM Transactions on Programming Languages and Systems*, **17(2)** (1995), 181–196, DOI 10.1145/201059.201061

[3] **Dean, J., G. DeFouw, D. Grove, V. Litvinov and C. Chambers,** Vortex: an optimizing compiler for object-oriented languages, in: *OOPSLA '96 Proceedings of the 11th ACM SIGPLAN conf. on Object-Oriented Programming, Systems, Languages, and Applications*, ACM New York, USA, 1996, 83–100, DOI 10.1145/236337.236344

[4] **Forgács, I.,** Double iterative framework for flow-sensitive interprocedural data flow analysis, *ACM Transactions on Software Engineering and Methodology*, **3**(1) (1994), 29–55, DOI 10.1145/174634.174635

[5] **Juhász, D. and T. Kozsik,** Superoptimization in LLVM, Technical Report, Dept. of Programming Languages and Compilers, Eötvös Loránd University, 2012.

[6] **Lattner, C. and V. Adve,** LLVM: a compilation framework for lifelong program analysis & transformation, in: *Proc. Int'l Symp. on Code Generation and Optimization*, ISBN 0-7695-2102-9, 2004, 75–86, DOI 10.1109/CGO.2004.1281665

[7] **Lerner, S., D. Grove and C. Chambers,** *Composing Dataflow Analyses and Transformations*, Technical Report, University of Washington Computer Science & Engineering, 2001.

[8] **Lerner, S., D. Grove and C. Chambers,** Composing dataflow analyses and transformations, in: *Proc. 29th ACM SIGPLAN-SIGACT symp. on Principles Of Programming Languages*, ACM New York, USA, 2002, 270–282, DOI 10.1145/503272.503298

[9] **Lerner, S., T. Millstein and C. Chambers,** Automatically proving the correctness of compiler optimizations, in: *Proc. ACM SIGPLAN 2003 conf. on Programming Language Design and Implementation*, ACM New York, USA, 2003, 220–231, DOI 10.1145/781131.781156

[10] **Muchnick, S.S.,** *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, San Francisco, USA, 1997. ISBN-13: 978-1558603202

[11] **Nielsen, F., H.R. Nielsen and C. Hankin,** *Principles of Program Analysis*, Springer-Verlag Berlin–Heidelberg, Corrected 2nd printing, 2005. ISBN 978-3-540-65410-0

[12] **Queva, M.,** *Phase-ordering in Optimizing Compilers*, Master's thesis, Supervised by C. Probst, IMM-Thesis-2007-71, Technical University of Denmark, 2007, `http://www2.imm.dtu.dk/pubdb/p.php?5406`

[13] **Reps, T., S. Horwitz and M. Sagiv,** Precise Interprocedural dataflow analysis via graph reachability, in: *Proc. 22nd ACM SIGPLAN-SIGACT symp. on Principles Of Programming Languages*, ACM New York, USA, 1995, 49–61, DOI 10.1145/199448.199462

[14] **Ryder, B.G. and M.C. Paull,** Incremental data-flow analysis algorithms, *ACM Transactions on Programming Languages and Systems*, **10(1)** (1988), 1–50, DOI 10.1145/42192.42193

[15] **Tóth, G., P. Hegedűs, Á. Beszédes, T. Gyimóthy and J. Jász,** Comparison of different impact analysis methods and programmer's opinion: an empirical study, in: *Proc. 8th Int'l Conf. on the Principles and Practice of Programming in Java*, ACM New York, USA, 2010, 109–118, DOI 10.1145/1852761.1852777

[16] *LLVM Language Reference Manual*, Last modified on 2011-10-13 18:04:49, Online, Accessed October 2011, `http://llvm.org/docs/LangRef.html`

[17] *Writing an LLVM Pass*, Last modified on 2011-10-11 02:03:52, Online, Accessed October 2011, `http://llvm.org/docs/WritingAnLLVMPass.html`

D. Juhász and T. Kozsik
Department of Programming Languages and Compilers
Eötvös Loránd University
Pázmány Péter sétány 1/C.
H-1117 Budapest
Hungary
`juhda@caesar.elte.hu`
`kto@elte.hu`