J S S Mahavidyapeetha
Sri Jayachamarajendra College of Engineering
(SJCE), Mysore - 570 006
An Autonomous Institute Affiliated to
Visvesvaraya Technological University, Belgaum

“Loop Fusion in LLVM Compiler”

Thesis submitted in partial fulfillment of curriculum prescribed for the award of the degree of

BACHELOR OF ENGINEERING
IN
COMPUTER SCIENCE AND ENGINEERING

by

Madhura Dinesh Kaushik
(4JC11CS058)
Sridhar G
(4JC11CS116)
Supreeth M S
(4JC11CS123)

Under the Guidance of
Sri. H D Nandeesh
Assisstant Professor,
Department of Computer Science & Engineering,
SJCE, Mysore

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
FEB 2015

J S S Mahavidyapeetha
Sri Jayachamarajendra College of Engineering
(SJCE), Mysore - 570 006
An Autonomous Institute Affiliated to
Visvesvaraya Technological University, Belgaum

CERTIFICATE

This is to certify that the work entitled **"Loop Fusion in LLVM Compiler"** is a bonafide work carried out by **Madhura Dinesh Kaushik, Sridhar G and Supreeth M S** in partial fulfillment of the award of the degree of **Bachelor of Engineering in Computer Science and Engineering of Visvesvaraya Technological University, Belgaum during the year 2014-15.** It is certified that all corrections / suggestions indicated during seminar presentation have been incorporated in the report. The project report has been approved as it satisfies the academic requirements in respect of project work prescribed for the Bachelor of Engineering degree.

Guide
Mr. H D Nandeesh
Assisstant Professor
Dept. of CS&E
SJCE, Mysore

Head of the Department
Dr. C N Ravikumar
Professor & Head
Dept. of CS&E
SJCE, Mysore

Place:Mysore
Date:19th March, 2015

Examiners: 1. ....
2. ....
3. ....

1 Abstract

This project is the implementation of a pass for loop fusion in LLVM compiler. Two loops, which are adjacent and have the same condition and increments with respect to the loop variable may be fused, i.e, their bodies may be executed one after the other with in a single loop. The decision to fuse the loops is taken based on the legality and profitability of the fusion. It should not be performed if the resulting code has anti-dependency or if the execution time of the program increases.

2 Acknowledgement

“**Experience is the best teacher**”, is an old adage. The satisfaction and pleasure that accompany the gain of experience would be incomplete without the mention of the people who made it possible.

First and foremost we pay due regards to this renowned institution, which provided us a platform and an opportunity for working on this project.

Our sincere regards to **Dr. Syed Shakeeb Ur Rahman**, Principal of S.J.C.E. for providing us an opportunity to enhance our knowledge by working on this project. We thank **Dr. C.N. Ravikumar**, Professor and Head of department of Computer science and Engineering for the implementation of plan of the project in our curriculum to enhance the knowledge and skills of the students.

We would like to express gratitude towards our guide, **Mr. H D Nandeesh** for his invaluable co-operation and support.

We are thankful to **Mr. N R Prashanth**, **Mr. Vikram T V** and **Mr. Vaivaswatha N** for sharing expertise, and their sincere and valuable guidance and encouragement extended to us.

Contents

List of Figures

3 Introduction

Loops are statements that allow us to execute one or more statements multiple times. Consider the two loops shown below:

```
for ( i = 0; i <= 10; i++)
{
        a[i] = i % 2;
}

for ( i = 0; i <= 10; i++)
{
        a[i] = a[i] + 2;
}
```

The statements inside both the loops are executed 11 times, once for each value of the variable i. i is called as the induction variable of the loop. These two loops have the same bounds, that is i from 0 to 10 and have the same increment for the loop variable, that is i++. Hence they can be fused into a single loop as follows :

```
for ( i = 0; i < 10; i++)
{
        a[i] = i % 2;
        a[i] = a[i] + 2;
}
```

3.1 Aim/Objective of the project work

The LLVM pass aims to improve the runtime of a program by fusing loops. In software engineering, it is often a better approximation that 90% of the execution time of a computer program is spent executing 10% of the code (known as the 90/10 law in this context). Most of this 10% of the code is comprised of loops of the program. So any optimization with respect to loops leads to better and/or faster code. This can be possible due to two aspects of the source code its ability to exploit temporal and spatial cache locality and its ability to reuse registers as much as possible and hence avoid spills to memory.

A CPU cache is a cache used by the central processing unit (CPU) of a computer to reduce the average time to access data from the main memory. The cache is a smaller, faster memory which stores copies of the data from frequently used main memory locations. Most CPUs have different independent caches, including instruction and data caches, where the data cache is usually organized as a hierarchy of more cache levels. Cache locality means that the same value or memory location is accessed again and again. There are two basic types of locality temporal and spatial. Temporal locality refers to situations where a resource that is used now is accessed again with in a small amount of time. Spatial locality means that a memory location might be accessed in a short time if a memory location near it has been accessed recently. If the same arrays or variables are being accessed in the loops that are to be fused, they need to be brought only once instead of twice into the cache. In such a case, all accesses with in the second loop are converted to cache access. On the other extreme, if all the variables are different and together they exceed the capacity of the cache, all accesses become memory accesses, decreasing the performance of the program.

Loops which access the same memory locations or elements in the statements within their bodies allow reuse of registers after fusion. For example, consider the loops given below:

```
for ( i = 0; i < 10; i++)
{
        A[i] = C[i] + D[i];
}

for ( i = 0; i < 10; i++)
{
        B[i] = C[i] * D[i];
}
```

In the above code, all the elements of the arrays C and D are loaded twice into registers; once for each loop. The first loop runs through all the elements of C and D before the second loop accesses any element of either array.

There is another advantage to be gained by fusing the loop. We can reuse the operands in the registers while executing the statements of the second loop. After fusion, the code looks like this:

```
for ( i = 0; i < 10; i++)
{
        A[i] = C[i] + D[i];
        B[i] = C[i] * D[i];
}
```

In the above fused code, in each iteration, the appropriate elements of arrays C and D need to be fetched only once for the two statements rather than twice (once for each loop) as in the original code. The pass implemented in this project tries to fuse those loops that result in better exploitation of cache locality and maximum register reuse.

3.2 Existing Solution

LLVM is a fairly new compiler, which was started in the year 2000. There does not exist a loop fusion pass for this infrastructure as yet. Hence there is no existing solution for the project. However other loop transformation passes such as LoopSimplify which is used to canonicalize natural loops and LoopUnroll which is used to unroll the iterations of a loop have been implemented in C++ language. Some of these passes are applied to the Intermediate Representation (IR) to transform it before applying the loop fusion transformation.

3.3 Proposed Solution

The first step towards the solution is to understand the LLVM infrastructure and its Intermediate Representataion (IR). Passes in LLVM can be of two types analysis passes and transform passes. Analysis passes compute information that other passes can use for debugging or for program visualization purposes. Analysis passes do not modify the program in any way. Transform passes can use or invalidate the analysis passes. Transform passes all change the program in some way. Loop fusion is a transforn pass.

Loops in LLVM have a unique structure. They are made up of basic blocks that include a preheader, a header (also called the exiting block), a loop latch and an exit block. To work on loops knowledge about this structure is necessary.

Next we write a function pass in C++ to achieve fusion of two loops after performing the basic checks of adjacency, same predicates and conditions. Fusion involves changing the IR, deleting the old loop, removing the references and updating the internal data structure (LoopInfo). Opt is used to run the pass on the IR generated from a C program. We then upgrade the pass to fuse more than two loops iteratively.

After the basic fusion is over, we add a module that prevents loops from fusing in the presence of anti dependence in the fused code. Anti dependence (also called Write After Read) in the fused code gives wrong results. Lastly, another module is added to check the profiatbility of fusing. Fusion may not always result in faster execution. The loop overhead is always reduced. However, the degree of exploitation of cache locality actually determines how advantageous fusion can be. If the cache is smaller than the number of elements to be accessed in one iteration of a loop, the elements are repeatedly replaced in the cache from the memory. Thus all cache accesses before fusion are converted to memory accesses after fusion. In such a case, fusion does more bad than good and is avoided.

Finally, after the pass has been written, it is tested on various inputs. Clang is a tool that is the front end for C in LLVM. It is used to generate the IR on which the pass should be run. The outputs of the program before and after loop fusion are compared.

3.4 Time Schedule(Gantt Chart)

The various phases involved in the project are Literature Survey, System design, System Implementation, System Testing and Report Writing. The time break up (includes start date and finish date) of the various phases of the project are as shown in the below gantt chart. All the mentioned phases have been completed till date according to the schedule.

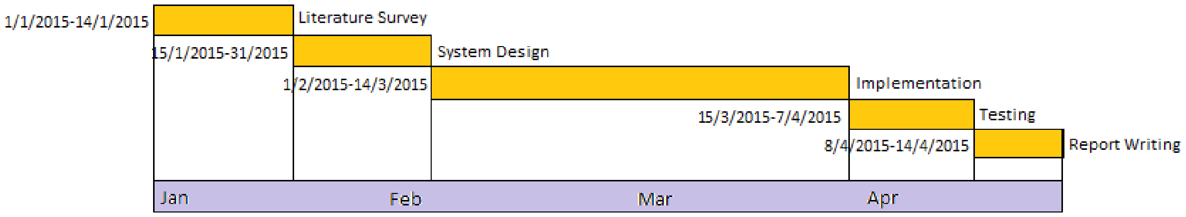


Figure 1: Gantt chart for project schedule

4 System Requirements and Analysis

LLVM stands for Low Level Virtual Machine. LLVM project was started in 2000 which is relatively new when compared to GCC, started in 1987. LLVM can provide the middle layers of a complete compiler system, taking intermediate form (IF) code from a compiler and emitting an optimized IF. This new IF can then be converted and linked into machine-dependent assembly code for a target platform. LLVM can accept the IF from the GCC toolchain, allowing it to be used with a wide array of extant compilers written for that project[3].

LLVM/Clang has a modular design which will help in static code analysis. LLVM supports a language-independent instruction set and type system. Each instruction is in static single assignment form (SSA), meaning that each variable (called a typed register) is assigned once and is frozen. This helps simplify the analysis of dependencies among variables. LLVM allows code to be compiled statically, as it is under the traditional GCC system, or left for late-compiling from the IF to machine code in a just-in-time compiler (JIT) fashion similar to Java. The type system consists of basic types such as integers or floats and five derived types: pointers, arrays, vectors, structures, and functions. A type construct in a concrete language can be represented by combining these basic types in LLVM. For example, a class in C++ can be represented by a combination of structures, functions and arrays of function pointers. LLVM is distributed under Open Source License.

Before the loop fusion pass is run some other passes need to be run on the IR. They are -

1. mem2reg - The mem2reg pass converts non-SSA form of LLVM IR into SSA form, raising loads and stores to stack-allocated values to “registers” (SSA values). Many of LLVM optimization passes operate on the code in SSA form and so does our loop fusion pass. This pass implements the standard “iterated dominance frontier” algorithm for constructing SSA form and construct the PHI nodes.

