Integer Range Analysis in LLVM

Maglione Sandro
10532096

June 21, 2020

Contents

Abstract

The purpose of the range analysis is to reduce any waste of memory by allocating the minimum number of bits to each variable while maintaining the correctness of the code. This compile-time optimization allows the programmer to focus more on the problem at hand and less on the optimal type to assign to each variable (int, short, long).

In this report, I introduce a LLVM pass that performs a integer range analysis. The pass implements a static analysis of the source code to infer the minimum and maximum values assignable to any variable in the code. We are then able to compute the range of values assignable to each variable and minimize the number of bits allocated for each of them.

The tools is composed of two LLVM passes: a Constant Range Analysis and a Branch Range Analysis. The first pass analyizes the original IR instructions obtained from a source code without branches i.e. where all ranges are constants. The second pass requires the IR to be optimized using some intermediate LLVM passes (e.g. mem2reg, constprop), and then it computes the range of all the variables from a source code with branches and loops.

The passes have been tested on some example IRs obtained from source files written in the C language. In this report I am going to present some of the results achieved by the tool, some of its assumptions and limitations, and how the tool can be expanded and optimized in future versions.

1 Introduction

One of the main goals of a compiler is to optimize the code written by the programmer in order to be able to improve the performance of the software and exploit at best the hardware resources. This procedure allows the programmer to focus all her effort and time on the overall structure of the code and less on minor optimizations of each function and memory management. The compiler can extract major information from a static analysis of the code. This analysis allows the compiler to remove any unused code and to optimize the memory utilization by extracting all sort of information only by analyzing the code a static-time. This procedure can drastically improve the developer experience as well as the overall performance of the final application.

1.1 What is a Range Analysis

One of the possible optimization that the compiler can perform is a range analysis. The goal of a range analysis is to optimize the allocation of memory for all variables in the code. In order to achieve that, the compiler scans the original code to infer the minimum and maximum values that each variable can assume during runtime. If the compiler detects an overall range (defined as the distance between the minimum and maximum value) which is bounded, then the compiler can **optimize the memory allocated to a specific variable by reducing the number of bits assigned to it**.

One simple but significant example is the index variable used in a for loop. This variable is used to specify the number of iterations performed by the loop. Because of that, its value is often limited between two well defined numbers. The range analysis can detect the minimum and maximum value of the variable and optimize the number of bits allocated to it. If we think about the frequency of such for-loop constructs in an average source code, we can understand how such a simple procedure can greatly improve the performance of the code.

Code 1 Example of a for loop in which the range of the index variable can be detected and optimized by the range analysis. The range of the i variable goes from a minimum value of 0, to a maximum of 20. Therefore, we can assign only 6 bit to the variable.

```
#include <stdio.h>

```

```
int main()
{
    for (int i = 0; i < 20; i++) {
        printf ("Code to optimize");
    }
}
```

1.2 LLVM

The tool here presented has been developed as a pass inside LLVM. LLVM is an open-source compiler infrastructure project. LLVM allows the developer to implement both a front end for any programming language, as well as a back end for any instruction set architecture. LLVM works by converting the intermediate representation (IR) generated by a compiler into a new optimized IR. The new IR can then be converted and linked into machine-dependent assembly language code for any specific target platform.

LLVM provides an API used to inspect and analyze the IR in order to further optimize the intermediate representation. The code which performs this optimization is called a pass. Each pass is written in C++. LLVM contains a predefined set of passes that can be used to perform some preliminary code optimizations. Furthermore, new passes can be implemented. Different passes can be run one after the other to generate the final IR to convert in assembly code. The constant range analysis pass works directly on the original IR generated by LLVM. The branch range analysis pass, instead, is designed to be run after some preliminary optimization passes provided by LLVM, such as mem2reg and constprop.

1.3 IR and SSA form

As discussed in the previous section, IR stands for Intermediate Representation and is an intermediate code representation used by LLVM to perform optimizations and analysis of the original source code. The IR generated by LLVM is a *.ll* file that contains a human-readable assembly-like code which can be inspected and optimized by implementing a LLVM pass.

