INDEX

Abstract

The purpose of the project is to develop a compiler optimization for C language using **LLVM**(compiler infrastructure) as a front end. The optimization pass described in this report deals with optimizations based on integer program variable ranges obtained through **Frama C.**

The idea behind this optimization is that once the ranges for the various numerical variables are determined, one can use that information to do a host of things like constant folding, checking for arithmetic overflows(and hence avoiding/handling it appropriately) etc.

For example, if the code has a conditional statement like *if (a < b) then <stmts 1> else <stmts 2>*. Now, here if we know the ranges of the variables 'a' and 'b' at this point, and if the ranges are completely disjoint (i.e. non-overlapping), one can successfully deduce that only one of *then* or *else* parts will always be executed and the other one will never be reachable. The range information can also be extended to expressions comprising of variables and constants(detailed information in the main report). Thus, we see that ,in general, logical operations can be constant folded with the help of range information.

As a secondary use of the range information of numerical variables, the code can be made to efficiently handle/avoid arithmetic overflows efficiently. The main purpose to include overflow checks for each arithmetic operation in a program is to avoid unexpected run time errors (or unexpected results). One way is to check for overflow for each arithmetic operation by calling a subroutine for every operation at runtime. Now, with the range information present, calling the overflow check subroutine every time can be avoided. It needs to be called only at times when we do not have the range information for some variable/expression in an arithmetic operation. This improves the runtime of the compiled code and ensures run time safety from arithmetic overflows at the same time.

I. Overview of LLVM

The LLVM compiler infrastructure project (formerly Low Level Virtual Machine) is a compiler infrastructure designed as a set of reusable libraries with well-defined interfaces. It is written in C++ and is designed for compile-time, link-time, run-time, and "idle-time" optimization of programs written in arbitrary programming languages. Originally implemented for C and C++, the language-agnostic design (and the success) of LLVM has since spawned a wide variety of front ends: languages with compilers that use LLVM include Common Lisp, ActionScript, Ada, D, Fortran, Ocaml, OpenGL Shading Language, Go, Haskell, Java bytecode, Julia, Objective-C, Swift, Python, Ruby, Rust, Scala,C#,and Lua.

The LLVM project started in 2000 at the University of Illinois at Urbana–Champaign, under the direction of Vikram Adve and Chris Lattner. LLVM was originally developed as a research infrastructure to investigate dynamic compilation techniques for static and dynamic programming languages. LLVM was released under the University of Illinois/NCSA Open Source License, a non-copyleft license. In 2005, Apple Inc. hired Lattner and formed a team to work on the LLVM system for various uses within Apple's development systems. LLVM is an integral part of Apple's latest development tools for Mac OS X and iOS.

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 tool chain, allowing it to be used with a wide array of extant compilers written for that project.

It can also generate relocatable machine code at compile-time or link-time or even binary machine code at run-time.

LLVM supports a language-independent instruction set and type system.[10] 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.

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

II. Overview of Frama C

Frama-C stands for Framework for Modular Analysis of C programs. Frama-C is a set of interoperable program analyzers for C programs. Frama-C has been developed by Commissariat à l'Énergie Atomique et aux Énergies Alternatives and Inria. Frama-C enables the analysis of C programs without executing them.

Frama-C has a modular plugin architecture.It relies on CIL (C Intermediate Language) to generate an abstract syntax tree. The abstract syntax tree supports annotations written in ANSI/ISO C Specification Language (ACSL).Several modules can manipulate the abstract syntax tree to add ANSI/ISO C Specification Language (ACSL) annotations. Among frequently used plugins are:

value analysis(what is used in this project): which computes a value or a set of possible values for each variable in a program. This plugin uses abstract interpretation techniques and many other plugins make use of its results.
jessie: to verify properties in a deductive manner. Jessie relies on the Why[2] back-end to enable proof obligations to be sent to automatic theorem provers like Z3, Simplify, Alt-Ergo or interactive theorem provers like Coq or Why. Using Jessie, an implementation of bubble-sort or a toy e-voting system can be proved to satisfy their respective specifications. Recently, the plugin wp has been developed, for similar purposes.
impact analysis: to highlight in the C source code the impacts of a modification.
slicing: this plugin enables to slice a program (program slicing). It enables to generate a smaller new C program which preserves some given properties.[3]
spare code: this plugin removes useless code from a C program

Frama-C can be used for the following purposes:

to understand C code which you have not written. In particular Frama-C enables to: observe a set of values, slice the program into shorter programs, navigate in the program.

to prove formal properties on the code. Using specifications written in ANSI/ISO C Specification Language enables to ensure properties of the code for any possible behavior. Frama-C handles floating point numbers.

to enforce coding standards or code conventions on C source code, by means of custom plugin(s).

to instrument C code against some security flaws.