2. indvars - This transformation analyzes and transforms the induction variables (and computations derived from them) into simpler forms suitable for subsequent analysis and transformation. All loops are transformed to have a single canonical induction variable which starts at zero and steps by one. The canonical induction variable is guaranteed to be the first PHI node in the loop header block. Any pointer arithmetic recurrences are raised to use array subscripts. The exit condition for the loop is canonicalized to compare the induction value against the exit value. Any use outside of the loop of an expression derived from the indvar is changed to compute the derived value outside of the loop, eliminating the dependence on the exit value of the induction variable. If the only purpose of the loop is to compute the exit value of some derived expression, this transformation will make the loop dead.

3. instcombine - Combine instructions to form fewer, simple instructions. This pass does not modify the CFG. This pass is where algebraic simplification happens. This pass can also simplify calls to specific well-known function calls (e.g. runtime library functions). For example, a call exit(3) that occurs within the main() function can be transformed into simply return 3. Whether or not library calls are simplified is controlled by the -functionattrs pass and LLVMs knowledge of library calls on different targets.

4. instnamer - This is a little utility pass that gives instructions names, this is mostly useful when diffing the effect of an optimization because deleting an unnamed instruction can change all other instruction numbering, making the diff very noisy.

5. loop - simplify - This pass is used to canonicalize natural loops. It performs several transformations to transform natural loops into a simpler form, which makes subsequent analyses and transformations simpler and more effective. Loop pre-header insertion guarantees that there is a single, non-critical entry edge from outside of the loop to the loop header. This simplifies a number of analyses and transformations. Loop exit-block insertion guarantees that all exit blocks from the loop (blocks which are outside of the loop that have predecessors inside of the loop) only have predecessors from inside of the loop (and are thus dominated by the loop header). This simplifies transformations such as store-sinking. This pass also guarantees that loops will have exactly one backedge. This pass obviously modifies the CFG, but updates loop information and dominator information.

5 Tools and Technology used

1. Clang: Clang is a compiler front end for the C, C++, Objective-C and Objective-C++ programming languages. It uses LLVM as its back end. It is designed to offer a complete replacement to the GNU Compiler Collection (GCC). It is open-source, developed by Apple; other companies such as Google, ARM, Sony and Intel are involved. Its source code is available under the University of Illinois/NCSA License.

   Clang is used in our project to compile the test cases into the .bc file. BC file is the file which consists of the IR representation of the input program which can be in any high level language.

Clang can provide the IR of the high level program in two formats:

(a) BC file

(b) LL file

The LL file is the user readable text format of the IR representation. This can opened in any of the standard text editors and can be examined to understand the Intermediate Representation (IR). It can even be edited to make some small changes to be incorporated into it. This flexibility is very helpful when changing a small thing in the IR without having to re-compile the whole thing.

A BC file is the binary representation of the LL file. This means that the IR of the input program is saved in the Binary format. This cannot be understood by the user and it cannot be edited by any editors. Since most of the computations done on the IR use the binary version of the same, this will be helpful and saves time in converting the user readable format into binary. Even if the input IR is not in the binary form, most of the systems convert it before using the same.

The commands used to get the LL and BC file of the input program are as follows:

```
clang -emit-llvm -S input.c – For LL file
clang -emit-llvm -C input.c – For BC file
```

2. OPT: The opt command is the modular LLVM optimizer and analyzer. It takes LLVM source files as input, runs the specified optimizations or analyses on it, and then outputs the optimized file or the analysis results.

   This command is extensively used in LLVM to run user-defined passes in the entire compiler architecture. First, the user writes a pass which may be ModulePass, FunctionPass, LoopPass or BasicBlockPass. These user-defined passes needs to be registered to the PassManager before it is ready to be run along with other passes. After the pass has been successfully compiled, it needs to be run. The command to run the pass using OPT is as follows:

```
opt -load path/to/.so/file -passName -S output.ll ¡ input.ll
```

In the above command, we are specifying the following things:
-load This specifies that the command needs to load the .so file and run on the input IR
-passName In a single pass, we can specify more than one passes. This will give the flexibility for the user to write multiple passes and to call the required one when running that .so file. Here, the required pass present in the so file is specified.

-S This specifies the output IR should be present in the LL format. This is in contrary of specifying the -C option, which gives the output IR in the BC format. The input file name and output file name is also specified in the usual conventions.

3. CSCOPE:Cscope is a console mode or text-based graphical interface that allows computer programmers or software developers to search C source code. It is often used on very large projects to find source code, functions, declarations, definitions and regular expressions given a text string. cscope is free and available under a BSD License.

   Since the code-base of the LLVM compiler architecture is huge, it becomes very difficult to find the required C++ files where some classes are defined and also to find the member functions available with a particular Class. At times it will become hard to find the required file as well.

   This problem is solved by using CSCOPE. What CSCOPE does is that, it creates a database of the list of the funcitons present in a particular file and where it is used and many other information.

   The process of creating this database is as follows: First, the list of files which needs to be indexed are created. This is usually done by using the 'find' command in UNIX. The source files are present in the 'src' directory and this directory is scanned and is used to index all the C++ files. Secondly, the list of filenames are given to the CSCOPE to create the database of the code using the command:

   ```
   cscope -d
   ```

   If the files are changed, then the database needs to be re-built.

4. CTAGS: Ctags generates an index (or tag) file of language objects found in source files that allows these items to be quickly and easily located by a text editor or other utility. A tag signifies a language object for which an index entry is available (or, alternatively, the index entry created for that object).

   It supports many programming languages. It is capable of generating tags for all types of C/C++ language tags, including class names, macro definitions, enumeration names, enumerators, function definitions, function prototypes/declarations, class, interface, struct, and union data members, structure names, typedefs, union names and variables (definitions and external declarations) It supports user-defined languages with the help of regular expressions. It supports the output of Emacs-style TAGS files. IT can also be used to print out a list of selected objects found in source files.

5. BYOBU: Byobu is an enhancement for the terminal multiplexers GNU Screen or tmux that can be used to provide on screen notification or status as well as tabbed multi window management. It is aimed at providing a better user experience for terminal sessions when connecting to remote servers.

   Byobu can be configured to run by default at every text login (SSH or TTY). That behavior can be toggled with the byobu-enable and byobu-disable commands -
   byobu-enable
   and
   byobu-disable

6. C++: C++ is a general-purpose programming language. It has imperative, object-oriented and generic programming features, while also providing the facilities for low-level memory manipulation.

   It is designed with a bias toward system programming (e.g., for use in embedded systems or operating system kernels), with performance, efficiency and flexibility of use as its design requirements. It is thus the natural choice for progrmming a pass for a compiler. C++ has also been found useful in many other contexts, including desktop applications, servers (e.g. e-commerce, web search or SQL servers), performance-critical applications (e.g. telephone switches or space probes), and entertainment software. C++ is a compiled language, with implementations of it available on many platforms and provided by various organizations, including the FSF, LLVM, Microsoftand Intel.

7. Git: Git is a distributed revision control system with an emphasis on speed, data integrity, and support for distributed, non-linear workflows. Git was initially designed and developed by Linus Torvalds for Linux kernel development in 2005, and has since become the most widely adopted version control system for software development. With the help of git we can easily keep track of the various versions of our code and revert back to the last stable version if something goes wrong.

8. Bitbucket: Bitbucket is a web-based hosting service for projects that use either the Mercurial or Git revision control systems. Bitbucket offers both commercial plans and free accounts. It offers free accounts with an unlimited number of private repositories. Bitbucket is written in Python using the Django web framework. Bitbucket is especially useful to share code among various members of the team. It allows people other than the owner of the code to view and download it. It is also used to keep a backup of the code. The official website is https://bitbucket.org/

6 Literature Survey

LLVM(2000) is a relatively new compiler compared to gcc(1987). The comparison of their performance is shown in the figure. It shows the time taken to execute a particular benchmark using both the compilers. The time taken by LLVM (29.61 seconds) is almost half as the time taken by GCC (around 60 - 61 seconds). Hence, though LLVM is still evolving, it gives a tough competition to GCC and is already being used in many applications.

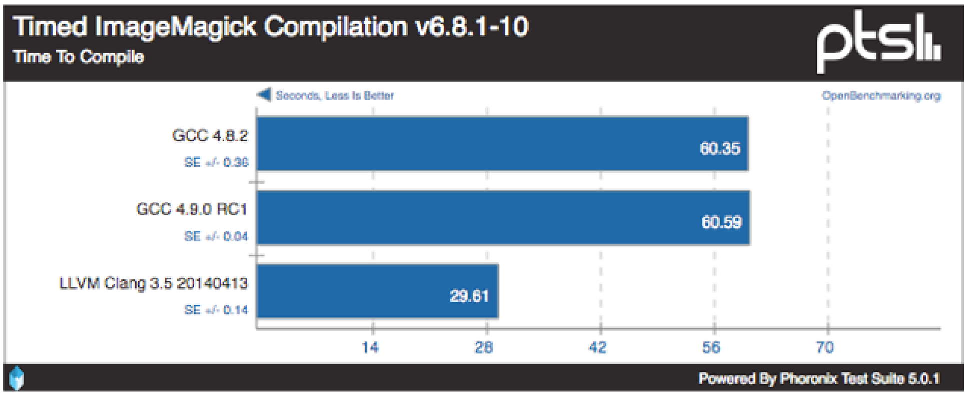
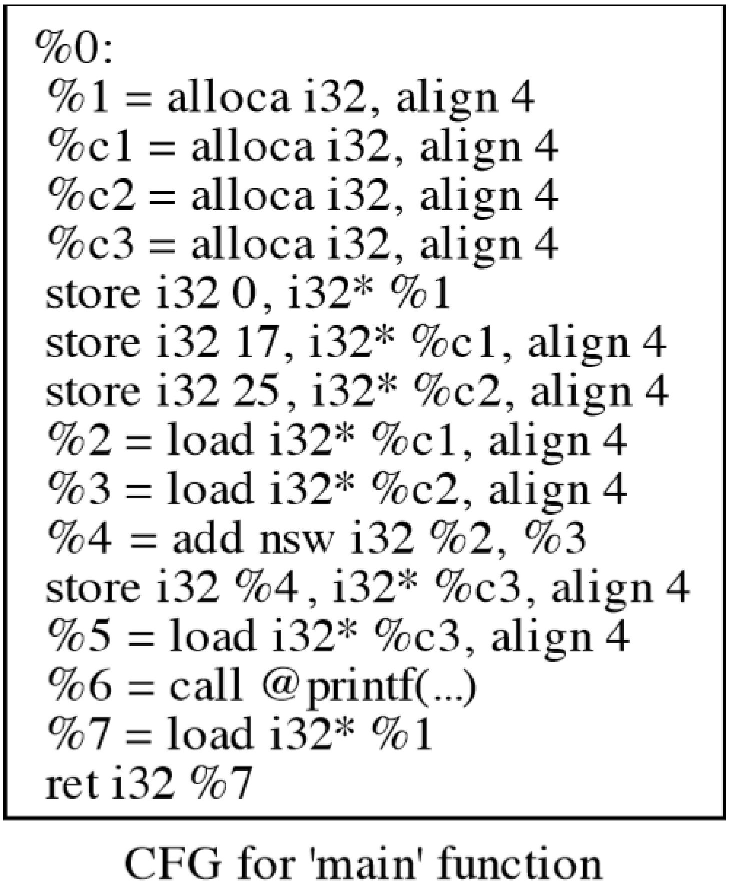


Figure 2: Performance comparison with GCC

The literature survey of this project involved understanding the LLVM infrastructure, mainly the LLVM IR and the Loop Represntation in the LLVM IR and also to know about the Dependencies that may occur between two loops. Some research papers and some chapters in books were used as the source. This chapter of the report is divided into sub-sections which are important in understanding LLVM and to implement Loop Fusion.

