SECOND ORDER PREDICTIVE COMMONING IN LLVM COMPILER INFRASTRUCTURE

by

N. MD FAIZAAN 2013103512
MUTHURAJ M T 2013103515
ASWIN P 2013103535

A project report submitted to the

FACULTY OF INFORMATION AND COMMUNICATION ENGINEERING

in partial fulfillment of the requirements for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ANNA UNIVERSITY, CHENNAI – 25

APRIL 2017

BONAFIDE CERTIFICATE

Certified that this project report titled **SECOND ORDER PREDICTIVE COMMONING IN LLVM COMPILER INFRASTRUCTURE** is the *bonafide* work of **N. MD FAIZAAN (2013103512)**, **MUTHURAJ M T (2013103515)** and **ASWIN P (2013103535)** who carried out the project work under my supervision, for the fulfillment of the requirements for the award of the degree of Bachelor of Engineering in Computer Science and Engineering. Certified further that to the best of my knowledge, the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or an award was conferred on an earlier occasion on these or any other candidates.

Place: Chennai
Date:

Dr. Arul Siromoney
Professor
Department of Computer Science and Engineering
Anna University, Chennai – 25

COUNTERSIGNED

Head of the Department,
Department of Computer Science and Engineering,
Anna University Chennai,
Chennai – 600025

ACKNOWLEDGEMENT

Foremost, we would like to express our sincere gratitude to our project guide, **Dr. Arul Siromoney** Department of Computer Science and Engineering, Anna University, for his continuous support and guidance which was instrumental in taking the project to successful completion.

We are grateful to **Dr.D.Manjula**, Head, Department of Computer Science and Engineering, Anna University, for providing conducive environment and facilities for the project.

We express our heartiest thanks to our panel of reviewers, **Dr. S. Valli**, Professor, Department of Computer Science and Engineering, Anna University, **Dr. S. Sudha**, Senior Assistant Professor, Department of Computer Science and Engineering, Anna University, and **Dr. B.L. Velammal** Professor, Department of Computer Science and Engineering, Anna University, for their valuable suggestions and critical reviews throughout the course of our project.

N. MD FAIZAAN **MUTHURAJ M T** **ASWIN P**

ABSTRACT

Compile time optimizations, including invariant code optimization and loop optimization are among the best candidates for improving the execution time of statically compiled programs, due their wide coverage, and untapped potential in terms of optimization. This project aims to implement one such optimization targeted at loops, known as Second Order Predictive Commoning.

Instead of implementing the optimization to a specific compiler front end for a language, we leverage the modular nature of LLVM to implement the optimization in a language neutral Intermediate Representation(IR) which can be plugged into the front end of any compiler. Second Order Predictive Commoning targets indexed array accesses inside loops, and seeks to *predict* the elements that will be used in subsequent iterations, and reuses them across iterations, eliminating redundant array accesses. We make use of the LLVM core classes for achieving this, through manipulation of the IR. Eliminating significant number of memory accesses is the main goal of this project, and this directly results in the improvement of the program execution times. The optimization is implemented in a loadable module that is triggered by using a special flag that is passed during compilation.

ABSTRACT

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

IR Intermediate Representation

IPLS Invariant-induced Pattern based Loop Specialization

LLVM Low Level Virtual Machine

OPT Optimizer for LLVM

RIP Relative Improvement Percentage

RTTI Run time type inference

SSA Static Single Assignment

CHAPTER 1

INTRODUCTION

1.1 OBJECTIVE

Low Level Virtual Machine (LLVM) by Lattner and Adve [4] is a compiler framework designed to support transparent, lifelong program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs. LLVM defines a common, low-level code representation in Static Single Assignment (SSA) form, and uses a universal Intermediate Representation (IR) for representing code at a level slightly above the assembly code. LLVM has matured into a full fledged compilation ecosystem with Clang being an excellent multistage optimizing compiler that can be a drop in replacement for GCC. We aim to implement Predictive commoning, a kind of loop optimization, in the LLVM compiler infrastructure.

1.2 PROBLEM STATEMENT

In loops, array accesses are often sequential in nature. This leads to use of same value from one iteration to next. Currently, the compiler loads the value again which is redundant. We try to optimize this step by removing the load and inserting a register swap.

