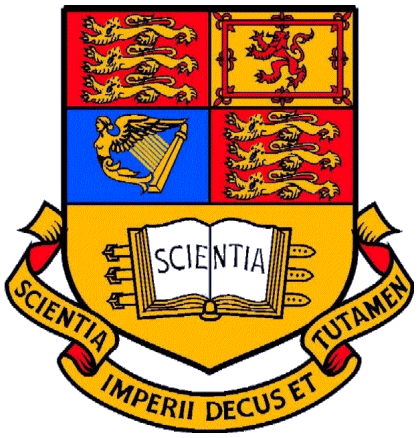
Imperial College London
Department of Computing



Generalizing loop-invariant code motion in a real-world compiler

Author:
Paul Colea

Supervisor:
Fabio Luporini
Co-supervisor:
Prof. Paul H. J. Kelly

MEng Computing Individual Project
June 2015

Abstract

Motivated by the perpetual goal of automatically generating efficient code from high-level programming abstractions, compiler optimization has developed into an area of intense research. Apart from general-purpose transformations which are applicable to all or most programs, many highly domain-specific optimizations have also been developed.

In this project, we extend such a domain-specific compiler optimization, initially described and implemented in the context of finite element analysis, to one that is suitable for arbitrary applications. Our optimization is a generalization of loop-invariant code motion, a technique which moves invariant statements out of program loops. The novelty of the transformation is due to its ability to avoid more redundant recomputation than normal code motion, at the cost of additional storage space.

This project provides a theoretical description of the above technique which is fit for general programs, together with an implementation in LLVM, one of the most successful open-source compiler frameworks. We introduce a simple heuristic-driven profitability model which manages to successfully safeguard against potential performance regressions, at the cost of missing some speedup opportunities. We evaluate the functional correctness of our implementation using the comprehensive LLVM test suite, passing all of its 497 whole program tests.

The results of our performance evaluation using the same set of tests reveal that generalized code motion is applicable to many programs, but that consistent performance gains depend on an accurate cost model. Applied unconditionally, our optimization introduces two major execution time improvements of over 30% in the test suite, but otherwise proves to be very detrimental. Coupled with the current cost model, all noticeable slowdowns are eliminated, but so is one of the previous speedups. We believe that with further refinement of the cost model, our technique could bring major improvements to a wide range of programs.

Acknowledgements

Firstly, I would like to wholeheartedly thank my supervisor, Fabio Luporini, for his proven enthusiasm and continuous feedback, and for always finding time for our lengthy, yet very pleasant discussions. I am thankful to Prof. Paul Kelly for steering the project in the right direction, for his helpful ideas and comments, and for the interest he has shown. An acknowledgement is due to the department's Computing Support Group, for being so helpful in providing me with the necessary resources for my experiments.

Lastly, I would like to express my sincerest gratitude towards my parents, Nicoleta and Horaţiu, and my grandparents, Maria and Enache, for shaping me into the person that I am today.

Contents

Chapter 1

Introduction

Compilers are arguably one of the fundamental pillars of computer science, allowing the field to thrive and reach the levels of complexity it enjoys today. Apart from mapping abstract programming languages to the concrete instruction set of a given architecture, modern compilers also employ a rich toolkit of optimizations which aim to improve the runtime behaviour of programs. Driven by the ever-increasing need for generating high-performance executable code, compiler optimization has evolved into a prominent area of research over the last four decades, spawning a plethora of efficient techniques.

In arithmetic-intensive programs, characteristic to fields such as scientific computing, a dominant fraction of the total running time is spent executing loops. For such applications, major improvement is thus achievable if the processing times and memory footprint of resource-intensive loops are reduced. The spectrum of compiler optimizations targeting loop constructs ranges from simple, universal procedures such as loop unrolling, to complex, specialized techniques such as loop blocking.

One of the most applicable and widely used loop optimizations is loop-invariant code motion (LICM). This transformation identifies expressions that are unnecessarily re-evaluated at successive loop iterations (*loop-invariant* expressions), and subsequently moves them to a program point where they execute fewer times. LICM is an attractive optimization because it almost invariably achieves a speedup in execution time, due to reducing the number of times certain statements are evaluated.

1.1 Motivation

Because of their generality, traditional compiler optimizations may achieve suboptimal results in application domains where programs adhere to a very particular, perhaps unusual structure. In such cases, specifically tailored transformations often produce superior code compared to even the most successful general-purpose compilers.

The core idea of this project stems from the previous work of Luporini et al. [27], who introduced such a toolkit of domain-specific optimizations in the context of the finite element method (see Section 4.2). The specialized compiler introduced by the authors, COFFEE, managed to outperform two of the most prominent general-purpose C compilers, through its use of domain-specific knowledge on the input programs.

In this project, we focus on one of the transformations introduced in COFFEE, which the authors refer to as *generalized loop-invariant code motion* (GLICM). GLICM pursues the same goals as loop-invariant code motion, but is able to identify and precompute more types of loop-invariant expressions. This optimization was developed in reaction to the limited amount of code motion performed by vendor compilers on finite element code, and single-handedly improved the execution time of certain programs by a factor of up to 2.5.

1.2 Project aim

Our foremost goal is to assess the usefulness and runtime impact of generalized loop-invariant code motion on typical, generic applications. Although GLICM was initially developed as a niche optimization for a specific field, we believe that the principles on which it rests can be suitably extended to a wider category of programs.

To achieve this, we set out to develop a general-purpose compiler optimization based on the operating principles of the generalized code motion procedure, as originally described by Luporini et al. Crucially, the project also aims to propose a safety model which ensures that our optimization is applied correctly and safely to arbitrary programs, without modifying their semantics or generating unwanted exceptions.

1.3 Main contributions

The main contributions of this project are the following:

- An implementation of generalized-loop invariant code motion as a stand-alone optimization pass in LLVM [20], a large open-source compiler framework used extensively across academia and industry (Chapter 5).
- An adaptation of the technique to fit the generality of arbitrary programs. In addition, the formulation of a set of safety requirements, rooted in compiler theory, necessary for correctly applying GLICM to arbitrary programs (Section 4.4).
- The development of a heuristic cost model which aims to apply generalized code motion only when the transformation is profitable (Section 5.4.5).
- An evaluation of our implementation's functional correctness and performance on a test suite consisting of hundreds of practical programs and benchmarks. We also document the effects of our optimization on finite element code, observing favourable speedups over Clang, LLVM's C and C++ compiler (Chapter 6).

Chapter 2

Background

This chapter begins by describing the important task of compilers and identifying the common high-level structure shared among most present-day compilers. Next, it offers an introduction to several relevant elements of compiler theory, such as control flow graphs, intermediate representations and the static single assignment property. We proceed to discuss the general requirements of compiler optimizations, and finally focus our attention on loop-invariant code motion.

2.1 Compilers overview

A compiler is a software tool which processes a program written in a source language and translates it to a target language program with equivalent behaviour. Most compilers map code written in a high-level programming language (such as C, C++, Haskell) to machine instructions for a given architecture (such as x86 or MIPS). Compilers have a monumental role in computer science, because they allow us to express programs concisely and conveniently using abstract, high-level programming languages, instead of low-level machine operations. Indeed, before the advent of compilers, programs were written directly in assembly language, an approach that could not have possibly scaled favourably to the depth and complexity of the software systems we use today.

Modern compilers achieve translation by invoking a well-defined sequence of transformations on the source program. A common compiler organization is the *three-phase compiler* [23, p. 8], which has the following structure:

1. the **front-end pass**, which begins by interpreting the source program as a character stream, delimiting it into a set of tokens (*lexemes*). These represent various syntactic constructs in the source language, such as operators (`+`, `/`), language-specific keywords (`for`, `static`, `class`) or special characters (`;`, {). Syntactic analysis (*parsing*) processes the stream of tokens to check if it represents a valid program, matching it against a formal definition of the source language. If the input program was found to be valid, the parser constructs the Abstract Syntax Tree (AST) form, a representation that reflects the abstract structure of the program's statements. The resulted AST is then checked for semantic errors, such as type inconsistencies (e.g. assigning an `double` value to a string variable), multiple declarations of variables with the same name, or references to undefined variables. Finally, the AST is traversed and converted to a language-independent form called the intermediate representation (IR).

2. the **optimizing pass** (*optimizer*) takes the IR program generated at the end of the front-end pass, manipulates it through various transformations, and produces an optimized version of it. Crucially, the optimized IR program must be semantically equivalent to the original version, i.e. it must produce exactly the same observable results. In the same time, the resulting IR program should perform better than the original program, with regard to aspects such as execution speed, memory bandwidth or energy consumption. While a considerable number of optimizations are common to all compilers, their implementation, ordering, cost models and scope differ vastly, making the optimization pass one of the most distinguishing components of a compiler. In providing its optimizations, a compiler depends heavily on the expressiveness and properties of its IR.

3. the **back-end pass** converts the optimized IR program to native code that executes on a target architecture, a process known as code generation. IR statements are rewritten to native instructions, and abstract storage locations used in the IR are mapped to machine registers (*register allocation*). During the code generation stage, the back-end pass applies its own target-dependent optimizations, such as instruction scheduling [23, p. 639], which reorders instructions such that they are executed more conveniently on the target machine.

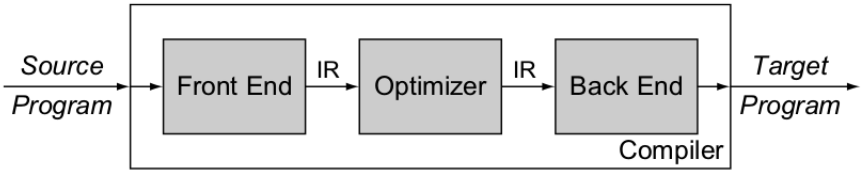


Figure 2.1: High-level structure of a typical three-pass compiler (taken from [23, p. 8]).

Not all compilers target machine instruction sets. Source-to-source compilers, for instance, translate programs from one high-level language to another [3]. Also, compilation does not necessarily have to be a separate, stand-alone process that is carried out before the program is executed. Just-in-time (JIT) compilation is a technique used by some languages based on virtual machines, and involves compiling portions of a program to machine code while the program is being interpreted. A prominent use case of JIT compilation is in the Java Virtual Machine (JVM), where sequences of bytecode are often compiled to native code in order to execute faster.

2.2 Control flow graphs

Throughout the compilation stages, it is often useful to represent the source program using various graph and tree structures [23, p. 226]. For instance, parse trees and abstract syntax trees are used by the front-end pass when performing syntactic and semantic analysis. Similarly, IR programs are modelled using a directed graph known as the control flow graph (CFG) [23, p. 231]. This graph describes the flow of control within

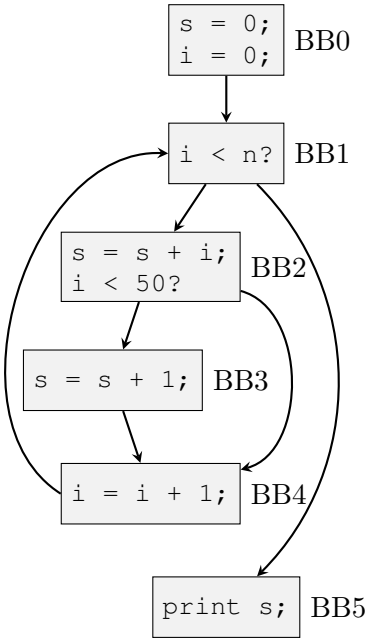
the given program, and is essential for implementing many compiler optimizations.

The nodes of the CFG are the program's *basic blocks*, sequences of non-branching statements that execute consecutively in the absence of exceptions. Edges between nodes represent potential transfers of control from one basic block to another during program execution. Control is transferred as a result of conditional and unconditional branches, which arise from high-level statements such as `if-then-else`, `for`, `while` or `break`. In the absence of branches, a program would execute all its constituent statements exactly once, in a top-to-bottom fashion.

If `A` and `B` are two basic blocks, and there is an edge between them in the CFG, we say that `A` is the *predecessor* of `B` and, equivalently, `B` is the *successor* of `A` [1, p. 529]. Figure 2.2 shows a pseudocode fragment and a corresponding CFG representation side-by-side. Here, `BB0` is the predecessor of `BB1` and, conversely, `BB1` is the successor of `BB0`. However, `BB1` is not the predecessor of `BB3`, because there is no direct edge connecting the two blocks (thus, the predecessor and successor relations are not transitive).

```
1  s = 0;
2  for (i = 0; i < n; i++) {
3    s = s + i;
4    if (i < 50) {
5      s = s + 1;
6    }
7  }
8  print s
```

(a) Source view.



(b) CFG view.

Figure 2.2: Example mapping between pseudocode and a corresponding control flow graph representation.

Practically, an edge between two basic blocks `A` and `B` means that, during at least one possible execution of the program, the last non-branching statement of `A` is followed by the first statement of `B`. Inspecting the CFG in our example, we observe two outgoing edges from `BB1`. One edge transfers control to `BB2` if the condition `i < n` holds, in which case the following statement would be `s = s + i`. On the other hand, if the loop

condition does not hold, control is transferred to `BB5`, and the first statement executed after the comparison is the `print` statement.

Each control flow graph has a conceptual *entry node*, which has a single edge to the first executable basic block in the program, but is otherwise empty. Conversely, an *exit node* is used to represent the end of the program's execution. Any basic block containing a statement that terminates execution will have an edge to the CFG's exit node [1, p. 529].

2.2.1 Dominator analysis

Dominator analysis is a technique applied by compilers at the level of the CFG, with the aim of reasoning about the structure of the underlying program [23, p. 478]. This analysis relies on the key notion of *dominators*. In a flow graph, a node `A` *dominates* a node `B` (A $\gg$ B) if every path from the entry node to `B` also includes `A` [1, p. 656]. In the CFG of Figure 2.2b, some examples of the dominance relationship are `BB0` $\gg$ `BB1`, `BB2` $\gg$ `BB3` and `BB1` $\gg$ `BB5`.

Based on this definition of dominance, every node in the control flow graph automatically dominates itself. *Strict dominance* between two nodes `A` and `B` requires that `A` $\gg$ `B` and, in addition, `A`$\neq$`B` [1, p. 657]. Thus, the three examples of dominance we offered in the previous paragraph also satisfy strict dominance. Furthermore, every non-entry node has an *immediate dominator*, the unique strict dominator which lies closest to it in the CFG. In our example, `BB1` strictly dominates `BB3`, but it is not its immediate dominator. Instead, the immediate dominator is `BB2`, the unique strict dominator with the shortest path to `BB3`.

Immediate dominators are used to condense the result of dominator analysis into a single data structure, the *dominator tree*. Every node in the dominator tree is a basic block of the CFG, and a node `A` is the unique parent of a node `B` if and only if `A` is the immediate dominator of `B`. From this representation, complete dominance relationships can be easily reconstructed, by noting that a node is dominated by all its tree ancestors. Figure 2.3 shows the dominator tree constructed for the flow graph used as an example throughout this section.

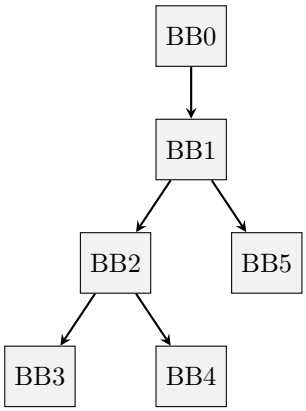


Figure 2.3: Dominator tree corresponding to the CFG in Figure 2.2b.

2.2.2 Natural loops

In the CFG representation, loop statements of the underlying program map to graph regions called *natural loops*. We use dominator analysis to define a natural loop as a set of CFG nodes with the following two properties [23, p. 665]:

1. There is an unique entry node (the *loop header*) which dominates all the other nodes in the loop.
2. There is a path from every node in the loop to the loop header, which only uses nodes from the loop.

In Figure 2.2b, we identify a natural loop (referred to as `L` in the remainder of this section) defined by nodes `BB1-4` and loop header `BB1`. It is easily observed that the first required condition holds, since `BB1` $\gg$ {`BB1`, `BB2`, `BB3`, `BB4`}. Furthermore, there is a path through the loop from each of these basic block back to `BB1`, the loop header. This natural loop corresponds to the pseudocode `for` loop defined between lines 2-7 in Figure 2.2a.

There are several regions in the control flow graph which have interesting properties with respect to natural loops [7]. Among these, we mention:

- The *loop preheader*: a basic block whose unique successor is the header of some natural loop. In our running example, `BB0` is the preheader of `L` since its unique successor is `BB1`, the header of `L`.
- *Back edges*: an edge between two basic blocks A and B, whereby B dominates A. Each natural loop has at least one back edge (also known as a *latch*) reaching back to the loop header from some node inside the loop. For instance, `L` has a single latch (the edge between `BB4` and `BB1`).
- *Loop exit blocks*: basic blocks which reside outside of a given loop, but have predecessors in that loop. In Figure 2.2b, `BB5` is an exit block of `L`.

2.3 Static single assignment form

The static single assignment (SSA) form is a desirable property of intermediate representations, which dictates that every variable in the program is only assigned to exactly once (hence the name). To achieve this, all variable definitions are altered, usually by appending a counter to the assigned variable's original name. The counter initially starts at 0 and is incremented by 1 each time the variable is redefined. Figure 2.4 shows how a sequence of pseudocode statements are rewritten such that they display the SSA property.

```
t = 0;11
a = t + 1;
b = a;
a = b + 5;
t = 9;
c = t / 5;
```

(a) Original pseudocode.

```
t0 = 0;
a0 = t0 + 1;
b0 = a0;
a1 = b0 + 5;
t1 = 9;
c0 = t1 / 5;
```

(b) After applying the SSA construction algorithm.

Figure 2.4: Example construction of the SSA form.

Another important aspect of the SSA form relates to how control flow is handled. To illustrate, consider the code snippet below, which contains a trivial C procedure for computing the signum function for a given integer.

```
1  int getSignum(int n) {
2    int sign;
3    if (n > 0) {
4      sign = 1;
5    } else if (n < 0) {
6      sign = -1;
7    } else {
8      sign = 0;
9    }
10   return sign;
11 }
```

Attempting to construct the SSA form for this example, we may end up with a result similar to the one below:

```
1  if (n0 > 0) {
2    sign0 = 1;
3  } else if (n0 < 0) {
4    sign1 = -1;
5  } else {
6    sign2 = 0;
7  }
8  return ?
```

We are faced with the problem that the variable `sign` has three different definitions, each of which occurs along a different path in the function's CFG. However, we must specify a single value as the argument of the `return` statement. SSA solves this problem by inserting a Phi function (ϕ) at the point where different control flow paths merge:

```
1  if (n0 > 0) {
2    sign0 = 1;
3  } else if (n0 < 0) {
4    sign1 = -1;
5  } else {
6    sign2 = 0;
7  }
8  sign3 = φ(sign0, sign1, sign2);
9  return sign3;
```

Phi functions are special statements whose purpose is to select between multiple definitions of the same base variable that reach the current basic block. Phi functions choose a single definition from their list of arguments, based on which block transfers control to the current basic block during program execution [23, p. 495]. In the above example, if `n = 5` at runtime, $\texttt{sign}_3$ will evaluate to $\texttt{sign}_0$, and if `n = -8`, it will evaluate to $\texttt{sign}_1$.

The SSA form is beneficial because it simplifies a number of data flow analyses, which are especially useful for ensuring the safe application of compiler optimizations such as loop-invariant code motion. Consequently, many intermediate representations (including the LLVM IR) exhibit this property. For a complete discussion on the SSA form and its applications, the interested reader is advised to consult Chapter 9.3 of [23].

2.4 Compiler optimizations

The field of compiler optimizations is arguably one of the most well-researched areas in computer science, and has spawned a wide spectrum of techniques that allow compilers to generate highly efficient target programs [2]. Most compiler optimizations aim to improve the program's overall execution speed, but it is not uncommon to target different aspects, such as code size (e.g. in embedded systems) or energy consumption (e.g. in portable electronic devices). A compiler in which the biggest fraction of the compilation time is attributed to optimizations is referred to as an *optimizing compiler*.

Two key properties that compiler optimizations need to meet are *safety* and *profitability* [23, p. 412]. An optimization is *safe* if it preserves the behaviour of the program. Since a compiler's foremost requirement is to output a target program equivalent to the source program, the compiler must only apply an optimization if it is provably safe to do so. Secondly, an optimization should be applied only when it is *profitable*. A good compiler will reason carefully about the runtime properties of the program and decide to apply a code transformation only when there is a strong likelihood that it will improve performance. To aid this decision, cost models and heuristic rules are sometimes used.

Compiler optimizations are the result of combining two types of processes: *analyses* and *transformations*. [23, p. 14]. Analyses inspect various regions of the control flow graph without modifying the program's structure, gathering safety and profitability information for transformations. The latter subsequently modify the IR in many different ways, e.g. by inserting and removing basic blocks, combining and moving statements, or inlining function calls.

2.4.1 Loop optimizations

Loop optimizations are an important category of optimizations, which have the goal of reducing the amount of CPU time spent in the program's loops. For application domains such as scientific computing, where loops often dominate execution times, scalar loop optimizations are a major source of performance improvement.

Loop optimizations depend on analyses which identify the natural loops in the program's CFG. Since natural loops consist of one or more basic blocks in the same procedure, loop optimizations are qualified as *regional optimizations* [23, p. 417] in some authors' taxonomy. These are optimizations that operate on regions of the CFG which are larger than a single basic block, but smaller than a full procedure.

Table 2.1 shows examples of several common loop transformations, adapted from material in [2] and [1, p. 848-849].

2.5 Loop-invariant code motion

Loop-invariant code motion (LICM) is a widely used loop optimization, which begins by searching a given natural loop for expressions whose result does not change between successive iterations. Once the appropriate expressions are identified, they are moved (*hoisted*, or *lifted*) before the loop entry. The result of each expression is stored in a temporary variable, and all uses of the expression in the original loop are subsequently replaced with references to that variable.