III a) **Generating annotations in the code**

After the ranges for the various numerical variables are obtained through Frama C, they are injected into the source code (C program) as annotations. The annotations are inserted as explained below.

Given a code snippet as :

```
a_max = r8_max(a_max, *(a + (i + j * 1001)));

i++;
```

It is turned into the following code snippet (assuming ranges for all (integer)variables are found)

```
char *acsl_assertion_28_before_call = "@assert i >= 0 && i <= 999 assert j >= 0 && j <= 999";
a_max = r8_max(a_max, *(a + (i + j * 1001)));

char *acsl_assertion_28_after_call = "@assert i >= 0 && i <= 999 assert j >= 0 && j <= 999";

char *acsl_assertion_29_before_instr = "@assert i >= 0 && i <= 999";

i ++;
```

The main idea is to insert a char *(string) variable (with automatic names) that contains the range information for the variables found in the expression that follows. The is done with the help of scripts. Now these annotations are carried over to the LLVM IR through a pass that is run after the compilation phase (in debug mode). Each annotation is put as meta data(named as "!acsl_range") for the appropriate instruction in the IR.

In LLVM IR, one can attach meta data nodes to a statement(instruction). This meta data contains information about the the instruction. Particularly, it is used for debugging purpose, where the meta data for each LLVM instruction contains the line no in the corresponding source language program. The same meta data can be used to store any other kind of information. So in this case, the meta data nodes are used to store the range for each LLVM instruction. To be more precise, since LLVM IR is SSA, most instructions are of the form *%3 = add nsw %1, %2* . Here %1,%2, and %3 refer to the virtual registers that hold some value from previous/current calculations. When attaching range meta data to an instruction, it is meant that the meta data is attached the the virtual register.

The meta data that is attached to the LLVM IR instructions are integer values for the range,with low and high values that are obtained by parsing the assertions(which is in the form of string) in the source code . On observation, one can find that some range information is repeated many times in the source code. This is due to the fact that once compiled, the source is broken into various basic blocks in the LLVM IR. Now to keep track of the range information same across all basic blocks, this strategy is used.

There is a scope for improving this present methodology of carrying forward the range information across the code (in IR). However, the main goal as of now was to get the result quickly, so a theory can be established and checked for the improvement that is achieved.

III b) **Propagation of Range meta data**

After generating meta data for the variables via the annotations in the source code, it is important that the range information is propagated to all the expressions that are present in the program. However complex an expression may be in the source code, it is compiled into a series of binary(or single) operations in the LLVM IR. Hence, the purpose of this pass is to propagate the range information through the expressions. The pass is called AnnotationPropagation.

For example, say there is an expression in the source code as given below :

```
a = b*c+d-e;
```

In the LLVM IR it is converted as something like this:

```
%2 = load i32* %b, align 4     !acsl_range !150
%3 = load i32* %c, align 4     !acsl_range !151
%4 = mul nsw i32 %2, %3
%5 = load i32* %d, align 4     !acsl_range !152
%6 = add nsw i32 %4, %5
%7 = load i32* %e, align 4    !acsl_range  !153
%8 = sub nsw i32 %6, %7
store i32 %8, i32* %a, align 4
```

In the above LLVM code , the range information is only present for %2, %3 , %5 and %7 ,i.e. for every variable it is present. However, we need it for b*c , (b*c)+d and ((b*c)+d)-e as well (%4,%5 and %8). Thus, is the need for propagation of the range information. This way the range information is present for any expression that may follow.

In order to aid this propagation,the LLVM class **ConstantRange** along with a custom class called **ConstantRangeUtil** is used. The Annotation pass propagates through the instructions one at a time, and if for an instruction (basically add , sub, mul, load , etc or anything that is a static assignment), which needs to have a range meta data (instructions like store,call etc dont need to have range meta data), the range meta data is not present, the data is calculated based on the operand(s) of that instruction. For eg:

```
%2 = load i32* %b, align 4     !acsl_range !150
%3 = load i32* %c, align 4     !acsl_range !151
%4 = mul nsw i32 %2, %3
```

In the above code snippet, range data needs to be calculated for %4 (i.e. the third instruction). This is calculated as follows:

Lower limit of range of % 4 = lower limit of range of %2 * lower limit of range of %3
Higher limit of range of %4 = higher limit of range of %2 * higher limit of range of %3

In general, the range for an instruction based on the operands is calculated as follows:

Let the instruction be NI that has two operands, LI(left Instruction) and RI(right instruction).
Let Low(I) be the lower limit of the range of instruction I.