6.1 LLVM IR

The core of LLVM is the intermediate representation (IR), a low-level programming language similar to assembly. IR is a strongly typed RISC instruction set which abstracts away details of the target. For example, the calling convention is abstracted through call and ret instructions with explicit arguments. Additionally, instead of a fixed set of registers, IR uses an infinite set of temporaries of the form %0, %1, etc. LLVM supports three isomorphic forms of IR: a human-readable assembly format, a C++ object format suitable for frontends, and a dense bitcode format for serialization. The IR is generated using a tool called clang which is the C language front end for LLVM. The IR for a simple C program to add two numbers and display the result is as shown in the figure below



```
%0:
 %1 = alloca i32, align 4
 %c1 = alloca i32, align 4
 %c2 = alloca i32, align 4
 %c3 = alloca i32, align 4
 store i32 0, i32* %1
 store i32 17, i32* %c1, align 4
 store i32 25, i32* %c2, align 4
 %2 = load i32* %c1, align 4
 %3 = load i32* %c2, align 4
 %4 = add nsw i32 %2, %3
 store i32 %4, i32* %c3, align 4
 %5 = load i32* %c3, align 4
 %6 = call @printf(...)
 %7 = load i32* %1
 ret i32 %7
```

CFG for 'main' function

Figure 3: Intermediate Representation in LLVM

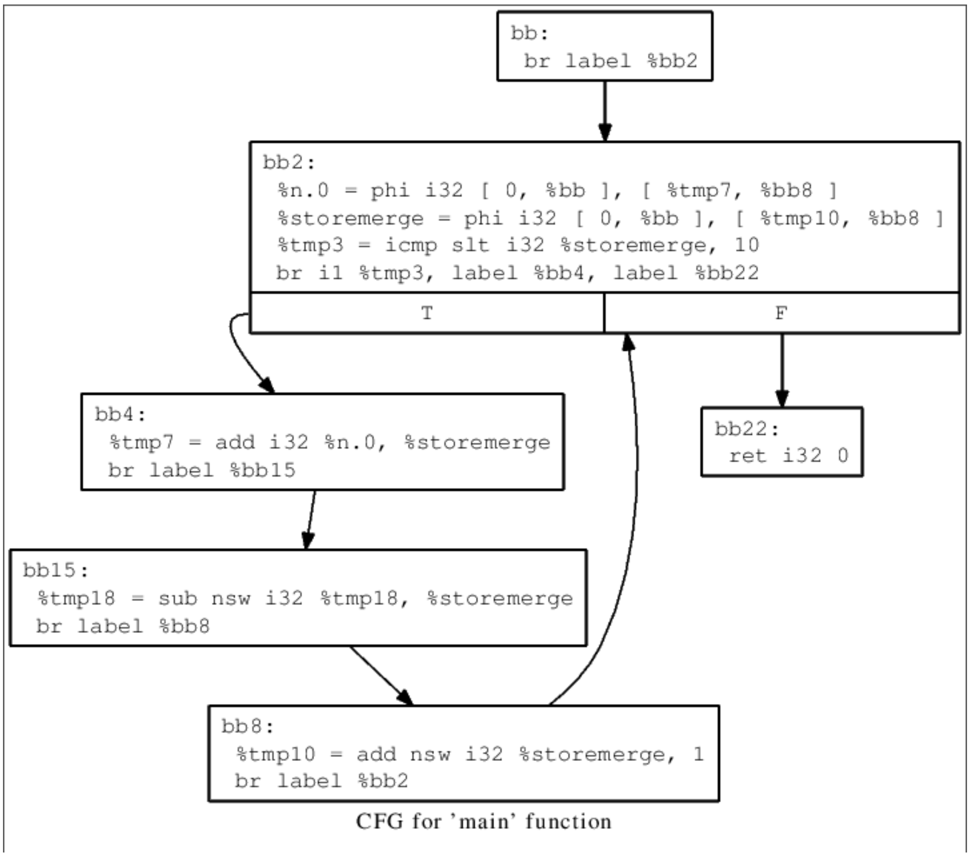
6.2 Loop Structure in LLVM IR

The picture clearly depicts the structure of the loop in LLVM IR. This structure of the loop is obtained after the input file is optimized using the -loop-simplify pass before giving as input to the Loop Fusion pass. This pass will make sure that all the loops present in the program are in the above structure irrespective of the state they were in before the transformation was applied onto it.

The instructions in the IR is represented as basic blocks. A basic block is a portion of the code within a program with only one entry point and only one exit point. This makes a basic block highly amenable to analysis. Compilers usually decompose programs into their basic blocks as a first step in the analysis process. The blocks to which control may transfer after reaching the end of a block are called that block's successors, while the blocks from which control may have come when entering a block are called that block's predecessors. The start of a basic block may be jumped to from more than one location. Basic blocks form the vertices or nodes in a control flow graph (CFG).

The main components of the Loop are explained as follows:

1. Loop Preheader: Loop Pre-header is the type of the Basic Block where in the initializations required for the loop are defined. This will usually be the place where the loop initialization variables will be declared. This part of the Basic Block may



```
bb:
 br label %bb2
```

```
bb2:
 %n.0 = phi i32 [ 0, %bb ], [ %tmp7, %bb8 ]
 %storemerge = phi i32 [ 0, %bb ], [ %tmp10, %bb8 ]
 %tmp3 = icmp slt i32 %storemerge, 10
 br i1 %tmp3, label %bb4, label %bb22
```

```
bb4:
 %tmp7 = add i32 %n.0, %storemerge
 br label %bb15
```

```
bb22:
 ret i32 0
```

```
bb15:
 %tmp18 = sub nsw i32 %tmp18, %storemerge
 br label %bb8
```

```
bb8:
 %tmp10 = add nsw i32 %storemerge, 1
 br label %bb2
```

CFG for 'main' function

Figure 4: Loop Structure in LLVM

not always be present for every loop. Hence, the Loop Simplify pass will make sure that this block is the predesessor of the Loop Header which will be exaplined later.

This block will make sure that the Loop Header is always the ONLY predesessor of the loop. This type of assumptions is mainly used during loop transformations.

2. Loop Header: Loop Header is usually the place where the initialization of the induction variable takes place. As explained before, induction variable is the one which changes during the course of the loop. It is the variable whose value is used and is incremented by a specific value for each iteration of the loop.

   Always, PHI Node is the first instruction of the Loop Header. This is to make sure that the Loop Header and the Induction variable is in SSA form. Static Single Assignment (SSA) is the form in which each variable is defined only once. The LLVM IR is in SSA form which enhances its capabilities.

   Header block is executed for every iteration of the loop. This is in contrast to Loop Pre-header, which is executed only once and is present outside the loop. The loop header is also called as the exiting block since it has a successor that is outside the loop.

3. Loop Body: Loop Body is the place where the actual contents of the loops exists. This is the place where the most of the logic of the loop exits. This may consist of more than one block of instructions. Each block in the loop body ends with a branch statement which branches to the Loop Header, Loop Exit-Block or to any other block inside the loop body.

   Body of the loop will be defined in such a way to accomplish a particular task such that each iteration of the loop completes part of that whole task. This project attempts to fuse the body of two loops. After fusing bodies of two loops, the last block of the first loop will be pointing to the first block of the second loop and the last block of the second loop will be pointing to the first block of the first loop. By doing this, we are fusing the bodies of two loops such that both the bodies are executed for iteration of the loop. The two bodies of the loop should be such that the cache locality of the two loops is satisfied and is both legal and profitable to fuse the two loops.

4. Loop Latch: Loop Latch is the block which is present as the last block of the loop. This block branches out to Loop Header. Thus this becomes the block predesessor of the Loop Header. Hence, the PHI Node present in the first instruction of the loop header should have an entry of the variable coming from the Loop Latch. The Loop Latch has a back edge to the Loop Header which is essential for a loop.

   Loop Latch basically contains the increment or decrement operations on the induction variable. This is the block where the inducation variable evolves through the loop. If the induction variable is a canonical induction variable, then the induction will ALWAYS be incremented by 1.

5. Exit Block: The exit block is the basic block to which the control branches when the loop condition becomes false. After each iteration the condition in is checked in the header block and if it evaluates to false, the branch is taken to the exit block instead of the body of the loop.

   The exit block usually has either some code or just a branch. In case of adjacent loops, the exit block of the first loop always contains only one instruction which is a branch instruction to the header of the second loop. In this case, the exit block of the first loop is also the preheader of the second loop.

6.3 Data Dependency

A data dependence exists between two instructions if both of them refer to the same memory location, and at least one of them involves a write or store. In compiler theory, the technique used to discover data dependencies among statements (or instructions) is called dependence analysis. There are three types of dependencies: data, name, and control.[7]

Data dependence exits when an instruction uses (reads or writes) a register or memory location which another instruction uses. The instructions that are data dependent on one another cannot be reorder in a program. There are three types of data dependences-

1. True dependence (RAW) - A true data dependencs refers to a situation where an instruction refers to a result that has not yet been calculated or retrieved. In this case an instruction uses as an operand data that has not been stored by a previous instruction yet.

   ```
   x = 1;
   y = x + 2;
   ```

   The second instruction reads x after the first instruction writes to it.

2. Anti-dependence (WAR) An anti-dependence refers to a situation where an instruction stores data after another instruction uses it as an operand.

   ```
   a = b + 10;
   b = 7;
   ```

   The second instruction stores a value to b after the first instruction reads it.

3. Output dependence (WAW) Output dependence exists when two instruction write to the same location. The order of the stores should be preserved to ensure that the instructions following the second store obtain the correct value when they try to read from that location.

   ```
   x = 10;
   y = x * 1;
   x = 20;
   z = x + 1;
   ```

   Both the first and the third instructions store into x. The values of x read by the second and fourth instructions are different.

The is another way to classify the dependences in the program based on whether the dependence still exists in the absence of the loop. There are two types of dependences in this category - Loop-Independent Dependence and Loop-Carried Dependence. In Loop-carried dependence dependence exists across iterations and if the loop is removed, the dependence no longer exists. For example, consider the code

```
for( i = 0; i < 10; i++ )
{
        a[i] = b[i];
}

for( i = 0; i < 10; i++ )
{
        c[i] = a[i+1];
}
```

In the above code the loops effectively store b[i+1] to c[i] using array a. However, when we fuse, the old values in array a is stored into array c instead of the new values in array a which is actually the values of array b. The unrolled iterations of the loops before and after fusion clearly highlight this issue.