1.3 NEED FOR THE SYSTEM

LLVM applies loop optimizations to the generated IR, as a part of the optimization passes run by default. They include Loop Deletion,

Loop extract, Loop reduce, Loop rotate, Loop Unroll, Loop unswitch and Loop simplify. However, LLVM does not apply optimization for loops in any predictive manner. While its GCC counterpart applies this optimization, LLVM applies no such optimization. We implement second order predictive commoning to the LLVM infrastructure as a loadable module which can be invoked using a special flag.

1.4 CHALLENGES

The main challenge of this project lies in program correctness. Any modification to the code must not result in incorrect code. For example, hoisting an array access must not result in undefined behavior. To overcome these, we employ Run Time Type Identification (RTTI) checks and make sure we don't optimize code incorrectly.

1.5 OVERVIEW OF THESIS

Chapter 2 discusses existing work in this area, including approaches to optimizations similar to what we are attempting. Chapter 3 explains the overall architecture and the design of various modules. Chapter 4 gives the implementation details of the project and discusses the results of optimization with emphasis on comparison between the execution times of various scenarios. Chapter 5 concludes the thesis and states the extensions that can be made to the project. Appendix A explains the basic set up needed to work with LLVM.

CHAPTER 2

RELATED WORK

Loops are a common target of various compiler optimization techniques. LLVM itself by default supports up to 10 optimizations targeted at loops. However there still is a lot left to be done especially in the area of reducing data accesses within the loops.

Oh *et al.* [5] discuss exploiting predictable patterns of values across loop iterations, as existing specializers cannot fully capitalize on this opportunity. To address this limitation, they have presented Invariant-induced Pattern based Loop Specialization (IPLS), the first fully-automatic specialization technique. Using dynamic information-flow tracking, IPLS profiles the values of instructions that depend solely on invariants and recognizes repeating patterns across multiple iterations of hot loops. IPLS then specializes these loops, using those patterns to predict values across a large window of loop iterations. This enables aggressive optimization of the loop.

Another style of optimization is with regard to invariants in a loop. LLVM includes loop invariant code motion but Sharma *et al.* [6] focuses on algebraic invariants found through a data-driven approach. The task of generating loop invariants lies at the heart of any program verification technique. A wide variety of techniques have been developed for generating linear invariants, including methods based on abstract interpretation and constraint solving among others. Recently, researchers have also applied these techniques to the generation of non-linear loop invariants. These techniques discover algebraic invariants.

Current parallelization techniques handle instruction level parallelization well in a satisfactory manner, but when it comes to parallelization across of loop iterations, its not exploited well. Aiken and Nicolau [1] present a new technique bridging the gap between fine- and coarse-grain loop parallelization, allowing the exploitation of parallelism inside and across loop iterations. Furthermore, they show that, given a loop and a set of dependencies between its statements, the execution schedule obtained by out transformation is time optimal: no transformation of the loop based on the given data-dependencies can yield a shorter running time for that loop.

Wolf and Lam [8] talk about improving the data locality. It is achieved by loop transformation algorithm which is based on two concepts: a mathematical formulation of reuse and locality, and a loop transformation theory that unifies the various transforms as uni-modular matrix transformations. The algorithm is successful in optimizing programs such as matrix multiplication, successive over-relaxation, LU decomposition without pivoting, and Givens QR factorization. Performance evaluation indicates that locality optimization is especially crucial for scaling up the performance of parallel code.

Amato *et al.* [2] propose a new technique combining dynamic and static analysis of programs to find linear invariants. They use a statistical tool, called simple component analysis, to analyze partial execution traces of a given program. They get a new coordinate system in the vector space of program variables, which is used to specialize numerical abstract domains.

Most conventional compilers fail to allocate array elements to registers because standard data-flow analysis treats arrays like scalars, making it impossible to analyze the definitions and uses of individual array elements. This deficiency is particularly troublesome for floating-point

registers, which are most often used as temporary repositories for subscripted variables. In Callahan *et. al* [3], the authors present a source-to-source transformation, called scalar replacement, that finds opportunities for reuse of subscripted variables and replaces the references involved by references to temporary scalar variables. The objective is to increase the likelihood that these elements will be assigned to registers by the coloring-based register allocators found in most compilers. In addition, they present transformations to improve the overall effectiveness of scalar replacement and show how these transformations can be applied in a variety of loop nest types.

Wolf and Lam [9] propose a new approach to transformations for general loop nests. In this approach, they unify all combinations of loop interchange, skewing and reversal as unimodular transformations. The use of matrices to model transformations has previously been applied only to those loop nests whose dependences can be summarized by distance vectors. Their technique is applicable to general loop nests where the dependences include both distances and directions.