Let High(I) be the higher limit of the range of instruction I.
If NI is add instruction:
Low(NI) = Low(LI) + Low(RI)
High(NI) = High(LI) + High(RI)

If NI is sub instruction:
Low(NI) = Low(LI) - High(RI)
High(NI) = High(LI) - Low(RI)

If NI is mul instruction:
Low(NI) = Low(LI) * Low(RI)
High(NI) = High(LI) * High(RI)

If NI is div instruction:
Low(NI) = Low(LI) / High(RI)
High(NI) = High(LI) / Low(RI)
If any range information for any of the two operands is missing, the range for the current instruction cannot be deduced and the instruction is left without any range meta data. Now, if this instruction is used in any subsequent expressions, those expressions(instructions) also wont have any range information attached as meta data to them.

This range information is useful in constant folding, generally in logical instructions.
For example, for the code snippet below, constant folding can be done if ranges of 'a' and 'b' are completely disjoint.

```
if (a < b){
    a = 5;
  }
else{
   b = 10;
  }
```

```
 %2 = load i32* %a, align 4
  %3 = load i32* %b, align 4
  %4 = icmp slt i32 %2, %3
  br i1 %4, label %5, label %6

; <label>:5                                     ; preds = %0  ;then
  store i32 5, i32* %a, align 4
  br label %7

; <label>:6                                     ; preds = %0  ;else
  store i32 10, i32* %b, align 4
  br label %7
```

In the above LLVM IR, the instruction *br i1 %4, label %5, label %6*
branches to basic block with label 5 ('then' branch) if a is less than b else it branches to basic block with label 6('else' branch). Now, here if we have the range information for 'a' and 'b', and if they are disjoint (i.e. non-overlapping), one of the basic blocks label 5 and label 6 will never be reached , and instead of the conditional branch instruction,we could have either branch to label 5 or branch to label 6.

IV a) **Integer Overflow Check**

In order to avoid runtime errors due to arithmetic overflow, is to compile the source code with integer overflow check option enabled. This option inserts run time checks for all arithmetic operations in LLVM IR. In clang compiler this option is enabled thorough the command line option as follows:

clang -S -emit-llvm -fsanitize=signed-integer-overflow -fsanitize=unsigned-integer-overflow -o test.s test.c

The *-emit-llvm* option compiles the source code to LLVM IR (front end), and the *-fsanitize=signed-integer-overflow* option inserts checks for singed integer overflows. Similarly, *-fsanitize=unsigned-integer-overflow* inserts checks for unsigned integer operations. With the code compiled to LLVM IR, instead of the target machine code, one can see how the overflow checks are inserted.

The following example will clearly delineate how the overflow checks are inserted:

Suppose, there is an expression in the C source code:

c = a+b;

This is converted to LLVM IR given below,if compiled with the overflow check option:

```
  %2 = load i32* %a, align 4
  %3 = load i32* %b, align 4
  %4 = call { i32, i1 } @llvm.sadd.with.overflow.i32(i32 %2, i32 %3)
  %5 = extractvalue { i32, i1 } %4, 0
  %6 = extractvalue { i32, i1 } %4, 1
  %7 = xor i1 %6, true
  br i1 %7, label %11, label %8, !prof !1

; <label>:8                                     ; preds = %0
  %9 = zext i32 %2 to i64
  %10 = zext i32 %3 to i64
  call void @__ubsan_handle_add_overflow(i8* bitcast ({ { [7 x i8]*, i32, i32 }, { i16, i16, [6 x i8] }* }* @1 to i8*),
i64 %9, i64 %10) #3
  br label %11

; <label>:11                                    ; preds = %8, %0
  store i32 %5, i32* %c, align 4
  ret i32 0
```

We can see how a function to check for overflow(*llvm.sadd.with.overflow.i32)* , is called right after loading 'a' and 'b' into the registers, and passing them as arguments to the overflow checker function for add instruction.

