A Tool for Range Analysis of Whole Programs

Victor Hugo Sperle Campos, Raphael Ernani Rodrigues, Igor Rafael de Assis Costa, Douglas do Couto Teixeira and Fernando Magno Quintão Pereira

[1]UFMG – 6627 Antônio Carlos Av, 31.270-010, Belo Horizonte, Brazil

{victorsc,raphael,igor,douglas,fernando}@dcc.ufmg.br

Abstract. *Range analysis is a compiler technique that determines statically the lower and upper values that each integer variable from a target program may assume during this program's execution. This type of inference is very important, because it enables several compiler optimizations, such as dead and redundant code elimination, bitwidth aware register allocation, and detection of program vulnerabilities. In this paper we present a tool that implements an inter-procedural, context-sensitive range analysis algorithm for the LLVM compiler. Our tool can be used as a standalone compiler pass that gives the user static information about the variables in a program, or it can be used to enable other compiler optimizations. This tool has been implemented to be able to scale to large code bodies. As an example, we have used it to analyze the whole gcc compiler is less than 15 seconds.*

1. Introduction

The analysis of integer variables on the interval lattice has been the canonical example of abstract interpretation since its introduction in Cousot and Cousot's seminal paper [4]. Compilers use range analysis to infer the possible values that discrete variables may assume during program execution. This analysis has many uses. For instance, it allows the optimizing compiler to remove from the program text redundant overflow tests [12] and unnecessary array bound checks [2]. Additionally, range analysis is essential not only to the bitwidth aware register allocator [1], but also to more traditional allocators that handle registers of different sizes [10]. Finally, range analysis has also seen use in the static prediction of branches [9], to detect buffer overflow vulnerabilities [11], to find the trip count of loops [7] and even in the synthesis of hardware [8].

We have implemented a range analysis tool on top of the LLVM compiler [6]. This tool can be used either as a standalone analysis, or it can be called by a client pass in order to feed this client with more information about the program. One of our main concerns when developing our range analysis was scalability, and we believe that our final product meets the requirements of industrial-quality software. We have used it to analyze a test suite with 2.72 million lines of C code. Our implementation is fast: it globally analyzes the gcc source code in less than 15 seconds, for instance. It is also precise: our results are similar to Stephenson *et al*'s [13], even though our analysis does not require a backward propagation phase. Furthermore, we have been able to find tight bounds to the majority of the examples used by Costan *et al.* [3] and Lakhdar *et al.* [5], who rely on much more costly methods. In Section 2 we provide a description of our implementation.

Software: Our tool is publicly available for download at `http://code.google.com/p/range-analysis/`. In our webpage we provide, in addition to the static range analysis itself, a profiler that records the minimum and maximum values assigned to each variable during program execution. We also provide a visual interval to our tool, which lets the user to see the control flow graph of the target program. This visual interface also lets the user to see the integer intervals that the analysis estimates to each variable.

2. Algorithm Description

The Interval Lattice. We perform arithmetic operations over the complete lattice $\mathcal{Z} = \mathbb{Z} \cup \{-\infty, +\infty\}$, where the ordering is naturally given by $-\infty < \ldots < -2 < -1 < 0 < 1 < 2 < \ldots + \infty$. For any $x > -\infty$ we define:

$$\begin{array}{ll} x + \infty = \infty, x \neq -\infty & x - \infty = -\infty, x \neq +\infty \\ x \times \infty = \infty \text{ if } x > 0 & x \times \infty = -\infty \text{ if } x < 0 \\ 0 \times \infty = 0 & (-\infty) \times \infty = \text{ not defined} \end{array}$$

From the lattice $\mathcal{Z}$ we define the product lattice $\mathcal{Z}^2$, which is partially ordered by the subset relation $\sqsubseteq$. $\mathcal{Z}^2$ is defined as follows:

$$\mathcal{Z}^2 = \emptyset \cup \{[z_1, z_2] |\ z_1, z_2 \in \mathcal{Z},\ z_1 \leq z_2,\ -\infty < z_2\}$$

The objective of range analysis is to determine a mapping $I : \mathcal{V} \mapsto \mathcal{Z}^2$ from the set of integer program variables V to intervals, such that, for any variable $v \in V$, if $I(v) = [l, u]$, then, during the execution of the target program, any value i assigned to v is such that $l \leq i \leq u$.