Predictive commoning is a rather new approach applied in optimising loops by combining and modifying two earlier known methods namely, Common Subexpression Elimination and Loop-Invariant Code Motion. It involves predicting whether array accesses will be reused in subsequent iterations and then tries to optimize those accesses by promoting them to registers.

CHAPTER 3

SYSTEM DESIGN

3.1 SYSTEM ARCHITECTURE

The goal of this project is to build a module that facilitates the necessary optimizations pertaining to predictive commoning. This entails a few key modules which are diagrammatically represented in Figure 3.1

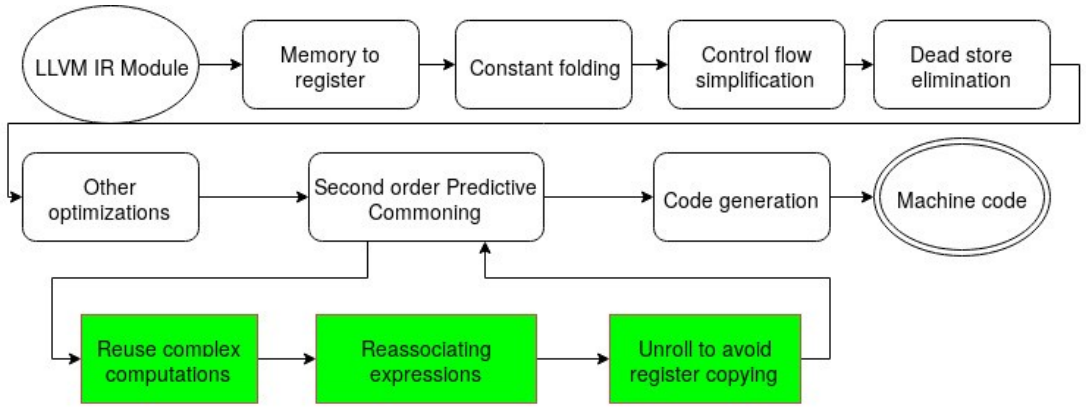

Figure 3.1 Architecture Diagram

The first module deals with identifying what parts of the code will be targeted for the optimization. LLVM API provides interfaces to identify and target code i.e. it helps to target only loops instead of running through the entire program.

3.2 SOFTWARE IMPLEMENTATION FLOW

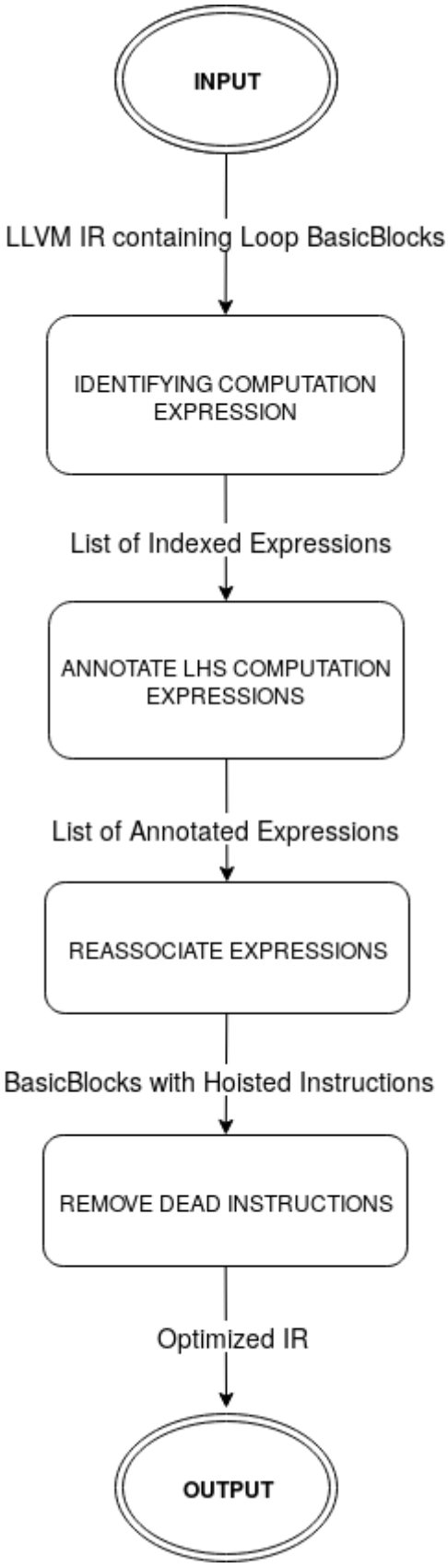

Figure 3.2 Software Flow Diagram

This overall architecture can be decomposed into a few highly cohesive modules. Each module is explained in detail in the following section.