A sequence of checks is then applied on the values returned by the overflow check function, and if the overflow is supposed to occur, then the basic block *<label>:8* is entered and the overflow is handled with the help of function *void @__ubsan_handle_add_overflow* , otherwise basic block *<label>:11* is entered , and the program runs normally.
The point worth noting is that the function call for overflow check does slow down the program.

Also, if we somehow know that there is no overflow occurring for sure , we can avoid that function call altogether. This is basis of one of the optimizations we can do with the range information for the variables.

The various functions to check for overflow for the basic signed arithmetic operations is as follows:

llvm.sadd.with.overflow.i32 - for add instruction
llvm.smul.with.overflow.i32 - for mul instruction
llvm.ssub.with.overflow.i32 - for sub instruction
llvm.sdiv.with.overflow.i32 - for div instruction

A noteworthy point is that these functions are for 32 bit signed operands. Similarly there are other functions to handle overflow for unsigned as well as non-32 bit integers.

In the next section , removal of the redundant integer overflow checks has been discussed in some detail. It discusses how the removal of the checks is done and why it improves the run time,while maintaining the safe behavior of the code.

IV b) **Removal of redundant Integer Overflow Checker**

As discussed in the previous section, if we can avoid the overflow checks for the instructions that we know are not going to overflow, we can improve the running time and have the same safety as was guaranteed by the overflow checker/handlers. The pass which removes the integer overflow checker functions if possible is called the RemoveIOC pass, which runs after the code has been compiled(in debug mode) using the overflow check option and the AnnotationPropagation pass.

The process of removing redundant checks, once we know the range information of the various instructions in the LLVM IR, is described as follows.

Let us consider the same example from the previous section. The expression *c = a+b;*
was converted to the following LLVM IR ,if compiled with the overflow check option:

```
  %2 = load i32* %a, align 4
  %3 = load i32* %b, align 4
  %4 = call { i32, i1 } @llvm.sadd.with.overflow.i32(i32 %2, i32 %3)
  %5 = extractvalue { i32, i1 } %4, 0
  %6 = extractvalue { i32, i1 } %4, 1
  %7 = xor i1 %6, true
  br i1 %7, label %11, label %8, !prof !1

; <label>:8                                     ; preds = %0
  %9 = zext i32 %2 to i64
  %10 = zext i32 %3 to i64
  call void @__ubsan_handle_add_overflow(i8* bitcast ({ { [7 x i8]*, i32, i32 }, { i16, i16, [6 x i8] }* }* @1 to i8*),
i64 %9, i64 %10) #3
  br label %11

; <label>:11                                    ; preds = %8, %0
  store i32 %5, i32* %c, align 4
  ret i32 0
```

Now, if we have range information for 'a' and 'b' (meta data for instructions %2, and %3 *),* and we know that the range of the add instruction can be obtained (i.e. it will not overflow), the checker function (*llvm.sadd.with.overflow.i32*) can be removed and an add instruction can be directly added after the instruction %3 as follows.

```
%2 = load i32* %a, align 4
  !acsl_range !190
%3 = load i32* %b, align 4
  !acsl_range !191
%4 = add nsw %2,%3, align 4 !acsl_range !192
br label %11
```

As can be noted that the basic block (*<label>:8*) is completely avoided, as there is no need for it. In case the range for the resultant add instruction were overflowing , or the range information on any of the operands were missing, then no part of the code would have been changed. Similarly, the pass checks for most binary operations(instructions) like mul, sub,div etc.

The pass modifies the code(in the LLVM IR) in a way such that the dependencies,integrity and

behavior of the code is not compromised. The code still behaves as it would do before the pass was run. Note that the AnnotationPropagation pass simply adds meta data nodes to the instructions and does not modify the direct code in any way.

However, since this pass deletes instructions ,if possible, it is necessary to verify that no behavior of the code is changed after the execution of the pass. The proof that the code is not changed in its behavior is as follows.