Loop-independent dependence is dependence exists that within an iteration; i.e., if the loop is removed, the dependence still exists. This kind of dependence does not create any problems while fusing of loops and hence need not be checked for.

```
//Loops Unrolled Before fusion

a[0] = b[0]; //Loop 1
a[1] = b[1];
a[2] = b[2];

c[0] = a[1]; //Loop 2
c[1] = a[2];
c[2] = a[3];

//Loops Unrolled After Fusion

a[0] = b[0];
c[0] = a[1];

a[1] = b[1];
c[1] = a[2];

a[2] = b[2];
c[2] = a[3];
```

If after fusion, there exists anti-dependence between within the loop body, the loops cannot be fused. This is because in this the value stored in a iteration will be read in the next iteration. Wrong outputs will be produced if loops are fused in this case.

6.4 Cache Locality

Cache locality of reference is a phenomenon describing the same value, or related storage locations, being frequently accessed. There are two basic types of reference locality temporal and spatial locality. Temporal locality refers to the reuse of specific data, and/or resources, within a relatively small time duration. Spatial locality refers to the use of data elements within relatively close storage locations.

- Temporal locality If at one point in time a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future. There is a temporal proximity between the adjacent references to the same memory location. In this case it is common to make efforts to store a copy of the referenced data in special memory storage, which can be accessed faster. Temporal locality is a special case of the spatial locality, namely when the prospective location is identical to the present location.

- Spatial locality If a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future. In this case it is common to attempt to guess the size and shape of the area around the current reference for which it is worthwhile to prepare faster access.

With respect to loop fusion, if the same locations are being accessed in the loops that are to be fused, they need to be brought only once instead of twice into the cache. In such a case, all accesses with in the second loop are converted to cache access instead of memory accesses. On the other hand, if all the locations are different and together they exceed the capacity of the cache, all accesses become memory accesses, degrading the performance of the program.

6.5 Register Reuse

Although most programmers would naturally write loop nests that achieve a high degree of reuse, there are many situations in which fusion of loop nests might produce good results. An important example is when the loop nests are produced as a result of some form of preprocessing, such as when Fortran 90 array statements are converted to scalar loops. Consider the following Fortran 90 example:

```
A(1:N) = C(1:N) + D(1:N)
B(1:N) = C(1:N) * D(1:N)
```

In this case both statements use identical sections of C and D. It is clear from these statements that the comment elements should be reused from registers. Loop fusion, will bring the references back together so the operands can be reused:

```
DO I = 1, 256
A(I) = C(I) + D(I)
B(I) = C(I) * D(I)
ENDDO
```

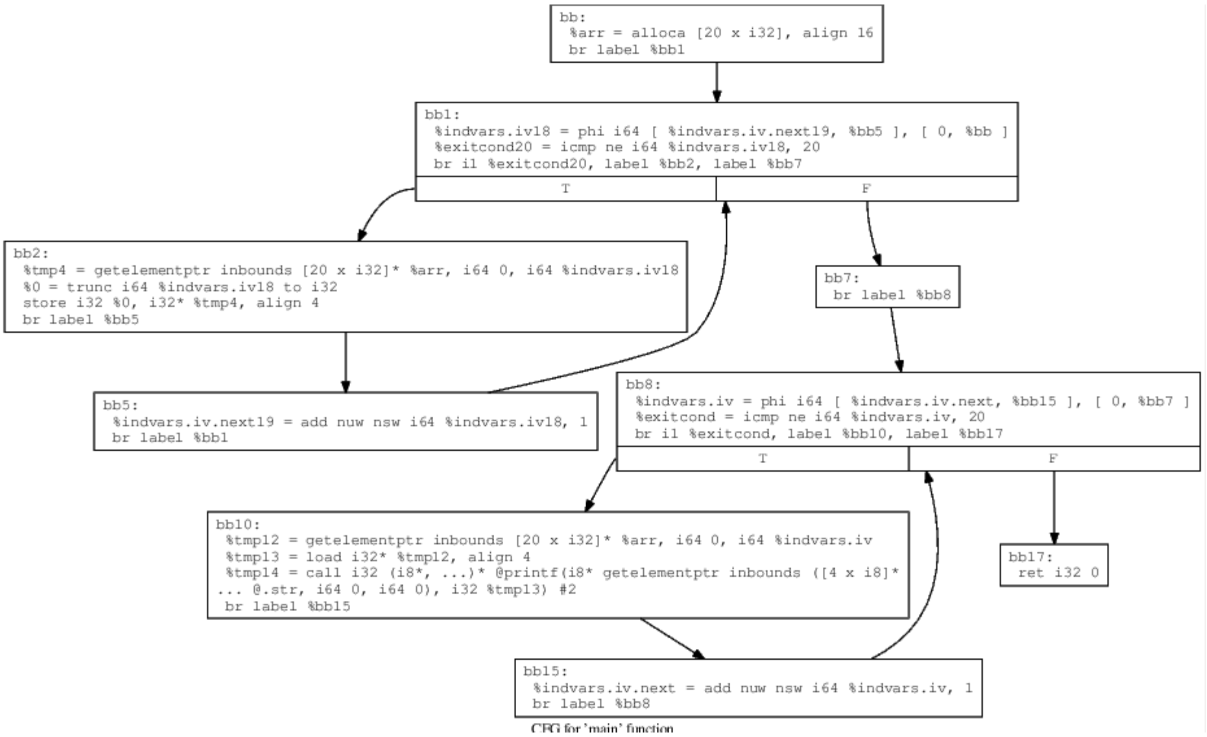
The appropriate sections of C and D need be fetched only once for the two statements, rather than twice as in the original scalarized code.

7 System Design

A control flow graph (CFG) is a representation, using graph notation, of all paths that might be traversed through a program during its execution. In a control flow graph each node in the graph represents a basic block, i.e. a straight - line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves. For example, consider the code:

```
for( i = 0;i < 10; i++ )
                  arr[i] = i ;
for( i = 0;i < 10; i++  )
                  printf ( "%d\t", arr[i] ) ;
```

The CFG for the above is:



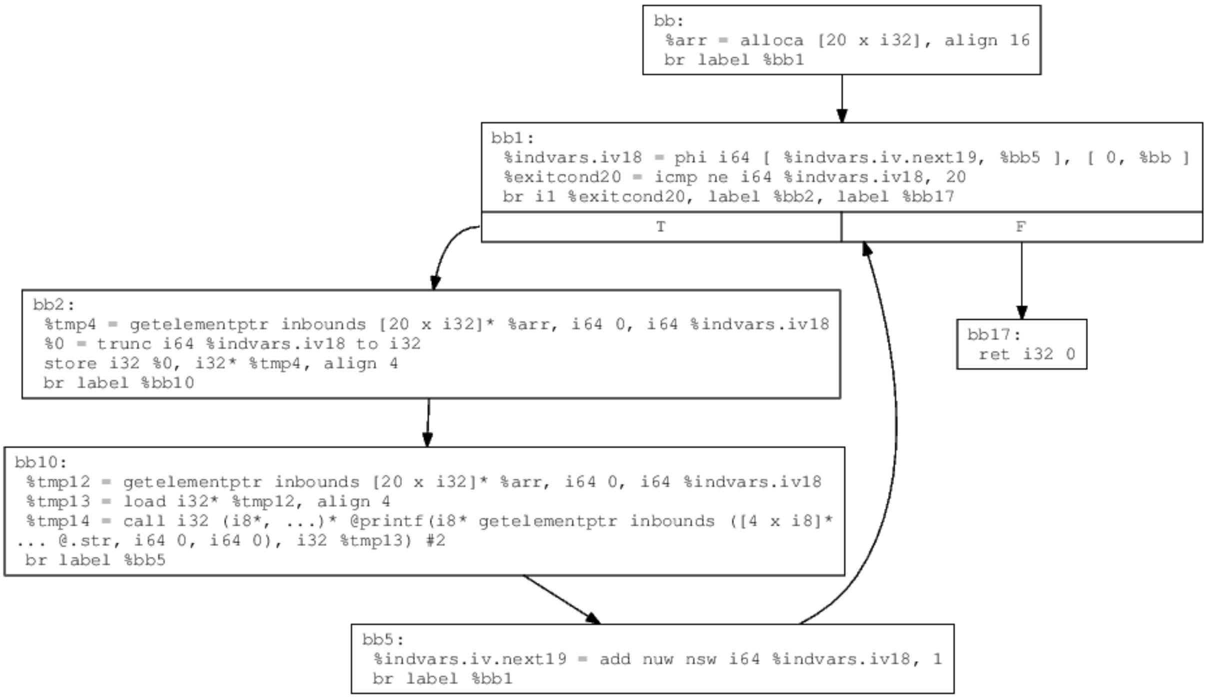
```
bb:
 %arr = alloca [20 x i32], align 16
 br label %bb1

bb1:
 %indvars.iv18 = phi i64 [ %indvars.iv.next19, %bb5 ], [ 0, %bb ]
 %exitcond20 = icmp ne i64 %indvars.iv18, 20
 br i1 %exitcond20, label %bb2, label %bb7

bb2:
 %tmp4 = getelementptr inbounds [20 x i32]* %arr, i64 0, i64 %indvars.iv18
 %0 = trunc i64 %indvars.iv18 to i32
 store i32 %0, i32* %tmp4, align 4
 br label %bb5

bb7:
 br label %bb8

bb5:
 %indvars.iv.next19 = add nuw nsw i64 %indvars.iv18, 1
 br label %bb1

bb8:
 %indvars.iv = phi i64 [ %indvars.iv.next, %bb15 ], [ 0, %bb7 ]
 %exitcond = icmp ne i64 %indvars.iv, 20
 br i1 %exitcond, label %bb10, label %bb17

bb10:
 %tmp12 = getelementptr inbounds [20 x i32]* %arr, i64 0, i64 %indvars.iv
 %tmp13 = load i32* %tmp12, align 4
 %tmp14 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]*
... @.str, i64 0, i64 0), i32 %tmp13) #2
 br label %bb15

bb17:
 ret i32 0

bb15:
 %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
 br label %bb8
```

Figure 5: Basic CFG for 2 loops

If we tranform the loop though loop fusion, we replace the two loops with a single one. After loop fusion of the above code, it would become

```
for( i = 0;i < 10; i++ )
{
                  a[i] = i;
                  b[i] = i;
}
```

The CFG for the above is:



```
bb:
 %arr = alloca [20 x i32], align 16
 br label %bb1
```

```
bb1:
 %indvars.iv18 = phi i64 [ %indvars.iv.next19, %bb5 ], [ 0, %bb ]
 %exitcond20 = icmp ne i64 %indvars.iv18, 20
 br i1 %exitcond20, label %bb2, label %bb17
```

```
bb2:
 %tmp4 = getelementptr inbounds [20 x i32]* %arr, i64 0, i64 %indvars.iv18
 %0 = trunc i64 %indvars.iv18 to i32
 store i32 %0, i32* %tmp4, align 4
 br label %bb10
```

```
bb17:
 ret i32 0
```

```
bb10:
 %tmp12 = getelementptr inbounds [20 x i32]* %arr, i64 0, i64 %indvars.iv18
 %tmp13 = load i32* %tmp12, align 4
 %tmp14 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]*
... @.str, i64 0, i64 0), i32 %tmp13) #2
 br label %bb5
```

```
bb5:
 %indvars.iv.next19 = add nuw nsw i64 %indvars.iv18, 1
 br label %bb1
```

CFG for 'main' function

Figure 6: CFG for fused loops

LLVM provides quite a few optimization algorithms that work on the IR. These algorithms are organized as passes. Each pass does something specific, like combining redundant instructions. Passes need not always optimize the IR, it can also do other operations like inserting instrumentation code, or analyzing the IR (the result of which can be used by passes that do optimizations) or even printing call graphs.