3.3 LIST OF MODULES

1. Identify and annotate computation Expressions
2. Re-associate Expressions and remove dead Instructions

3.4 MODULES SPLIT-UP AND DETAILED DESIGN

3.4.1 Identify and annotate computation Expressions

This module is responsible for identification of array indexing operations. It does this by connecting the multiple instructions that make up an indexing expression and grouping them as one basic block.

It repeats the process for all the instructions and returns a list of such expressions.

Algorithm 1 Identifying indexed sequences

$V \leftarrow$ Empty vector
for each $getElementPtrInst$ **do**
  $dominantLoad \leftarrow findDominantLoad()$
  $dominatedLoad \leftarrow findDominatedLoad()$
  if $dominantLoad \neq$ null AND $dominatedLoad \neq$ null **then**
    $V.push(indexedexpressions)$
  end if
end for
return V

Once we have identified the necessary expressions, we then annotate them with useful meta data. It is one of the ways through which information is sent across passes. We use this to tag two pieces of meta data, namely

- Type of indexed expression

- Initial value of an indexed expression

Algorithm 2 Initial value calculation

```
for each GEPInstruction do
  dominantLoad ← findDominantLoad()
  for each instruction between dominantLoad AND
  GEPInstruction do
    if typeof instruction == arithmetic Op then
      if operand == constant then
        return constant
      end if
    end if
    return 0
  end for
end for
```

Algorithm 3 Annotate computation expressions

```
Input: Vector of indexed expressions V
for each indexed expression in V do
  node ← getSuccessor()
  if typeof node == load then
    Tag as LHS
    initialValue ← getIntialValue()
    Tag initialValue to indexed expression
  end if
end for
return V
```

3.4.2 Re-associate expressions and remove dead Instructions

The optimization work begins with this phase. We hoist the RHS indexed expressions out of the loop into new registers. After hoisting, these are then initialized with the value retrieved from the meta data attached earlier. We then proceed to add re-associating expressions i.e

the swap instructions between the registers.

Algorithm 4 Re-associating expressions

```
Input: Vector of tagged indexed expressions V
Create a BasicBlock above the loop header
for each RHS indexed expression do
    Create a register in the hoisted block
    Get initial value of indexed expression from meta data
    Load value into register
end for
```

Once all the optimizations are done, we need to find and eliminate all instructions which are now deemed unnecessary. Here, we make use LLVM's advanced Uses and Users APIs to complete the task.

Algorithm 5 Re-associating expressions

```
for each instruction in BasicBlock do
    users ← getAllUsers(instruction)
    if users is empty then
        removeInstruction(instruction)
        replaceAllUsesWith(undefinedValue)
    end if
end for
```

CHAPTER 4

IMPLEMENTATION AND RESULTS

4.1 IMPLEMENTATION

4.1.1 Identify and annotate computation Expressions

Input: IR of the un-optimized code is provided as input to the first pass of the Identifying common expression phase.

Output: Annotated version of the input code is obtained, with LHS and RHS expressions marked, and the optimizable computations annotated.

Code snippet for re-associating the Expressions

```
void setAllMetadata(BasicBlock *BB) {
LLVMContext& context = BB->getContext();
MDBuilder builder(context);
auto *RHS = MDNode::get(context, builder.createString("RHS"));
auto *LHS = MDNode::get(context, builder.createString("LHS"));
auto *deadInst = MDNode::get(context, builder.createString(""));
...
auto onlyGEPs = getGEPs(BB);
...
```

4.1.2 Re-associate expressions

Input: The annotated IR is sent as input to the reassociating expressions pass.

Output: The output IR consists of the first iteration array access hoisted out of the loop. A new basic block is added to apply the predictive commoning to the loop. All the remaining instructions are marked as dead in the IR.