In Figure 2.5a, the expressions `x * x` and `x + 5` at lines 4 and 5 are candidates for code motion, since the value of `x` is not redefined anywhere in the loop. Figure 2.5b shows the same loop after LICM was applied. The two expressions have been moved outside of the loop, their results stored in temporaries `t1` and `t2` respectively, and references to them rewritten to use the said temporaries.

```
1  x = 5;
2  for (i = 0; i < 50; i++) {
3    a = i * i;
4    b = a + x * x;
5    c = b * (x + 5);
6    d = a + b * c;
7  }
```

(a) Original loop.

```
1  x = 5;
2  t1 = x * x;
3  t2 = x + 5;
4  for (i = 0; i < 50; i++) {
5    a = i * i;
6    b = a + t1;
7    c = b * t2;
8    d = d + c;
9  }
```

(b) After loop-invariant code motion.

Figure 2.5: Applying loop-invariant code motion at source level.

Transformation	Original code	Optimized code

Loop unrolling (by factor of 4)	`for (i=0; i<100; i++) {`
	`  s += A[i];`
	`}`
		`for (i=0; i<100; i+=4) {`
		`  s += A[i];`
		`  s += A[i+1];`
		`  s += A[i+2];`
		`  s += A[i+3];`
		`}`

Loop unswitching	`for (i=0; i<100; i++) {`
	`  s += A[i];`
	`  if (0 < n && n < 100) {`
	`  A[i] = A[n];`
	`  }`
	`}`
		`if (0 < n && n < 100) {`
		`  for (i=0; i<100; i++) {`
		`  s += A[i];`
		`  A[i] = A[n];`
		`  }`
		`} else {`
		`  for (i=0; i<100; i++) {`
		`  s += A[i];`
		`  }`
		`}`

Loop interchange	`for (i=0; i<100; i++) {`
	`  for (j=0; j<1000; j++) {`
	`  C[i][j] = A[j][i] *`
	`  B[j][i];`
	`  }`
	`}`
		`for (j=0; j<1000; j++) {`
		`  for (i=0; i<100; i++) {`
		`  C[i][j] = A[j][i] *`
		`  B[j][i];`
		`  }`
		`}`

Loop fusion	`for (i=0; i<n; i++) {`
	`  s1 += A[i];`
	`}`
	`for (i=0; i<n; i++) {`
	`  s2 += B[i];`
	`}`
		`for (i=0; i<n; i++) {`
		`  s1 += A[i];`
		`  s2 += B[i];`
		`}`

Loop fission	`for (i=0; i<n; i++) {`
	`  s1 += A[i];`
	`  s2 += B[i];`
	`}`
		`for (i=0; i<n; i++) {`
		`  s1 += A[i];`
		`}`
		`for (i=0; i<n; i++) {`
		`  s2 += B[i];`
		`}`

Loop skewing	`for (i=0; i<128; i++) {`
	`  for (j=0; j<128; j++) {`
	`  s+= A[i][j] * B[j][i];`
	`}`
		`for (i=0; i<16; i+=8) {`
		`  for (j=0; j<16; j+=8) {`
		`  for (x=i; x<i+8; x++) {`
		`  for (y=j; y<j+8; y++) {`
		`  s+= A[x][y] *`
		`  B[y][x];`
		`  }`
		`  }`
		`  }`
		`}`

Table 2.1: Examples of common loop optimizations.

2.5.1 Finding loop-invariant expressions

Loop-invariant code motion relies on data-flow analysis, a collection of analysis techniques which reason about the flow of values during program execution. In particular, *reaching definitions analysis* is used to discover suitable candidate statements for code hoisting. A definition of a variable *reaches* a statement if (a) the statement reads the value of the variable and (b) there exists a path in the flow graph from the definition to the respective statement along which the variable is not redefined [23, p. 491]. Thus, if d is a reaching definition of x at statement s, then there exists a possible program execution in which s may use the value of x as it was defined by d.

LICM can only legally hoist expressions whose value remains constant after each loop iteration. Such an expression e is called *loop-invariant*, and it has the property that for all variables x it references, all definitions of x reaching e originate from outside e's loop. Equivalently, on any flow path possibly taken by the program during runtime, no value read by e is redefined within the loop. If this is the case, then moving e outside its loop could not possibly lead to a different result, and e becomes a candidate for hoisting.

Computing reaching definition information normally involves tracking all definitions of variables in the CFG and solving a set of simultaneous data-flow equations. However, if the IR used by the compiler obeys the SSA property, the task becomes trivial. Since the SSA property ensures that all variables are assigned exactly once (see Section 2.3), checking if a loop statement is hoistable only amounts to verifying that the (unique) definitions of its operands are all placed outside the loop.

2.5.2 Profitability

Loop-invariant code motion improves overall program execution time by reducing the number of times loop-invariant expressions are computed by a factor equal to the loop size. In general, the optimization is profitable, but the achieved speed-up will depend on the nature of the hoisted instructions, as well as the size of the targeted loop.

In some cases, LICM can incur performance penalties due to the introduction of excessively many temporary variables. During execution, these will typically be stored in physical machine registers. If there are not enough registers available for storing all the precomputed results, *register spilling* will occur in the loop [2], a process which frees registers by transferring their stored values to memory. This introduces considerable memory latency in the program, which may outweigh any benefits achieved by code motion.

In cases where machine registers are sparse, one could prioritize hoisting expensive statements (such as function calls to complex routines, or floating-point arithmetic operations) over cheaper ones such as bitwise operations or shifts, which are executed efficiently by the CPU.

2.5.3 Safety

A central principle underlying the safe application of loop-invariant code motion is that no instruction should be executed in the modified program if it is not executed without the optimization [1, p. 641]. The basis of this requirement from a safety standpoint is

that executing such an instruction may throw an exception in the optimized program which would not have existed in the original one. Apart from different behaviour, lifting this restriction also affects the speed of the optimized program, which might end up executing more statements than the unmodified program.

Loop rotation

In order to preserve behaviour, LICM must transform the input program in a manner that satisfies the safety requirement identified above. First, we consider the aspects of hoisting invariant expressions out of loops that may not execute at all during a run of the program. Usually, these are loops based on `for` and `while` constructs, which will be skipped entirely if the loop condition does not hold initially. Clearly, it is not sound to hoist invariant statements out of such loops, since they would be executed even when their loop is not entered.

To allow safe code motion, program loops are typically rewritten using a transformation known as *loop rotation*, or *loop inversion*. Loop rotation replaces `for` and `while`-based loops with an equivalent `do-while` formulation, which tests for the initial loop condition in a wrapping `if` statement [1, p. 642]. After applying this technique, invariant statements can be safely hoisted before the loop entry, because they will execute if and only if the initial loop condition holds.

Figure 2.6 shows how this transformation is used to rewrite a loop and allow safe code motion. In Figure 2.6a, we would not hoist any invariant loop instructions out of the program, because the `i`-loop might not execute at all for some values of `n`. Applying loop rotation, we obtain the equivalent program in Figure 2.6b. Invariant statements are now hoistable to the program point before the entry to the `do-while` loop, and they will be executed only if the initial loop condition holds. We note that if `n` were known to be greater than 0 at compile-time, the wrapping `if (i < n)` statement could be eliminated altogether.

```
1  for (i = 0; i < n; i++) {
2    a = i * i;
3    b = a + x * x;
4    c = b * (x / n);
5    d = a + b * c;
6  }
```

(a) Original loop.

```
1  i = 0;
2  if (i < n) {
3    do {
4      a = i * i;
5      b = a + x * x;
6      c = b * (x / n);
7      d = a + b * c;
8      i = i + 1;
9    } while (i < n);
10 }
```

(b) After loop rotation.

Figure 2.6: Applying loop rotation to a `for` loop.

Figure 2.7 gives the control flow graph representation of this loop, after hoisting the invariant expressions `x * x` and `x / n`. The empty regions following the body of the loop and the if statement are labelled as `loop.exit` and `if.exit`. We note that `BB2`, the basic block which contains the hoisted statements, is a loop preheader (see Section 2.2.2), because its unique successor is the header of the loop. Indeed, in all instances where code motion is applied, loop-invariant instructions will be moved to a basic block which is the preheader of their loop.

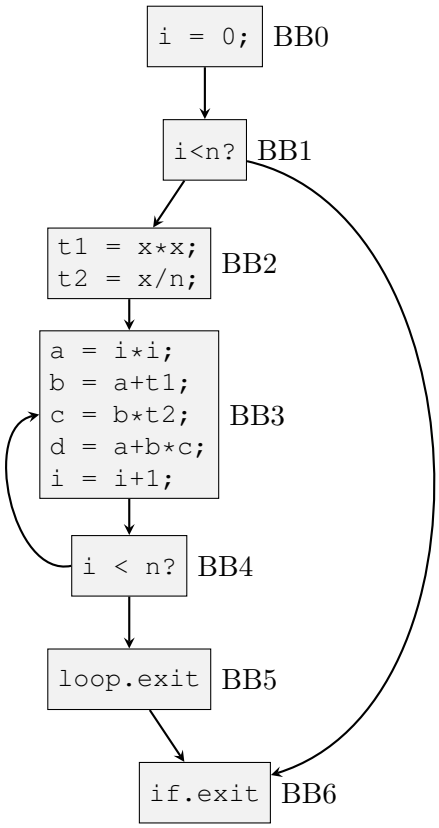


Figure 2.7: CFG view of the program in Figure 2.6b after applying loop-invariant code motion.

Hoisting instructions which are guaranteed to execute

Instructions which are not guaranteed to execute during a run of their loop should not be hoisted, even if they are invariant. If such statements were to be hoisted, the optimized program would execute them unconditionally each time before entering the loop, whereas the original program would have executed them only for some executions of the loop.

The most important category of invariant expressions LICM should avoid hoisting are those which are guarded by conditional statements inside the loop. These are not guar-

anteed to execute, since they are skipped whenever the guarding conditional expression evaluates to false. Of course, we assume that the values of the conditionals are not known at compile time, since in this case control flow statements could be simply removed.

To illustrate the potential hazards of hoisting such expressions, consider the pseudocode in Figure 2.8. Here, the sub-expression `2 / (m - n)` on line 3 is loop-invariant and guarded by a conditional statement which cannot be evaluated at runtime. If we were to hoist this expression and evaluate it outside of the loop, the optimized program would perform a division by 0 when `m` is equal to `n`, raising an arithmetic exception. This fault would not have occurred in the original program, since the conditional check prevents the division from happening.

```
1  // m, n are supplied at runtime
2  for (i = 0; i < n; i++) {
3    if (i % 3 == 0 && m != n) {
4      a = a + 2 / (m - n);
5    }
6    a = a + n * n;
7  }
```

Figure 2.8: Loop containing an invariant instruction which is not guaranteed to execute.

To check that an expression is guaranteed to execute in its loop, a convenient solution is to first apply loop rotation (if necessary) and then rely on dominator analysis. Specifically, a statement is guaranteed to execute in a loop `L` if the basic block containing it dominates all the exit blocks of `L` (see Section 2.2.2). Intuitively, this means that any code following the loop cannot be reached without first executing the expression.

Consider Figure 2.9, which presents the CFG representation of the loop in Figure 2.8 after loop rotation was applied. The invariant expression which we identified as unsafe to hoist was `2 / (m - n)`, and its basic block is `BB3`. `BB3` does not dominate the (unique) loop exit block `BB6`, due to the edge between `BB2` and `BB4`. Thus, dominator analysis reveals that we cannot hoist this sub-expression, since there are possible executions of this loop which do not evaluate it. On the other hand, the basic block of the statement `a = a + n * n` on line 7 dominates the exit block of the loop. Since this invariant statement is guaranteed to execute, it can be safely moved to the loop preheader.

Exceptions caused by loop instructions

An invariant expression should not be hoisted if any of the loop statements possibly executing before it may throw a runtime exception. If the invariant expression were hoisted, the optimized program might, again, execute some instructions which would not have been reached in the original one. If the invariant expression itself may cause execution to terminate suddenly, the optimized program might fail due to a different exception than the original program, leading to changed behaviour.

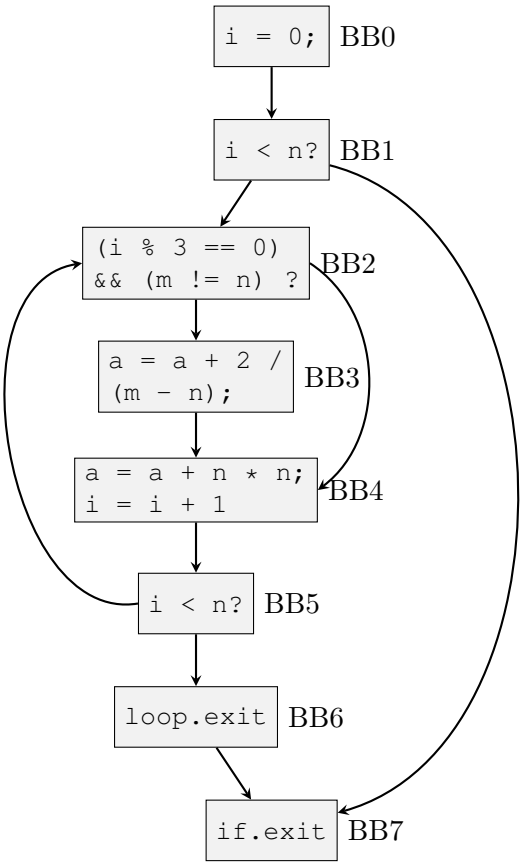


Figure 2.9: CFG of the program in Figure 2.8 after applying loop rotation.

Aliasing

Loop-invariant statements which interact with memory bear additional safety hazards. Upon identifying an instruction which loads data from memory as loop-invariant, LICM must check that there are no statements inside the loop that may write data to that same memory location. In the code snippet below, `A[1] * 5` is an invariant expression, but `B` is an alias of `A`, so the memory location pointed to by `A[1]` is actually written to when storing into `B[1]` at the second iteration of the loop. If we were to ignore aliasing and precompute the result of `A[1] * 5` in a temporary variable, the values in `B` at the end of the loop would be {`6, 7, 8, 9, 10`} in the optimized program, as opposed to {`6, 7, 38, 39, 49`} in the original program.

```
1 A[5] = {1, 2, 3, 4, 5};
2 B = A;
3 s = 0;
4 for (i = 0; i < 5; i ++) {
5   B[i] = B[i] + A[1] * 5;
6 }
```

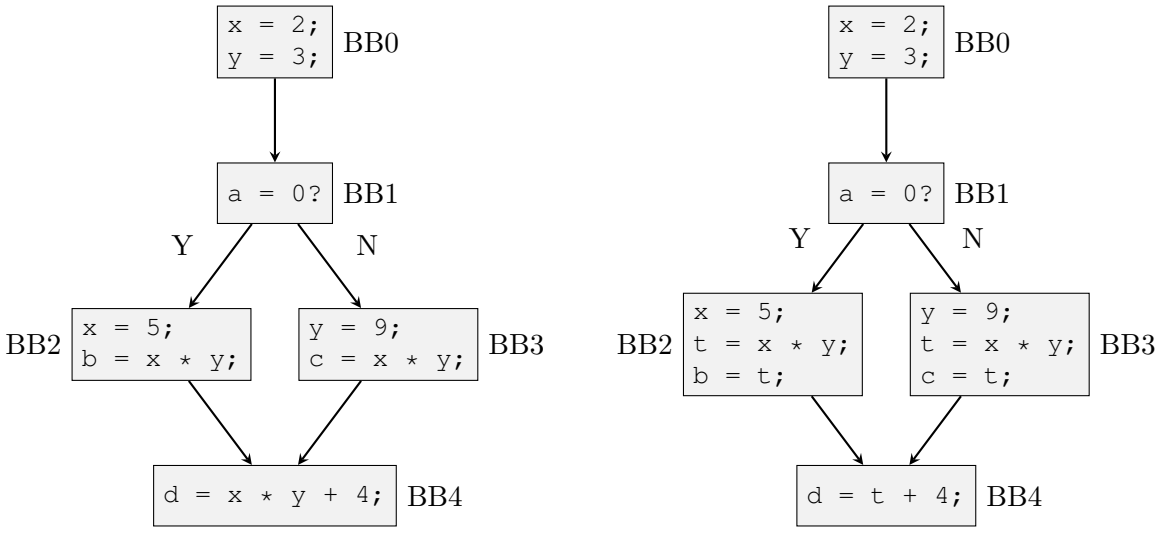
To avoid these fallacies, LICM makes use of alias analysis, a technique that discovers which memory locations are possibly accessed in multiple ways (i.e. through more than one identifier). Alias analyses are prudent, in that they classify two pointers as non-aliasing

only if this is provably true. If the analysis is not able to make a definitive decision on two pointers, it cautiously lists them as possible aliases. These properties enable optimizations such as LICM to correctly modify the placement of memory-accessing statements. In loops where memory is accessed many times and in complex patterns, code motion will perform better if an extensive alias analysis is used, at the expense of compilation time overhead.

2.5.4 Comparison with partial redundancy elimination

Loop-invariant code motion can be regarded as a specialized case of an optimization called *partial redundancy elimination* [1, p. 639]. To elaborate, we first need to introduce the notion of redundant and partially redundant expressions. An expression is *redundant* (or *fully redundant*) if it has already been evaluated on all paths leading to its program point, and none of its operands changed since the last evaluation. If the same condition holds for some, but not all the paths leading to its program point, an expression is *partially redundant* [23, p. 552].

If the evaluation of an expression is redundant at a program point, it can be completely eliminated while maintaining correct program behaviour. Consider the CFG in Figure 2.10a. Here, the sub-expression `x * y` in `d = x * y` is redundant, because it was already evaluated along all paths leading to this block, i.e. in `BB2` and `BB3`. Instead of re-evaluating it in `BB4`, we store the expression's value at `BB2` and `BB3` in a temporary `t`, and then use `t` to rewrite our statement as `d = t + 4`. Since neither `x` nor `y` change after the expression's evaluations in `BB2` and `BB3`, this transformation is safe. As the authors of [1] note, removing redundant instructions reduces the actual number of evaluations of an expression at execution time, and not the number of different positions in the CFG where the expression may be evaluated.

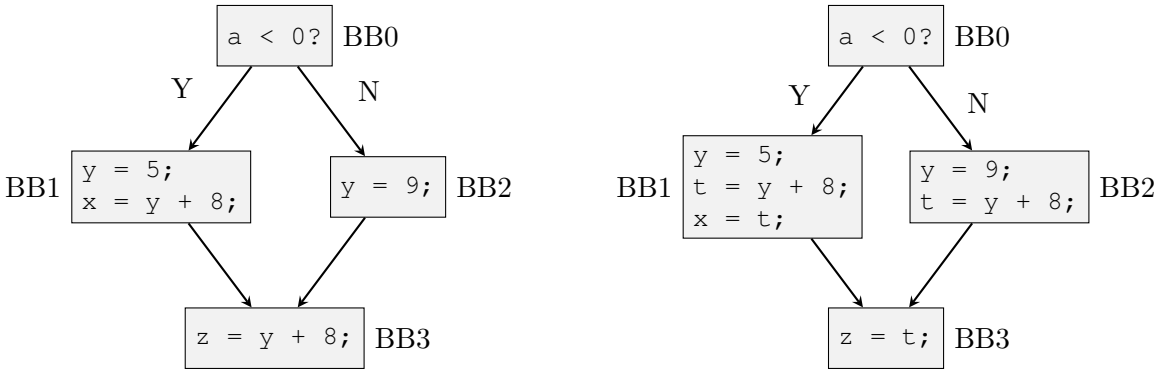


(a) Original code. **(b)** After eliminating redundant expressions.

Figure 2.10: Elimination of a redundant re-evaluation in basic block `BB4`.

Partial redundancy elimination (PRE) consists of identifying partially redundant expressions and inserting additional evaluations along different program paths such that the expressions become fully redundant. Once this is achieved, the newly redundant expressions are eliminated.

In Figure 2.11a, the expression `y + 8` is partially redundant in `BB3`, because its is evaluated in `BB1`, but not in `BB2`. An evaluation is inserted at the end of `BB2`, which now makes the expression fully redundant. We eliminate the redundant expression by storing all evaluations of `y + 8` in the same temporary value `t` and rewriting our statement in `BB3` as `z = t`. This is possible because `y` is not redefined following any evaluations of `y + 8` in `BB1` and `BB2`.

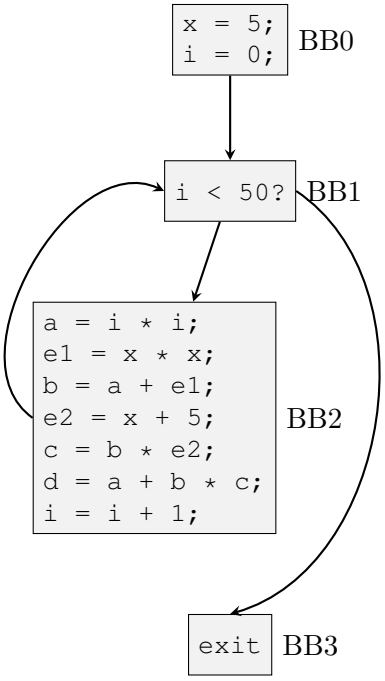


(a) CFG with a partial redundancy. **(b)** After PRE.

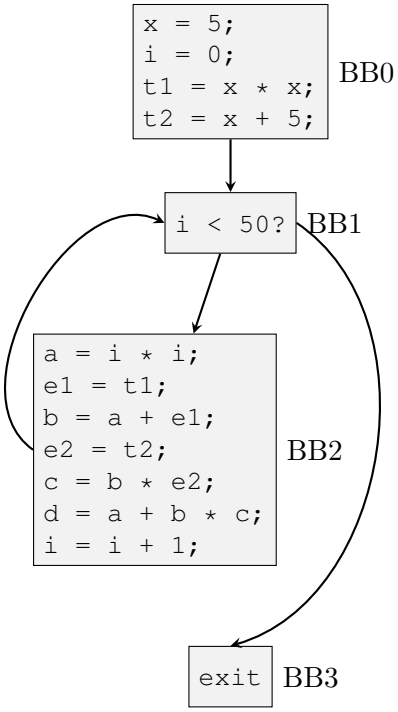
Figure 2.11: Applying Partial redundancy elimination to an `if` statement.

We now consider the implications of optimizing natural loops using partial redundancy elimination. Figure 2.12a depicts the CFG representation of the loop in Figure 2.5a, the only change being that the expressions `x * x` and `x + 5` are now stored in two separate variables, `e1` and `e2`, before being used.