The main characteristic of the IR generated by LLVM is that it is in **SSA form**. SSA stands for Static Single Assignment and it is a particular representation of the original source code in which each variable is assigned exactly once, and every variable is defined before it is used. This property enables any sort of code analysis and optimization because it introduces a series of assumptions in the analyzed code which can be exploited by the compiler a static-time to infer more information from the original code.

Code 2 Here we can see an example of an IR generated by LLVM. Each IR is composed by a series of instructions grouped in different basic blocks. As we can see, each variable is assigned exactly once in the whole code, and every variable is defined before it is used.

```
define dso_local i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  %d = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 0, i32* %a, align 4
  store i32 10, i32* %b, align 4
  %0 = load i32, i32* %a, align 4
  %add = add nsw i32 %0, 3
  store i32 %add, i32* %c, align 4
  %1 = load i32, i32* %b, align 4
  %add1 = add nsw i32 %1, 10
  store i32 %add1, i32* %d, align 4
  ret i32 0
}
```

1.4 mem2reg

The branch range analysis is based on some preliminary LLVM passes which perform some intermediate optimization exploited by the tool's pass to improve the analysis. The most important of these preliminary passes is called mem2reg.

The main result of the mem2reg pass is to convert the original IR in a 'pruned' SSA form. The pass achieve this result by promoting memory references to be register references. Therefore, *load*, *alloca*, and *store* instructions are removed and *phi* instructions are introduced. A *phi* instruction is used to inform the compiler of the possible values assumed by a variable. In fact, it is possible that a variable is assigned or modified in different basic blocks of the code. The *phi* instruction contains a list of all the possible values assumed by a variable at the entry of a basic block. This information allows the tool to infer the range of the variable based on the list of values that it can assume.

The other LLVM passes which are used by the tool are *constprop*, *dce*, *simplifycfg*, and *gvn*.

Code 3 Example of a IR generated using the preliminary passes mentioned above. As we can see, alloca, store, and load instructions have been removed and phi instructions have been introduced at the entry of the basic blocks.

```
define dso_local i32 @fun() #0 {
entry:
  br label %for.cond

for.cond:
      ; preds = %for.body, %entry
  %a.0 = phi i32 [ 1, %entry ], [ %add, %for.body ]
  %j.0 = phi i32 [ 10, %entry ], [ %inc, %for.body ]
  %cmp = icmp slt i32 %j.0, 30
  br i1 %cmp, label %for.body, label %for.end

for.body:
      ; preds = %for.cond
  %add = add nsw i32 %a.0, 10
  %inc = add nsw i32 %j.0, 1
  br label %for.cond

for.end:
      ; preds = %for.cond
  ret i32 %a.0
}
```

2 Installation

In order to use the tool you must clone and build the LLVM environment in your local machine. The LLVM project provides many resources online on how to install the infrastructure. In this report, I am going to highlight the main step required to setup the environment, from the actual build of LLVM to the installation of the tool.

2.1 Prerequisites

The prerequisites to install LLVM successfully are listed on the official LLVM guide. The main softwares required are:

- CMake
- GCC
- Python
- zlib
- GNU make

2.2 Setting up the LLVM environment

The first required step is the installation of LLVM. You must clone the open-source LLVM repository, hosted on Github, on your local machine. In order to do that, just navigate to your choosen directory and run the *git clone* command below:

git clone https://github.com/llvm/llvm-project.git

This command will create a *llvm-project* folder inside your directory. The folder contains all the files required to build and start using LLVM.
Navigate to the newly created folder and create a new *build* folder.

cd llvm-project
mkdir build
cd build

The next step is launching the *cmake* command to setup the build process. Many options are available. The following command is the default recommended on the Clang getting started guide:

cmake -DLLVM_ENABLE_PROJECTS=clang -G "Unix Makefiles" ../llvm

Finally, launch the *make* command to build the environemnt and install LLVM on your local machine. This builds both LLVM and Clang for debug mode. Note that generally this step may require some time to be completed.

make -j4

For more details, visit the Getting Started Guide available on the offical Clang-LLVM website.

2.3 Importing the passes

Once the build process of the LLVM environment is completed, we are ready to import and build the tool. The code for the tool's passes is available on Github.