Code snippet for annotating the Expressions

```
IRBuilder<> loopBuilder(lbody, lbody->getFirstInsertionPt());
int i = 0;
for (auto it : newAllocs) {
auto newLoad = loopBuilder.CreateAlignedLoad(it, 4);
oldLoads[i]->replaceAllUsesWith(newLoad);
i++;
}
i = 0;
for (auto store: oldStores) {
auto *user = cast<Instruction>(store->getOperand(0));
IRBuilder<> tmpLoad(store->getNextNode());
tmpLoad.CreateAlignedStore(user, tmpStores[i], 4);
for (auto it = newAllocs.rbegin(); it != newAllocs.rend(); ++it) {
auto prevInst = tmpLoad.CreateAlignedLoad(*it, 4);
tmpLoad.CreateAlignedStore(prevInst, *(++it), 4);
}
auto prevInst = tmpLoad.CreateAlignedLoad(tmpStores[i], 4);
tmpLoad.CreateAlignedStore(prevInst, *newAllocs.rbegin(), 4);
i++;
}
```

4.2 MODULES

4.2.1 Identify and annotate computation Expressions

The two modules are implemented in three passes, of which the first pass alone constitutes the first module. In the first pass, we iterate through the input IR to find the basic blocks with loops, using the `getLoopAnalysisUsage` class. After finding the loop containing block, we iterate through the basic blocks found in that block, to find initial iteration of the loop. We define `getInitial()` function to get the initial iteration. Upon getting the initial iteration, we check for dominant and dominated expressions. If a basic block inside the loop contains both dominant and dominated load expressions, we set the metadata using the `setAllMetaData()` function. The `MDNode::get()` function is used to retrieve the text Metadata node needed to annotate the expresssions, using the `setUsersMetadata(dominant, gepIT, deadInst)` function call. These steps are all iteratively done by invoking the `runOnLoop()` function.

4.2.2 Re-associate expressions and remove dead Instructions

The second pass is run on each function, and it hoists the indexed expressions out of the loop and adds a new basic block, to add the swapping instructions at the end. On finding the loop containing blocks, we get the annotations by using the `instr->getMetadata()` function.

By ensuring that the annotations are what we require, we create a new basic block, and also hoist the instructions out of the loop. `builder.CreateAlloca()` function is used to create a new allocation for the hoisted first iteration value. The old loads are collected, and pushed in a vector. Once the new loads with array access eliminated are constructed, all the old loads are replaced with the new loads.

`CreateAlignedStore` is used to create the replacing store instructions that will be used as the swapping block at bottom of the loop body.

```
IRBuilder<> loopBuilder(lbody, lbody->getFirstInsertionPt());
int i = 0;
for (auto it : newAllocs) {
auto newLoad = loopBuilder.CreateAlignedLoad(it, 4);
oldLoads[i]->replaceAllUsesWith(newLoad);
i++;
}
```

Replacing the instructions with IRBuilder

Once the two initial passes are done, we eliminate the dead instructions by eliminating its uses and then removing it using the `i->eraseFromParent()` function call.

4.3 TESTING

We test the program for correctness, and non breakage of code that has no potential for optimization. The module is compiled and run on several single source programs in the LLVM test suite [7]. They run successfully after being compiled with optimization switch, and do not introduce any errors in the generated code. Therefore, the optimized code runs as expected, with the added advantage of shorter execution time. The code with no potential for optimizations is left untouched by the optimization pass. Table 4.1 contains all the various programs that the module was tested against, and the effect for optimization.

Table 4.1 Test cases

Test Program from LLVM Suite	**Optimizable**	**RIP**	**Correctness**
Adobe-C++/stepanov_vector.cpp	No	-	Yes
Adobe-C++/benchmark.cpp	Yes	24%	Yes
Adobe-C++/loop_unroll.cpp	No	-	Yes
Stanford/BubbleSort.cpp	No	-	Yes
Stanford/FloatMM.cpp	No	-	Yes
Stanford/IntMM.c	No	-	Yes
Stanford/Queens.c	Yes	18%	Yes
Stanford/QuickSort.c	No	-	Yes
Stanford/Towers.c	No	-	Yes
Stanford/Treesort.c	No	-	Yes
Polybench/linear-algebra/solvers/dynprog	Yes	9%	Yes
Polybench/linear-algebra/solvers/gramschmidt	No	-	Yes
Shootout-C++/fibo.cpp	Yes	26%	Yes
Shootout-C++/hash.cpp	No	-	Yes

4.4 EXPERIMENTAL RESULTS

The proposed optimization is used on 3 different scenarios. The Relative Improvement Percentage (RIP) is the difference of optimization times between the two versions with and without applying the optimization.

$$RIP\% = \frac{T_{unoptimized} - T_{optimized}}{T_{optimized}} x100$$

If $RIP\%$ is negative, it is better to not apply the optimization.