LLVM does not automatically choose to run any passes, anytime on its own. Passes have to be explicitly selected and run on each module. This gives you the flexibility to choose transformations and optimizations that are most suitable for the code in the module.

There is an LLVM binary called opt, which lets you run passes on bitcode files from the command line. You can write your own passes (in C/C++, as a shared library). This can be loaded and executed by +opt+.

A pass manager“ is responsible for loading passes, selecting the correct objects to run them on (for example, a pass may work only on functions, individually) and actually runs them. opt is a command-line wrapper for the pass manager.

LLVM defines two kinds of pass managers-the FunctionPassManager manages function or basic-block passes. These lighter weight passes can be used immediately after each generated function to reduce memory footprint. The PassManager manages module passes for optimizing the entire module.

FunctionPasses may overload three virtual methods to do their work. All of these methods should return true if they modified the program and false if they didn't.

virtual bool doInitialization(Module &M);
The doInitialization method is designed to do simple initialization type of stuff that does not depend on the functions being processed.

virtual bool runOnFunction(Function &F) = 0;
The runOnFunction method will run for every function.

virtual bool doFinalization(Module &M);
The doFinalization method is called when the pass framework has finished calling runOnFunction for every function in the program being compiled.

The project is mainly divided into 4 modules

1. Basic checks: The for loop will have 4 parts-initialization statement, test expression, update statement and code. Before fusing the 2 loops, we need to check if these constraints of the loops match. If the loops are

```
for( i = 0;i < 10; i++ )
                a[i] = i;
for( j = 0; j < 10; j++ )
                b[j] = j;
```

   The Initialization part i.e. i = 0 and j = 0 match. i.e. both the induction variables are assigned with 0. The condition of the 2 loops, i ¡ 10 and j ¡ 10 match. i.e. both the loops run until their respective induction variables become 10. Both the variables are incremented by 1. Hence the above code passed the first module. Such variables which run from 0 to some value and are incremented by 1 after each iteration are called as Canonical Induction Variables.

   Lets consider a case where we cannot fuse the loops. In the code below, the array a is modified after the first loop and before the second loop. If we fuse, the old values in array a is accessed in the second loop which conmputes the wrong results.

```
for( i = 0;i < 10; i++ )
{
                arr[i] = i  ;
}
arr[7] = 70;
for( i = 0;i < 10; i++  )
{
                sum += arr[i];
}
```

2. Legality (Checking dependencies): There are situations in which fusion of loop nests might produce wrong results, for example, in case of anti-dependency between instructions across iteratons of the loop. Anti-dependency means that one instruction writes into a location that is read in the previous instruction. Hence for each instruction of the body of loop 1 that involves either a load or store, we check if dependency exists between it and each instruction that involves a load or store in the body of the second loop.

3. Profitability: There are many situations in which fusion of loop nests might produce good results. But there are cases where fusing loops might take more time. For example, consider the loops

```
for( i = 0; i < 100; i++ ){
                a[i] = i;
                b[i] = i;
                c[i] = i;
}

for( i = 0; i < 100; i++ ){
                a[i] = i;
                d[i] = i;
                e[i] = i;
}
```

While running the first loop, the 3 arrays will be loaded onto the cache, and the corresponding indices of the array will be updated. If we fuse the loops the result would be as shown below

```
for( i = 0; i < 100; i++ )
{
                a[i] = i;
                b[i] = i;
                c[i] = i;
                a[i] = i;
                d[i] = i;
                e[i] = i;
}
```

Since the cache size is limited and the number of array elements to be loaded into the cache exceeds the size of the cache, it can't load all the arrays into the cache simultaneously. Hence in each iteration, it dynamically loads the arrays into the cache on demand. The number of memory accesses is increased in this case as the cache content is changed within the iteration. In this case instead of just accessing the cache the memory has to be accessed as well. As we know the time taken for a memory access is at least 100 times more than the time taken for a cache access. Hence we need to check if fusing the loops will really increase the performance by decreasing the runtime and if it doesn't, the loops should remain as they are.

4. Loop Fuse: If the loops pass all the above 3 tests, then it can be fused. In LLVM's representation of loops, every loop will have a header, a body, a latch and a exit block. Since all the conditions for the loops taken in the example passed, i.e. header, latch and exit block had the same values in both the loops, they are fused. The Algorithm is as follows

   (a) Replace the use of induction variable of 2nd loop with that of 1st loop

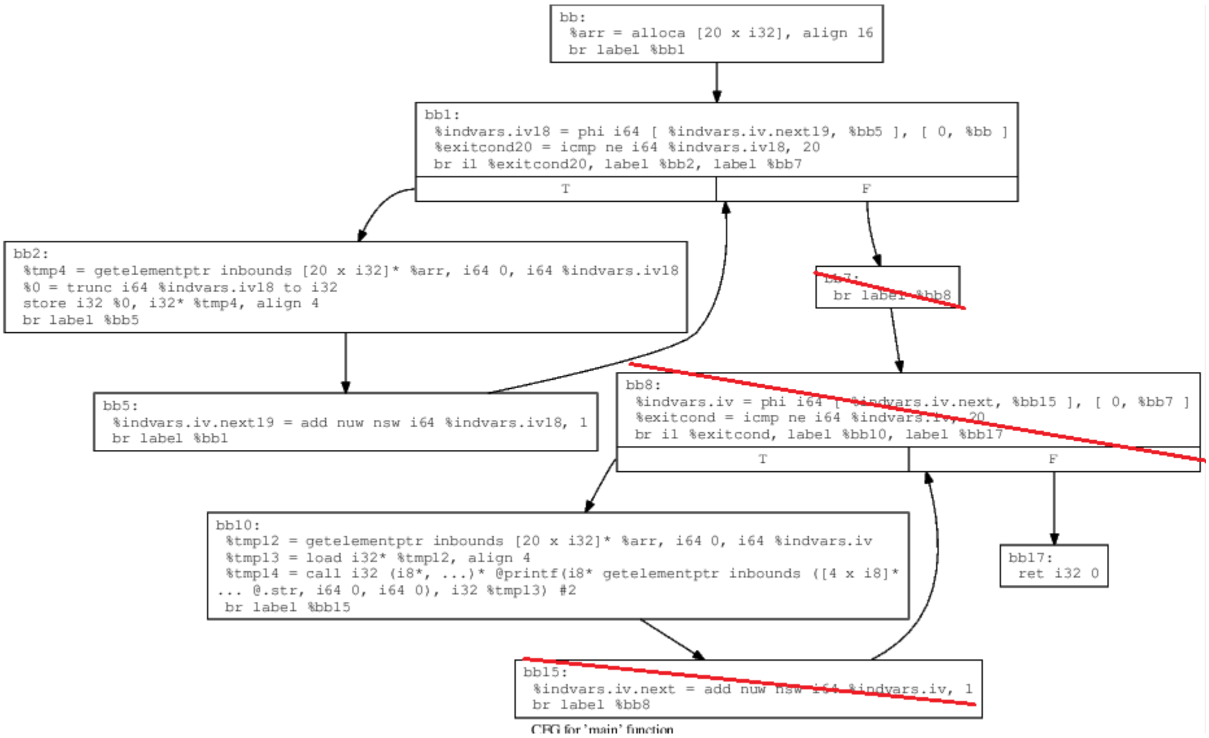
   (b) Combine the bodies of loop 1 and loop 2

   (c) Set the succesor of 2nd loop's body to exit block of 1st loop

   (d) Point the exit block of loop 1 to exit block of loop 2

   (e) Delete the unwanted basic blocks of 2nd loop

The unwanted blocks of the 2nd loop mentioned in the last step of the above algorithm are the header, latch and the exit block of loop 2 (basically all basic blocks other than those that form the body of the loop). The blocks that have to deleted are shown striked in figure 7.



```
bb:
 %arr = alloca [20 x i32], align 16
 br label %bb1
```

```
bb1:
 %indvars.iv18 = phi i64 [ %indvars.iv.next19, %bb5 ], [ 0, %bb ]
 %exitcond20 = icmp ne i64 %indvars.iv18, 20
 br i1 %exitcond20, label %bb2, label %bb7
```

```
bb2:
 %tmp4 = getelementptr inbounds [20 x i32]* %arr, i64 0, i64 %indvars.iv18
 %0 = trunc i64 %indvars.iv18 to i32
 store i32 %0, i32* %tmp4, align 4
 br label %bb5
```

```
bb7:
 br label %bb8
```

```
bb5:
 %indvars.iv.next19 = add nuw nsw i64 %indvars.iv18, 1
 br label %bb1
```

```
bb8:
 %indvars.iv = phi i64 [ %indvars.iv.next, %bb15 ], [ 0, %bb7 ]
 %exitcond = icmp ne i64 %indvars.iv, 20
 br i1 %exitcond, label %bb10, label %bb17
```

```
bb10:
 %tmp12 = getelementptr inbounds [20 x i32]* %arr, i64 0, i64 %indvars.iv
 %tmp13 = load i32* %tmp12, align 4
 %tmp14 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]*
... @.str, i64 0, i64 0), i32 %tmp13) #2
 br label %bb15
```

```
bb17:
 ret i32 0
```

```
bb15:
 %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
 br label %bb8
```

Figure 7: Blocks to be Deleted

In compiler design, Static Single Assignment form (often abbreviated as SSA form or simply SSA) is a property of an intermediate representation (IR), which requires that each variable is assigned exactly once, and every variable is defined before it is used. Existing variables in the original IR are split into versions and new variables are typically indicated by the original name with a subscript, so that every definition gets its own version. All LLVM instructions are represented in the Static Single Assignment (SSA) form. Essentially, this means that any variable can be assigned to only once. Such a representation facilitates better optimization, among other benefits. A consequence of single assignment are PHI () nodes. These are required when a variable can be assigned a different value based on the path of control flow. For example, consider the value of b at the end of execution of the snippet below:

```
a = 1;
if (v < 10)
    a = 2;
b = a;
```

The value of b cannot be determined statically. The value of 2 cannot be assigned to the original a, since a can be assigned to only once. There are two a s in there, and the last assignment has to choose between which version to pick. This is accomplished by adding a PHI node:

```
a1 = 1;
if (v < 10)
     a2 = 2;
b = PHI(a1, a2);
```

The PHI node selects a1 or a2, depending on from where the control reached the PHI node. The argument a1 of the PHI node is associated with the block a1 = 1;“ and a2 with the block a2 = 2;“. Hence for step 1, we need to replace all uses of 2nd PHINode with 1st PHINode inside the 2nd loop.