These two expressions are evaluated on some path up to their current program point (i.e. the path followed along the loop's latch). However, they are not evaluated on the path that enters the natural loop from `BB0`, which makes the expressions only partially redundant. Using PRE, we insert evaluations of the two expressions on the remaining paths needed to achieve full redundancy, and then eliminate the redundant evaluations. We obtain the CFG shown in Figure 2.12b, which is almost identical to the CFG resulting after loop-invariant code motion.



(a) A loop with partially redundant expressions.



(b) After PRE.

Figure 2.12: Applying partial redundancy elimination to a natural loop.

Chapter 3

The LLVM project

This chapter introduces the LLVM compiler infrastructure, the environment we have chosen for implementing our optimization. We discuss the architecture of the language agnostic optimizer and offer a brief introduction into the rich LLVM intermediate representation, supporting it with numerous examples from its type system and instruction set. We present the LLVM command-line tools which we found to be most useful for our goals, together with a suggested workflow for manipulating simple programs. Finally, we introduce the testing infrastructure used across the project, which we shall refer to when discussing our evaluation methodology.

3.1 Introduction

LLVM is an open-source framework providing a modern *collection of modular and reusable compiler and toolchain technologies* [20]. The project stemmed from the work of Chris Lattner, who first implemented core elements of LLVM to support the research of his master thesis in 2002 [26]. One of the key strengths of LLVM is that it faces active development from an expert community of contributors, and is widely used across the industry and academia alike [11]. The project received the ACM Software System Award in 2012 [4] as an acknowledgement of its contribution to compiler research and implementation.

LLVM officially acknowledges more than 10 main sub-projects [20], which range in diversity from a debugger [9] to a symbolic execution tool [8]. In addition to these, the official website presents a long list of miscellaneous projects that are based on components of the LLVM infrastructure [19].

All projects which are part of the LLVM ecosystem are built upon the core libraries, which are arguably the centrepiece of LLVM. They host the source- and target-independent optimizer (Section 3.3), the implementation of the LLVM intermediate representation (Section 3.2), and a suite of command-line tools useful for code manipulation (Section 3.4). The core libraries also implement various back-end passes that translate IR to machine code for different platforms (x86, PowerPC, Nvidia GPUs).

For building a complete compiler for a source language, the LLVM core libraries readily provide the optimizing pass and code generation for common architectures, the front-end being the only missing component. Clang is by far the most prominent front-end implemented in LLVM [5], and it targets the family of C languages (C/C++ and

Objective-C/C++). Coupled with the core libraries, Clang is a powerful compiler producing high-performance code, and positions itself as a direct competitor to both gcc and the Intel compiler.

3.2 The LLVM IR

The LLVM IR (in some contexts known as *LLVM assembly*) is an expressive code representation designed to support most practical high-level languages [15]. In the same time, the IR's instruction set closely resembles the instruction set of a microprocessor, facilitating code generation. It obeys the SSA property discussed in Section 2.3, which reduces the complexity of many compiler analyses. In its textual form, the LLVM IR is clear and human-readable, allowing new developers to accommodate to it quickly.

Program structure representation

Hierarchically, an LLVM IR program contains the following components:

- **Modules:** Each module is the result of compiling a single translation unit (source file) of the original program using an LLVM-based front-end such as Clang. The most important components of a module are the declarations and definitions of global variables and functions. When compiling a program that consists of multiple source files, modules have to be linked together, in the same way C object files are linked to produce a final executable.
- **Functions:** Functions are introduced using the `define` and `declare` keywords. Various compilation parameters, such as visibility, garbage collection strategy, or memory alignment, can be specified on a per-function basis through special attributes in the IR . Each function definition minimally consists of one or more basic blocks, together with a list of parameters and a return type.
- **Basic blocks:** The LLVM IR implements basic blocks as we have defined them in Section 2.2, i.e. as sequences of non-branching statements, executed in succession. Basic blocks are named by labels, and delimited by *terminator instructions*, which transfer control to other basic blocks.
- **Instructions:** Instructions are the basic unit of the LLVM IR, and they group together in basic blocks. Section 3.2 briefly introduces the instruction set.

Identifiers

In the LLVM IR, entities and values are named using two types of identifiers:

- **Global** (reserved for functions and global variables), prefixed by @.
- **Local** (used by local function variables and custom types), prefixed by %.

Identifiers are prefixed with these special characters mainly in order to avoid clashes between variable names and reserved keywords in the IR [15].

LLVM identifiers can be derived from source-level identifiers, by adding various prefixes, suffixes or integers. Alternatively, identifiers may be generated by the compiler with no naming hints from the source program, in which case they are represented by a single

unsigned integer (a so-called *unnamed identifier*). To obtain the integer, a counter is initialized to 0 in each program function, and subsequently incremented by 1 whenever a new identifier is assigned.

Type system

The LLVM IR maintains detailed type information about all the entities in each module. This provides transparent information to the optimizer, and allows the IR to resemble the source code much more closely. Table 3.1 introduces some of the most common LLVM types. Note that there is no dedicated boolean type in the IR. Instead, the `i1` type (together with two constant values, `true` and `false`) is used as a substitute.

The type system includes two additional types that are not used in relation to program values. One is the **label** type, used by the labels of basic blocks. The other is the **metadata** type, used to reference *metadata nodes*, special IR constructs that hold miscellaneous data about the program, such as debug information.

Type	Description	Example(s)
Integer	N-bit wide integers, for N between 1 and $2^{33} - 1$	`i1` is a 1-bit integer, while `i39` is a 39-bit integer
Floating point	Floating point types of different widths	`half`, `float` and `double` are 16-, 32- and 64-bit floating point types, respectively
Pointer	Holds the memory location of a data item	`double*` is a pointer to a `double`
Structure	A collection of different types, stored at contiguous memory locations	`{i35, i64*}` denotes a structure holding an `i35` and a pointer to an `i64`
Array	A C-like array, a sequence of elements of the same type	`[16 x float]` represents an array holding 16 elements of type `float`
Function	Function signature (one type for the returned value and a list of parameter types)	`float (i32*, {i32, i32}*)` is a function that takes a pointer to an `i32` and a pointer to a structure containing two `i32`s, and returns a `float`
Vector	Vectors of multiple elements, used by SIMD instructions	`<16 x float>` is a vector of 16 floating point values
Void	Has no size and represents no value	A function with no return value has the `void` return type

Table 3.1: Common types of the LLVM type system.

Instruction set

The LLVM instruction set is designed to resemble assembly languages closely in the amount of low-level operations it can express. Based on their purpose, instructions are grouped into several major categories, which are shown in Table 3.2.

Key	Category	Description
	Terminator instructions	Signal the end of basic blocks and transfer control to another part of the program
	Binary operations	Perform arithmetic calculations on two operands of the same type
	Bitwise operations	Perform bitwise calculations on two operands of the same type
	Memory accessing and addressing	Handle all interactions with memory (accessing, allocation, reading and writing)
	Conversion operations	Convert values between different types
	Miscellaneous	Instructions which cannot be placed in any of the above categories

Table 3.2: The main instruction categories in the LLVM IR.

Table 3.3 introduces several important instructions for each category, together with examples, using the color coding introduced in the table above. The list of instructions we have chosen is representative for our purposes, but far from complete. For an exhaustive presentation of the instruction set, the reader is advised to refer to the official documentation of the LLVM language [15].

We note that the syntactic structure of the LLVM IR is somewhat different from other assembly languages, since instructions that produce values explicitly assign them to an identifier. Thus, most instructions in the IR share a common high-level syntax:

```
<identifier> = <instruction> <optional parameters> <operands>
```

The form above clearly emphasizes the IR's SSA property, while also promoting readability. By contrast, in other popular assembly languages such as x86 or MIPS, the assignment of values to their destinations is implicit.

Type	Instruction	Explanation	Example
	`ret`	Returns control (and possibly a value) from the current function back to its caller	`ret i32 %a` exits the current function and returns the value `%a`, which is of type `i32`
	`br`	Transfers control flow to other basic blocks in the same function, either conditionally or unconditionally	`br i1 %cmp, label %a, label %b` branches to basic block `a` if the value in `%cmp` is 1, branches to `b` otherwise.
	`add, sub, mul, div`	Arithmetic operations on integer values. `fadd, fsub, fmul, fdiv` used for floating-point numbers	`%add = fadd double %1, %2` adds `%1` and `%2` (of type `double`) and stores the result in `%add`.
	`urem, srem`	Compute the remainder from the unsigned and signed division of their operands	`%c = urem i45 %a, %b` performs an unsigned division of `%a` to `%b` and stores the remainder in `%c`
	`and, or, xor`	Logical bitwise operations	`%0 = and i1 %x, %y` computes `%x & %y` and stores the result in `%0`
	`shl, lshr, ashr`	Arithmetic and logical bit shifts	`%0 = lshr i8 48, 2` shifts the bits in 48 by 2 positions to the right (effectively dividing by 4) and writes the result to `%0`
	`getelementptr`	Computes the address of an element in an aggregate data type (e.g. an array); for an in-depth presentation of this instruction, the reader is advised to consult [21]	`%idx = getelementptr [5 x float], [5 x float]* %arr, i32 0, i32 3` computes the address of the fourth element of a float array with 5 elements, pointed to by `%arr`
	`load`	Reads a value from a given memory location	`%n = load double, double* %ptr` reads a `double` value from the memory address pointed to by `%ptr` into `%n`.
	`store`	Stores a value at a given memory location	`store i32 %val, i32* %ptr` writes the value stored in `%val` to the memory location pointed to by `%ptr`
	`alloca`	Allocates memory on the stack	`%arr = alloca double, i64 10` allocates ten `doubles` on the stack and returns a pointer to the address of the first element.
	`sext, zext, fpext`	Extend type of their operand to one represented by more bits	`%b = zext i8 %a to i32` extends an `i8` value `%a` with zeros up to the size of the destination type (`i32`), placing the result in `%b`.
	`trunc, fptrunc`	Truncate type of their operand to one represented by fewer bits	`%a = trunc i32 %b to i8` truncates the higher order bits of `%b` down to the size of the destination type (`i8`), placing the result in `%a`
	`call`	Transfers control to a called function	`%a = call i32 @f()` calls the function named `f` and stores the result it returns (an `i32` value) in `%a`
	`cmp`	Compares between two values of the same type	`%cmp = icmp eq i32 %i, 10` stores the value 1 into `%cmp` if `%i` is equal (`eq`) to 10, or 0 otherwise
	`phi`	Implements the Phi node of the SSA form	`%i = phi i64 [%x, %bb1], [%y, %bb2]` sets `%i` to `%x` if control arrives from `bb1`, or to `%y` if it arrives from `bb2`

Table 3.3: Examples of instructions in the LLVM IR instruction set.

In-memory and on-disk representation

The IR's implementation in the core libraries provides the in-memory representation used by various tools, such as the optimizer, during compile time. Discussing the implementation of the IR is beyond the scope of this paper, and the reader is advised to refer to the LLVM developer's handbook [17] for an overview of the IR's implementation. Complete information is available from LLVM's online source code documentation [16].

On disk, the IR is represented in two ways:

- as *bitcode files*, which encode an IR program as a stream of bits [13], and typically use the `.bc` extension. These files can be executed directly using a bitcode interpreter (see Section 3.4).
- as *assembly files*, human-readable ASCII text files represented by the `.ll` extension. Most importantly, they allow compiler writers to visually inspect the IR after it is generated or subsequently manipulated.

Complete example

We conclude this section by providing a full example of the LLVM IR in Figure 3.1, which showcases all the language features we have discussed so far. The IR is obtained as is by compiling the simple C program below using Clang, with all compiler optimizations turned off.

```
1  int halfsum(int *A, int n) {
2    if (n <= 0) {
3     return -1;
4    }
5    int s = 0;
6    for (int i=0; i<n; i++) {
7      s += A[i];
8    }
9    return s >> 1;
10 }
```

Visually, each basic block of the function is clearly represented by a label (e.g. `for.cond`, `if.then`, `return`) and delimited by one of the instruction set's terminator instructions. We believe that most of the IR is self-explanatory, except possibly for the comparisons on lines 11 and 26, which use two specifiers that we have not yet mentioned: `sle` (signed less than or equal) and `slt` (signed less than). With this clarification, the reader can verify that the given intermediate representation mirrors the behaviour of the source program.

```
1  define i32 @halfsum(i32* %A, i32 %n) {
2  entry:
3    %retval = alloca i32, align 4
4    %A.addr = alloca i32*, align 8
5    %n.addr = alloca i32, align 4
6    %s = alloca i32, align 4
7    %i = alloca i32, align 4
8    store i32* %A, i32** %A.addr, align 8
9    store i32 %n, i32* %n.addr, align 4
10   %0 = load i32, i32* %n.addr, align 4
11   %cmp = icmp sle i32 %0, 0
12   br i1 %cmp, label %if.then, label %if.end
13
14 if.then:
15   store i32 -1, i32* %retval
16   br label %return
17
18 if.end:
19   store i32 0, i32* %s, align 4
20   store i32 0, i32* %i, align 4
21   br label %for.cond
22
23 for.cond:
24   %1 = load i32, i32* %i, align 4
25   %2 = load i32, i32* %n.addr, align 4
26   %cmp1 = icmp slt i32 %1, %2
27   br i1 %cmp1, label %for.body, label %for.end
28
29 for.body:
30   %3 = load i32, i32* %i, align 4
31   %idxprom = sext i32 %3 to i64
32   %4 = load i32*, i32** %A.addr, align 8
33   %arrayidx = getelementptr inbounds i32, i32* %4, i64 %idxprom
34   %5 = load i32, i32* %arrayidx, align 4
35   %6 = load i32, i32* %s, align 4
36   %add = add nsw i32 %6, %5
37   store i32 %add, i32* %s, align 4
38   br label %for.inc
39
40 for.inc:
41   %7 = load i32, i32* %i, align 4
42   %inc = add nsw i32 %7, 1
43   store i32 %inc, i32* %i, align 4
44   br label %for.cond
45
46 for.end:
47   %8 = load i32, i32* %s, align 4
48   %shr = ashr i32 %8, 1
49   store i32 %shr, i32* %retval
50   br label %return
51
52 return:
53   %9 = load i32, i32* %retval
54   ret i32 %9
55 }
```

Figure 3.1: Unoptimized IR program containing the definition of a single function, which computes half the sum of an array of integers.

3.3 The optimizer

The source- and target-independent optimizer consumes LLVM bitcode files, analyses and transforms them, and finally emits a bitcode file with identical behaviour, but more efficient runtime properties.

The basic unit of the LLVM optimizer is the **pass**. A pass is a single, distinct operation applied to the input program. Based on their nature, passes fall into two categories:

- **Analysis passes**, which traverse the program and collect information that is used by subsequent passes. Analysis passes do not modify the program at all. As an example, the `loops` analysis pass inspects the program and identifies the natural loops in the body of each function.
- **Transform passes**, which modify the input program in some particular way in order to improve its efficiency. These passes perform a variety of transformations, such as reordering, moving or replacing instructions, rearranging basic blocks, restructuring loops, or vectorization. Most transform passes require information from analysis passes in order to ensure that the changes they propose are safe and result in a semantically equivalent program.

This clear twofold separation between passes mirrors an idea we outlined in Section 2.4, where we argued that optimization consists of analysis followed by transformation.

Passes are also distinguished based on which region of a program they operate on. From this point of view, the main types of passes are module passes, function passes, loop passes, basic block passes and call graph passes (which traverse the program's call graph from callees to callers). Within each of these subcategories, we commonly find both analysis and transform passes.

For a pass to be enabled in the optimizer, its implementation must be registered with an object called the **pass registry**, to which it provides information such as its name, description and command-line flag. Each pass also specifies all the transform and analysis passes that must be scheduled before its execution. Optionally, each pass specifies which analysis results become outdated after its execution and which remain unchanged. For instance, a pass that creates new loops will invalidate the result of the `loops` analysis, which identifies natural loops in the program. On the other hand, a pass that rewrites instructions with constant operands will not affect a dominator tree analysis pass.

The pass registry is not responsible for generating an optimal sequence of optimizations. Instead, any tool that builds upon the optimizer is responsible for specifying the desired running order for optimizations, by adding them to a queue called the **pass manager**. Clang uses its own optimization sequence, suitable for C-based languages, but other front-ends may choose to apply optimizations differently. The order in which passes are queued is the order in which they will execute, subject to dependency constraints.

The pass manager uses the information gathered by the pass registry and is responsible for efficient and correct pass execution. It achieves this by:

- scheduling the prerequisites of each pass to be executed before that respective pass;

- reusing the results of analyses between passes. Analysis information required by a pass is recomputed only when it is believed that an earlier pass invalidated it. If a pass does not specify which analyses it preserves, the pass manager will assume that all analyses are invalidated.

3.4 The LLVM toolchain

LLVM provides more than a dozen command-line tools, all of which manipulate source and IR programs in different ways [14]. Some commonly used command-line tools are:

- `clang`: the LLVM-based front-end for the C family of programming languages. This tool is not distributed with the core LLVM project, but maintained in its own separate repository. The user can either build the tool from source, download a pre-built binary for their architecture, or obtain the tool from official software repositories on some Linux distributions. Many of the command line options accepted by `clang` are identical to `gcc`'s for compatibility reasons. Conveniently, `clang` can compile source files to LLVM IR if the `-emit-llvm` flag is passed.
- `opt`: the LLVM optimizer. As with `clang`, the user can specify a general optimization level (such as `-O1`, `-O2` or `-O3`) which applies a preconfigured sequence of optimizations. Alternatively, the user can provide a list of passes which will be scheduled and executed by the pass manager in the order in which they are given.
- `llc`: the LLVM static compiler, which compiles LLVM programs (encoded as bitcode files) to assembly language for a specified target architecture. Users can also cross-compile code for a different architecture than that of the host machine.
- `llvm-as`: the LLVM assembler, which converts human-readable LLVM assembly language files to bitcode files.
- `llvm-dis`: the LLVM disassembler, performing the reverse operation, i.e. converting bitcode files to ASCII `.ll` files.
- `lli`: the LLVM interpreter, which interprets a program in LLVM bitcode format, or executes it using a JIT compiler.
- `llvm-link`: the LLVM bitcode linker, which links multiple bitcode files in a single bitcode file, useful for compiling programs consisting of multiple source files.

Figure 3.2 shows how these different tools may be used to compile, optimize and execute a single-source C program.

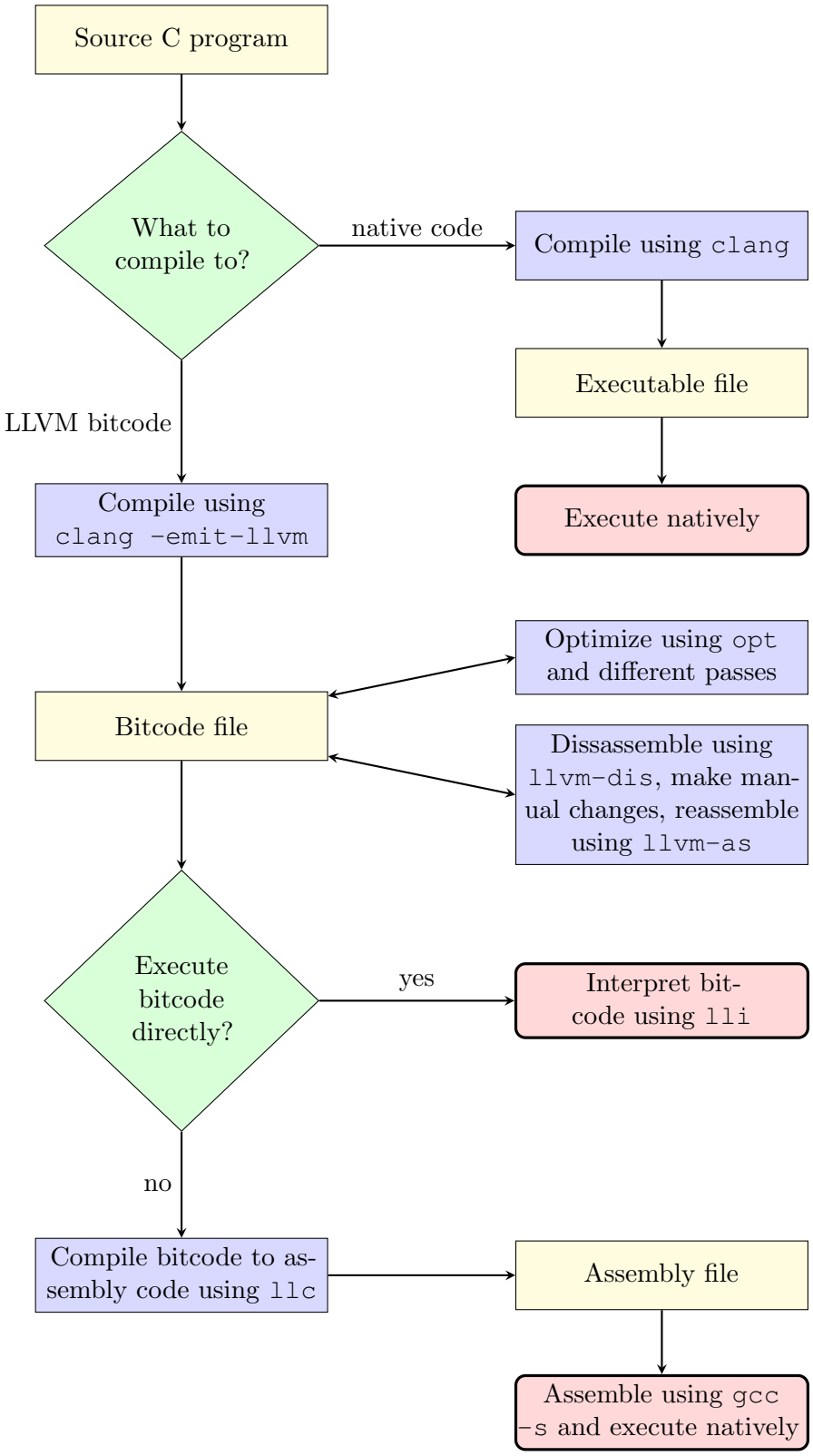


Figure 3.2: Methodology for compiling and executing a single-source C program.