We evaluate our optimization with three different scenarios which cover most use cases of our optimization

4.4.1 Performance evaluation

Scenario 1

In this scenario, we consider two array accesses in one statement. We use the code for calculating the Fibonacci sequence.

The results of applying our optimization are shown in Figure 4.1

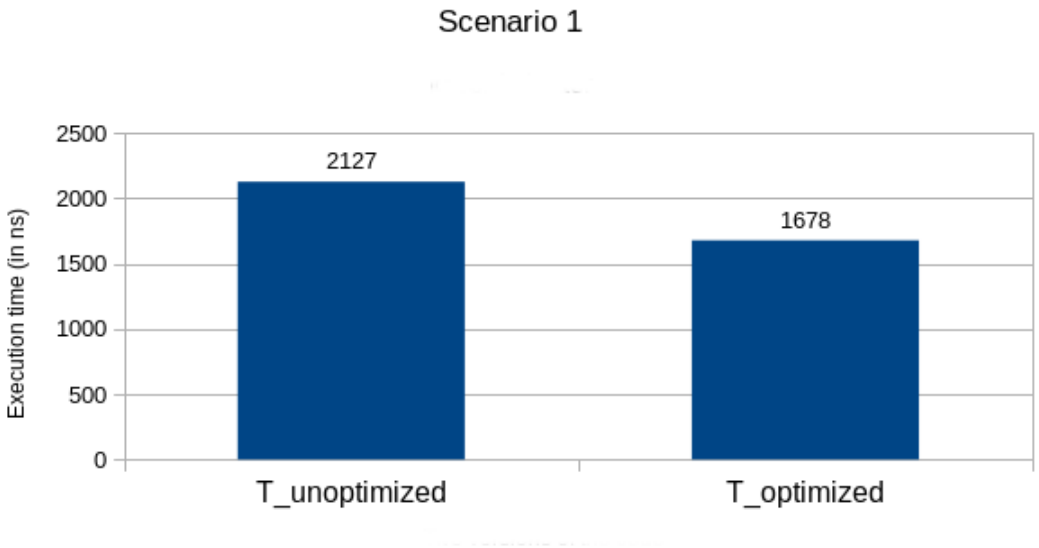

Figure 4.1 Execution times for Scenario 1

Scenario 2

The optimization quality is proportional to the number of array accesses in the loop. This is depicted by this scenario which involves only one array access.

The results of applying our optimization are shown in Figure 4.2

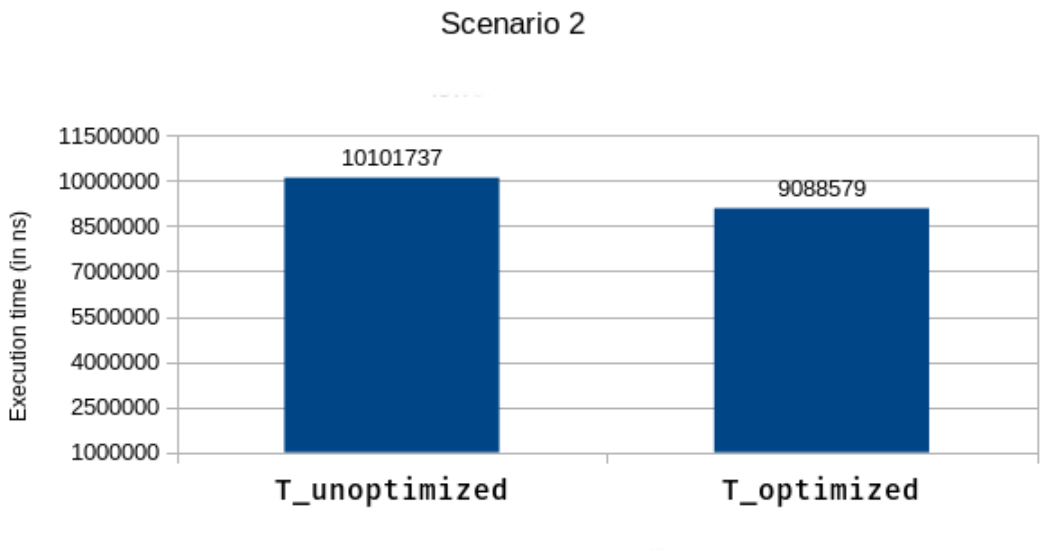

Figure 4.2 Execution times for Scenario 2

Scenario 3

Not applying the optimization when not needed is also an evaluation of our system. Therefore, this scenario contains code which cannot be optimized.