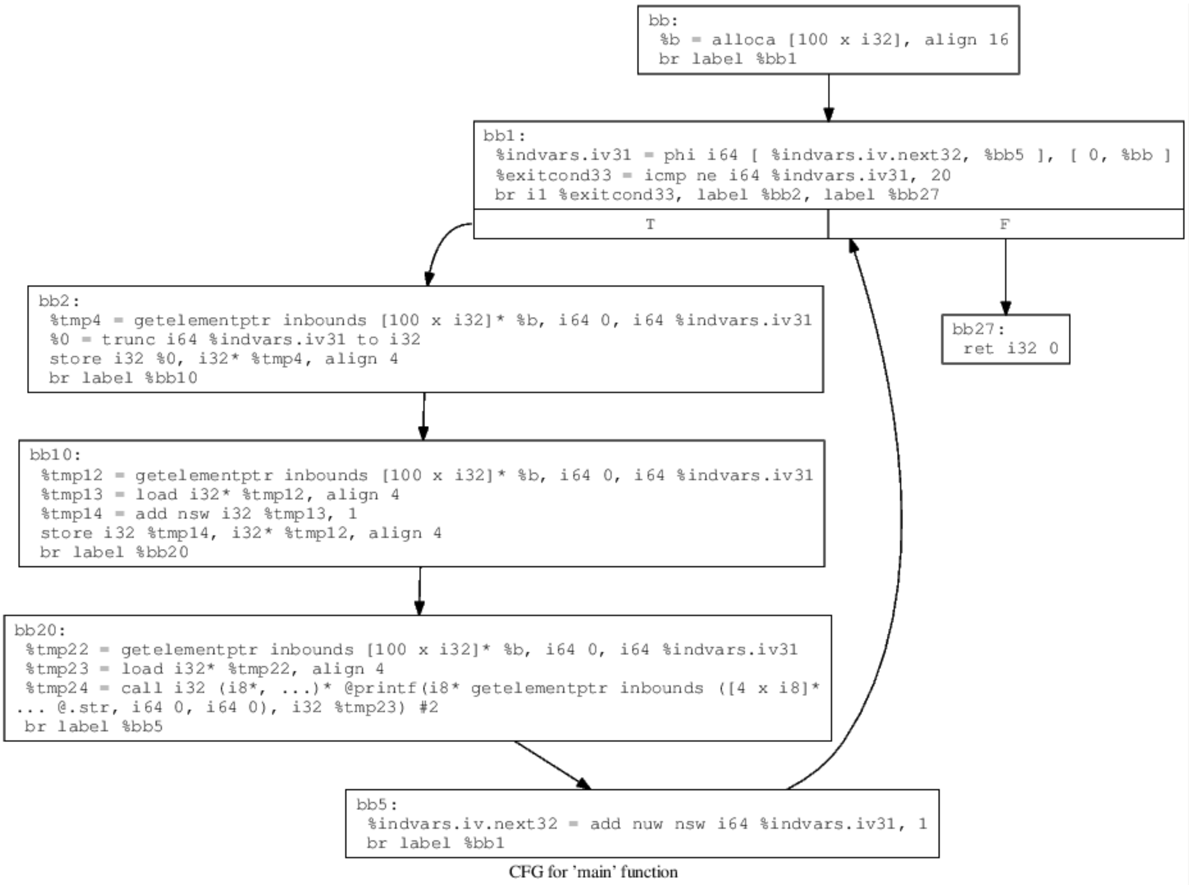
5. Fusing multiple loops: If there were more than 2 loops and if there is a possibility to fuse, then the Algorithm should be followed for all the loops. If it passes the 3 modules defined, it is good to fuse.

```
for ( i = 0 ; i < n ; i ++ )
                b[i] = i ;

for ( i = 0 ; i < n ; i ++ )
                b[i]++ ;

for ( i = 0 ; i < n ; i ++ )
                printf ( "%d\t", b[i] ) ;
```

From the loops, it is clear that loops 1 and 2 can be fused Again, loop 3 can also be fused with the loop obtained after fusing loop 1 and loop 2. Hence the output will be

```
for( i = 0; i < n; i++ )
{
                b[i] = i ;
                b[i]++ ;
                printf ( "%d\t", b[i] ) ;
}
```

The CFG for the above will be



```
bb:
 %b = alloca [100 x i32], align 16
 br label %bb1

bb1:
 %indvars.iv31 = phi i64 [ %indvars.iv.next32, %bb5 ], [ 0, %bb ]
 %exitcond33 = icmp ne i64 %indvars.iv31, 20
 br i1 %exitcond33, label %bb2, label %bb27
T | F

bb2:
 %tmp4 = getelementptr inbounds [100 x i32]* %b, i64 0, i64 %indvars.iv31
 %0 = trunc i64 %indvars.iv31 to i32
 store i32 %0, i32* %tmp4, align 4
 br label %bb10

bb27:
 ret i32 0

bb10:
 %tmp12 = getelementptr inbounds [100 x i32]* %b, i64 0, i64 %indvars.iv31
 %tmp13 = load i32* %tmp12, align 4
 %tmp14 = add nsw i32 %tmp13, 1
 store i32 %tmp14, i32* %tmp12, align 4
 br label %bb20

bb20:
 %tmp22 = getelementptr inbounds [100 x i32]* %b, i64 0, i64 %indvars.iv31
 %tmp23 = load i32* %tmp22, align 4
 %tmp24 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]*
... @.str, i64 0, i64 0), i32 %tmp23) #2
 br label %bb5

bb5:
 %indvars.iv.next32 = add nuw nsw i64 %indvars.iv31, 1
 br label %bb1
```

CFG for 'main' function

Figure 8: CFG for nested fused loops

8 System Implementation

For the transformation to occur, we need to write a function pass. A FunctionPass executes on each function in the program independent of all of the other functions in the program. FunctionPasses do not require that they are executed in a particular order, and FunctionPasses do not modify external functions. The implemention of the system is in C++. The header file which contains all the functions used in the program is as given below

```
//       Header file for the LoopFuse pass

#include ”llvm/IR/Instructions.h”
#include ”llvm/IR/Constants.h”
#include ”llvm/Analysis/LoopInfo.h”
#include ”llvm/Analysis/DependenceAnalysis.h”
#include ”llvm/Analysis/ScalarEvolution.h”
#include ”llvm/Support/raw_ostream.h”
```

```
using namespace llvm ;

class LoopFuse: public FunctionPass
{
        private:

        // DATA MEMBERS:

        // LoopInfo used to store all loop information and is
           accessible to all member functions
        LoopInfo *LI ;

        ScalarEvolution *SE ;

        // These two vectors are used to store the body of the
           two loops
        std::vector<BasicBlock *> blocks1, blocks2 ;

        // Function varialbe obtained from runOnFunction() and
           is accessible to all member functions
        Function *Fun ;

        // Vector to store all the Loop * contents
        std::vector<Loop *> loopVector ;

        // enum to store the type of edge
        enum edge_type { BAD_EDGE, ANTI, TRUE, OUTPUT } ;

        // Structure to store the data of the Graph edge
        struct graphEdge
        {
                edge_type type ;
                int count ;
                bool set ;
        } **graphMatrix ;

        // MEMEBER FUNCTIONS:

        // This function is used to ask the PassManager to run
           any dependent passes before running this pass
        void getAnalysisUsage ( AnalysisUsage &AU ) const ;

        // Function template to swap any two values
        template <class T>
        void swap ( T &a, T &b ) ;
```

```
// Function to return the loop termination condition ( i
   .e. < <= > >= == != )
int predicateNum ( Loop *l ) ;

// Function to return the loop limiting value ( i.e. if
   i<20 is the condition , returns 20
int limitValue ( Loop *l ) ;

// Function to check if the two loops are adjacent
bool adjacent ( Loop *l1 , Loop *l2 ) ;

// Function to remove all the references of all
   instructions in a particualar Basic Block
void dropReference ( BasicBlock *b ) ;

// Delete a Block
void deleteBlock ( BasicBlock *b ) ;

// Function to remove un-wanted basic blocks from the
   function
void cleanUp ( Loop *l ) ;

// Function to get dependency present between two loops
void getDependency ( Loop *l1 , Loop *l2 ) ;

// Function to fuse the body of the two loops
void fuseBody ( Loop *l1 , Loop *l2 ) ;

// Function to perform basic checks on the two loops
bool basicCheck ( Loop *l1 , Loop *l2 ) ;

// Helper function to check and fuse two loops
bool fuseCheck ( Loop *l1 , Loop *l2 ) ;

// Function to display the contents of the LoopInfo.
   TEMPORARY. Delete later
void display ( LoopInfo *LI ) ;

// Disaply Sub-Loops
void displaySubLoops ( Loop *l ) ;

// Dependency Analysis
bool analyzeLoop ( Loop *l1 , Loop *l2 ) ;

// Build blocks present in the fused loop
std::vector<BasicBlock *> buildBlocks ( Loop *l ) ;
```

```
        // Function to build the LoopVector using LoopInfo
           contents
        void buildVector () ;

        // Function to initialize the graph matrix and also the
           contents of the graph edge
        void initGraph () ;

        // Function to build the Graph of loops
        void buildGraph () ;

        // Function to update the graph of loops
        void updateGraph ( Loop *l1 , Loop *l2 , edge_type type )
           ;

        // Function to Dump Graph of loops
        void dumpGraph () ;

        public:
        static char ID ;

        LoopFuse() : FunctionPass(ID) { }

        // Main function which implements LoopFuse
        bool runOnFunction ( Function &F ) ;

} ;
```

We have seen that there are 4 steps in the loop fusion

1. Basic checks
2. Legality
3. Profitability
4. Loop Fuse

If the 2 loops are taken to be l1 and l2, to check if they are adjacent, we first check if the pre-header of the second loop and the exit block of the first loop are the same. If they are different, there is some code in the input program between the 2 loops and they are not adjacent. If they are the same, we further check whether that basic block has only one instruction, which is the branch instruction to the header of the second loop. Also, the case where 2 loops have more than one basic block between them but still are adjacent because those basic blocks have only branch instructions are handled. Such cases arise when we try to fuse the loop obtained after the fusion of two loops with a third loop. The code is as follows

```
bool LoopFuse::adjacent ( Loop *l1, Loop *l2 )
{
        BasicBlock *b1 = l1->getExitBlock() ;
        BasicBlock *b2 = l2->getLoopPreheader() ;

        // If exit block and preHeader are not same
        if ( b1 != b2 )
        {
                if ( b1->size() != 1 )
                        return false ;

                if ( b1->getTerminator()->getSuccessor(0) != b2
                   )
                        return false ;

                if ( b1 == nullptr || b2 == nullptr )
                {
                        errs() << "NULL pointer encountered\n" ;
                        return false ;
                }

                // These steps are done if the two loops have
                   some Blocks between them, but they contain
                   ONLY Branch instruction.
                // Hence, this still counts as adjacent.
                // The intermediate block is removed from the
                   CFG and also from the LoopInfo
                PHINode *phi = l2->getCanonicalInductionVariable
                   () ;

                phi->setIncomingBlock(1,b1) ;
                b1->getTerminator()->setSuccessor(0,l2->
                   getHeader()) ;

                b2->dropAllReferences() ;
                LI->removeBlock(b2) ;
                l1->getParentLoop()->removeBlockFromLoop(b2) ;
                deleteBlock(b2) ;
                return true;
}

// Helper function to check and fuse two loops
bool LoopFuse::FuseCheck ( Loop *l1, Loop *l2 )
{
        // Check if the two loops are adjacent
        if ( !adjacent(l1,l2) )
        {
                errs() << ”The two loops are not adjacent.
                   CANNOT fuse \n” ;
                return false ;
        }

        // Check if the predicate is same
        if ( predicateNum(l1) != predicateNum(l2) )
        {
                errs() << ”The loop check condition is not same.
                    CANNOT fuse \n” ;
                return false ;
        }

        // Check if the limit integer is same
        if ( limitValue(l1) != limitValue(l2) )
        {
                errs() << ”The loop check limiting value is not
                   same. CANNOT fuse \n” ;
                return false ;
        }

        return true ;
}
```

Hence the implementation is basically done with the help of pointers to various basic blocks of the program.