3.5 Testing infrastructure

LLVM relies on two types of tests for evaluation [18]. One is a collection of small unit tests which stress specific features of LLVM subcomponents (such as individual optimization passes), and the other is a sizeable collection of real-world programs, together with the supporting infrastructure to evaluate them using a full-fledged compiler. This section describes the latter, which we used extensively when carrying out performance evaluation.

The LLVM whole program tests (also known as the *LLVM test suite*) consist of hundreds of practical applications and benchmarks written in C and C++. Due to its scale, the test suite is maintained in a separate repository, as opposed to the LLVM source tree. There are three types of whole-program tests:

- *Single-source tests*, small programs that consist of a single source file.
- *Multi-source tests*, larger applications which are built by compiling and linking multiple source files.
- *External tests*, for which the test suite does not provide the source code, but only the Makefiles necessary to compile them. The most prominent external test supported by LLVM is arguably the SPEC benchmark [25].

The aim of the test suite is to evaluate the correctness of the compilation process for a given compiler, and the performance of the resulting executable programs. Most often, the compiler under test is Clang, but using different compilers is also supported. In summary, the testing procedure is the following:

1. Compile (and possibly link) each program's source files, timing the process.
2. If any compilation errors occur, or the compilation time exceeds a specified threshold, the test case is reported as failing at compile-time.
3. Execute the program and time its duration.
4. If the program produces an output that is different from a stored reference output, or the execution time is greater than a runtime threshold, the test case is marked as failing at runtime.
5. After all tests are processed, a report is generated containing all the information gathered during the test run.

To run the test suite conveniently, a command-line tool called `lnt` [22] is provided, a wrapper around a Makefile-based infrastructure which LLVM had used initially. `lnt` provides a simple interface for running all or a subset of the test suite, and allows users to configure per-run parameters, such as the path to the compiler under test or the amount of parallelism in the build process. After the run completes, a report in JSON format is produced and stored in a local SQLite database. Running another one-line `lnt` command, the tester can start a local server backed by the database of test reports, and then use a web browser to visualize and compare the results of different runs.

Chapter 4

Generalized loop-invariant code motion

This chapter introduces generalized code motion, the main contribution of this project. First, we present the methodology of our optimization, and we describe the application domain which motivated its development. Afterwards, we identify and discuss the main sources of runtime improvements and slowdowns, and address the safety concerns of our transformation. Finally, we present two ideas that may further improve the versatility and applicability of our technique.

4.1 Overview

The optimization technique we have developed and implemented in this project is called *generalized loop-invariant code motion* (GLICM). GLICM shares the same underlying mechanics and motivation as regular loop-invariant code motion, but (as the name implies) generalizes the procedure by identifying more types of invariant expressions and hoisting code at multiple program locations.

In Section 2.5, we described how LICM identifies statements which are invariant with respect to an IR loop, and then hoists these statements in the preheader block of the respective loop, subject to several safety considerations. By contrast, our optimization finds expressions which are invariant not with respect to the current loop, but to some parent loop (*superloop*), a higher-level loop enclosing it. For an expression to be invariant under this notion, none of its operands must be redefined anywhere in the chosen parent loop, except if the operand is the induction variable of the current loop. After such invariant expressions are identified, our optimization proceeds with the following steps:

1. Place a new, empty loop in the preheader of the parent loop. We shall refer to this loop as the *cloned loop*. The cloned loop must crucially have the same iteration pattern as the loop that is currently being processed.

2. Hoist the invariant expressions to the cloned loop. Because the results of these expressions must be remembered at each iteration of the original loop, we cannot simply store them in scalar variables. Instead, they are stored in temporary one-dimensional arrays, which we allocate before the entry of the cloned loop. These arrays will be as large as the size of the loop, and will have the appropriate type for storing the result of each hoisted statement.

3. Replace references to the hoisted statements in the original loop with references to the appropriate temporary array element.

To help the reader envision our transformation, we will apply GLICM at source level on a pseudocode example. Consider the code snippet in Figure 4.1a, which shows a triple nested loop performing some calculations on a two-dimensional array (we omitted initialization code for brevity). We shall use our transformation to hoist invariant statements from the innermost `k`-indexed loop out to the preheader of its immediate parent, the `j`-indexed loop.

The following steps are taken:

1. We inspect the body of the `k`-loop, looking for invariant expressions. We identify the expression `B[i][k] * B[k][i]` at line 6 as a candidate for generalized hoisting. This expression is invariant with respect to the `j`-loop, because `i` is not redefined within the body of that loop, and `k` is the induction variable of the current loop.

2. We create an empty clone of the `k`-loop in the preheader of the `j`-loop (see Figure 4.1b). Note that this has the same iteration space as the original loop.

3. We hoist the statement `B[i][k] * B[k][i]` to the body of the cloned loop. We allocate a one-dimensional array of 100 `doubles` and store the value of this expression at the `i`th iteration of the cloned loop into the `i`th element of the temporary array. The state of the code after this stage is shown in Figure 4.1c.

4. Finally, we replace references of the invariant expression in the original loop with references to the array in which its result was stored (see Figure 4.1d). This concludes the optimization process.

We note two distinctive characteristics of GLICM that follow from this overview. Firstly, as opposed to regular loop-invariant code motion, GLICM cannot hoist statements out of arbitrary loops. The supported loops have to be part of a loop nest, and their iteration must be governed by an induction variable. For loops with different types of termination conditions, generating the cloned loop and assigning results to temporary arrays becomes a highly non-trivial problem, which we have not yet investigated.

Secondly, this description of the technique only suits loops with induction variables that are initialized to 0 and subsequently incremented by 1 at each loop iteration.

```
1  double B[100][100] = ...;
2  s = 0;
3  for (i=0; i<100; i++) {
4    for (j=0; j<10; j++) {
5      for (k=0; k<100; k++) {
6        s += B[i][k] * B[k][i];
7        s += B[j][k] * B[k][j];
8      }
9    }
10 }
```

(a) Original loop nest.

```
1  double B[100][100] = ...;
2  s = 0;
3  for (i=0; i<100; i++) {
4    for (k = 0; k < 100; k++) {
5    }
6    for (j=0; j<10; j++) {
7      for (k=0; k<100; k++) {
8        s += B[i][k] * B[k][i];
9        s += B[j][k] * B[k][j];
10     }
11   }
12 }
```

(b) After insertion of cloned loop.

```
1  double B[100][100] = ...;
2  s = 0;
3  for (i=0; i<100; i++) {
4    double t[100];
5    for (k = 0; k < 100; k++) {
6      t[k] = B[i][k] * B[k][i];
7    }
8    for (j=0; j<10; j++) {
9      for (k=0; k<100; k++) {
10       s += B[i][k] * B[k][i];
11       s += B[j][k] * B[k][j];
12     }
13   }
14 }
```

(c) After generalized hoisting.

```
1  double B[100][100] = ...;
2  s = 0;
3  for (i=0; i<100; i++) {
4    double t[100];
5    for (k = 0; k < 100; k++) {
6      t[k] = B[i][k] * B[k][i];
7    }
8    for (j=0; j<10; j++) {
9      for (k=0; k<100; k++) {
10       s += t[k];
11       s += B[j][k] * B[k][j];
12     }
13   }
14 }
```

(d) After replacing original references.

Figure 4.1: Example application of GLICM at source level.

4.2 Original motivation

Generalized loop-invariant code motion was introduced by Luporini et al. in [27]. Their paper documents the improvements brought by this optimization and others in the context of the finite element method, a numerical technique for approximating solutions of partial differential equations (PDEs), with wide applications in computational science and simulation. In finite element analysis, the full domain of a PDE is discretized into many smaller regions, on which complex mathematical operations (*kernels*) are applied. This process is known as local assembly. The results of local computations are then combined into a global system of equations, which, when solved, yields the final approximation.

A novel and increasingly popular approach for consistently achieving efficient, high-performance numerical solutions is to automatically generate the local assembly code from an abstract mathematical specification of the problem, using a tool such as Firedrake [6]. The generated code typically includes a multiple nested loop enclosing a complex chain of arithmetic expressions in the innermost loop, such as the one in Figure 4.2.

For this variety of loop nests, the authors noted that popular compilers such as the Intel

compiler and gcc perform suboptimally, since they hoist expressions from the innermost loop only if they do not depend on the `k` variable [27, p. 7]. This limitation prevents many mathematical expressions in the `k`-loop from being precomputed in advance. Since local assembly operations are executed many times over (on every element in the discretized domain of the PDE), missing these hoisting opportunities prevents significant performance gains.

To mitigate the shortcomings faced by general-purpose compilers on this type of programs, [27] introduces GLICM as one of several optimizations performed on local assembly code after generation. In the experiments presented in the paper [27, p. 14], the execution time of several finite element calculations is reduced by a factor of up to 3 as a result of GLICM, showing the applicability and potential of this optimization.

```
1  for (int i = 0; i < 14; ++i) {
2    for (int j = 0; j < 10; ++j) {
3      for (int k = 0; k < 10; ++k) {
4        A[j][k] += (((D0[i][k] * D0[i][j]) + (((a2 * D3[i][k]) +
5            (a5 * D2[i][k]) + (a8 * D1[i][k])) *
6            ((a2 * D3[i][j]) + (a5 * D2[i][j]) +
7            (a8 * D1[i][j]))) + (((a1 * D3[i][k]) +
8            (a4 * D2[i][k]) + (a7 * D1[i][k])) * ((a1 *
9            D3[i[j]) + (a4 * D2[i][j]) + (a7 *
10           D1[i][j]))) + (((a0 * D3[i][k]) + (a3 *
11           D2[i][k]) + (a6 * D1[i][k])) * ((a0 *
12           D3[i][j]) + (a3 * D2[i][j]) + (a6 *
13           D1[i][j]))))) * det * W[i]);
14     }
15   }
16 }
```

Figure 4.2: Generated local assembly code for a Helmholtz equation (after simplifying variable names for clarity).

4.3 Trade-offs

This section documents the advantages and disadvantages of generalized loop-invariant code motion. Because this technique is aggressive in the way it modifies a program, by adding new loops and allocating multiple arrays, balancing between the various trade-offs becomes an important task.

4.3.1 Benefits

Precomputing invariant expressions

The main benefit introduced by GLICM is that it prevents the redundant evaluation of invariant expressions belonging to multiply-nested loops. Instead, these expressions are evaluated at an appropriate program point and their results made available in the original loop through a temporary array.

As we saw in the previous section, GLICM offers significant performance improvements when applied to arithmetic-intensive loop nests, where the largest fraction of the execution time is devoted to computing arithmetic operations in the CPU. This is due to the high execution costs of these operations (especially multiplications and divisions on floating-point numbers) relative to other instructions. Another important source of runtime improvement stems from precomputing the results of calls to complex functions.

Recall that generalized code motion is applied to the current loop and some chosen parent loop, which does not necessarily have to be the immediately enclosing loop. To quantify exactly how much redundant computation is avoided, we will consider a general case, whereby we are processing a loop lying at depth i in a multiple loop nest. Thus, our loop (we shall refer to it as L_i) is enclosed in $i-1$ outer loops, $L_1, L_2 ... L_{i-1}$.

Assume we choose $L_j, 1 \leq j \leq i-1$ as the parent loop of L_i. Any statements of L_i which are invariant with respect to L_j (according to our definition in Section 4.1) will be moved by GLICM before the entry to L_j and wrapped in a cloned loop with the same iteration pattern as L_i. If we take n_k to be the number of times loop L_k executes (its trip count), the instructions hoisted by generalized code motion in the cloned loop will be executed $n_1 * ... * n_{j-1} * n_i$ times in the transformed program. By contrast, they were executed $n_1 * ... * n_{j-1} * n_j * ... * n_i$ in the original program. The number of times these statements are evaluated has thus been reduced by a total factor of $n_j * ... * n_{i-1}$.

Figure 4.3a presents a slightly modified version of the code we used as an example when we presented GLICM. The updated pseudocode now includes a new statement on line 9, `A[k] * B[k][k]`, which is invariant not only with respect to the `j`-loop, but also with respect to the `i`-loop, which means it can be hoisted outside of it. Figure 4.3b shows the program code after applying generalized code motion on the `k`-loop, considering both the `i`-loop and the `j`-loop as potential parent loops.

In the original version of the program, all the statements of the `k`-loop would have been executed 10^5 times. After the transformation, the statement on line 7, hoisted in the preheader of the `j`-loop, is executed 10^4 times, while the statement on line 9, hoisted in the preheader of the outermost loop, is executed 10^2 times. The reader can verify that the total number of evaluations for both statements was reduced by a factor which corresponds to the formula given above.

Better cache performance

Significant increases in cache performance can be achieved as a result of GLICM if the unoptimized loop contains large invariant sub-expressions which include many memory accesses. By hoisting these statements, memory accesses are moved towards the outer loops in the nest, and are replaced by a single access to the one-dimensional array storing the expression's result. This frees up the CPU cache in the original loop, creating more space for other data items that may be required by unhoistable statements.

```
1  double B[100][100] = ...;
2  double A[100] = ...;
3  s = 0;
4  for (i=0; i<100; i++) {
5    for (j=0; j<10; j++) {
6      for (k=0; k<100; k++) {
7        s += B[i][k] * B[k][i];
8        s += B[j][k] * B[k][j];
9        s += A[k] * B[k][k];
10     }
11   }
12 }
```

(a) Original loop.

```
1  double B[100][100] = ...;
2  double A[100] = ...;
3  s = 0;
4  double t1[100];
5  for (k=0; k<100; k++) {
6    t1[k] = A[k] * B[k][k];
7  }
8  for (i=0; i<100; i++) {
9    double t2[100];
10   for (k=0; k<100; k++) {
11     t2[k] = B[i][k] * B[k][i];
12   }
13   for (j=0; j<10; j++) {
14     for (k=0; k<100; k++) {
15       s += t2[k];
16       s += B[j][k] * B[k][j];
17       s += t1[k];
18     }
19   }
20 }
```

(b) After generalized loop-invariant code motion.

Figure 4.3: Applying GLICM with respect to two different parent loops.

Enabling automatic vectorization

Automatic vectorization is a popular approach for exploiting parallelism in a program. This technique identifies code regions containing instructions that operate on single data elements, and rewrites them to use SIMD (Single Instruction, Multiple Data) instructions, which process *vectors* of multiple elements at the same time [24]. An important requirement of vectorization is that there are no data dependencies between the vectorized instructions, since in this case they may not be executable in parallel.

Loops are a typical target for automatic vectorization, because they often process collections of elements sequentially. The cloned loop introduced by GLICM is a good candidate for vectorization, because there are no dependencies between executions of the same instruction during consecutive loop iterations. Otherwise, these instructions would not be invariant, and would have never been hoisted in the first place. If the cloned loop is vectorized, the invariant statements will be executed not only fewer times, but also faster in the optimized program.

4.3.2 Disadvantages

Storage costs

Generalized code motion creates additional storage constraints on the program, since it may allocate several temporary arrays at multiple program points for each loop it processes. To avoid quickly running out of memory or stack space (depending on where the arrays are allocated), an upper bound must be placed on the amount of total storage space introduced by the transformation. This could be achieved by setting the threshold at a percentage of the total memory used by the program, allowing the user to control

the amount of space GLICM is allowed to use, or relying on heuristic methods.

Runtime penalties

While our optimization can achieve impressive speed-ups for some loops, it can equally deteriorate execution times if applied in the wrong manner. If the hoisted invariant expressions are trivial (they are composed of few primitive operations) and inexpensive (e.g. bitwise operations, comparisons), precomputing them in an additional loop and then loading them from an array may actually be slower than recomputing them on the spot. GLICM should thus be driven by a cost model which avoids these cases, and only moves instructions out from a loop when it is profitable to do so.

In Section 4.3.1, we argued that temporary arrays may improve performance by decreasing memory traffic in the loop and freeing up the CPU cache. However, the reverse could happen if invariant instructions operating on scalar values in the original loop are replaced with references to many temporary arrays. Portions of these arrays will be brought into the cache during execution, which leaves less space for storing other memory items accessed inside the loop.

Compilation cost

On its own, GLICM is a costly transformation, because it requires alias analysis (and possibly reaching definitions analysis) to be performed on multiple regions of the control flow graph. When applied as a step in a pipeline of optimizations, generalized code motion may also affect the time required by subsequent transformations, since it increases the size of the program.

Increased compilation times are an inevitable downside of any compiler optimization, and as such we regard this factor as less important compared to the potential improvements in execution time. Nonetheless, implementations of GLICM should strive to minimize compilation overhead through the use of efficient algorithms and data structures.

4.4 Generalizing safety requirements

To preserve program behaviour, GLICM needs to assess the safety of any potential transformations before modifying the program's IR. This section describes an important contribution of the project, namely a set of safety considerations we have developed, which ensure that GLICM can be extended to safely operate on arbitrary general-purpose programs. Since our transformation generalizes loop-invariant code motion, these guidelines are similarly an extension of the requirements of LICM, outlined in Section 2.5.3.

Loop rotation

Firstly, loop rotation must be applied to all loops in the unoptimized program. The `do-while` formulation provided by this transformation will be useful when considering which invariant expressions can be appropriately hoisted. Figure 4.4 revisits loop rotation, showing how the technique is applied on a modified version of the loop nest given in Figure 4.1.

```
1  double B[n][n] = ...;
2  s = 0;
3  for (i=0; i<n; i++) {
4    for (j=0; j<n; j++) {
5      for (k=0; k<n; k++) {
6        s += B[i][k] * B[k][i];
7        s += B[j][k] * B[k][j];
8      }
9    }
10 }
```

(a) Original loop.

```
1  double B[n][n] = ...;
2  s=0;
3  i=0;
4  if (i<n) {
5    do {
6      j=0;
7      if (j<n) {
8        do {
9          k=0;
10         if (k<n) {
11           do {
12             s+=B[i][k]*B[k][i];
13             s+=B[j][k]*B[k][j];
14             k=k+1;
15           } while (k<n);
16         }
17         j=j+1;
18       } while (j<n);
19     }
20     i=i+1;
21   } while (i<n);
22 }
```

(b) After loop rotation.

Figure 4.4: Applying loop rotation on a multiple nested loop.

Hoisting instructions which are guaranteed to execute

A key idea which we stressed when discussing the safety of code motion is that invariant statements should be executed in the optimized program only if they are executed in the unoptimized version. For LICM, we stated that an instruction can be safely hoisted if its basic block dominates all the exit blocks of its loop (Section 2.5.3). Equivalently, the expression should be guaranteed to execute whenever its loop executes. For generalized code motion, this condition is not sufficient.

To understand why, consider the code in Figure 4.5a, and assume GLICM is used to hoist invariant statements from the `j`-loop to the preheader of the `i`-loop. Here, the expression `B[j] * B[m]` is invariant with respect to the parent loop, and it is guaranteed to execute in the `j`-loop. The optimized program produced by GLICM is given in Figure 4.5b. The problem with hoisting this expression is that, when `m` is not an integer between 0 and 9, `B[j] * B[m]` is not executed in the original program (because the `j`-loop will never be entered), but it is executed in the optimized program. The optimized program will access invalid memory in the `B` array, leading to undefined behaviour which is not present in the unoptimized code.

The generalized hoisting procedure should only move invariant statements which are guaranteed to execute not only in their loop, but in *the parent loop*. In control flow graph terminology, this requires the invariant instructions to *dominate all the exit blocks of the parent loop* after loop rotation was applied. Thus, if there exists a path in the parent loop along which an invariant statement of the current loop is not executed, that statement should not be hoisted.

```
1  int m = atoi(argv[1]);
2  int A[10], B[10];
3  ...
4  for (int i=0; i<10; i++) {
5    if (i == m) {
6      for (int j=0; j<10; j++) {
7        A[i]+=B[j]*B[m];
8      }
9    }
10 }
```

(a) Original loop nest.

```
1  int m = atoi(argv[1]);
2  int A[10], B[10];
3  ...
4  int t[10];
5  for (int j = 0; j < 10; j++) {
6    t[j] = B[j]*B[m];
7  }
8  for (int i=0; i<10; i++) {
9    if (i == m) {
10     for (int j=0; j<10; j++) {
11       A[i]+=t[j];
12     }
13   }
14 }
```

(b) After GLICM.

Figure 4.5: Applying GLICM on a loop guarded by a conditional statement inside the parent loop.

Previously, we saw that statements may not be guaranteed to execute due to a guarding conditional statement in the current loop. The wider implication of this safety requirement is that if it is possible for the current loop, together with any of its superloops up to (but excluding) the parent loop, to not execute at all during an execution of the parent, then no invariant instructions should be moved by generalized code motion. If such statements were hoisted, they would be executed in the optimized program and not in the original one. Thus, GLICM can safely process a given loop if it can determine at compile-time that it executes whenever the chosen parent loop does.

Consider the pseudocode in Figure 4.4b, which shows a multiple nested loop after loop rotation. Suppose we wish to hoist instructions out of the `k`-loop to the preheader of the `j`-loop. Before moving the invariant expression `B[i][k] * B[k][i]` to this program point, we must ensure that it is executed whenever the parent loop is entered (i.e. whenever `0 < n`). The only other conditional guarding the expression (`k < n`) is the initial test for entering the `k`-loop. Since `k` is also initialized to 0, and the value of `n` is not modified, the current loop is guaranteed to execute whenever the parent loop is entered. Thus, the expression can be hoisted safely.

Exceptions caused by loop instructions

Generalized code motion should not move invariant expressions to some parent loop's preheader if are any statements between their program point and the entry to the parent loop which may throw exceptions. This ensures that the optimized program does not have altered exception behaviour, and that it does not execute more instructions than necessary when the loop nest exits early.

Aliasing

When discussing the safety of LICM in Section 2.5.3, we stressed that we must avoid hoisting expressions which read data from memory if the same memory location may be written to by other statements in the loop. In other words, we must use pointer aliasing

information when considering to hoist statements that involve memory.

In generalized code motion, we rely on alias analysis for avoiding the same type of hazards, but we require a larger area of the CFG to be inspected. Assume our transformation is currently processing a loop `L` inside a loop nest, and let `P` denote the parent loop we have chosen as a hoisting point. Further assume that we identified an expression of `L` which is invariant with respect to `P` and reads data from a memory location α. Generalized code motion can only hoist this expression out to `P`'s preheader if there is no other statement in the entire body of `P` which may write to α.