The first step is to access the *Transforms* directory inside the llvm-project folder. Open the llvm-project folder and naviagate to the *llvm-project/llvm/lib/Transforms* folder. This folder contains all the transform passes available inside your LLVM installation. We are going to import both the constant range analysis pass and the branch range analysis pass as two different passes inside LLVM.

- Create two directories: **ConstantRange** and **BranchRange**.
- Update the *CMakeLists.txt* file inside */llvm-project/llvm/lib/Transforms* by adding the newly created directories: *add_subdirectory(ConstantRange) add_subdirectory(BranchRange)*
- Open *ConstantRange* directory
- Copy and paste the **ConstantRange.cpp** file inside the directory
- Create a new *CMakeLists.txt* file inside the directory
- Add the following code to the *CMakeLists.txt* file to allow LLVM to recognize the new pass:

```
add_llvm_loadable_module( LLVMConstantRange
  ConstantRange.cpp

  PLUGIN_TOOL
  opt
  )
```

- Navigate back to the *Transforms* folder and open *BranchRange* directory
- Copy and paste the **BranchRange.cpp** file inside the directory
- Create a new *CMakeLists.txt* file inside the directory
- Add the following code to the *CMakeLists.txt* file to allow LLVM to recognize the new pass:

```
add_llvm_loadable_module( LLVMBranchRange
  BranchRange.cpp

  PLUGIN_TOOL
  opt
  )
```

- Navigate back to the *llvm-project/build* folder
- Launch again the build command: **make -j4**

2.4 Running the code

After all the previous steps are completed, the passes are installed and built inside the LLVM environment and can be used to perform the range analysis.

The two command to use for the passes are **-const-range** for the constant range analysis and **-branch-range** for the branch range analysis.

- Import your source code in C language in a folder of your choice (in this guide I assume the name of the file is *example.c*)
- Navigate to the *llvm-project/build/bin* folder

- Run the command

  ./clang -S -emit-llvm [...path to your C code]/example.c

  for the constant range analysis, which generates directly the IR in *.ll* file, and

  ./clang -c -O0 -emit-llvm [...path to your C code]/example.c -o example.bc -Xclang -disable-O0-optnone

  for the branch range analysis, which generates a *bytecode* file

- For the branch range analysis pass, we need to compute an intermediate IR code using some LLVM preliminary passes. Run the following command to build the *.ll* version of the previously generate *bytecode* file:

  ./llvm-dis example.bc -o=example.ll

- Run the following command to apply the preliminary passes to the IR file:

  ./opt -mem2reg -constprop -dce -simplifycfg -gvn -S example.ll -o example-super.ll.

  This command generates another *example-super.ll* file which contains the IR code used by the branch range analysis pass

- Finally, run the command to launch the tool:

  ./opt -load ../lib/LLVMConstantRange.so -const-range <example.ll >/dev/null -debug

  to perform the constant range analysis on the IR file, or

  ./opt -load ../lib/LLVMBranchRange.so -branch-range <example-super.ll >/dev/null -debug

  to perform the branch range analysis on the optimized IR file

3 Constant Range Analysis

In this section I am going to introduce a simple example that aims to showcase the tool's constant range analysis algorithm. As already explained, the constant range analysis operates on the original IR of LLVM to extract the constant value of each variable in the source code. The assumption in the constant range analysis pass is that no branch is present in the code. Therefore, all the variables resolve to a constant range.

3.1 Example code

The following example illustrates all the major features provided by the tool for the constant range analysis. Here below is the source code written in C language and its translation in the default IR computed by LLVM.

Code 4 Example code for the Constant Range Analysis algorithm.

```c
int main() {
  int a = 9;
  int b = a - 6;
  int c = 3 + b;
  int d = c + c;
  return 0;
}
```

Code 5 Example code for the Constant Range Analysis converted in the default IR of LLVM.

```
define dso_local i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  %d = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 9, i32* %a, align 4
  %0 = load i32, i32* %a, align 4
  %sub = sub nsw i32 %0, 6
  store i32 %sub, i32* %b, align 4
  %1 = load i32, i32* %b, align 4
  %add = add nsw i32 3, %1
  store i32 %add, i32* %c, align 4
```

```
16   %2 = load i32 , i32* %c , align 4
17   %3 = load i32 , i32* %c , align 4
18   %add1 = add nsw i32 %2, %3
19   store i32 %add1 , i32* %d , align 4
20   ret i32 0
21 }
```

The source code is composed of four variables: *a*, *b*, *c*, *d*. Each variable depends on another previous variable and/or a constant. From the example, we can clearly infer the constant value assumed by each of the variable inside the source code.