To test dependence between the two loops, we build 2 vectors, one for each loop. These vectors contain pointers to all the basic blocks of the particular loop. In the function to analyze dependency, we consider each block that is not the header or the latch of the loop, i.e., the blocks which are bodies of the loop. In all such basic blocks we check whether each instruction in it is a load or store instruction since dependences can exist only between loads and stores. For all pairs of such instructions, taking one from each of the loops we call the depends() function to get the dependency between them.

To check for profitability, a graph is constructed. The graph has loops as nodes and the edges between these loops can be of two types - good edge or bad edge. A good edge means the loops should be fused and a bad edge means fusion should not be done. The graph is imlemented as an array of a structure that stores for each loop the weight between them (iteration count) and the type of edge between them. The initGraph() and updateGraph() functions are implemented as shown below

```
void LoopFuse::initGraph ()
{
        unsigned int size = loopVector.size() ;

        graphMatrix = new graphEdge * [size] ;
        for ( unsigned int i = 0 ; i < size ; i ++ )
        {
                graphMatrix[i] = new graphEdge [size] ;
                for ( unsigned int j = 0 ; j < size ; j ++ )
                        graphMatrix[i][j].set = false ;
        }

        return ;
}

void LoopFuse::updateGraph ( Loop *l1 , Loop *l2 , edge_type type
   )
{

        std::vector<Loop *>::iterator it = find ( loopVector.
           begin() , loopVector.end() , l1 ) ;
        int i = it - loopVector.begin() ;

        it = find ( loopVector.begin() , loopVector.end() , l2 ) ;
        int j = it - loopVector.begin() ;

        graphMatrix[i][j].type = type ;
        graphMatrix[i][j].count = limitValue( l1 ) ;
        graphMatrix[i][j].set = true ;

        graphMatrix[j][i].type = type ;
        graphMatrix[j][i].count = limitValue( l1 ) ;
        graphMatrix[j][i].set = true ;

        return ;
}
```

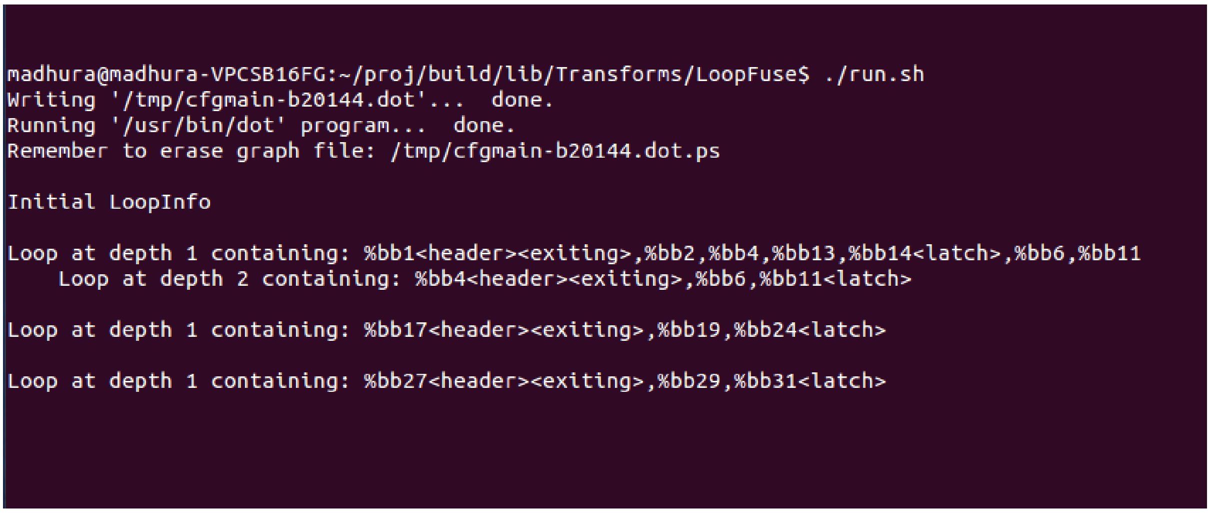
Apart from these, we make use of two more functions, buildGraph() to build the graph at the beginning of the pass and dumpGraph() to display the contents of the graph matrix. Each pair of loop which has a good edge can be fused. Once the fusion is done, the graph is updated for the single fused loop that is obtained. Two nodes are combined to form a single node that represents the loop obtained after fusion.

An important data structure that is used in the implementation is LoopInfo. The LoopInfo class that is used to identify natural loops and determine the loop depth of various nodes of the CFG. The loops identified may actually be several natural loops that share the same header node and not just a single natural loop. I keeps track of the following information about the loops in the program -

- Number of loops in each function
- Loop Nestedness
- Basic Blocks that are a part of each loop

- Loop Header, Loop Latch, Exit blocks

The LoopInfo for a C program which has four loops is as shown in the figure. The first loop has another loop within it, i.e., it is a doubly nested loop. The subloop is shown as having depth 2. Also note that the basic blocks of the subloop are shown to be present in both the subloop as well as its parent loop. It designates the basic blocks as ⟨*header*⟩⟨*exiting*⟩ and ⟨*latch*⟩. The basic blocks that do not have such tags form the body of the loop.



```
madhura@madhura-VPCSB16FG:~/proj/build/lib/Transforms/LoopFuse$ ./run.sh
Writing '/tmp/cfgmain-b20144.dot'...  done.
Running '/usr/bin/dot' program...  done.
Remember to erase graph file: /tmp/cfgmain-b20144.dot.ps

Initial LoopInfo

Loop at depth 1 containing: %bb1<header><exiting>,%bb2,%bb4,%bb13,%bb14<latch>,%bb6,%bb11
    Loop at depth 2 containing: %bb4<header><exiting>,%bb6,%bb11<latch>

Loop at depth 1 containing: %bb17<header><exiting>,%bb19,%bb24<latch>

Loop at depth 1 containing: %bb27<header><exiting>,%bb29,%bb31<latch>
```

Figure 9: Initial LoopInfo

9 System Testing and Results

The loop fusion pass is run on different kind of inputs. The test cases are written in C language and contains programs with all kinds of loops like singly nested and doubly neated loops, for, while and do - while loops etc. For example, consider the code for singly nested loops

```
for( i = 0;i < 10; i++ )
                arr[i] = i ;
for( i = 0;i < 10; i++  )
                printf ( "%d\t", arr[i] ) ;
```

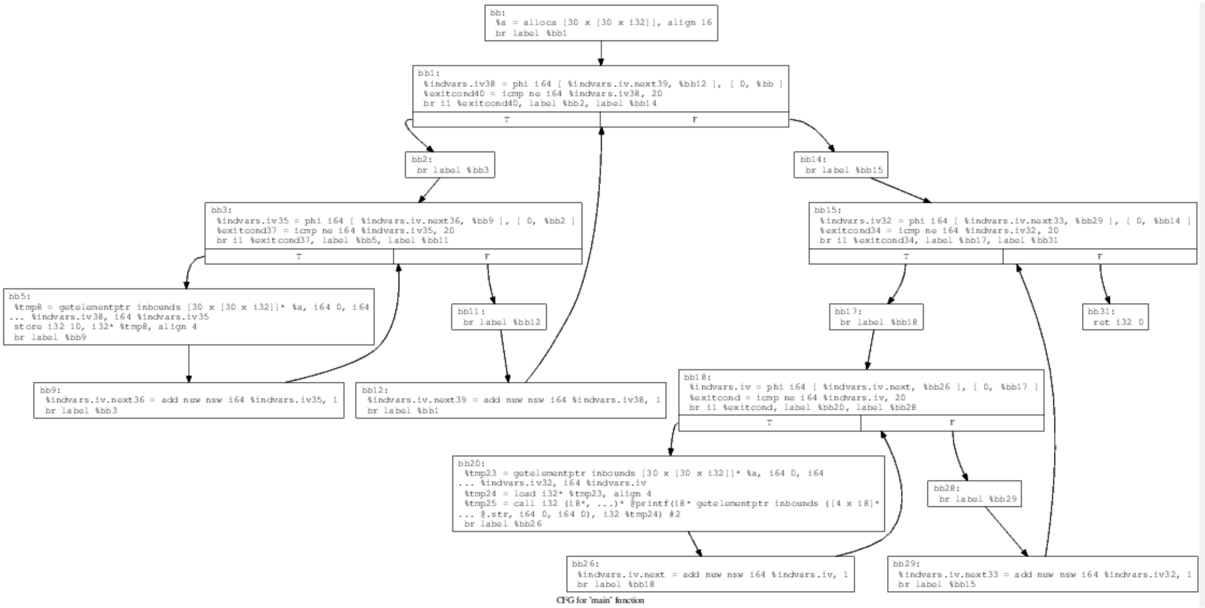
If we tranform the loop though loop fusion, we replace the two loops with a single one. After loop fusion of the above code, it would become

```
for( i = 0;i < 10; i++ )
{
                a[i] = i;
                b[i] = i;
}
```

It is run on doubly nested loops. For example, consider the code

```
for ( i = 0 ; i < n ; i ++ )
        for( j = 0; j < n; j++ )
                a[i][j] = 10 ;
for ( i = 0 ; i < n ; i ++ )
        for( j = 0; j < n; j++ )
                printf ( "%d\t", a[i][j] ) ;
```