Consider the C code snippet in Figure 4.6. Here, pointers `A` and `B` are aliases of each other, since they point to the same memory. Consider the sub-expression `A[k] * A[k]` in the `k`-loop, which is invariant with respect to the outermost loop. In theory, this expression could be precomputed by GLICM outside of the `i`-loop. However, the statement `B[i] = A[i] * A[i]` on line 6 writes to memory locations which our invariant statement reads. Based on this insight, GLICM will choose to not move the invariant expression out to the preheader of the `i`-loop. Had the instruction been hoisted, the optimized code would have produced a different value in `s` following the execution of the nested loop, altering the program's intended behaviour.

```
1  double *A = malloc(n * sizeof(double));
2  double *B = A;
3  double s = 0;
4  ...
5  for (i = 0; i < n; i++) {
6    B[i] = A[i] * A[i];
7    for (j = 0; j < n; j++) {
8      for (k = 0; k < n; k++) {
9        s += A[k] * A[k];
10     }
11   }
12 }
```

Figure 4.6: Pointer aliasing in a multiple loop nest.

4.5 Comparison with loop interchange and LICM

The single most important goal underlying our optimization is to avoid more redundant recomputation than typical loop-invariant code motion is able to achieve. Generalized loop-invariant code motion is the methodology we developed to reach this end, but other transformations may be used to achieve similar results.

In some cases, loop interchange followed by loop-invariant code motion is one such alternative. Figure 4.7 compares the optimized programs obtained by applying both this sequence of transformations and GLICM to a simple loop nest. In Figure 4.7c, the `i`- and `j`-loops are interchanged, which causes an expression to become invariant with respect to its loop, and thus hoisted by the subsequent code motion pass. For this particular loop

nest, the combination of loop interchange and LICM seems to produce superior code, because a single scalar value (as opposed to an array) is used for storing the result of the hoisted sub-expression.

```
1  double A[100][100];
2  double B[100][100];
3  ...
4  for (int i=0; i<100; i++) {
5    for (int j=0; j<100; j++) {
6      B[i][j]+=A[j][j]*A[99-j][j];
7    }
8  }
```

(a) Original code.

```
1  double A[100][100];
2  double B[100][100];
3  ...
4  double t[100];
5  for (int j=0; j<100; j++) {
6    t[j] = A[j][j]*A[99-j][j];
7  }
8  for (int i=0; i<100; i++) {
9    for (int j=0; j<100; j++) {
10     B[i][j]+=t[j];
11   }
12 }
```

(b) After GLICM.

```
1  double A[100][100];
2  double B[100][100];
3  ...
4  for (int j=0; i<100; i++) {
5    double t=A[j][j]*A[99-j][j];
6    for (int i=0; j<100; j++) {
7      B[i][j]+=t;
8    }
9  }
```

(c) After loop interchange and LICM.

Figure 4.7: Comparison between GLICM and loop interchange followed by LICM.

In general, however, GLICM is not substitutable by this sequence of two optimizations. Firstly, loop interchange is only applicable to *perfect loop nests* [2, p. 362], i.e. nests in which only the innermost loop contains code statements. GLICM does not suffer from this limitation. Secondly, GLICM generally helps avoid more redundant computation than loop interchange combined with loop-invariant code motion.

Consider the code in Figure 4.8a. The expression on line 5 is loop-invariant, so it can be hoisted out of the `j`-loop. However, once this is done, the expression on line 6 will remain stuck in the loop, since it is not invariant and we cannot perform loop interchange any further, due to compromising our perfect loop nest. Alternatively, if we begin by interchanging the two loops, the expression on line 6 will become invariant with respect to the `i`-loop, now enclosing it. Even though we are now able to hoist this expression, the overall situation is no better, since now the statement on line 5 is stuck in the innermost loop.

In conclusion, no matter the order in the loop nest, only a single expression at a time can be hoisted out of the innermost loop. An optimal outcome (shown in Figure 4.8b) is achieved instead by hoisting one of the expressions using LICM and the other one using

GLICM.

```
1  int A[100];
2  int x=0, y=0;
3  for (int i=0; i<100; i++) {
4    for (int j=0; j<100; j++) {
5      x+=A[i][i]*A[n-i][n-i];
6      y+=A[j][n-j]*A[n-j][j];
7    }
8  }
```

(a) Original loop nest.

```
1  int A[100];
2  int x=0, y=0;
3  int t1[100];
4  for (int j=0; j<100; j++) {
5    t1[j]=A[j][n-j]*A[n-j][j];
6  }
7  for (int i=0; i<100; i++) {
8    int t2 = A[i][i]*A[n-i][n-i];
9    for (int j=0; j<100; j++) {
10     x+=t2;
11     y+=t1;
12   }
13 }
```

(b) After LICM and GLICM.

Figure 4.8: A loop nest where loop interchanging and LICM hoist a suboptimal number of invariant expressions.

Even for programs such as the one in Figure 4.7, where we can use both methods to hoist the same number of invariant instructions, the GLICM-optimized version might perform better in practice. Firstly, the cloned loop introduced by GLICM is vectorizable. If automatic vectorization is enabled, the hoisted operations will be executed using SIMD instructions, achieving a speed-up compared to the scalar versions. Secondly, loop interchanging might have other detrimental effects on the runtime of the program, such as increasing cache misses due to inefficient memory access patterns.

4.6 Possible extensions

More types of induction variables

In Section 4.1, we noted that GLICM operates only on non-innermost loops with a special form of induction variable, which is initialized by 0 and incremented by 1 after each iteration. This restriction simplifies the task of storing precomputed results, since it allows us to place the value of a sub-expression at the `ith` iteration of the cloned loop into the `ith` element of the appropriate temporary array. We propose two distinct approaches which could be used to make GLICM applicable to loops with other types of induction variables:

- Invoke an optimization dedicated to rewriting loops prior to applying GLICM. This optimization would attempt to transform loops to a form which uses the supported type of induction variable.
- GLICM could use an additional analysis module to compute the trip count of loops which use unsupported types of induction variables. This information is necessary in order to know how much space to allocate for temporary arrays. For a given loop, GLICM would then proceed in its normal fashion, picking a suitable parent loop, identifying appropriate invariant expressions, and hoisting them to the cloned loop. As a new step in the process, GLICM would introduce a fresh induction variable, starting from 0 and incremented by 1 at each iteration of the cloned loop. This will be used for storing results computed in the cloned loop into temporary arrays. Another fresh induction variable with the same iteration pattern is introduced in the original loop, in order to load the precomputed results in the correct order. Figure 4.9 illustrates how GLICM could be applied using this extended methodology.

```
1 int s=0;
2 int A[30];
3 for (int i=0; i<10; i++) {
4   for (int j=0; j<30; j+=3) {
5     s+=A[k]+A[k/3];
6   }
7 }
```

(a) Original loop nest.

```
1  int s=0;
2  int A[30];
3  int t[10];
4  int fresh1=0;
5  for (int j=0; j<30; j+=3) {
6    t[fresh1]=A[k]+A[k/3];
7    fresh1++;
8  }
9  for (int i=0; i<10; i++) {
10   int fresh2=0;
11   for (int j=0; j<30; j+=3) {
12     s+=t[fresh2];
13     fresh2++;
14   }
15 }
```

(b) After GLICM.

Figure 4.9: Applying GLICM on a loop with a non-standard induction variable.

Nested cloned loops and multi-dimensional temporary arrays

When we described the methodology of generalized code motion in Section 4.1, we said that an expression in a loop `L` is invariant with respect to a parent loop `P` if each of its operands is either defined outside of `P`, or is the induction variable of `L`. We may generalize this definition to allow for the presence of more than one induction variable among an invariant expression's operands.

We propose the following amended notion of invariance: an expression in a loop `L` is invariant with respect to a parent loop `P` if each of its operands is either defined outside of `P`, or is the induction variable of `L`, *or is the induction variable of a superloop of `L` which is also a subloop of `P`*. As a result of this extension, generalized code motion can now hoist even more expressions to outer levels of loop nests, at the cost of multi-dimensional temporary arrays and nested cloned loops.

In Figure 4.10a, we revisit the code fragment we used in our initial exposure of GLICM. We note that the sub-expression `B[j][k] * B[k][j]` in the `k`-loop (the current loop) is invariant with respect to the `i`-loop (the parent loop) in the light of our revised definition of invariance. This is because `k` is the induction variable of the current loop, and `j` is the induction variable of the `j`-loop, which is both a superloop of the current loop and a subloop of the parent loop. This expression is thus hoistable, but because it contains induction variables of two loops, we will require a nested cloned loop and a two-dimensional temporary array for holding its evaluation. The transformed code is shown in Figure 4.10b.

Although we acknowledge the feasibility of extending GLICM in this direction, we have not fully investigated the performance and safety implications of this generalization. Thus, we can only list this development as a possible, but not necessarily worthwhile, vector of future exploration.

```
1  double B[100][100] = ...;
2  s = 0;
3  for (i=0; i<100; i++) {
4    for (j=0; j<10; j++) {
5      for (k=0; k<100; k++) {
6        s += B[i][k] * B[k][i];
7        s += B[j][k] * B[k][j];
8      }
9    }
10 }
```

(a) Original loop

```
1  double B[100][100] = ...;
2  s = 0;
3  double[10][100] t;
4  for (j=0; j<10; j++) {
5    for (k=0; k<10; k++) {
6      t[j][k] = B[j][k] * B[k][j];
7    }
8  }
9  for (i=0; i<100; i++) {
10   for (j=0; j<10; j++) {
11     for (k=0; k<100; k++) {
12       s += B[i][k] * B[k][i];
13       s += t[j][k];
14     }
15   }
16 }
```

(b) After generalized loop-invariant code motion

Figure 4.10: Storing precomputed results in two-dimensional temporary array.

Chapter 5

Implementation

This chapter treats the implementation of generalized code motion in the LLVM project. Firstly, we motivate our choice of compiler, and briefly describe the main development tools we used. We proceed to pinpoint the location of our implementation in the LLVM core libraries, highlight key aspects of the pass registration process (discussed in Section 3.3) and discuss GLICM's pass dependencies. The core sections of this chapter describe how our implementation guarantees the safety requirements of the transformation, and the methodology of the generalized code hoisting process. Since our implementation relies heavily on the LLVM libraries and is thus not self-contained, we will favour high level descriptions to source code listings throughout the chapter.

5.1 Choice of compiler

Our decision to use LLVM as the platform for implementing generalized code motion was influenced by many factors. Among these, we list the following:

- LLVM is arguably the most extensive and successful open-source compiler framework. It supports a variety of source languages and target architectures, and its state-of-the-art optimizer is the building block of Clang, which is in turn among the most efficient C and C++ compilers.
- The official documentation is clear, rich and well-structured, offering detailed guidance on how to obtain, build and use the source code of the project. This alleviates most of the initial learning curve caused by installing and setting up a system as large as LLVM.
- The project's source code is generally well-documented, well-organized and well-tested. The codebase is fairly easy to navigate, and the automatically generated online documentation is helpful for untangling the relationships between code components.
- Although the LLVM project is arguably biased towards C-based languages, its intermediate representation is designed to express generic source programming languages. Thus, even though our optimization is mostly motivated by C-like languages, LLVM provides a platform for potentially integrating it with other front-ends.
- Since LLVM is under active development, the implementation of new optimization techniques is open to public scrutiny from an expert community. This could lead

to improvements and refinements in our technique, should it be included in the main repositories, as well as increased interest in the area.

- The LLVM project uses a comprehensive test suite, which contains both tiny, single-file programs and large, full-fledged applications. This provides us with a way to evaluate our technique in terms of correctness, impact on runtime performance and applicability in different domains.
- Integrating a new optimization in the LLVM core libraries is done with minimum overhead, due to the inherent modularity in the design of the optimizer (see Section 3.3).

5.2 Development tools

Version control

The source code of all the official LLVM projects is maintained using Subversion repositories. In addition, automatically updated Git mirrors of these repositories are provided, either hosted on LLVM's own Git servers, or on Github. Due to the seamless integration provided by LLVM, we preferred to use Git over Subversion for version control, due to our familiarity with the former.

To set up a working LLVM environment, we required four different Git repositories, as per the official starting guide [10]. These are used to host:

- Clang;
- `compiler-rt` (a library supporting code generation);
- The LLVM test-suite;
- The LLVM core libraries.

Since we needed the first three repositories for building and testing LLVM, but otherwise did not intend to modify any of their source files, we cloned them directly from the LLVM Git servers. The implementation of generalized code motion would live entirely in the core libraries, so we only required changes in the latter repository to be versioned. Our approach was to create a fork of the repository's Github mirror[1], and develop our pass on a separate branch, constantly merging it against the most recent changes.

IDE

The LLVM project is entirely implemented in C++, and its build process relies on an infrastructure of hierarchical Makefiles. Although development on the command-line is possible, we immediately acknowledged the positive impact a good IDE would provide on such a large codebase. Fortunately, LLVM supports the use of CMake[2], a tool which can automatically generate project files for many IDEs.

Initially, we attempted to use Eclipse CDT, an extension of Eclipse for C and C++. Although Eclipse CDT is abundant with features, we were confronted with an extremely

[1] https://github.com/llvm-mirror/llvm
[2] http://www.cmake.org/

slow and cumbersome build cycle, and unacceptable source file indexing times. After several attempts at improving performance, we switched to Code::Blocks[3], a much lighter IDE written in C++. Although it may suffer from occasional buggy behaviour, Code::Blocks compensates by being much more responsive and using less than half the memory needed by Eclipse CDT.

5.3 Pass setup

5.3.1 Organization

Our implementation of generalized loop-invariant code motion consists of a single C++ source file, `GLICM.cpp`. The location of this file relative to (a simplified view of) the LLVM codebase is given in Figure 5.1. Together with many other scalar optimizations (techniques that focus on improving execution speed on a single thread [23, p. 539]), generalized code motion lives in the `Scalar` subdirectory of `Transforms`, which hosts all transformation passes.

Figure 5.1: Location of generalized loop-invariant code motion in the LLVM codebase.

Before deciding to implement GLICM as a separate optimizer pass, we contemplated the possibility of extending the existing implementation of loop-invariant code motion (`LICM.cpp`). This was motivated by similarities in the methodology of the two optimizations, and the fact that LICM provided helper functions and methods that we would also require. We ultimately decided against this approach based on several considerations:

- The implementation of loop-invariant code motion is highly non-trivial, and our

[3] http://www.codeblocks.org/

own code would have added extra complexity to it, affecting readability and maintainability. Furthermore, we would have had to extend many functions and methods for supporting the generalized case, which faced the risk of introducing accidental bugs.

- Generalized code motion modifies a program's control flow graph by creating new basic blocks for the cloned loop, whereas LICM only moves instructions from one basic block to another. Hence, although both optimizations serve the same purpose, they affect the program structure in radically different ways. Combining the two transformations in a unified pass would have conflicted with LLVM's design philosophy.

- The LICM pass is invoked liberally in optimization sequences, since it invariably speeds up programs and has a small processing overhead. For instance, in the complex optimization pipeline employed by Clang, LICM may be invoked up to 4 times[4]. For generalized code motion, a single invocation at the right place in the sequence should be sufficient for hoisting invariant expressions unsupported by LICM. With a single pass for both transformations, generalized code motion would be attempted an excess number of times, causing slowdowns in compilation times and little, if any, additional runtime gains.

- We would have been burdened by many merge conflicts in `LICM.cpp` whenever syncing our forked repository with the main branch of LLVM.

5.3.2 Pass registration and dependencies

In `GLICM.cpp`, the generalized code motion pass is declared as a `struct` that inherits from `LoopPass`, a class containing base functionality for all passes that process loops. The `GLICM` structure contains the private fields and methods needed across the lifetime of our transformation, and overrides two crucial virtual methods:

- `getAnalysisUsage`, which specifies a list of passes that must be run prior to invoking GLICM.

- `runOnLoop`, which describes the optimization process.

To register the pass with the pass registry (discussed in Section 3.3) and to initialize its pass dependencies, several special macros are used. Figure 5.2 illustrates the registration of our transformation, which specifies a short (`GLICM`) and a descriptive (`Generalized Loop-Invariant Code Motion`) name, as well as a command-line flag (`glicm`) for invoking our pass using `opt`.

Macros are also used to initialize the five pass dependencies of GLICM, which are declared as prerequisites in `getAnalysisUsage`:

- `DominatorTreeWrapperPass`, a pass which computes a dominator tree for the program.

- `LoopInfoWrapperPass`, an analysis pass that collects information about natural loops in the program. It exposes several useful utility functions that we use in our implementation.

[4]See http://llvm.org/docs/doxygen/html/PassManagerBuilder_8cpp_source.html.

```
1  INITIALIZE_PASS_BEGIN(GLICM, "glicm",
2                        "Generalized Loop-Invariant Code Motion",
3                        true, /* Modifies the CFG of the function */
4                        false) /* Is not an analysis pass */
5  INITIALIZE_PASS_DEPENDENCY(DominatorTreeWrapperPass)
6  INITIALIZE_PASS_DEPENDENCY(LoopInfoWrapperPass)
7  INITIALIZE_PASS_DEPENDENCY(LoopSimplify)
8  INITIALIZE_PASS_DEPENDENCY(ScalarEvolution)
9  INITIALIZE_AG_DEPENDENCY(AliasAnalysis)
10 INITIALIZE_PASS_END(GLICM, "glicm", "GLICM", true, false)
```

Figure 5.2: Pass registration and initialization of pass dependencies.

- `LoopSimplify`, a transformation which ensures that every loop has a preheader where instructions can be safely hoisted.
- `ScalarEvolution`, an analysis pass which provides useful information about loop induction variables.
- `AliasAnalysis`, an interface for alias analysis. As opposed to the other dependencies in this list, which represent passes with an unique implementation, alias analysis is an analysis group (AG) in LLVM terminology, an interface that is implemented by multiple passes. By specifying this form of dependency, we require the results of some implementation of alias analysis to be made available to GLICM, without dictating the use of a specific analysis technique.

We note the absence of loop rotation (which we discussed in Sections 2.5.3 and 4.4) in the list of dependencies. This transformation, implemented by the `LoopRotate` pass, cannot be scheduled before other passes using `getAnalysisUsage` due to its current configuration. To address this, loop rotation should be explicitly scheduled before generalized code motion in the pipeline of optimizations.

5.4 Pass implementation

All LLVM `LoopPasses` must supply an implementation of the `runOnLoop` method, which contains all the optimization's logic. When the given pass is scheduled to run on an LLVM module, this method will be called for every loop of every function in that module. Loops in the same function are processed in a top-to-bottom, innermost-first fashion.

In our implementation of generalized loop-invariant code motion, `runOnLoop` performs the following tasks:

1. Checks that the current loop meets the required conditions for processing.
2. Finds statements that can be hoisted to the parent loop.
3. Uses a profitability model to filter out statements that should not be hoisted.
4. Creates the cloned loop, if there are statements remaining after the previous step.

5. Hoists invariant instructions to the cloned loop, storing results in temporary arrays and replacing references in the original loop.

Compared to the description presented in Chapter 4, our implementation suffers from two main limitations:

1. Only the immediate superloop of the current loop is considered as a potential hoisting point. This restriction was introduced in the early stages of implementation, and time constraints later prevented us from rigorously implementing and testing the more general case. We will use the term *parent loop* to refer to the immediate parent of the current loop in the remainder of this section.

2. When processing a loop with GLICM, we rely on knowing loop's trip count for creating the cloned loop and allocating temporary arrays. Unfortunately, the LLVM libraries do not provide a reliable way of obtaining the upper bound of a loop's iteration variable unless that bound is constant. Thus, we only apply generalized code motion to loops with constant trip counts.

5.4.1 Checking suitability of current loop

The first step of generalized code motion is to check that the currently inspected loop is suitable for the transformation. The code sample in Figure 5.3 shows the conditions our implementation tests for, given a current loop `L`:

- The current loop must have a parent loop.
- The parent loop must have a dedicated preheader block, where invariant instructions can be hoisted (this is checked by the condition on line 11).
- The current loop has a *canonical induction variable*, which starts at 0 and is incremented by 1 after each iteration. The pointer to the Phi function defining the loop's induction variable is provided by the `LoopInfoWrapperPass` analysis.
- The current loop's trip count must be known at compile-time. In the code sample, we obtain the value of the trip count from the `SCEV` object, which holds the results of the `ScalarEvolution` analysis.
- The body of the parent loop contains no exceptions. The `loopMayThrow` helper iterates over all the basic blocks of `ParentLoop`, searching for instructions that may throw exceptions. This is a coarser check compared to our discussion on exception behaviour in Section 4.4.
- The current loop is guaranteed to execute whenever the parent loop is executed. For this requirement, we implemented the `subloopGuaranteedToExecute` helper function, which checks that the current loop's header dominates all the exit blocks of the parent loop. The approach is similar to the technique described in Section 4.4, but checks that the loop itself (as opposed to individual instructions) is guaranteed to execute.

```
1  bool GLICM::runOnLoop(Loop *L, LPPassManager &LPM) {
2  ...
3  Loop *ParentLoop = L->getParentLoop();
4  PHINode *IndVar = L->getCanonicalInductionVariable();
5  unsigned TripCount = SCEV->getSmallConstantTripCount(L);
6  ...
7  if (!(ParentLoop &&
8        ParentLoop->isLoopSimplifyForm() &&
9        IndVar &&
10       TripCount > 0 &&
11       !loopMayThrow(ParentLoop) &&
12       subloopGuaranteedToExecute(CurLoop, ParentLoop, DT))) {
13   return false;
14 }
15 ...
16 }
```

Figure 5.3: Testing that GLICM is applicable for the current loop.

If not all of these conditions are met, GLICM cannot be applied to the current loop. In this case, the function returns `false`, signifying that the input IR program was left unchanged.