The tool computes the constant value of *a* as 9 from the *store* instruction at line 9. After that, a *sub* instruction is found. The tool resolves the value of *%0* in the operation to *%a*. Since we already computed the constant value of *%a* as 9, the tool performs the subtraction and assigns *9 - 6 = 3* to *%sub*. The following store instruction tells the tool that the value just computed is assigned to *%b*.
The same happens for the *add* instruction at line 14: the value of *%add* is computed as the sum of the constant value of *%b* and 3, and then the following store assigns the computed value of 6 to *%c*.
Finally, the last *add* instruction computes the sum between two other variables. The tool resolves the value of *%2* and *%3* to *%c*. The last store then assigns the sum *6 + 6 = 12* to *%d*.

At the end of the analysis, the tool reports the constant value of all the variables found in the code. The output of the constant range analysis on this example code is represented here below.

Code 6 Result of the Constant Range Analysis on the example code.

```
--- VALUE RANGES ---
 retval(0, 0)
 a(9, 9)
 sub(3, 3)
 b(3, 3)
 add(6, 6)
 c(6, 6)
 add1(12, 12)
 d(12, 12)
```

As we can see, the tool was able to infer correctly the constant value of all the variables in the code.

4 Branch Range Analysis

In this section I am going to present one example that illustrates all the major features of the branch range analysis algorithm. In order to perform the branch range analysis, we need to compute an optimized version of the default IR computed by LLVM. All the steps to produce the optimized IR are presented in the *Installation* section.

4.1 Example code

In the branch range analysis we introduce loops and branches. This makes the analysis more complex but also more interesting and useful in practice. Generally, every source code has multiple branches and loops. The tool is designed to extract as many information as possible at compile-time to improve the memory allocation of the variables in the source code. In the following example, I am going to explain all the major steps performed by the tool to extract all the ranges possible from the source code. Here below is reported the source code of the example and the optimized IR computed by LLVM.

Code 7 Source code example for the Branch Range Analysis algorithm.

```
int fun()
{
    int j = 10;
    int a = 1;

    for (; j < 30; ++j)
    {
        a = a + 10;
    }

    return a;
}
```

Code 8 Oprimized IR generated by LLVM from the example source code.

```
define dso_local i32 @fun() #0 {
entry:
  br label %for.cond

for.cond:
    ; preds = %for.body, %entry
```

```
7   %a.0 = phi i32 [ 1, %entry ], [ %add, %for.body ]
8   %j.0 = phi i32 [ 10, %entry ], [ %inc, %for.body ]
9   %cmp = icmp slt i32 %j.0, 30
10  br i1 %cmp, label %for.body, label %for.end
11
12 for.body:
13      ; preds = %for.cond
14  %add = add nsw i32 %a.0, 10
15  %inc = add nsw i32 %j.0, 1
16  br label %for.cond
17
18 for.end:
19      ; preds = %for.cond
20  ret i32 %a.0
21 }
```

4.2 Branches and Phi nodes

The first major difference we can notice compared to the constant range analysis example is the format of the IR code. As previously explained, the optimizations passes applied to the original IR allows the tool to extract more information from the code. As we can see, *alloca*, *load*, and *store* instructions have been removed.

We notice the introduction of two *phi* instructions at line 7 and line 8. These instructions allows the tool to understand the possible values assumed by the variable *%a.0* and *%j.0*. These two variables, in fact, assume different values when the *for.cond* basic block is reached from the *entry* basic block above it, or from the *for.body* basic block below it.

That is the principal difference with respect to the constant range analysis pass: with the introduction of branches, a single code scan is not enough anymore to extract all the information available on ranges. This can be noticed from this example, where the *phi* instructions introduce a dependency on a basic block with is located after the current one. This happens because from the *for.body* basic block the code loops back to the *for.cond* basic block with the unconditional *br* instruction at line 16.