A Holistic View of our Range Analysis Algorithm. We perform range analysis in a number of steps. First, we convert the program to a suitable intermediate representation that makes it easier to extract constraints. From these constraints, we build a dependence graph that allows us to do range analysis sparsely. Finally, we solve the constraints applying different fix-point iterators on this dependence graph. Figure 1 gives a global view of this algorithm. Some of the steps in the algorithm are optional. They improve the precision of the range analysis, at the expense of a longer running time. The last phase happens per strong component; however, if we opted for not building these components, then it happens once for the entire constraint graph. Nevertheless, the use of strongly connected components is so essential for performance and precision that it is considered optional only because we can easily build our implementation without this module.

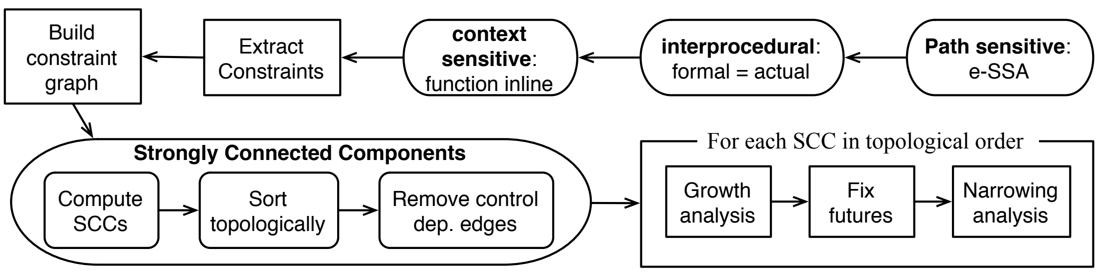
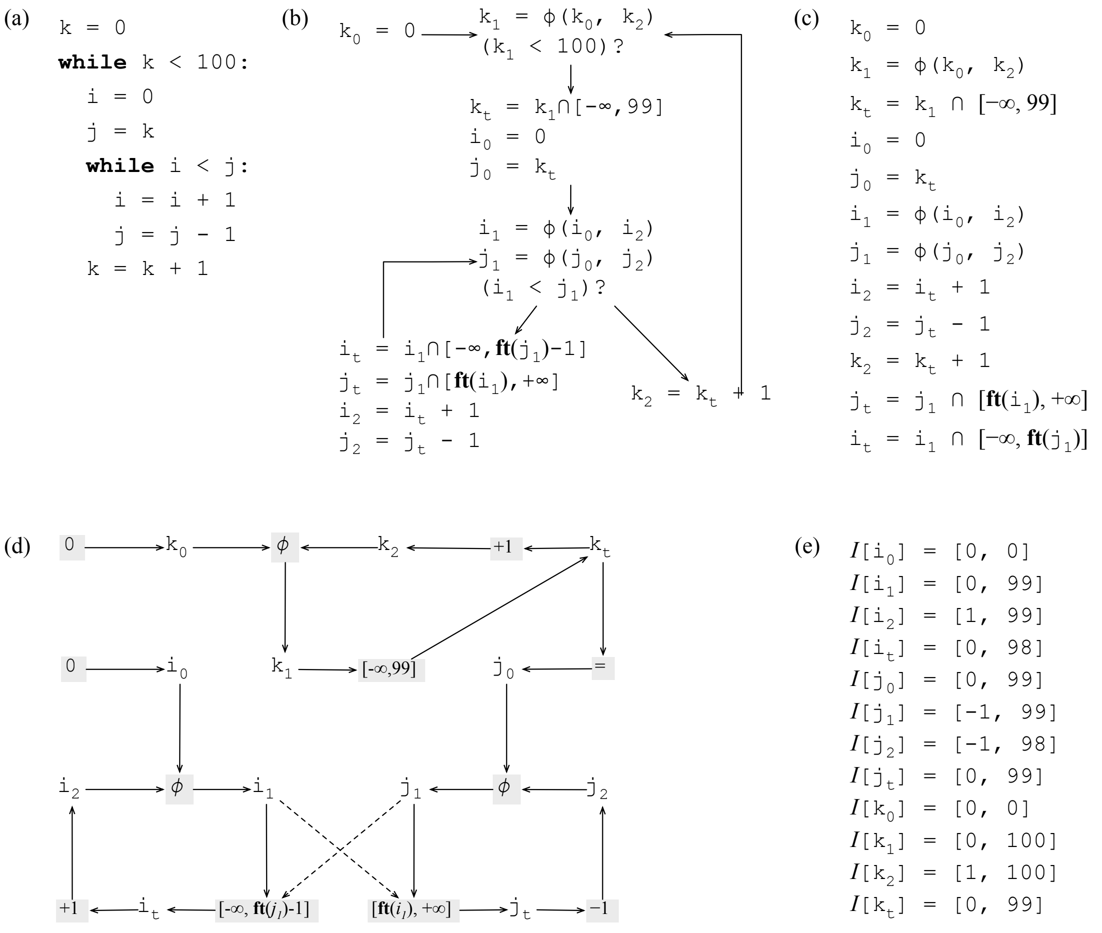

Build constraint graph
Extract Constraints
context sensitive: function inline
interprocedural: formal = actual
Path sensitive: e-SSA
Strongly Connected Components
Compute SCCs
Sort topologically
Remove control dep. edges
For each SCC in topological order
Growth analysis
Fix futures
Narrowing analysis

Fig. 1. Our implementation of range analysis. Rounded boxes are optional steps.