5.4.2 Conditions for hoisting an instruction

To decide which instructions of the current loop can be hoisted to the preheader of the parent loop, our implementation uses the helper method in Figure 5.4. This decision requires that several facts about the given instruction hold simultaneously:

- The instruction has loop-invariant operands with respect to the parent loop. Recall that the LLVM IR obeys the SSA property, meaning that each named value in a program is assigned to exactly one. We iterate over each of the instruction's operands and check that either (a) the operand is the induction variable of the current loop, (b) the operand is a constant, or (c) the (unique) instruction defining the operand lies outside the parent loop. If, for each of the instruction's operands, at least one of these conditions holds, the instruction is invariant in the sense required by GLICM.

- If the instruction is a `load`, it cannot be hoisted if it is marked as `volatile` (a directive which indicates that its location in the program should not be changed), or if it reads from a memory location which may be written to by another instruction in the parent loop. Also, `call` instructions cannot be hoisted if they write to memory locations which are read in the parent loop. Effectively, this ensures the aliasing constraints we stressed in Section 4.4. These checks are made by the `canSinkOrHoistInst` helper, which was initially a private method in the LICM pass, but was subsequently extracted to a shared header to promote code reuse. This method also disallows code hoisting altogether for certain categories of instructions, such as `phi` and branch statements.

  Of this method's arguments, the `CurAST` object is used for fulfilling aliasing queries. `CurAST` is a so-called alias set tracker (AST), an LLVM data structure

that constructs sets of possibly aliasing pointers [12]. AST's are initialized from an `AliasAnalysis` object, which holds the results of the alias analyses scheduled before the current pass, but which exposes a limited interface. Program regions of interest (such as basic blocks or individual instructions) can be subsequently added to an AST for processing, which causes existing alias sets to be updated and possibly new ones to be created. The AST used by our implementation is built from all the basic blocks of the parent loop, since this is the CFG region GLICM is responsible for.

- The instruction is guaranteed to execute in its current loop. Below, this is checked by `isSafeToExecuteUnconditionally`, yet another method in `LICM.cpp` that we were able to reuse. Together with the condition that the current loop itself is guaranteed to execute in the parent loop (as explained in the previous section), this ensures that the given instruction executes whenever the parent loop executes. Thus, hoisting it would preserve program behaviour.
- The identifier defined by the instruction is not used outside of the current loop. We defer the explanation of this item to Section 5.4.4, which discusses our approach to hoisting invariant instructions.

```
1 bool GLICM::canHoist(Instruction *I) {
2   return hasLoopInvariantOperands(I, ParentLoop) &&
3          canSinkOrHoistInst(*I, AA, DT, CurLoop, CurAST, SafetyInfo) &&
4          isSafeToExecuteUnconditionally(*I, DT, CurLoop, SafetyInfo) &&
5          !isUsedOutsideOfLoop(I, CurLoop, LI);
6 }
```

Figure 5.4: Testing that a given instruction can be safely hoisted by generalized code motion.

5.4.3 Inserting the cloned loop

This section describes how our implementation accomplishes the insertion of an empty clone of the current loop in the program's control flow graph. We illustrate the application of the algorithm using an artificial example, which aims to insert a cloned loop with canonical induction variable `%i` and trip count 100 in the preheader of a parent loop. For convenience, we label the header and preheader blocks of the parent loop as `P.Header` and `P.Preheader` respectively. The CFG region we will be operating on is represented in the figure below.

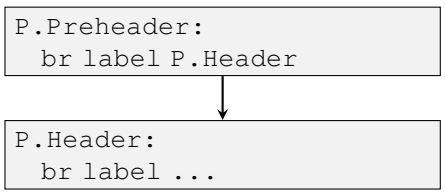


Figure 5.5: CFG region containing the preheader and header of the parent loop.

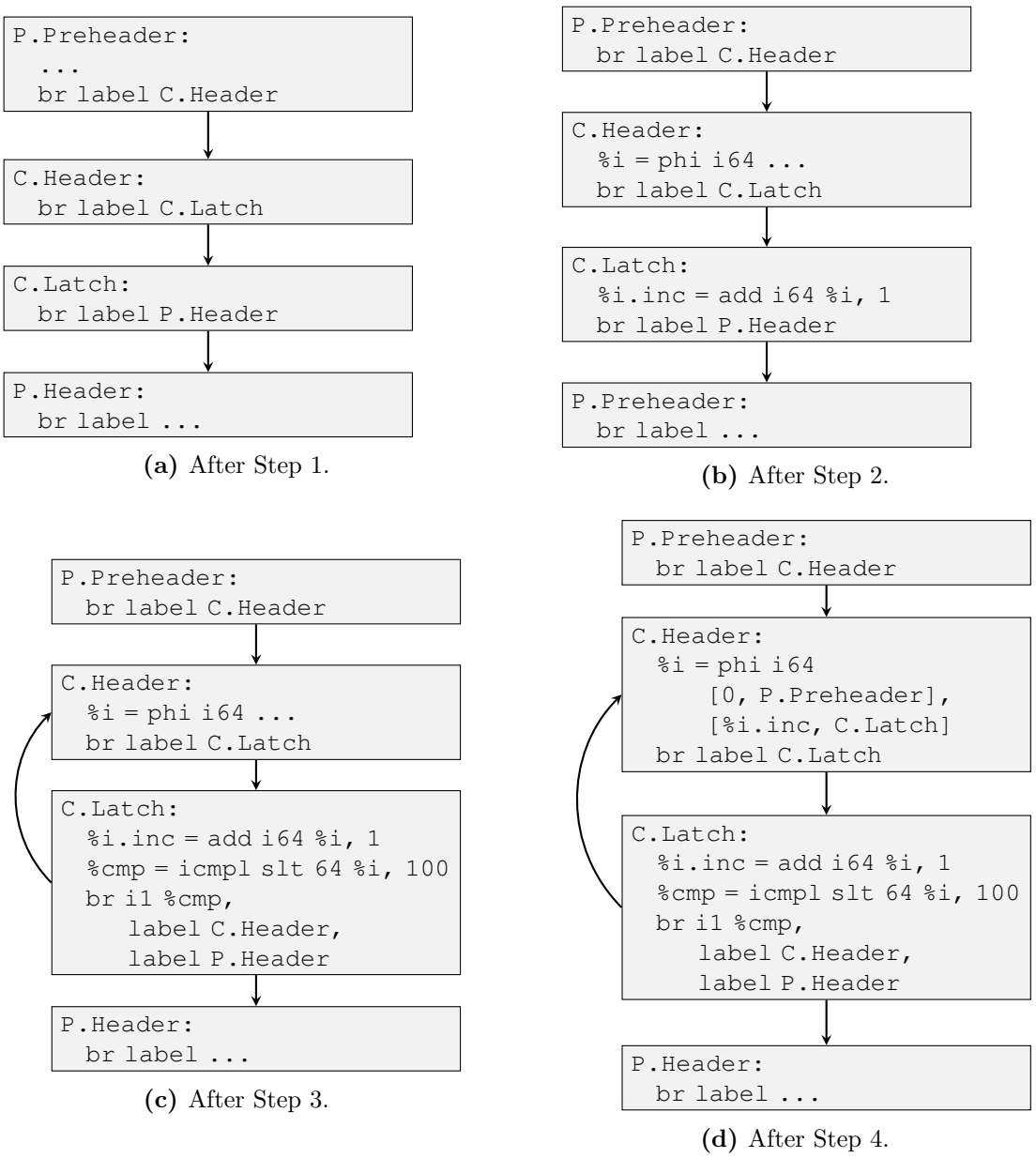
The algorithm consists of four main stages, and is tailored to the specific types of loops handled by generalized code motion, characterized by canonical induction variables and constant trip counts. The effect of each stage on the initial control flow graph is given in Figure 5.6.

Step 1 We *split the control flow edge* between `P.Preheader` and `P.Header`, creating a new basic block between the two. This block will be the header of the cloned loop (hence, we label it `C.Header`, and will contain all the invariant instructions hoisted from the current loop. Next, we split the edge between the newly created `C.Header` and `P.Header`, inserting yet another basic block. This basic block will be used to handle control flow in the cloned loop, and is labelled `C.Latch`.

Step 2 In the cloned loop header, we insert a Phi function corresponding to the cloned loop's induction variable, `i`. Momentarily, this Phi function is empty (i.e. it does not select between any values). In `C.Latch`, we create an instruction that increments the induction variable value by 1, writing the result to `%i.inc`.

Step 3 In this step, the edges of the control flow graph are manipulated in order to reorganize the two basic blocks we inserted so far into a proper loop. To achieve this, we first insert a comparison that tests whether `%i.inc` is less than the known trip count, 100. We change the unconditional branch at the end of `C.Latch` to a conditional one, which branches back to `C.Header` if the loop condition `%i.inc < 100` holds, or to `P.Header` if the cloned loop has finished execution.

Step 4 This step completes the definition of `%i` in Step 1. Control flow can arrive to `C.Header` from two sources. When the basic block is entered from `P.Preheader`, the cloned loop is at its first iteration, hence `%i` is initialized to 0. On the other hand, when control arrives from `C.Latch`, the basic block is re-entered following a previous iteration of the cloned loop. Thus, `%i` is set to `%i.inc`, the incremented iteration variable. This marks the end of the procedure.



(a) After Step 1.

(b) After Step 2.

(c) After Step 3.

(d) After Step 4.

Figure 5.6: The four steps of the cloned loop insertion algorithm.

5.4.4 Code motion methodology

We now describe the core of our implementation, the complete process which achieves the program transformations advertised by GLICM. To help the reader visualize the distinct stages of the hoisting process, we will conceptually apply generalized code motion to the C program in Figure 5.7. Our aim is to hoist the redundant sub-expression `A[j] * j` in the `j`-loop to a cloned loop placed in the preheader of its parent, the `i`-loop. A suitable LLVM intermediate representation of the program, containing only the CFG regions of interest to our optimization (the `i` and `j` loops, together with bordering basic blocks), is given in Figure 5.8.

```
1  int main() {
2    int A[100];
3    for (int k=0; k<100; i++) {
4       A[k] = k % 5;
5    }
6    int s = 0;
7    for (int i=0; i<100; i++) {
8      for (int j=0; j<100; j++) {
9        s+= A[j]*j;
10     }
11   }
12   printf("%d\n", s);
13 }
```

Figure 5.7: Source program used for outlining the stages in the implementation of generalized code motion.

Stage 1: Computing the hoistable set

Once we ascertain that the current loop is supported by generalized code motion (see Section 5.4.1), we begin our search for invariant expressions. We make use of the function's dominator tree (computed by the `DominatorTreeWrapperPass` pass dependency listed in Section 5.3.2) for traversing the basic blocks of the current loop. Starting from the loop header, we recursively visit the dominator tree in depth first order, stopping when we reach nodes lying outside of our loop. For each visited node, we inspect the instructions of the underlying basic block from top to bottom. This direction of traversal (both with respect to the loop's basic blocks and individual instructions in a single basic block) ensures that we visit variable definitions before the instructions that use them.

As GLICM processes a loop's basic blocks, it constructs and maintains a *hoistable set* of instructions. The hoistable set is a custom data type we implemented, a sequential container that iterates over its elements in insertion order, and also provides methods for membership testing. As instructions are inspected during the dominator tree traversal, they are added to the hoistable set if and only if they *could* be moved to a hypothetical cloned loop (subject to the constraints in Section 5.4.2), *provided that all other members of the hoistable set were hoisted prior to the instruction.* Using a hoistable set allows us to keep track of which instructions in the current loop would be potentially hoisted by GLICM, but without actually moving them.

In our example in Figure 5.8, the `j`-loop is defined by two basic blocks, `for.j.header` and `for.j.inc`. As the loop is traversed, the following definitions are added to the hoistable set, specifically in this order:

- `%arrayidx`: meets all hoisting requirements (uses the iteration variable of the current loop).
- `%1`: meets all hoisting requirements, provided that `%arrayidx` (now a member of the hoistable set) is hoisted.
- `%2`: meets all hoisting requirements (uses the iteration variable of the current loop)

- `%mul`: meets all hoisting requirements, provided that `%1` and `%2` (members of the hoistable set) are hoisted.
- `%j.inc`: meets all hoisting requirements (uses the iteration variable of the current loop).
- `%exitcond.j`: meets all hoisting requirements, provided that `%j.inc` is hoisted.

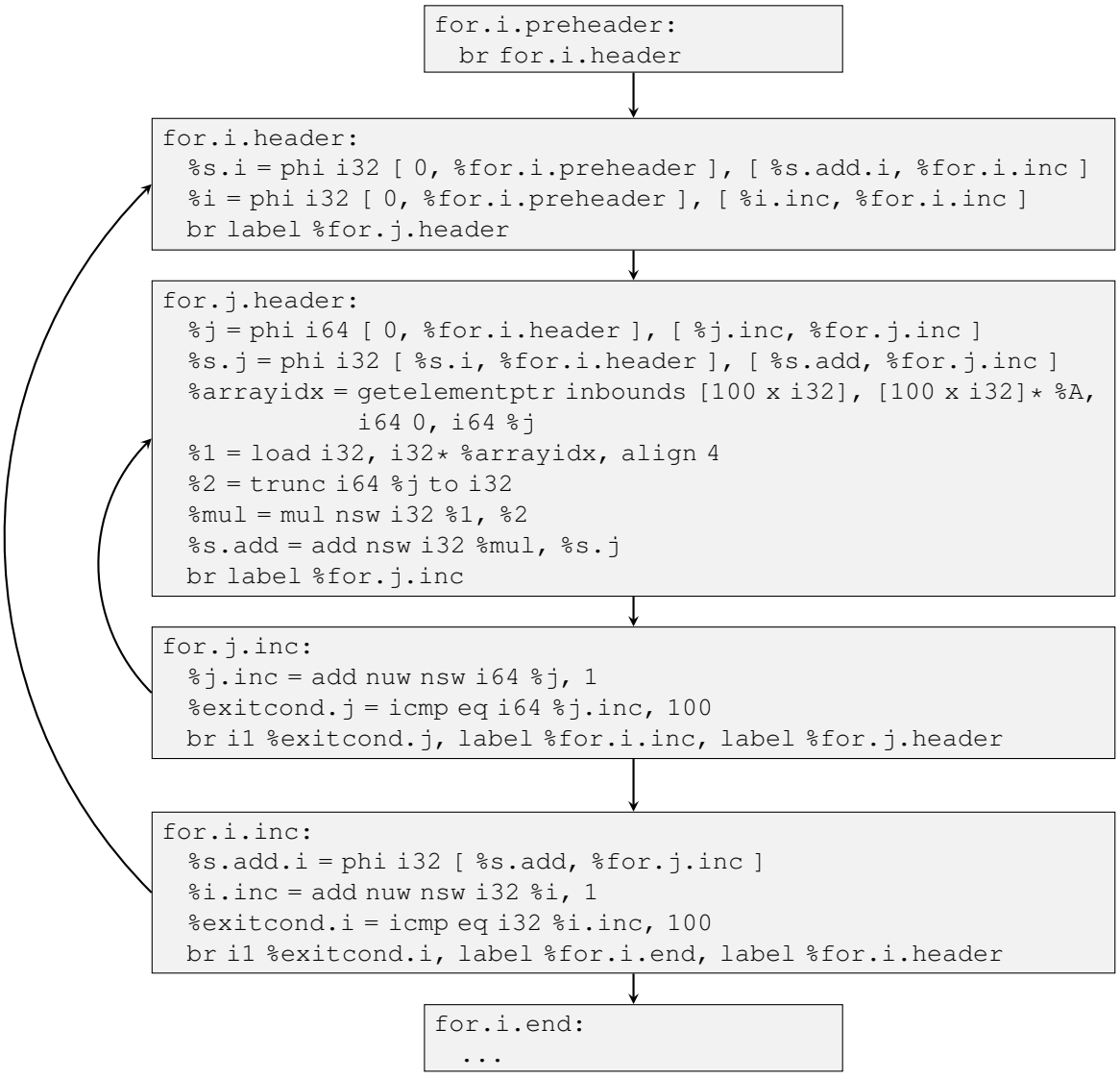


Figure 5.8: Initial IR of the nested loop in Figure 5.7.

Stage 2: Applying the profitability model

After computing a hoistable set, we use a heuristics-driven cost model which filters out those instructions which should not be hoisted due to profitability concerns. The cost model is discussed in more detail in Section 5.4.5. For now, it suffices to say that the cost model takes the initial hoistable set as input and reduces it to a possibly smaller (or even empty) hoistable set.

When applied to the example hoistable set computed in the previous stage, the cost model will decide against hoisting its last two definitions (`%j.inc` and `%exitcond.j`). Intuitively, this is because these two instructions are part of the current loop's control flow, and because it is not efficient to precompute an instruction as inexpensive as a comparison into a temporary array. Therefore, the hoistable set now contains four definitions: `%arrayidx`, `%1`, `%2` and `%mul`. Note that, together, they form the IR representation of the source code sub-expression we want to hoist, `A[j] * j`.

Stage 3: Hoisting instructions

The instructions remaining in the hoistable set after the cost model was applied are those that our optimization will hoist. Firstly, our implementation checks if the hoistable set is empty. If this is true, there are no instructions to hoist out of the current loop, and thus the entire procedure terminates. Otherwise, we use the method described in Section 5.4.3 to create the empty cloned loop after the preheader block of the parent loop.

We proceed to iterate over all the instructions of the hoistable set, and move each instruction to the header of the cloned loop, placing it immediately before the last statement in the basic block (the delimiting branch statement). With this approach, hoisted instructions are guaranteed to appear in the same order in the cloned loop as they initially did in the current one. Consequently, there is no risk of breaking the semantic correctness of IR program by having instructions attempt to use values defined later in the program. This is the main reason why we required holding instructions in a data structure that preserves insertion order.

As a final step that needs to be taken when moving invariant instructions, any occurrences of the current loop's induction variable among the instruction's operands must be replaced with the induction variable of the cloned loop. Thus, in our case, the use of `%j` in the definition of `%arrayidx` must be changed. Figure 5.9 shows the state of the IR after this stage of the generalized code motion process.

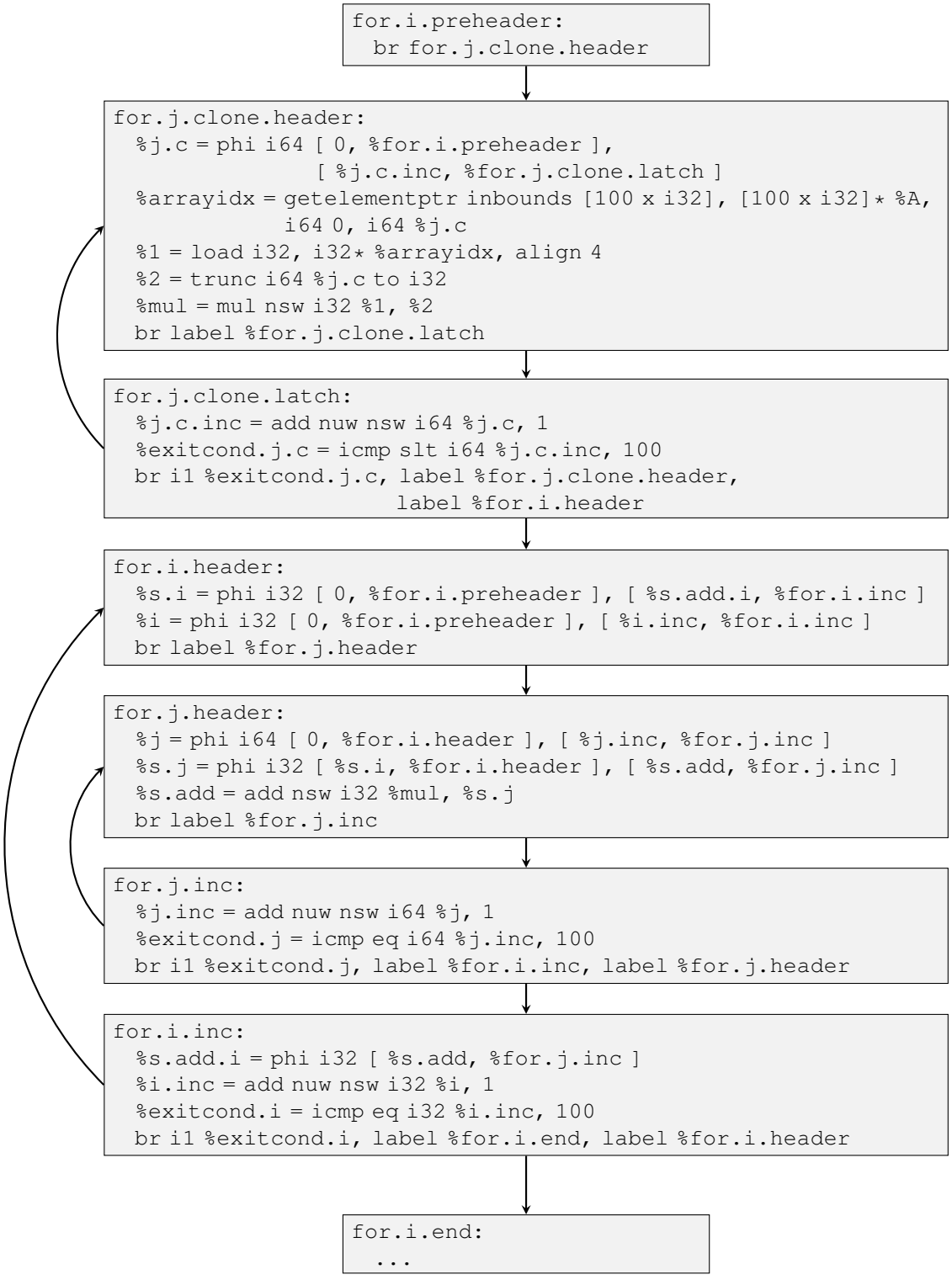


Figure 5.9: IR of the nested loop in Figure 5.7, after several invariant instructions have been moved to the cloned loop.

Stage 4: Storing and loading precomputed expressions

To complete the process of generalized code motion, we must store the results of certain instructions in the cloned loop into temporary arrays, and then load the precomputed results into the main loop. Our implementation achieves this through the following steps:

1. Iterate over all the hoisted instructions now present in the cloned loop. If the value defined by an instruction is used in the original loop, then it must be stored in a temporary array. We take note of this by adding the respective instruction to a special list L. Otherwise, all the users of that value are in the cloned loop. In this case, the instruction is most likely a single step in a longer chain of calculations that result in an invariant expression. The result of such instructions need not be stored in temporary arrays.