Another new instruction introduced is the *cmp* instruction at line 9. This instruction performs a *slt* (Signed Lower Than) comparison between the value of *%j.0* and a constant value of 30. This result is then utilized by the conditional *br* instruction at line 10 to branch to *for.body* or *for.end* when the condition is verified and not verified respectively.

4.3 Loop ranges

The first source of information about the ranges assumed by the variables in the code is related to the loop condition. We can intuitively notice from the source code in C that, no matter the computations inside the loop body, the value of the variable j must be between 10 and 30 at the end of the function. In fact, the variable j is not involved in any computation, and its role is to determine the number of iterations of the loop. This construct is heavily used in all programming languages and source codes, and it can be greatly exploited by the range analysis to infer the ranges of the variables.

Furthermore, once we know that the loop tripcount is from 10 to 30, we can infer that the number of iterations performed of the loop body is 20. This additional information allows the tool to extract even more information on the value range of the variable a. Variable a is initially initialized to 1. After that, its value is updated inside the loop body of 10 at each iteration. Since we know that the number of iterations is equal to 20, we can infer the range of a from a minimum value of 1 to a maximum value of *1 + (20 x 10)*, which is equal to 201.

At the end of the computation, the tool is able to assign a well-defined range to both variable j, by exploiting the information on the loop condition. and variable a, by exploiting the information about its initial value and the tripcount of the loop previously computed.

4.4 Algorithm

The previous two sections explain the rationale behind the algorithm and what information the tool can extract from the example source code. In this section, I am going to highlight the steps performed by the algorithm to actually compute the information on the ranges.

The pass contains a vector called *workList*. This *workList* contains all the basic blocks that are still to visit by the algorithm. The algorithm iterates over the elements inside the list until the list becomes empty. Initially, the only basic block inside the list is the *entry* block, which is present in every IR produced by LLVM.

For every basic block in the list, the algorithm inspects all the instructions contained in the basic block. For each possible instruction. the pass makes some computations to extract as many information as possible based on the current knowledge of the variable ranges. The first instruction encountered is the *br* instruction of the *entry* block at line 3. The algorithm recognizes the *for.cond* basic block in the branch instruction. Since the *for.cond* has never been visited before, the algorithm adds it to the *workList* vector and marks it as visited.

All the instruction of the *entry* block have been inspected. The next basic block inside the *workList* is *for.cond*. The algorithm proceeds to inspect all the instructions inside this basic block.

- The first *phi* instruction assigns 1 or *%add* to the value of variable *%a.0*. The range of *%add* is currently completely unknown. The pass assumes a possible range which spans from -infinite to +infinite in such case. Nonetheless, the *phi* instruction informs the analysis that the value of *%add* originates from the *for.body* basic block. Therefore, the algorithm inspects the *for.body* basic block and finds that the value of *%add* is dependent on the value of *%a.0*. Specifically, *%add* is updated with and *add* instruction. From this inspection, the algorithm infers that the minimum value assumed by *%a.0* at line 7 must be the value coming from the *entry* basic block. Therefore, after the first *phi* instruction, the range of *%a.0* is from a minimum of 1 to a maximum of +infinite.

- The analysis of the second *phi* instruction at line 8 is the same explained above for variable *%a.0*. This time, the value of *%j.0* is dependent on *%inc*, which is updated with an *add* instruction in basic block *for.body*. Therefore, at the end of this inspection, the range of *%j.0* is from 10 to +infinite.

- The next instruction is the *cmp* at line 9. The algorithm inspects the condition imposed by the instruction to infer the range of variable *%j.0* when the condition is verified and when the condition is not verified. The comparison is *slt*, therefore the condition is *%j.0* less than the constant value of 30. In the taken case, the possible range of *%j.0* is from -infinite to a maximum value of 29, while in the not taken case, the range spans from a minimum value of 30 to a maximum of +infinite. This information will be exploited by the following conditional *br* instruction.