The results of applying our optimization are depicted in 4.3

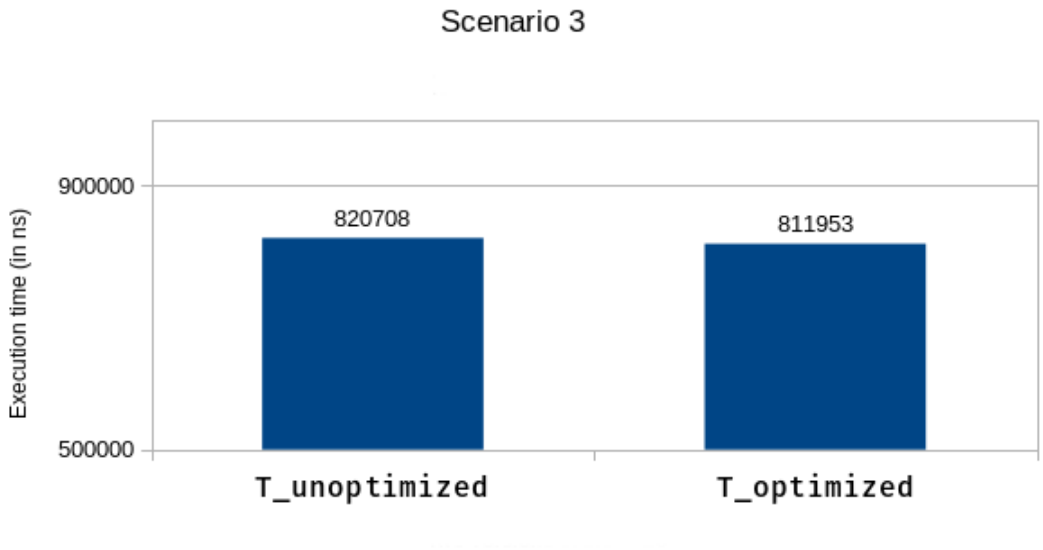

Figure 4.3 Execution times for Scenario 3

4.4.2 Evaluation

For each scenario, we then find the RIP according to formula specified in 4.4. The results are below

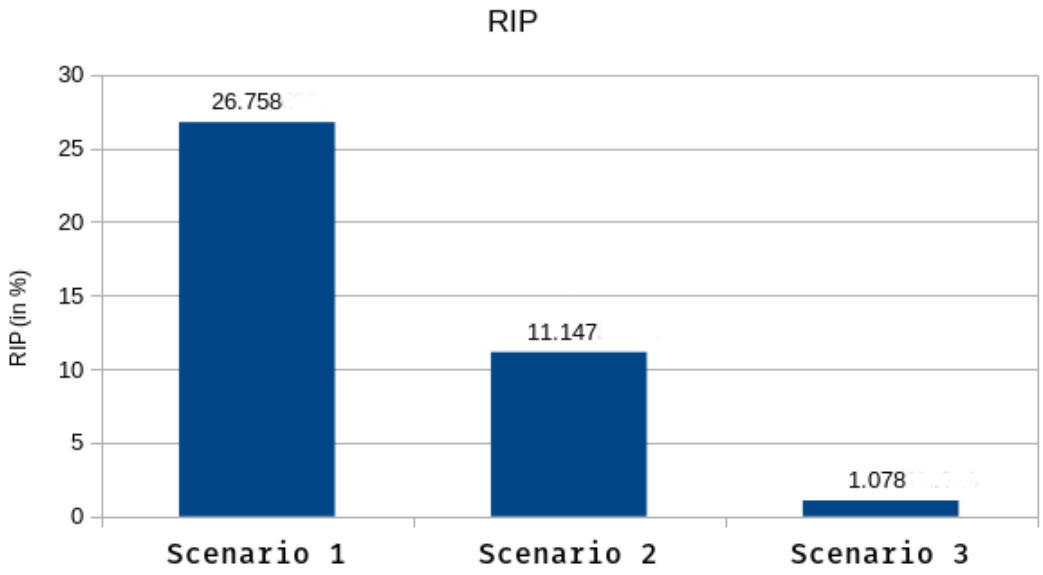

Figure 4.4 RIP of all scenarios

We can deduce the following observations

1. The performance is directly proportional to the number of array accesses present in the loop.
2. There is no improvement or deterioration when the optimization is not applicable.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 CONTRIBUTION