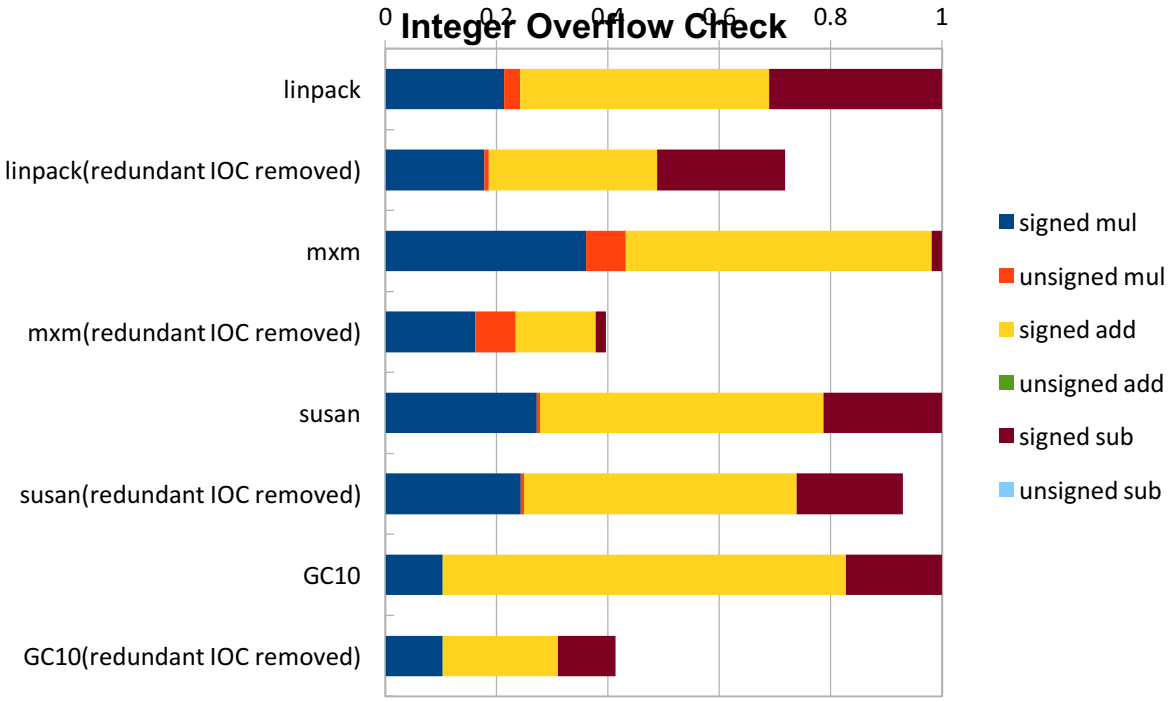
On observation, it can be ascertained that each original arithmetic operation is in a separate block, because of the overflow checker function. This is because, depending on the result given back by the checker function, the program may either continue normally, or go to the overflow handler routine. Since, by definition every basic block needs to end in a branch(with no branch via br instruction in between the basic block), each arithmetic operation is put in a separate basic block. And in the corresponding basic block all the instructions ,after the operands are loaded in the registers , are basically the ones related to the overflow checking. Hence, if only these instructions are deleted there will not be any change in the behavior of the program. But care is taken to delete the instructions from the end of the basic block (rather than from the point where the arithmetic instruction should be added if there is no overflow).

In the next section , the results of running the pass through some test benchmarks are discussed and also the metrics that were observed in relation to performance are mentioned and briefly explained.