2. For promoting clarity in the structure of the IR, we create an additional basic block dedicated to array declarations, before the entry to the cloned loop. For each instruction I in L, we allocate an array (denoted by A) in this basic block. The array will hold elements of the same type as the value produced by I, and its size will be equal to the trip count of the current loop.

3. Let $i.c$ be the induction variable of the cloned loop. Before and after each instruction I in L, we insert two additional instructions, respectively: a `getelementptr` which computes α, the address of $A[i.c]$, and a `store` which writes the result of I to α.

4. Let i be the induction variable of the current (original) loop. For each instruction I in L, we insert two additional instructions at the beginning of the current loop's header: a `getelementptr` which computes α', the (same) address of $A[i]$, and a `load` which reads the value at memory location α' into a fresh scalar variable, V.

5. Replace all uses of I in the original loop with V. Now, any instruction in the original loop that required the value defined by I will use V, the precomputed result of the instruction, loaded from a temporary array. At this point, generalized code motion finishes processing the current loop.

In Section 5.4.2, we stated, without justification, that the result of an instruction must not be used outside of the current loop for that instruction to be hoistable. This final step of code motion provides the reason for this restriction: if a hoisted invariant value were to be referenced outside the current loop, we would have no way of replacing those uses with loads from a temporary array, since we would be outside of the induction variable's scope.

Figure 5.10 shows the final IR produced by generalized loop-invariant code motion for our ongoing example. Of all the instructions hoisted in the cloned loop, only `%mul` is stored in a temporary array, because it is the only one with lingering uses in its original loop. The value of `%mul` at a given iteration of the `j`-loop is loaded from the temporary array into `%glicm.load`. Finally, we replace the use of `%mul` with `%glicm.load` in the `%s.add` instruction.

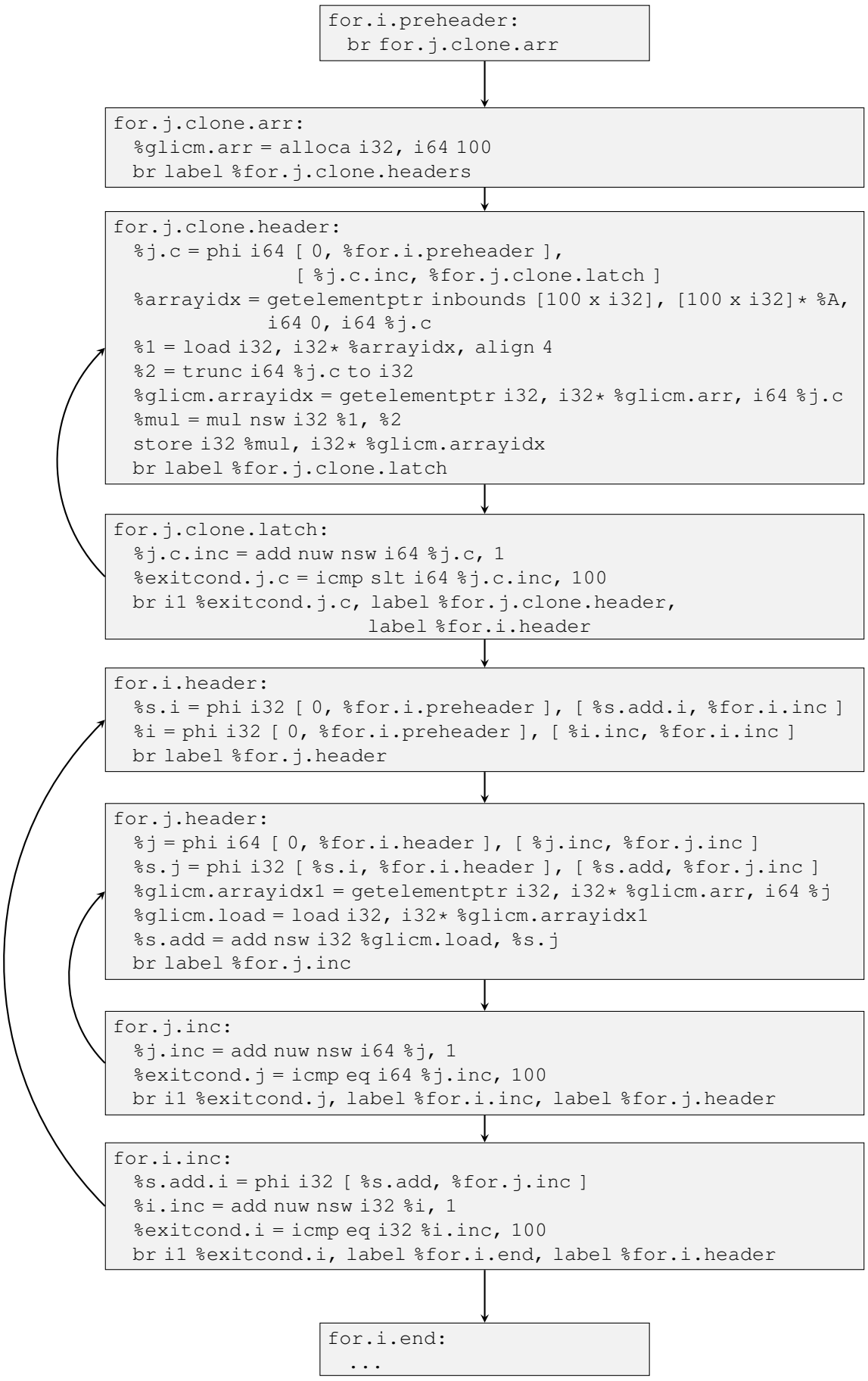


Figure 5.10: IR of the nested loop in Figure 5.7, after completing generalized code motion.

5.4.5 Cost model

The cost model we implemented is driven by heuristic considerations, drawn from our experiences with GLICM on a wide variety of programs. Its purpose is to prevent certain legally hoistable instructions from being moved to the cloned loop, if there is sufficient reason to believe that it would lead to degraded performance.

In our initial implementation of GLICM, we would immediately hoist invariant instructions to the cloned loop during the traversal of the dominator tree. This approach is used by the LICM implementation, which does not have an associated cost model, and therefore moves any hoistable instructions unconditionally. Once we realized the need for a cost model, the technique became inappropriate, since it presented us with two equally inconvenient choices:

- Hoist all invariant instructions and then attempt to 'unhoist' the unprofitable ones, which requires us to remember us the initial locations of those instructions in the original loop.
- Perform several profitability checks on instructions before hoisting them, but without much knowledge on that instruction's impact on the other hoistable instructions of the loop.

It was because of these shortcomings that we devised the hoistable set data structure and changed the hoisting process to its current form.

The implementation of the cost model is effectively a single pass over the initial hoistable set, removing invariant instructions that should rather be kept in the current loop. When an instruction is removed, any users of that instruction's result must also be taken out of the set. Otherwise, there would be instructions living in the cloned loop that require values defined at a later program point, causing a semantic inconsistency with the SSA form.

The cost model avoids precomputing instructions based on two heuristic checks, which we outline below. If an instruction fits at least one of them, the profitability model removes it from the hoistable set. The cost model can be easily extended, by adding new heuristics or by performing additional passes on the hoistable set.

1. Unprofitable instructions

We deem an instruction in the hoistable set *unprofitable* if redundantly computing it in the current loop is actually preferred to loading its precomputed result from an array. The canonical example is the LLVM `getelementptr` instruction, which is used for address computation when accessing elements of arrays and other aggregate data structures. Consider the simple pseudocode loop nest below:

```
1  for (i=0; i<100; i++) {
2    for (j=0; j<100; j++) {
3      s+=A[j]*i;
4    }
5  }
```

In LLVM IR, the innermost loop would be roughly translated into following instruction sequence (which uses a relaxed syntax, to promote clarity):

```
1  %0 = getelementptr %A, %j
2  %1 = load %0
3  %2 = mul %0, %i
4  %s.add = add %s, %2
```

The `getelementptr` instruction defining `%0` is loop-invariant with respect to the outer loop, and therefore hoistable by GLICM. Assuming the result of this instruction is stored in a temporary array `%t`, the inner loop will contain the following after generalized code motion:

```
1  %glicm.idx = getelement ptr %t, %j
2  %glicm.load = load %glicm.idx
3  %1 = load %glicm.load
4  %2 = mul %0, %i
5  %s.add = add %s, %2
```

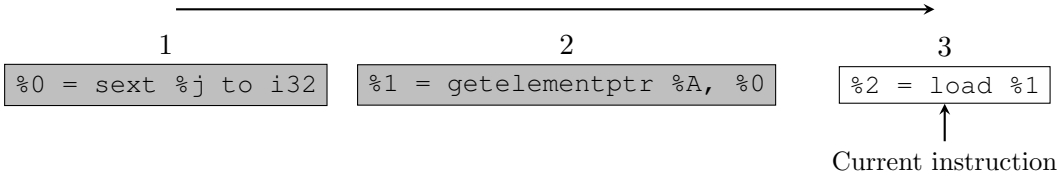
In this unfortunate case, generalized code motion did not optimize the source program, but rather achieved the opposite. One `getelementptr` instruction was hoisted from the current loop, but only to be replaced by another `getelementptr`, together with a load from a temporary array. Furthermore, the cloned loop which was hypothetically inserted before the outer loop also incurs additional overheads, both in terms of execution time and memory footprint.

There are other types of instructions which produce similar detrimental effects when precomputed. For example, it would not make much sense to hoist an invariant `load` instruction, store the value it produces in a temporary array, and then load the precomputed value from the temporary array in the original loop. On different grounds, we should also avoid precomputing inexpensive instructions such as bitwise operations, since redundantly re-evaluating them on the spot will most likely be faster than loading data from an array.

Our cost model limits the instructions which can be evaluated into temporary arrays to binary arithmetic operations and function calls (we refer to such instructions as *profitable*). We impose this restriction with the hope that the processing time saved by cutting redundant evaluations outweighs the memory latency caused by loading from temporary arrays. Instructions such as `getelementptr` and `load` will still be hoisted, but only if they have a role in defining a larger sub-expression.

To achieve this, our implementation iterates over the hoistable set in *reverse order* and removes any unprofitable instructions whose result is not used by other remaining members of the set. We justify the reversed iteration order using the example hoistable set below, which contains three invariant instructions. Assume that we are processing the set in normal order, and that we have iterated over the first two instructions so far.

According to our heuristic, the third and last instruction should be removed, because it is unprofitable and has no further uses in the set. However, once it is removed, the second instruction (a `getelementptr`) becomes unprofitable, and must now be removed. Thus, with this approach, we often have to revert decisions concerning previously inspected



instructions. In some cases, this could lead to a large cascade effect, causing unnecessary complexity in the cost model. By iterating over the set in reverse order, the instructions we visit never use a value defined later in the program, and thus we are never required to reconsider previous decisions.

2. Single-instruction invariant sub-expressions

The other heuristic we applied in our cost model acts against precomputing invariant sub-expressions that consist of a single instruction. This restriction is based on our practical experiences with GLICM, which showed that some programs perform poorly if such small sub-expressions are hoisted. Based on this criterion, our implementation filters out instructions from the hoistable set if they neither use nor are used by any other members of the set. Together, these two conditions check that a given instruction does not contribute to the computation of a larger sub-expression.

This rule also removes any single-instruction sub-expressions involving the original loop's induction variable from the hoistable set. In particular, we mention the instruction in the current loop which increments the induction variable at the end of an iteration. Without the cost model, this instruction would have its result precomputed in the cloned loop for every loop that is processed by GLICM.

Chapter 6

Evaluation

In this chapter, we describe our approach to evaluating the implementation of generalized code motion, both in terms of functional correctness and runtime impact, on a wide set of real-world programs. We present the results obtained on several types of benchmarks, and discuss some fundamental challenges in evaluating the performance of compiler optimizations.

Our evaluation was carried out exclusively on C and C++ programs, due to a number of reasons. Firstly, the LLVM test suite (see Section 3.5) readily provides an infrastructure for benchmarking a heterogeneous collection of programs written in these two languages. Constructing an equally comprehensive collection of tests in a different source language, and compiling it using an appropriate language front-end, would have been infeasible in the time frame of the project.

Secondly, we wanted to evaluate the performance of generalized code motion in the context of automatically generated local assembly programs, the original motivation for the technique (see Section 4.2), which are also written in C. Lastly, the prominent SPEC benchmark [25], which we also refer to, contains mostly C and C++ programs.

6.1 Safety and functional correctness

When discussing the fundamental properties of compiler transformations in Section 2.4, we identified safety as the most important characteristic for any optimization. Based on empirical evidence derived from extensive testing, we have satisfying guarantees that our implementation of GLICM adheres to the safety requirement. This section provides justification for this claim, and also describes our approach towards verifying the functional correctness of our implementation.

6.1.1 Unit tests

In the early stages of the implementation, we relied on several simple C programs for checking the high-level correctness of our pass. These programs act as unit tests, since they exhibit behaviour that should prevent our safety model from applying generalized code motion. Table 6.1 describes the particular safety requirement stressed by each test.

Each test consists of a single source file, containing one or two functions that process a statically initialized multi-dimensional array, derive a value from the it (e.g. by adding

all its elements) and print the result at the end of the program run. Programs are tested using the following process:

1. The program is compiled to LLVM IR using `clang -O0 -emit-llvm` (no optimizations).
2. We invoke `opt` on the resulting bitcode file, applying a minimal list of optimizations which we found to create generalized code motion opportunities, followed by the GLICM pass itself.
3. (Optional) The post-GLICM bitcode file is disassembled using to a human-readable assembly file using `llvm-dis`, in order to visually inspect the generated IR and spot any anomalies.
4. The bitcode file produced by GLICM is executed using `lli`, the IR interpreter, and its output is written to a file on disk.
5. The actual output of the run is compared to a stored reference output, produced by a `gcc`-compiled executable of the program.

The command-line invocations corresponding to the steps above, for a hypothetical C program `code.c`, are the following:

```
clang -O0 -emit-llvm code.c -c -o code.bc
opt -tbaa -basicaa -scalarrepl -mem2reg -loop-rotate -indvars
    -instcombine -reassociate -licm -glicm code.bc -o code-glicm.bc
llvm-dis code-glicm.bc
lli code-glicm.bc > code.actual_output
diff code.actual_output code.reference_output
```

To allow for more flexibility in the process, we wrote a Makefile with separate targets for each stage of the testing procedure. We achieved complete automation by using a script that iterates over all the test files and applies the steps above. Although simple, this testing methodology allowed us to smoke test our pass for major defects, before carrying out extensive, more costly evaluation.

Program	**Tested feature**
`call.c`	invariant function call
`complex-expr.c`	large invariant sub-expression
`controlflow.c`	invariant sub-expression wrapped in conditional
`currentloop-aliasing.c`	invariant read with aliasing write in current loop
`inv-and-var.c`	current loop with both invariant and variant expressions
`parentloop-aliasing.c`	invariant read with aliasing write in parent loop
`simple-expr.c`	simple invariant sub-expression
`subloop-guaranteed.c`	current loop guaranteed to execute
`subloop-notguaranteed.c`	current loop not guaranteed to execute

Table 6.1: Unit tests for GLICM.

6.1.2 LLVM test suite

LLVM's official test suite, consisting of 497 test cases at the time of this writing, provided us with a great spectrum of real-world programs and benchmarks for stress-testing the stability of our implementation. With GLICM enabled in Clang (see Section 6.2.1), we attained a 100% pass rate in the test suite, a result which has persisted across hundreds of runs. The implications of this are twofold.

Firstly, the transformation proves itself to be robust, by never causing Clang to fail while processing a source file. Thus, our implementation takes edge cases into consideration, manages its data structures cautiously, and produces semantically correct LLVM IR.

Secondly, generalized code motion preserves program behaviour, since any test case affected by our pass produces identical results compared to the reference output (see Section 3.5 for an overview of the testing process). Therefore, our implementation of GLICM is functionally correct, and the safety requirements we proposed were not found to be vulnerable.

6.2 Performance evaluation methodology

We evaluate the effects of our optimization on individual programs by compiling them with gcc and three different versions of Clang, and then comparing the running times of the generated executables. Table 6.2 describes the four different compilation techniques we use, and introduces a notational shorthand for each.

Notation	Description
GCC	`gcc -O3`
CLANG	`clang -O3`
GLICM$_{CM}$	`clang -O3` (including GLICM)
GLICM$_{\overline{CM}}$	`clang -O3` (including GLICM without the cost model)

Table 6.2: Four different compilation techniques used for performance evaluation.

We are mainly concerned with benchmarking GLICM$_{CM}$ and GLICM$_{\overline{CM}}$ against CLANG, in order to identify the net differences introduced by our pass. Occasionally, we shall also compare the performance of the different Clang variants to that of GCC.

6.2.1 Enabling GLICM in Clang

To obtain GLICM$_{CM}$ and GLICM$_{\overline{CM}}$, we added our generalized code motion pass to the pass manager which defines Clang's optimization sequence[1]. As some authors have noted [23, p. 9], finding the ideal placement of an optimization in relation to tens of other interrelated transformations is a daunting, if not infeasible task. However, we believe that we have chosen an appropriate point for scheduling GLICM in Clang's complex

[1]Practically, we inserted our pass in the `populateModulePassManager` method defined in `lib/Transforms/IPO/PassManagerBuilder.cpp`

optimization pipeline.

Firstly, our pass is scheduled after loop rotation (which we discussed in Sections 2.5.3 and 4.4), LICM and expression reassociation, a transform pass which rewrites expressions to a form that promotes more code motion opportunities. Furthermore, GLICM is invoked after function inlining, an optimization which replaces certain function calls with the actual body of the called function. If a function containing its own loops is inlined in a callee loop, new nested loops will be created, which introduces more opportunities for GLICM.

Lastly, like most other scalar optimization passes, generalized code motion runs before any vectorization passes. Thus, the cloned loop inserted by GLICM can be potentially rewritten by Clang's automatic vectorization passes.

6.2.2 Experimental setup

All our performance tests were run on a remote machine with an Intel I7-2600 CPU running at 3.4 GHz on 8 virtual cores, with a 32KB L1 cache and a 256KB L2 cache. We took several steps to ensure that our performance evaluation produces accurate and reliable results:

- Each collection of tests was run multiple times with the same configuration parameters, to promote statistical significance in our findings. To this end, we repeated test runs between 30 and 100 times, depending on their size and complexity.
- We disabled frequency scaling techniques in the CPU, so that our machine's cores always run at constant speed. This eliminates any variations in program runtime behaviour which are due to fluctuations in the CPU's workload.
- Apart from kernel and OS processes, no other applications were allowed to run on the machine while tests were being executed, in order to minimize CPU and memory contention in the system.

6.3 Results

6.3.1 Local assembly code

As outlined in Section 4, the initial motivation for generalized code motion lies in compute-intensive applications, such as the finite element method, where the particularities of the code might render traditional optimizations ineffective. To validate our implementation, we wanted to confirm that it creates major execution time improvements in such application domains.

To this end, we carried out performance evaluation on two programs that compute local assembly kernels for the finite element method (see Section 4.2 for an overview of the technique). The local assembly computations are characteristic to a Helmholtz and an elasticity partial differential equation, respectively. Both programs contain three equivalent *variants* of the same local assembly computation:

- *Variant 0*, which contains unoptimized C code that was automatically generated by the Firedrake tool [6] from a high-level mathematical description of the problem.

- *Variant 1*, the code in Variant 0 after some optimization by COFFEE, the compiler introduced by Luporini et al. in [27]. To create this variant, COFFEE applies a subset of its specialized toolset of optimizations (including GLICM) directly on the source code of Variant 0.
- *Variant 2*, the code in Variant 0 fully optimized by the COFFEE compiler.

The two programs contain an instrumentation function that executes every code variant repeatedly for a fixed amount of time (by default, 100ms), maintaining counters for the number of completed executions. At the end of a run, the code variant with the greatest execution counter is declared the fastest.

We compiled the programs with the four different compilers we described previously (CLANG, GCC, GLICM$_{CM}$ and GLICM$_{\overline{CM}}$) and then ran the experiment. Our expectation was that GLICM$_{CM}$ performs better than GCC and CLANG for Variant 0, which is the unoptimized, automatically generated code version. On the other hand, we did not expect Variant 0 to be faster than either Variant 1 or 2 using any of the compilers, since the latter are code versions produced by COFFEE using a tailored domain-specific set of optimizations, in addition to generalized code motion. However, we did expect GLICM$_{CM}$ to not cause Variant 1 or 2 to execute slower, through correct assumptions made by its cost model.

Table 6.3 presents the results of our experiments, averaged over 100 runs. We draw the following key points based on this data, which mostly confirms our initial assumptions:

- GLICM$_{CM}$ performs much better than CLANG for Variant 0. The speed-up factors achieved by our optimization are 2.22 and 1.64 for the Helmholtz and elasticity kernels, respectively. For one of the equations, GLICM$_{CM}$ is superior to GCC by a factor of 1.41, while for the other GCC generates code that runs roughly 10% faster. Thus, based on our experiments, it can be said GLICM$_{CM}$ performs at least as well as GCC for Variant 0.
- Our cost model performs efficiently on local assembly code. Consider the performance of Variants 1 and 2 for the elasticity equation, which are both 40% slower with GLICM$_{\overline{CM}}$ than CLANG. With GLICM$_{CM}$, the slowdowns disappear entirely. In the case of the Helmholtz equation, the cost model does not eliminate the penalties completely, yet it reduces them significantly, i.e. from 8% and 15.2% to 2% and 3.77% respectively.
- Although not related to our optimization and the purpose of our experiment, we note that GCC outperforms CLANG massively for Variants 1 and 2 of the local assembly code.

Helmholtz equation			
	Variant 0	*Variant 1*	*Variant 2*
GCC	72934	192921	221094
CLANG	46429	123982	146818
GLICM$_{CM}$	103390	121413	141275
GLICM$_{\overline{CM}}$	98996	114097	124355
Elasticity equation			
	Variant 0	*Variant 1*	*Variant 2*
GCC	26096	47905	63684
CLANG	14306	24761	30508
GLICM$_{CM}$	23459	24816	30554
GLICM$_{\overline{CM}}$	22194	14820	18012

Table 6.3: Number of executions of each local assembly code variant using four different compilers. A greater number signifies better performance.