Array access is the costliest memory operation in a standard program, since other accesses do not involve any calculation before accessing the memory location itself. This project implements second order predictive commoning in LLVM compiler infrastructre as a loadable module. The loadablity ensures that it does not necessitate that entire LLVM infrastructure be compiled with this module to make use of this optimization. With appropriate version of Clang, we can dynamically load this module with any version of LLVM later than 3.

5.2 FUTURE WORK

The loadable module is invoked using a special flag when invoking the compiler. Second order predictive commoning can help drastically reduce the number of loads and stores, which are among the costliest memory operations. Array accesses are dominant in many calculations, including common operations like JPEG and MPEG decoding. All programs that have need for significant repetitive array accesses, may be compiled with this improvement in clang and LLVM, to take advantage of this fact. As an improvement over this project, we can eliminate the need to have a specific form of array access to be optimized, and generalize it to cover even more array access formats.

We can also send the patch to upstream developers, and merge it to

the mainline release in the next LLVM release schedule. This will enable us to get a closer code scrutiny, and will help us in identifying ways to further generalize this optimization, with the help of more experienced LLVM maintainers.

APPENDIX A

Setting up LLVM

All LLVM passes are subclasses of the Pass class, which implement functionality by overriding virtual methods inherited from Pass. Depending on how your pass works, you should inherit from the ModulePass , CallGraphSCCPass, FunctionPass , or LoopPass, or RegionPass, or BasicBlockPass classes, which gives the system more information about what your pass does, and how it can be combined with other passes. One of the main features of the LLVM Pass Framework is that it schedules passes to run in an efficient way based on the constraints that your pass meets (which are indicated by which class they derive from).

A.1 SETTING UP THE BUILD ENVIRONMENT

```
add_llvm_loadable_module( LLVMHello
Hello.cpp

PLUGIN_TOOL
opt
)
```

A.2 BASIC CODE REQUIRED

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
struct Hello : public FunctionPass {
static char ID;
Hello() : FunctionPass(ID) {}

bool runOnFunction(Function &F) override {
errs() << "Hello: ";
errs().write_escaped(F.getName()) << '\n';
return false;
}
}; // end of struct Hello
}  // end of anonymous namespace

char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass",
false /* Only looks at CFG */,
false /* Analysis Pass */);
```

REFERENCES

[1] Alexander Aiken and Alexandru Nicolau, "Optimal loop parallelization", In *ACM SIGPLAN Notices*, volume 23, pp. 308–317, 1988.

[2] Gianluca Amato, Maurizio Parton, and Francesca Scozzari, "Discovering invariants via simple component analysis", *Journal of Symbolic Computation*, vol. 47, num. 12, pp. 1533–1560, 2012.

[3] David Callahan, Steve Carr, and Ken Kennedy, "Improving register allocation for subscripted variables", *ACM Sigplan Notices*, vol. 25, num. 6, pp. 53–65, 1990.

[4] Chris Lattner and Vikram Adve, "LLVM: A compilation framework for lifelong program analysis & transformation", In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, p. 75, 2004.

[5] Taewook Oh, Hanjun Kim, Nick P Johnson, Jae W Lee, and David I August, "Practical automatic loop specialization", In *ACM SIGARCH Computer Architecture News*, volume 41, pp. 419–430, 2013.

[6] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V Nori, "A data driven approach for algebraic loop invariants", In *European Symposium on Programming*, pp. 574–592, 2013.

[7] LLVM Maintainer Team, "LLVM Test Suite". https://github.com/llvm-mirror/test-suite, 2017.

[8] Michael E Wolf and Monica S Lam, “A data locality optimizing algorithm”, In *ACM SIGPLAN Notices*, volume 26, pp. 30–44, 1991.

[9] Michael E Wolf and Monica S Lam, “A loop transformation theory and an algorithm to maximize parallelism”, *IEEE Transactions on Parallel and Distributed systems*, vol. 2, num. 4, pp. 452–471, 1991.