- The last instruction of the *for.cond* basic block is a conditional *br* instruction. The algorithm utilizes both the previous ranges computed on the *cmp* instruction as well as the current range assigned to variable *%j.0* to compute the range that this variable can assume in case of branch taken to basic block *for.body* and in case of branch not taken to basic block *for.end*. Specifically, the algorithm already computed the minimum value of *%j.0* to be 10 and it also knows that, in case of branch taken, the maximum value possible for *%j.0* must be 29. Therefore, inside the *for.body* basic block, the range of *%j.0* is from a minimum value of 10 to a maximum value of 29. For the branch not taken case instead, the range of *%j.0* in the basic block *for.end* is from 10 to +infinite, since no other additional information can be inferred. Finally, both *for.body* and *for.end* are inserted inside the *workList*.

The next basic block in the list is *for.body*. This basic block contains two *add* instructions and one *br* instruction. The current ranges assumed by variable *%a.0* and variable *%j.0* are respectively from 1 to +infinite and from 10 to 29. Based on this information, the algorithm is able to compute the range of *%inc* from 11 to 30. Since the values of *%inc* are changed from the default of -infinite to +infinite, we know that we have some new information to propagate to the next basic block. Therefore, even if the *for.cond* basic block is marked as already visited, the algorithm adds it to the *workList*.

The next basic block in the list is *for.end*. Since the only instruction contained in the block body is *ret*, the algorithm cannot extract any additional information from it and just moves to the next basic block.

The algorithm now visits the *for.cond* basic block for the second time. From the inspection of the *for.body* block previously computed, we have a series of new information that can be exploited to improve the analysis. We now know that the tripcount of the loop from *for.cond* to *for.body* and back to *for.cond* is based on the value of *%j.0*. We also computed that the possible range assumed by *%j.0* in the loop is from 10 to 29. Therefore, we compute the tripcount to be 20. Armed with this new information, we can completely infer the range of variable *%a.0*. In fact, since the varaible *%a.0* is updated by 10 at each iteration of the loop body, we can assign it a range that spans from a minimum value of 1 to a maximum value of *1 + (10 * 20)*, which is equal to 201. The range of *%j.0* was already known to be from 10 to 30 inside the *for.cond* basic block. Furthermore, we can now limit the range of *%j.0* inside the *for.end* basic block to a maximum value of 30 since we know that the range of *%j.0* cannot be greater than 30. Therefore, the new range of *%j.0* inside *for.end* becomes a constant of 30.

Completed this analysis, the algorithm terminates because, once the ranges becomes fixed and are not updated anymore, no basic block is added to the *workList*.

4.5 Results

The previous example aims to illustrate all the possible information that the tool can extract from a simple source code. Here below we can see the output of the algorithm on the example proposed.

Code 9 Output result of the branch range analysis algorithm on the example source code explained above.

```
--- VALUE-RANGES ---
BB: entry

BB: for.cond
     a.0(1, 201) = 201 {9bit}
     j.0(10, 30) = 21 {6bit}

BB: for.body
     j.0(10, 29) = 20 {6bit}
     add(-Inf, +Inf) = MAX
     inc(11, 30) = 20 {6bit}

BB: for.end
     j.0(30, 30) = 1 {2bit}
```

Based on this analysis, we can allocate 9 bits to the variable a and 6 bit to the variable j. This causes a reduction of 23 bit, from the standard 32 bit of an int variable, for variable a and of 26 bit for variable j.

Overall, this analysis yields a memory improvement of **71.8%** for the allocation of variable a, and of **81.3%** for the allocation of variable j. This causes an overall improvement of **76.6%** for the memory allocation of the whole function.

5 Future work

In this section I am going to highlight some of the current limitations of the tool and some proposes for future improvements.

5.1 Limitations

The tool supports all the most common instruction inside the IR form of LLVM. Nonetheless, the inspection of some instructions is still missing.

The range analysis can infer a great amount of information when the variables are initialized and updated internally inside the function body. The algorithm supports **scanf** instructions. However, the range analysis performs rather poorly over user-defined variables. This happen because the range of a user-define variable generally cannot be limited to a predefined minimum and maximum value, since we can never know at compile-time what value the user is going to insert. This lack of range for user-defined variables often causes a reduction of the performance of the range analysis. The same applies for argument variables of a function.