6.3.2 LLVM test suite

We carried out extensive experiments on the programs in the LLVM test suite, using the infrastructure described in Section 3.5. To mark the source files which were actually transformed by our optimization during compilation with GLICM$_{\overline{CM}}$ and GLICM$_{CM}$, we placed several debug prints in our pass' implementation, which documented important events such as hoisted instructions or allocated arrays. When invoking Clang, we enabled debugging information to be displayed exclusively for our pass, with a special command-line flag[2]. This caused events related to GLICM to be written in per-file compilation logs by the testing framework. With the help of `grep`, we were able to isolate the programs of interest from the rest of the test suite.

This section presents our performance evaluation results on the programs distributed with the official LLVM test repository. As per Section 3.5, these are of two types:

- *Single-source tests*, programs consisting of a single source file. These are further structured into three main subdirectories, `Benchmarks`, `Regression` and `UnitTests`, containing to mini-benchmark programs, regression tests and unit tests, respectively.
- *Multi-source tests*, applications that are built from multiple source files. Multi-source test cases are also divided among three subcategories, `Benchmarks` and `UnitTests`, as before, and `Applications`, which hosts real-world programs that were minimally adapted for integration with the test suite.

When discussing runtime effects, we will be completely ignoring programs with an execution time of under 50ms. We introduce this restriction because we found execution times of programs to vary within this order of magnitude between successive test runs. For such programs, we could thus not be sure if the incurred speedup is circumstantial, or actually a consequence of our transformation. On similar grounds, we ignore programs that exhibit either speedups or slowdowns of under 1%.

[2] `-mllvm -debug-only=glicm`

GLICM$_{\overline{CM}}$

Let us first discuss the results obtained for GLICM$_{\overline{CM}}$, which are shown in Tables 6.4 and 6.5 for single-source and multi-source tests, respectively. The tables describe the speedups and slowdowns achieved by GLICM$_{\overline{CM}}$ compared to CLANG and GCC, for most of the programs modified by our pass. Several test cases in both the single-source and multi-source suites have been omitted due to space considerations.

The ratios displayed for each test case are obtained by dividing the execution times of the original programs, compiled with CLANG and GCC respectively, to the execution time of the program optimized with GLICM$_{\overline{CM}}$. Thus, values greater than 1 indicate speedups, while values less than 1 indicate slowdowns compared to the initial programs. The execution times we used for aggregating these metrics were averaged over 100 repeated runs of the test suite for CLANG and GLICM$_{\overline{CM}}$, and over only 3 runs for GCC, since our main concern is comparing GLICM$_{\overline{CM}}$ to CLANG.

At first glance, experimental results show that generalized code motion was applied to a considerable amount of programs, constituting approximately a third of the test suite. However, for both sets of programs, the mostly insignificant improvements introduced by generalized code motion are largely overshadowed by performance regressions, both numerically and in terms of magnitude. Multi-source programs were particularly affected, often doubling their execution time when compiled with GLICM$_{\overline{CM}}$, with several extreme outliers.

The cause of the overwhelming number of regressions is the unconditional application of GLICM to any non-outermost program loop in the test suite. Without the cost model, any instruction which passes the invariance and safety tests is hoisted, leading to many cases of bitwise operations, loads, or address computations being moved to the cloned loop and subsequently stored in temporary arrays.

However, we do note that major improvement was achieved for two test cases:

- `mandel`, a single-source program which computes elements of the Mandelbrot set. Here, generalized code motion hoisted an invariant floating point division (together with other less profitable statements) from a doubly-nested loop in the main body of the program. With GLICM$_{\overline{CM}}$, the program achieved speedups of 1.614 and 1.734, when compared to CLANG and GCC respectively.
- `power`, a multi-source program which is part of a benchmark centred on evaluating dynamic data structures [28]. GLICM modified this test case by precomputing instructions that we normally deem unprofitable (including two `getelementptrs` and a comparison) into temporary arrays, in a small function that dominates the program's execution time. We were surprised by the speedup achieved for this test case, as it contradicts our intuition of profitability. When compiled with GLICM$_{\overline{CM}}$, this particular program executes 1.495 times faster than the version produced by CLANG, and 1.441 times faster compared to GCC.

Total test cases		313
Total source files affected by GLICM		55
Total test cases affected by GLICM		55
Improvements	**CLANG**	**GCC**
Benchmarks/Misc/mandel	1.614	1.734
Benchmarks/Misc/matmul_f64_4x4	1.018	2.004
Benchmarks/Polybench/linear-algebra/solvers/durbin/durbin	1.012	0.698
Regressions	**CLANG**	**GCC**
Benchmarks/BenchmarkGame/puzzle	0.185	0.149
Benchmarks/Shootout/matrix	0.251	0.231
Benchmarks/Misc/fp-convert	0.299	0.294
Benchmarks/Shootout-C++/matrix	0.329	0.485
UnitTests/Vectorizer/gcc-loops	0.338	0.570
Benchmarks/Polybench/medley/floyd-warshall	0.402	0.474
Benchmarks/Misc/dt	0.502	0.500
Benchmarks/Misc-C++/oopack_v1p8	0.546	0.577
Benchmarks/Polybench/linear-algebra/kernels/syrk	0.585	0.597
Benchmarks/Polybench/datamining/correlation	0.669	0.667
Benchmarks/Polybench/linear-algebra/kernels/3mm	0.707	0.712
Benchmarks/Polybench/linear-algebra/kernels/2mm	0.709	0.746
Benchmarks/Polybench/linear-algebra/kernels/syr2k	0.712	1.120
Benchmarks/Polybench/linear-algebra/kernels/gemver	0.719	0.574
Benchmarks/Polybench/linear-algebra/kernels/atax	0.744	0.494
Benchmarks/Polybench/linear-algebra/kernels/gemm	0.758	0.833
Benchmarks/Polybench/datamining/covariance	0.762	0.756
Benchmarks/Polybench/linear-algebra/kernels/mvt	0.778	0.596
Benchmarks/Polybench/stencils/adi	0.810	0.858
Benchmarks/Polybench/linear-algebra/solvers/dynprog	0.829	1.395
Benchmarks/Polybench/linear-algebra/kernels/doitgen	0.851	0.883
Benchmarks/Linpack/linpack-pc	0.868	1.494
Benchmarks/Misc/ReedSolomon	0.890	0.863
Benchmarks/Polybench/stencils/fdtd-2d	0.936	0.940
UnitTests/Vector/multiplies	0.945	0.000
Benchmarks/Misc/oourafft	0.949	0.941
Benchmarks/Stanford/Puzzle	0.974	1.053
Benchmarks/McGill/misr	0.976	1.236
Benchmarks/Polybench/linear-algebra/solvers/gramschmidt	0.978	0.970
Benchmarks/Polybench/linear-algebra/kernels/cholesky	0.979	1.002
Benchmarks/Polybench/linear-algebra/kernels/bicg	0.986	0.636
Benchmarks/Polybench/linear-algebra/kernels/symm	0.987	0.963
Benchmarks/Polybench/stencils/jacobi-2d-imper	0.988	1.005

Table 6.4: Speedups and slowdowns of GLICM$_{\overline{CM}}$ over CLANG and GCC for single-source programs in the LLVM test suite. This table omits the 10 least significant slowdowns.

Total test cases		184
Total source files affected by GLICM		220
Total test cases affected by GLICM		94
Improvements	**CLANG**	**GCC**
Benchmarks/Olden/power	1.495	1.441
Benchmarks/Prolangs-C++/life	1.014	1.092
Applications/SIBsim4	1.013	0.961
Applications/oggenc	1.013	1.026
Benchmarks/Ptrdist/bc	1.012	1.022
Applications/lemon	1.011	0.958
Regressions	**CLANG**	**GCC**
Applications/viterbi	0.111	0.135
Benchmarks/TSVC/GlobalDataFlow-flt	0.170	0.174
Benchmarks/TSVC/Symbolics-fltc	0.189	0.226
Benchmarks/TSVC/Equivalencing-flt	0.195	0.133
Benchmarks/TSVC/ControlLoops-flt	0.231	0.225
Benchmarks/TSVC/Equivalencing-dbl	0.268	0.279
Benchmarks/TSVC/Expansion-flt	0.286	0.334
Benchmarks/TSVC/CrossingThresholds-flt	0.312	0.354
Benchmarks/TSVC/InductionVariable-flt	0.343	0.323
Benchmarks/TSVC/ControlLoops-dbl	0.348	0.345
Benchmarks/TSVC/NodeSplitting-flt/	0.359	0.403
Benchmarks/TSVC/GlobalDataFlow-dbl	0.361	0.359
Benchmarks/TSVC/Packing-flt	0.366	0.430
Benchmarks/TSVC/Symbolics-dbl	0.373	0.436
Benchmarks/TSVC/Searching-dbl	0.377	0.491
Benchmarks/TSVC/Packing-dbl	0.388	0.441
Benchmarks/TSVC/Searching-flt	0.392	0.483
Benchmarks/TSVC/CrossingThresholds-dbl	0.419	0.460
Benchmarks/TSVC/ControlFlow-flt	0.426	0.440
Benchmarks/TSVC/IndirectAddressing-flt	0.434	0.435
Benchmarks/Trimaran/enc-rc4	0.436	0.509
Benchmarks/TSVC/Expansion-dbl	0.445	0.490
Benchmarks/TSVC/InductionVariable-dbl	0.456	0.445
Benchmarks/TSVC/NodeSplitting-dbl	0.461	0.466
Benchmarks/Prolangs-C/gnugo	0.481	0.500
Benchmarks/TSVC/LoopRerolling-flt	0.482	0.452
Benchmarks/TSVC/ControlFlow-dbl	0.497	0.509
Benchmarks/TSVC/IndirectAddressing-dbl	0.522	0.519

Table 6.5: Speedups and slowdowns of GLICM$_{\overline{CM}}$ over CLANG and GCC for multi-source programs in the LLVM test suite. This table omits the 20 least significant slowdowns.

GLICM$_{CM}$

The impact of generalized code motion on single-source and multi-source programs, when applied together with its cost model, is described in Tables 6.6 and 6.7. We immediately notice that the number of programs modified by our optimization is now drastically lower. The cost model proved to be efficient in eliminating all but four regressions, all of which are much less severe than those we had experienced initially. Due to relying on purely heuristic measures that do not take wider program context into account, the cost model is not immune to inaccurate behaviour. As a consequence, several speedups are also lost with GLICM$_{CM}$, including one of the two most significant improvements we noted in the previous section.

The improved single source program which we have discussed earlier, `mandel`, maintains its behaviour also in the presence of the cost model. We believe that the speedup achieved in this case (which is impressive, given that it is induced by a single optimization) is a glimpse into the potential of GLICM in real-world applications.

Total test cases		313
Total source files affected by GLICM		6
Total test cases affected by GLICM		6
Improvements	**CLANG**	**GCC**
Benchmarks/Misc/mandel	1.624	1.744
Regressions	**CLANG**	**GCC**
Benchmarks/Polybench/stencils/adi/adi	0.995	1.055

Table 6.6: Speedups and slowdowns of GLICM$_{CM}$ over CLANG and GCC for single-source programs in the LLVM test suite.

Total test cases		184
Total source files affected by GLICM		17
Total test cases affected by GLICM		12
Improvements	**CLANG**	**GCC**
Applications/oggenc	1.013	1.026
Regressions	**CLANG**	**GCC**
Benchmarks/TSVC/CrossingThresholds-flt	0.987	1.118
Benchmarks/TSVC/CrossingThresholds-dbl	0.994	1.093
Benchmarks/7zip	0.998	1.035

Table 6.7: Speedups and slowdowns of GLICM$_{CM}$ over CLANG and GCC for multi-source programs in the LLVM test suite.

6.3.3 SPEC CPU 2006

Wanting to investigate more applications which could potentially benefit from generalized code motion, we turned to the SPEC family of benchmarks [25], which are used as an industry standard. Fortunately, several SPEC-based benchmarks are supported as external tests by the LLVM testing infrastructure, which meant that the benchmark programs could be readily built and run using the same tools we have used so far.

The specific benchmark we chose use is CPU 2006, the most recent version of SPEC's well renowned CPU-intensive benchmark. This benchmark consists of integer and floating point applications written in C, C++ and Fortran. Given our purposes, we only evaluated the performance of C and C++ benchmarks. Unfortunately, 6 of the 17 floating point benchmarks, which we were of considerable interest to us, are implemented in Fortran, and are thus unreachable by our optimization.

Table 6.8 shows the results following compilation and subsequent execution of the CPU 2006 benchmark using GLICM$_{\overline{CM}}$. Although generalized code motion modified 11 of the 16 test cases that were executed in total, there was no significant improvement detected in any of the programs. Moreover, regressions continued to appear due to the application of GLICM in the absence of a cost model.

Enabling the cost model, the total number of affected programs decreases to 4. In addition, we found that there were neither reliable speedups nor slowdowns in the test suite, since the execution times of the affected programs was modified only marginally. Although we would have enjoyed for GLICM to achieve tangible speedups on the applications in the CPU 2006 benchmark, we are content with the fact that our cost model seems to always cull the number of slowdowns to a bare minimum.

Total test cases		16
Total source files affected by GLICM		117
Total test cases affected by GLICM		11
Regressions	**CLANG**	**GCC**
External/SPEC/CFP2006/433_milc	0.764	0.771
External/SPEC/CFP2006/447_dealII	0.860	0.095
External/SPEC/CINT2006/464_h264ref	0.876	0.850

Table 6.8: Speedups and slowdowns of GLICM$_{\overline{CM}}$ over CLANG and GCC for the SPEC CPU2006 benchmark.

6.4 Challenges and limitations

The performance evaluation of our new loop optimization was challenged by several factors. While we managed to mitigate some of the impediments, others have proven to be inherently difficult problems with no optimal solution.

Firstly, the results of our entire performance evaluation are determined by GLICM's place in Clang's optimization pipeline. In Section 6.2.1, we justified GLICM's position in the optimization sequence by relating to other optimization passes which we know affect GLICM's performance. However, considering the often subtle interactions between Clang's different passes, there may possibly be a more optimal position in the pipeline for invoking generalized code motion.

Secondly, a considerable fraction of the programs in the LLVM test suite cannot be reliably used as speed benchmarks, because their execution times are in the order of tens of milliseconds or less. Despite our best efforts, we found that runtime variations of this magnitude between identical runs of the same test case are not uncommon. One solution to this problem would have been to wrap such programs in a loop and execute them many (e.g. 1000) times, but this approach does not scale too well. In the end, we tested the programs in test suite without altering them, at the potential cost of not detecting some speedup opportunities.

There is yet another major limitation we faced when evaluating GLICM using real, general-purpose applications. No matter what effect GLICM may have on a particular loop nest, our evaluation methodology only reflects changes in the overall runtime of an entire program. Thus, major local improvements and regressions alike are obstructed if the loop which experiences them is computationally insignificant in the optimized program.

Chapter 7

Conclusion

In this report, we described our approach towards mapping a highly domain-specific compiler optimization to a general-purpose environment, by adapting the initial description of the technique and introducing additional safety considerations. Afterwards, we outlined our strategy for implementing generalized code motion as a pass in the LLVM optimizer, and lastly discussed our evaluation methodology.

We believe that we managed to fulfil most of the goals we set to achieve at the beginning of this project. Firstly, we provided a functionally correct implementation of our optimization pass in the most popular open-source compiler framework. Most importantly, this allowed us to reach our central objective, which was to evaluate the effects of GLICM on programs and applications of all sizes and shapes, ranging from full database applications to focused micro-benchmarks.

The main conclusion we draw from our experiments is that generalized code motion cannot be applied unconditionally to arbitrary programs, because it may have severely detrimental effects on their runtime if instructions are hoisted inappropriately. Instead, the transformation should be driven by the considerations of a cost model. Although our proposed version proves to be efficient in cutting slowdowns, it also suffers from inaccuracy, since it simultaneously prevents some improvements. This situation arises due to coarseness in the model, which does not take wider program context into account, but rather relies on local information pertaining to the currently processed loop.

Objectively, the results of the performance evaluation show that our optimization does not generally introduce as many speedups as we had initially hoped, most likely due to the considerable trade-offs it involves. However, this does not necessarily mean that GLICM does not have the potential of performing better on general-purpose programs, provided that a more accurate cost model is used. We did observe two cases of major improvement introduced by generalized code motion, which highlight the impact a single optimization can make if applied opportunely. Lastly, we confirmed that GLICM achieves a considerable performance improvement on local assembly kernels when compared to plain Clang and, occasionally, the GNU C compiler.

7.1 Future work

Despite its achievements, the project certainly has its shortcomings, and remains open-ended for future development. Below, we suggest a list of possible directions for extending our work:

- The cost model used by GLICM can be supplemented with more fine-grained checks, which, as we discussed, may increase the overall usefulness of the transformation.
- Although we believe our pass implementation generally adheres to good software design practices and also to the LLVM style guide, we did not have the possibility of fine-tuning for performance due to time constraints. A particular direction of improvement would be reimplementing the hoistable set data structure, which currently uses memory rather inefficiently.
- The implementation of GLICM could be extended to support hoisting instructions not only to the immediate parent of the current loop, but also to other superloops.
- The applicability of our optimization can be further assessed by testing against more benchmarks and applications. Since GLICM is enabled directly into Clang, the evaluation process is in no way tied to the LLVM testing infrastructure. Different C and C++ programs can be compiled with Clang and then benchmarked using another tool set, or even directly, e.g. with the Unix `time` command. Alternatively, one could focus on exploring the benefits GLICM may have on other computational kernels specific to the finite element method.
- A significant step forward would be to engage in discussions with LLVM developers concerning the inclusion of GLICM in the main LLVM repositories. If this were to happen, the optimization would become usable by the entire community, potentially knowing uses that we have not envisioned.

References

[1] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools, 2nd Edition*. Addison Wesley, 2006.

[2] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. "Compiler Transformations for High-performance Computing". In: *ACM Comput. Surv.* 26.4 (Dec. 1994), pp. 345–420. ISSN: 0360-0300. DOI: 10.1145/197405.197406. URL: http://doi.acm.org/10.1145/197405.197406.

[3] compilers.net. *Types of Compilers*. URL: http://www.compilers.net/paedia/compiler/index.htm (visited on 06/14/2015).

[4] Association for Computing Machinery. *ACM Software System Awards*. URL: http://awards.acm.org/software_system/year.cfm (visited on 01/30/2015).

[5] Clang contributors. *clang: a C language family frontend for LLVM*. URL: http://clang.llvm.org/ (visited on 01/30/2015).

[6] Firedrake contributors. *The Firedrake project*. URL: http://www.firedrakeproject.org/ (visited on 06/03/2015).

[7] GCC contributors. *Loop Representation - GCC Internals*. URL: https://gcc.gnu.org/onlinedocs/gccint/Loop-representation.html (visited on 05/25/2015).

[8] KLEE contributors. *KLEE*. URL: https://klee.github.io/ (visited on 06/01/2015).

[9] LLDB contributors. *The LLDB debugger*. URL: http://lldb.llvm.org/ (visited on 01/30/2015).

[10] LLVM contributors. *Getting Started with the LLVM System*. URL: http://llvm.org/docs/GettingStarted.html (visited on 06/01/2015).

[11] LLVM contributors. *List of LLVM users*. URL: http://llvm.org/Users.html (visited on 06/01/2015).

[12] LLVM contributors. *LLVM Alias Analysis Infrastructure*. URL: http://llvm.org/docs/AliasAnalysis.html (visited on 06/02/2015).

[13] LLVM contributors. *LLVM Bitcode File Format*. URL: http://llvm.org/docs/BitCodeFormat.html (visited on 06/01/2015).

[14] LLVM contributors. *LLVM Command Guide*. URL: http://llvm.org/docs/CommandGuide/ (visited on 01/30/2015).

[15] LLVM contributors. *LLVM Language Reference*. URL: http://llvm.org/docs/LangRef.html (visited on 01/30/2015).

[16] LLVM contributors. *LLVM Online Documentation*. URL: http://llvm.org/doxygen (visited on 06/02/2015).

[17] LLVM contributors. *LLVM Programmer's Manual*. URL: http://llvm.org/docs/ProgrammersManual.html (visited on 06/02/2015).

[18] LLVM contributors. *LLVM Testing Infrastructure Guide*. URL: http://llvm.org/docs/TestingGuide.html (visited on 06/02/2015).

[19] LLVM contributors. *Projects using LLVM*. URL: http://llvm.org/ProjectsWithLLVM (visited on 01/30/2015).

[20] LLVM contributors. *The LLVM Compiler Infrastructure*. URL: http://www.llvm.org (visited on 01/30/2015).

[21] LLVM contributors. *The Often Misunderstood GEP Instruction*. URL: http://llvm.org/docs/GetElementPtr.html (visited on 06/01/2015).

[22] LNT contributors. *LNT - LLVM Performance Tracking Software*. URL: http://llvm.org/docs/lnt/ (visited on 06/02/2015).

[23] Keith Cooper and Linda Torczon. *Engineering a Compiler, 2nd Edition*. Morgan Kaufmann, 2011.

[24] Intel Corporation. *Using Automatic Vectorization*. URL: https://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/GUID-32ED933F-5E8A-4909-A581-4E9DB59A6933.htm (visited on 01/30/2015).

[25] The Standard Performance Evaluation Corporation. *SPEC Homepage*. URL: https://www.spec.org (visited on 06/02/2015).

[26] Chris Lattner. "LLVM: An Infrastructure for Multi-Stage Optimization". MA thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002. URL: http://llvm.org/pubs/2002-12-LattnerMSThesis.pdf.

[27] Fabio Luporini et al. "COFFEE: an Optimizing Compiler for Finite Element Local Assembly". In: *CoRR* abs/1407.0904 (2014). URL: http://arxiv.org/abs/1407.0904.

[28] Anne Rogers et al. "Supporting Dynamic Data Structures on Distributed-memory Machines". In: *ACM Trans. Program. Lang. Syst.* 17.2 (Mar. 1995), pp. 233–263. ISSN: 0164-0925. DOI: 10.1145/201059.201065. URL: http://doi.acm.org/10.1145/201059.201065.