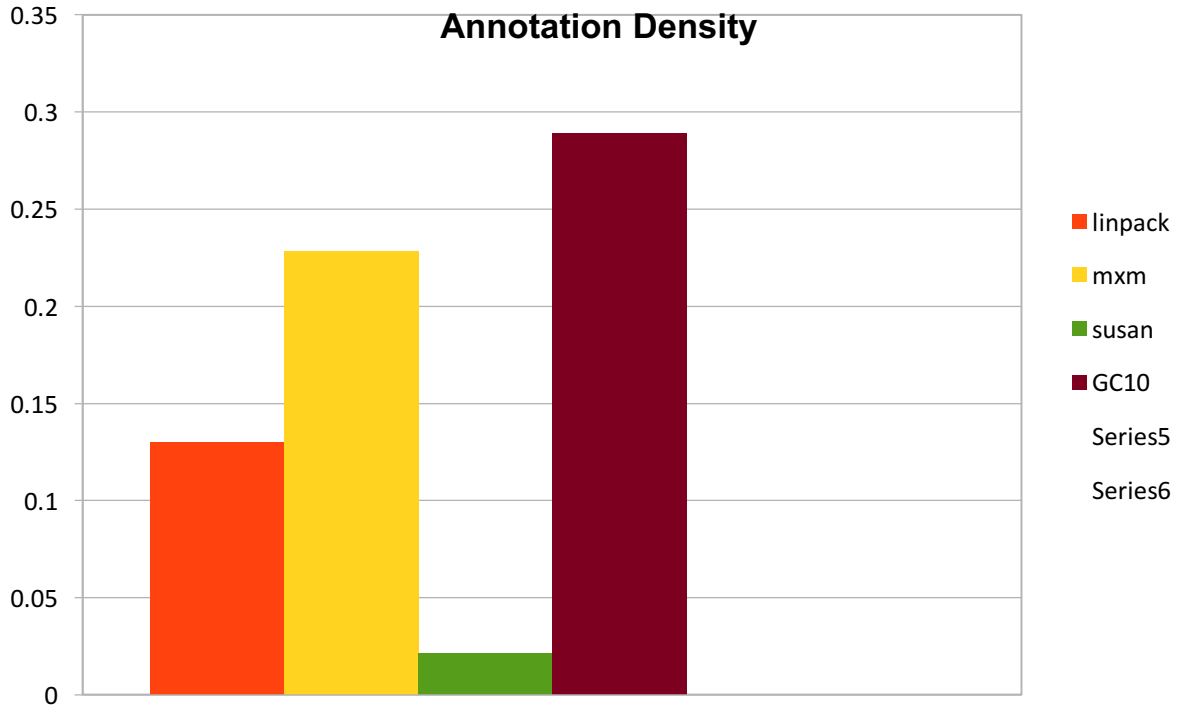
V) **RESULTS**

The optimization pass was run for the test benchmarks. The metrics to observe was the reduction in the number of overflow checker functions in each benchmark. Also, another metric to observe for each benchmark was the running time for the normal code(with integer overflow checks enabled) and the running time after the RemoveIOC pass was run through the code.

The graphs below show the results in a comprehensive way.



In the graph above, each horizontal bar represents the number of instructions in each benchmark(namely linpack,mxm,susan and GC10), before and after removing redundant integer overflow checks. The various colors used in every bar correspond to the various integer overflow checker functions that were there in the LLVM IR for the corresponding benchmark. As we can observe, in all cases the no of overflow checker functions have been reduced. In the next graph a rough correlation between the annotation density and the number of removed checker functions has been discussed.



The above graph shows the annotation density for the various benchmarks that have been tested. Annotation density refers to the number of annotations in the source code per line of source code. These annotations refer to the annotations that were injected into the C source code.

It is worth noting that the GC10 benchmark has the greatest annotation density and susan the least. Also, in the previous graph, we could see that GC10 had the most number of overflow checks removed , and susan had the least number of overflow checks removed. From the results, it seems that more the annotation density, more overflow checks can be removed. However, there is no direct correlation between the two. Although, as a rule of thumb, one can say that more annotation density could mean removal of more redundant integer overflow checks. The reason why annotation density is not directly correlated to the removal of overflow checks, is that it mostly depends on which variable is used how often. For example, say the annotation density is very good in a source code (range information for all variables are present except one variable say 'x'). Now, if all expressions contain 'x' , then no expression will have any range information present, which would mean that most overflow checks cannot be removed. However, that such a case is unlikely to occur, and so the more the annotation density , the better the optimization.

Fewer function calls do suggest better running time. However, this needs to be realized in practice. Hence, the run time test below attests to the hypothesis.

//linpack_bench
IOC REMOVED: 1188074
IOC NOT REMOVED: 2320145

//susan
IOC REMOVED: 212013

IOC NOT REMOVED: 216013

//mxm
IOC REMOVED: 268016
IOC NOT REMOVED: 292018

//NEC-Matrix
IOC REMOVED: 4000
IOC NOT REMOVED: 8000

In the above tests IOC means Integer Overflow Checks. The runtimes were obtained through scripts. We can observe that in all cases the running time was better after removing redundant integer overflow checks(All compilations were done in debug mode to facilitate propagation of annotations).

VI) Conclusion

The range information as can be imagined can be very useful for a host of optimizations. This project just deals with a couple of them. The main core of optimizations is in the loops. Loop optimizations are the heart of most compiler optimizations. LLVM has a host of optimizations for loops.

The idea for further usage of range information, can be to aid any of the existing optimizations in the LLVM framework library. For example, most loop optimizations make use of scalar evolution. Scalar Evolution for a variable (particularly within a loop) means how the variable changes within the loop. This information is used for many loop optimizations. Now, in order to computer the Scalar Evolution for a variable, LLVM uses some opaque classes like SCEV(which are associated with variables). Sometimes the SCEV for a particular variable cannot be computed. Now the idea is to somehow use the range information for that variable ,if available, to aid in the computation of its SCEV. If SCEV can be obtained for some variable through the use of range information, it will indirectly aid in loop optimizations too.