(a)

```
k = 0
while k < 100:
  i = 0
  j = k
  while i < j:
    i = i + 1
    j = j - 1
  k = k + 1
```

(c)

$k_0 = 0$
$k_1 = \phi(k_0, k_2)$
$k_t = k_1 \cap [-\infty, 99]$
$i_0 = 0$
$j_0 = k_t$
$i_1 = \phi(i_0, i_2)$
$j_1 = \phi(j_0, j_2)$
$i_2 = i_t + 1$
$j_2 = j_t - 1$
$k_2 = k_t + 1$
$j_t = j_1 \cap [\mathbf{ft}(i_1), +\infty]$
$i_t = i_1 \cap [-\infty, \mathbf{ft}(j_1)]$

(e)

$I[i_0] = [0, 0]$
$I[i_1] = [0, 99]$
$I[i_2] = [1, 99]$
$I[i_t] = [0, 98]$
$I[j_0] = [0, 99]$
$I[j_1] = [-1, 99]$
$I[j_2] = [-1, 98]$
$I[j_t] = [0, 99]$
$I[k_0] = [0, 0]$
$I[k_1] = [0, 100]$
$I[k_2] = [1, 100]$
$I[k_t] = [0, 99]$

Fig. 2. Range analysis by example. (a) Input program. (b) Internal compiler representation. (c) Constraints of the range analysis problem. (d) The constraint graph. (e) The final solution.

We will illustrate the mandatory parts of the algorithm via the example program in Figure 2. Figure 2(a) shows a program taken from the partition function of the quicksort algorithm used by Bodik *et al.* [2]. We have removed the code that performs array manipulation from this program, as it plays no role in our explanation. Figure 2(b) shows one possible way to represent this program internally. A good program representation allows us to find more precise results. In this example we chose a program representation called Extended Static Single Assignment form [2], which lets us to solve range analysis via a path sensitive algorithm. Figure 2(c) shows the constraints that we extract from the intermediate representation seen in part (b) of this figure. From these constraints we build the *constraint graph* in Figure 2(d). This graph is the main data-structure that we use to solve range analysis. For each variable v in the constraint system, the constraint graph has a node n_v. Similarly, for each operation $v = f(\ldots, u, \ldots)$ in the program, the graph has an *operation node* n_f. For each operation $v = f(\ldots, u, \ldots)$ we add two edges to the graph: $\overrightarrow{n_u n_f}$ and $\overrightarrow{n_f n_v}$. Some edges in the constraint graph are dashed. These are called *control dependence edges*. If a constraint $v = f(\ldots, \mathbf{ft}(u), \ldots)$ uses a *future* bound from a variable u, then we add to the constraint graph a control dependence edge $\overrightarrow{n_u n_f}$. Futures are symbolic expressions, which denote intervals whose limits are not known immediately, but might be known after the analysis starts solving the constraints. The final solution to this instance of the range analysis problem is given in Figure 2(e).