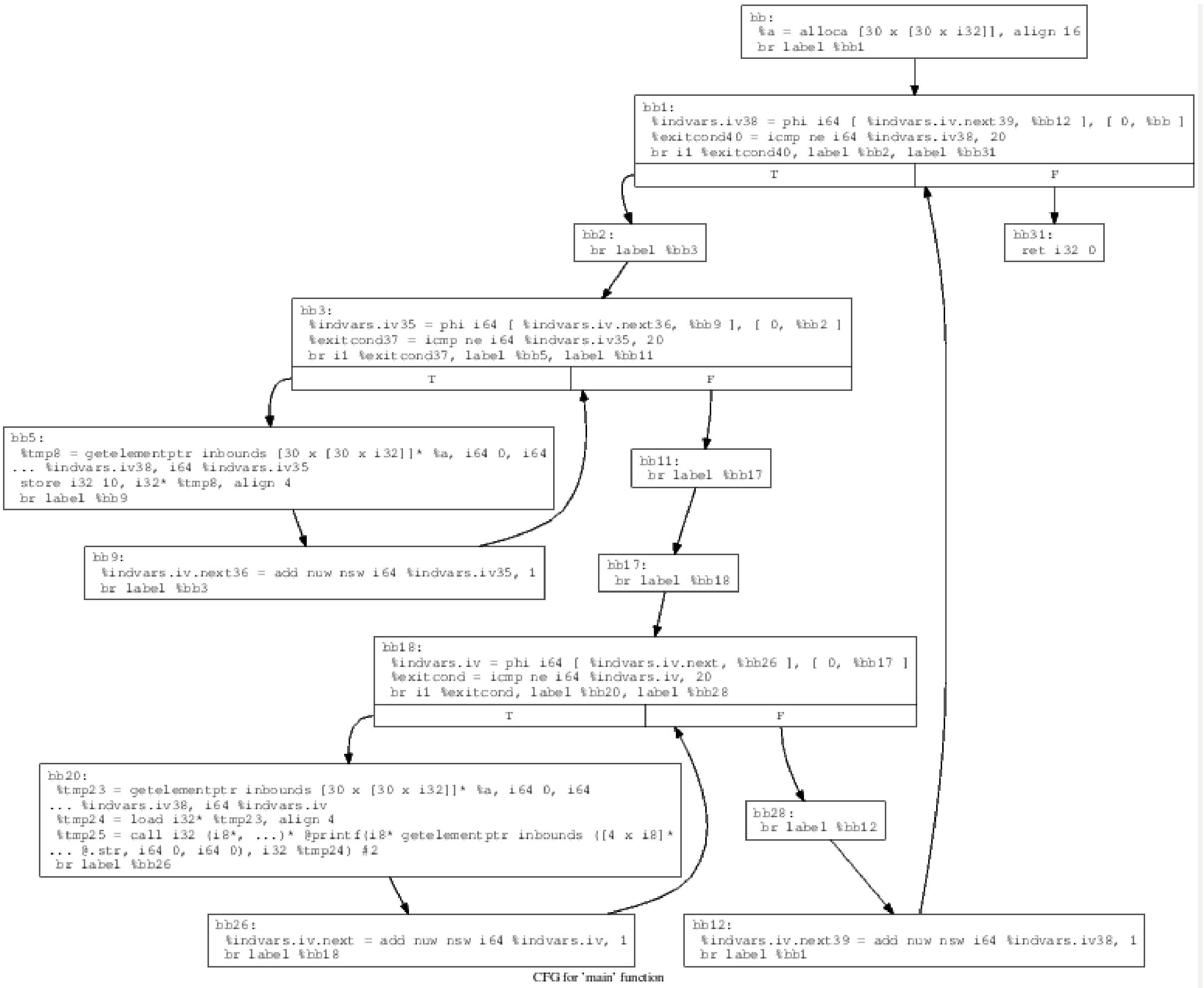
The CFG for the above is



```
bb:
 %a = alloca [30 x [30 x i32]], align 16
 br label %bb1

bb1:
 %indvars.iv38 = phi i64 [ %indvars.iv.next39, %bb12 ], [ 0, %bb ]
 %exitcond40 = icmp ne i64 %indvars.iv38, 20
 br i1 %exitcond40, label %bb2, label %bb14

bb2:
 br label %bb3

bb14:
 br label %bb15

bb3:
 %indvars.iv35 = phi i64 [ %indvars.iv.next36, %bb9 ], [ 0, %bb2 ]
 %exitcond37 = icmp ne i64 %indvars.iv35, 20
 br i1 %exitcond37, label %bb5, label %bb11

bb15:
 %indvars.iv32 = phi i64 [ %indvars.iv.next33, %bb29 ], [ 0, %bb14 ]
 %exitcond34 = icmp ne i64 %indvars.iv32, 20
 br i1 %exitcond34, label %bb17, label %bb31

bb5:
 %tmp8 = getelementptr inbounds [30 x [30 x i32]]* %a, i64 0, i64
... %indvars.iv38, i64 %indvars.iv35
 store i32 10, i32* %tmp8, align 4
 br label %bb9

bb11:
 br label %bb12

bb17:
 br label %bb18

bb31:
 ret i32 0

bb9:
 %indvars.iv.next36 = add nuw nsw i64 %indvars.iv35, 1
 br label %bb3

bb12:
 %indvars.iv.next39 = add nuw nsw i64 %indvars.iv38, 1
 br label %bb1

bb18:
 %indvars.iv = phi i64 [ %indvars.iv.next, %bb26 ], [ 0, %bb17 ]
 %exitcond = icmp ne i64 %indvars.iv, 20
 br i1 %exitcond, label %bb20, label %bb28

bb20:
 %tmp23 = getelementptr inbounds [30 x [30 x i32]]* %a, i64 0, i64
... %indvars.iv32, i64 %indvars.iv
 %tmp24 = load i32* %tmp23, align 4
 %tmp25 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]*
... @.str, i64 0, i64 0), i32 %tmp24) #2
 br label %bb26

bb28:
 br label %bb29

bb26:
 %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
 br label %bb18

bb29:
 %indvars.iv.next33 = add nuw nsw i64 %indvars.iv32, 1
 br label %bb15
```

Figure 10: CFG for doubly nested loops

If we tranform the loop though loop fusion, we replace the two loops with a single one. After loop fusion the CFG would be



```
bb:
 %a = alloca [30 x [30 x i32]], align 16
 br label %bb1

bb1:
 %indvars.iv38 = phi i64 [ %indvars.iv.next39, %bb12 ], [ 0, %bb ]
 %exitcond40 = icmp ne i64 %indvars.iv38, 20
 br i1 %exitcond40, label %bb2, label %bb31

bb2:
 br label %bb3

bb31:
 ret i32 0

bb3:
 %indvars.iv35 = phi i64 [ %indvars.iv.next36, %bb9 ], [ 0, %bb2 ]
 %exitcond37 = icmp ne i64 %indvars.iv35, 20
 br i1 %exitcond37, label %bb5, label %bb11

bb5:
 %tmp8 = getelementptr inbounds [30 x [30 x i32]]* %a, i64 0, i64
... %indvars.iv38, i64 %indvars.iv35
 store i32 10, i32* %tmp8, align 4
 br label %bb9

bb11:
 br label %bb17

bb9:
 %indvars.iv.next36 = add nuw nsw i64 %indvars.iv35, 1
 br label %bb3

bb17:
 br label %bb18

bb18:
 %indvars.iv = phi i64 [ %indvars.iv.next, %bb26 ], [ 0, %bb17 ]
 %exitcond = icmp ne i64 %indvars.iv, 20
 br i1 %exitcond, label %bb20, label %bb28

bb20:
 %tmp23 = getelementptr inbounds [30 x [30 x i32]]* %a, i64 0, i64
... %indvars.iv38, i64 %indvars.iv
 %tmp24 = load i32* %tmp23, align 4
 %tmp25 = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4 x i8]*
... @.str, i64 0, i64 0), i32 %tmp24) #2
 br label %bb26

bb28:
 br label %bb12

bb26:
 %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
 br label %bb18

bb12:
 %indvars.iv.next39 = add nuw nsw i64 %indvars.iv38, 1
 br label %bb1
```

Figure 11: CFG for doubly nested fused loops

Profitability tests are done where a check is done if the loop fusion process increases or decreases the perofrmance. If it increases, the fusion of loops is done. If it doesn't, it is kept as it is.

10 Applications

A compiler is a computer program (or set of programs) that transforms source code written in a programmimg language (the source language) into another computer language (the target language, often having a binary form known as object code)[1]. Compilers are used all over the world by programmers to compile the code they write. Any optimization in a compiler is thus beneficial for a wide variety of users.

LLVM infrastructure project was started in 2000 which is relatively new when compared to GCC which was started in 1987. LLVM provides the middle layers of a complete compiler system, taking Intermediate Representation (IR) code from a compiler and emitting an optimized IR. It is widely used by many firms including Apple for the development of Mac OSX and iOS, and Sony in the Software Development Kit (SDK) of its PS4 console. It is often extensively used in Android, iOS to compile the kernel source code.

LLVM pasases can work on machines with any kind of architecture. Originally written for C and C++, LLVM has spawned a wide variety of front ends for languages like Python, Runy, Go, Fortran and many more. Thus writing a single LLVM pass is sufficient to work with all high level languages.

Loop fusion in any compiler improves the runtime speed. The 90% - 10% rule states that 90% of the code takes 10% of the time. The remaining 10% takes the other 90% of the time. A majority of the time consuming 10% of the code involves loops. Hence any optimization related to loops leads to much faster code than when compared to optimizations with respect to other parts of the program. The added overhead of numerous optimizations during compilation is worth it since most of the programs are executed many times after compiling once. For example, consider the code for an andriod mobile application. The application is compiled once by the developer and the executable is distributed to all the users. Millions of users may download that application and each of them might run it hundreds to thousands of times. Hence only the developer is affected by the slow compilation due to the optimization overheads whereas many more users are benefitted by the faster execution time. Also, while competing in markets, user satisfaction is extremely important and faster executables help in achieving it.

11 Conclusion and Future Work

The system implemeted as part of this project fused singly nested loops successfully. It scans the program from the beginning and on finding two loops that can be fused, fuses them after performing legality and profitability checks. This new fused loop is added to the list of loops of the program. It becomes a candidate for fusion and can be fused again with a third loop.

Loop fusion failed when loops are not adjacent or their limiting values or conditions are not the same. It was prevented when it wasn't legal (there was anti dependence) and when it wasn't profitable.

Furure work for the project includes fusion of loops that are more deeply nested than doubly nested loops like triply nested loops that have three induciton variables. Also, we can try to fuse loops in which the value of the induction variable is changed within the body of the loops. In such cases, the trip count of the loops have to be compared before fusion.

Many different heuristics can be taken to check the profitability of fusion apart from the ones implemented in this project. The decision of fusion can be taken based on the pressure on the registers. This depends on the number of registers in the system which in turn depends on the architecture of the system. Another consideration could be the number of iterations of the loops. It may be possible that fusing a loop with 50 iterations and another with 100 iterations; only for 50 iterations in each loop is more profitable than fusing 2 loops with 100 iterations each. It involves fission of the loop with more iterations and the fusion with the other loop. The existing graph structure can be made use of while implementing this idea in the future.

12 Publication Details

This is a research project since no solution exists. LLVM is an open source infrastructure, i.e., its source code is availabe to the users via a free license. There does not exit any pass for loop fusion in this infrastructure. The solution implemented in this project can be submitted to the LLVM infrastructure as a patch.

References

[1] Compilers: Principles, Techniques, and Tools by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman

[2] Efficient Polynomial-Time Nested Loop Fusion with Full Parallelism (http://www.utdallas.edu/ edsha/papers/tim/ijca00.pdf)

[3] The LLVM Compiler Infrastructure: http://www.llvm.org

[4] LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation (http://llvm.cs.uiuc.edu/)

[5] Optimizing Compilers for Modern Architectures: A Dependence-Based Approach by Randy Allen and Ken Kennedy

[6] Removing Impediments to Loop Fusion through Code Transformations (blainey@ca.ibm.com)

[7] Wikipedia: http://en.wikipedia.org/wiki/Data_dependency