The major limitation of the tool is the lack of support for floating point variables. The introduction of floating point ranges complicates a lot the analysis. Many researches and proposes has been advanced for the range analysis of floating point variables, and other solutions are available online.

5.2 Possible improvements

Here I propose some improvements that can be implemented inside the tool for future versions:

- Adding support for floating point range analysis
- Adding support for more IR instructions that may introduce an improvement in performance. Generally speaking, not all instruction may be useful to extract information on ranges. Nonetheless, some advancement can be made by implementing a more thorough analysis of all the IR instruction of LLVM.
- Adding support for function parameters. Currently the tool does not support argument variables in input to the inspected function. It may be possible to improve the performance in case the argument variable analysis yields a more restrictive range.

- Adding support for pointer instructions. A pointer instruction introduces back *load* and *store* instructions in the branch range analysis IR code. Since pointers are broadly used in the C language, this improvement may greatly expand the scope of the tool.

References

[1] Patryk Zadarnowski Adam Wiggins. *Value Range Propagation in LLVM.* 2003. URL: https://www.jantar.org/talks/zadarnowski03vrp.pdf.

[2] Mark Stephenson Jonathan Babb and Saman Amarasinghe. *Bitwidth Analysis with Application to Silicon Compilation.* URL: http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TM-602.pdf.

[3] F. Kenneth Zadeck Barry K. Rosen Mark N. Wegman. *Global Value Numbers and Redundant Computations.* URL: https://www.cse.wustl.edu/~cytron/531Pages/f11/Resources/Papers/valnum.pdf.

[4] Douglas do Couto Teixeira and Fernando Magno Quintao Pereira. *The Design and Implementation of a Non-Iterative Range Analysis Algorithm on a Production Compiler.* URL: https://homepages.dcc.ufmg.br/~fernando/publications/papers/SBLP2011_douglas.pdf.

[5] Marcel Gort and Jason H. Anderson. *Range and Bitmask Analysis for Hardware Optimization in High-Level Synthesis.* URL: http://legup.eecg.utoronto.ca/marcelaspdac13.pdf.

[6] Universidade Federal de Minas Gerais – Department of Computer Science – Programming Languages Laboratory. *Writing an LLVM Pass.* URL: https://www.inf.ed.ac.uk/teaching/courses/ct/19-20/slides/llvm-2-writing_pass.pdf.

[7] Jason R. C. Patterson. *Accurate Static Branch Prediction by Value Range Propagation.* URL: http://www.lighterra.com/papers/valuerangeprop/Patterson1995-ValueRangeProp.pdf.

[8] Fernando Magno Quintão Pereira. *Range Analysis - Program Analysis and Optimization – DCC888.* URL: https://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/RangeAnalysis.pdf.

[9] LLVM Project. *Getting Started with the LLVM System.* URL: http://llvm.org/docs/GettingStarted.html.

[10] LLVM Project. *Getting Started: Building and Running Clang.* URL: http://clang.llvm.org/get_started.html.

[11] Fernando Magno Quintao Pereira Raphael Ernani Rodrigues Victor Hugo Sperle Campos. *A Fast and Low-Overhead Technique to Secure Programs Against Integer Overflows*. URL: https://dl.acm.org/doi/10.1109/CGO.2013.6494996.

[12] Uffe Sørensen. *Static Single-Assignment Form and Value Range Propagation for Uppaal*. URL: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=5088C9739B6204085046CF7DD52424F2?doi=10.1.1.453.4840&rep=rep1&type=pdf.

[13] Igor Rafael de Assis Costa Victor Hugo Sperle Campos Raphael Ernani Rodrigues and Fernando Magno Quint˜ao Pereira. *Speed and Precision in Range Analysis*. URL: https://homepages.dcc.ufmg.br/~fernando/publications/papers/SBLP12_victor.pdf.

[14] Andrzej Warzyński. *Wring an LLVM Pass: 101*. 2019. URL: https://llvm.org/devmtg/2019-10/slides/Warzynski-WritingAnLLVMPass.pdf.

[15] Weng Fai Wong Yang Ding. *Bit-Width Analysis for General Applications*. URL: https://core.ac.uk/download/pdf/4384351.pdf.