3. How to Create a LLVM pass using our Analysis

Our analysis can be used independently from other compiler clients, to identify logical problems in the analyzed code. However, the Range Analysis is more often used as a tool to identify dead code, memory overflow and redundant checks; hence, enabling other compiler analysis. We will illustrate this last mode via an example. In order to use our range analysis, one can write a LLVM pass that calls it. There is vast documentation [1] about how to write LLVM passes in the web. The program below, which is self-contained, is an example of such a pass. This program is a LLVM pass; that is, it is called as part of the LLVM tool chain, and receives as input a data-structure of the `Function` type, that describes the program. An iterator over such a data-structure returns a list of basic blocks. Iterators over basic-blocks return a list of instructions.

```
//We are ommiting the LLVM includes
#include "../RangeAnalysis/RangeAnalysis.h"
using namespace llvm;
class ClientRA: public llvm::FunctionPass {
public:
  static char ID;
  ClientRA() : FunctionPass(ID){ }
  virtual ~ClientRA() { }
  virtual bool runOnFunction(Function &F){
    IntraProceduralRA<Cousot> &ra = getAnalysis<IntraProceduralRA<Cousot>>();
    errs() << "\nCousot Intra Procedural analysis
              (Values -> Ranges) of " << F.getName() << ":\n";
    for(Function::iterator bb=F.begin(), bbEnd=F.end(); bb!=bbEnd; ++bb){
      for(BasicBlock::iterator I=bb->begin(), IEnd=bb->end(); I!=IEnd; ++I){
        if(I->getOpcode() == Instruction::Store){
          const Value *v = &(*I);
          Range r = ra.getRange(v);
          r.print(errs());
          I->dump();
        }
      }
    }
    return false;
  }

  virtual void getAnalysisUsage(AnalysisUsage &AU) const {
    AU.setPreservesAll();
    AU.addRequired<IntraProceduralRA<Cousot> >();
  }
};

char ClientRA::ID = 0;
static RegisterPass<ClientRA>
       X("client-ra", "A client that uses RangeAnalysis", false, false);
```

Our Range Analysis interface provides a method, *getRange*, that returns a *Range* object for any variable in the original code. This object, of type *Range*, contains the range information related to the variable. There are many versions of our range analysis pass, e.g., intra/inter procedural, with different narrowing operators, etc. In this example

[1] `http://llvm.org/docs/WritingAnLLVMPass.html`

we are using the intra-procedural version using Cousot & Cousot's original narrowing operator [4].

In order to use the example client, you need to give it a bitcode input file. Below we show how to do this. First, we can translate a c file into a bitcode file using clang:

```
clang -c -emit-llvm test.c -o test.bc
```

Next thing: we must convert the bitcode file to e-SSA form. We do it using the *vssa* pass. The e-SSA conversion module is distributed together with our analysis; however, its use is not mandatory. It increases the precision of the analysis, at the expenses of a small slowdown. In our experiments, we have observed that this slowdown is very small.

```
opt -instnamer -mem2reg -break-crit-edges test.bc -o test.bc
opt -load LLVM_SRC_PATH/Debug/lib/vSSA.so -vssa test.bc -o test.essa.bc
```

Notice that we use a number of other passes too, to improve the quality of the code that we are producing: *instnamer* just assigns strings to each variable. We only use this pass for aesthetic reasons. This pass will help our visual interface to look nicer. *mem2reg* maps variables allocated in the stack to virtual registers. Without this pass everything is mapped into memory, and then our range analysis will not be able to find any meaningful ranges to the variables. *break-crit-edges* removes the critical edges from the control flow graph of the input program. This pass will increase the precision of our range analysis (just a tiny bit though), because the e-SSA transformation will be able to convert more variables into a better format. We run the example client with the commands below:

```
opt -load LLVM_SRC_PATH/Debug/lib/RangeAnalysis.so
    -load LLVM_SRC_PATH/Debug/lib/ClientRA.so -client-ra test.essa.bc
```

This sequence of calls will cause our example pass to be dynamically loaded by the LLVM framework. It will receive the bitcodes that we had produced before, and will produce, as output, a list of variables, followed by their intervals.

References

[1] Rajkishore Barik, Christian Grothoff, Rahul Gupta, Vinayaka Pandit, and Raghavendra Udupa. Optimal bitwise register allocation using integer linear programming. In *LCPC*, volume 4382 of *Lecture Notes in Computer Science*, pages 267–282. Springer, 2006.

[2] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM, 2000.

[3] Alexandru Costan, Stephane Gaubert, Eric Goubault, Matthieu Martel, and Sylvie Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In *CAV*, pages 462–475, 2005.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[5] Lies Lakhdar-Chaouch, Bertrand Jeannet, and Alain Girault. Widening with thresholds for programs with complex control graphs. In *ATVA*, pages 492–502. Springer-Verlag, 2011.

[6] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE, 2004.

[7] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *CGO*, pages 136–146. IEEE, 2009.

[8] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(11):1355–1371, 2001.

[9] Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *PLDI*, pages 67–78. ACM, 1995.

[10] Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation by puzzle solving. In *PLDI*, pages 216–226. ACM, 2008.

[11] Axel Simon. *Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities*. Springer, 1th edition, 2008.

[12] Marcos Rodrigo Sol Souza, Christophe Guillon, Fernando Magno Quintao Pereira, and Mariza Andrade da Silva Bigonha. Dynamic elimination of overflow tests in a trace compiler. In *CC*, pages 2–21, 2011.

[13] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *PLDI*, pages 108–120. ACM, 2000.