Tool for Static Range Analysis of Whole Programs

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

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

{victorsc,raphael,igor,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 describe an inter-procedural, context-sensitive range analysis algorithm that we have implemented in the LLVM compiler. During the effort to produce an industrial-quality implementation of our algorithm, we had to face a constant tension between precision and speed. The foremost goal of this paper is to discuss the many engineering choices that, due to this tension, have shaped our implementation. Given the breath of our evaluation, we believe that this paper contains the most comprehensive empirical study of a range analysis algorithm ever presented in the compiler related literature.*

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 [8]. 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 [29] and unnecessary array bound checks [5]. Additionally, range analysis is essential not only to the bitwidth aware register allocator [2, 33], but also to more traditional allocators that handle registers of different sizes [16, 26, 27]. Finally, range analysis has also seen use in the static prediction of branches [25], to detect buffer overflow vulnerabilities [28, 36], to find the trip count of loops [20] and even in the synthesis of hardware [6, 19, 21].

Given this great importance, it comes as no surprise that the compiler literature is rich in works describing in details algorithmic variations of range analyses [14, 21, 30, 32]. On the other hand, none of these authors provide experimental evidence that their approaches are able to deal with very large programs. There are researchers who have implemented range analyses that scale up to large programs [4, 35]; nevertheless, because the algorithm itself is not the main focus of their works, they neither give details about their design choices nor provide experimental data about it. This scenario was recently changed by Oh *et al.* [24], who introduced an abstract interpretation framework which processes programs with hundreds of thousands of lines of code. Nevertheless, Oh *et al.* have designed a very simple range analysis, which does not handle comparisons between variables, for instance. They also do not discuss the precision of their implementation, but only its runtime.

We have implemented our algorithm in the LLVM compiler [18] as a pass, that can be used either as a stand alone analysis pass or called by a client pass. 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 [30], 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.* [7] and Lakhdar *et al.* [17], who rely on much more costly methods. In Section 2 we provide a description of our implementation.

Our demo tool provides a visual interface that allows the user to try our range analysis. The tool runs our analysis in the source code of user's programs. The user can produce the CFG, the Constraint Graph and evaluate the analysis precision with the execution of an instrumented version of his program.

2. Algorithm Description

The Interval Lattice. Following Gawlitza *et al.*'s notation, we shall be performing 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 valued 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.

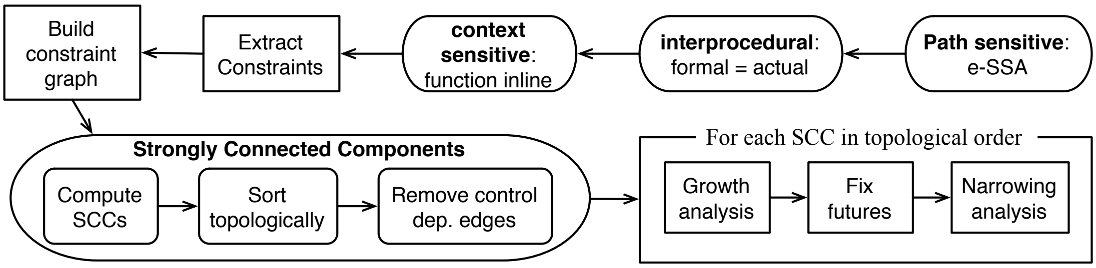

Fig. 1. Our implementation of range analysis. Rounded boxes are optional steps.

We will illustrate the mandatory parts of the algorithm via the example program in Figure 2. Figure 2(a) shows an example program taken from the partition function of the quicksort algorithm used by Bodik *et al.* [5]. 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, 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 constraint $v = f(\ldots, u, \ldots)$ in the constraint system, the graph has an *operation node* n_f. For each constraint $v = f(\ldots, u, \ldots)$ 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 $\overrightarrow{n_u n_f}$. 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 stand-alone, to identify logical problems in the analyzed code. However, the Range Analysis can be used as a tool to identify dead code, memory overflow, redundant checks and many other analyses. Then, our analysis tool was designed to allow being called from client passes. 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.

```
//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() { }
```

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

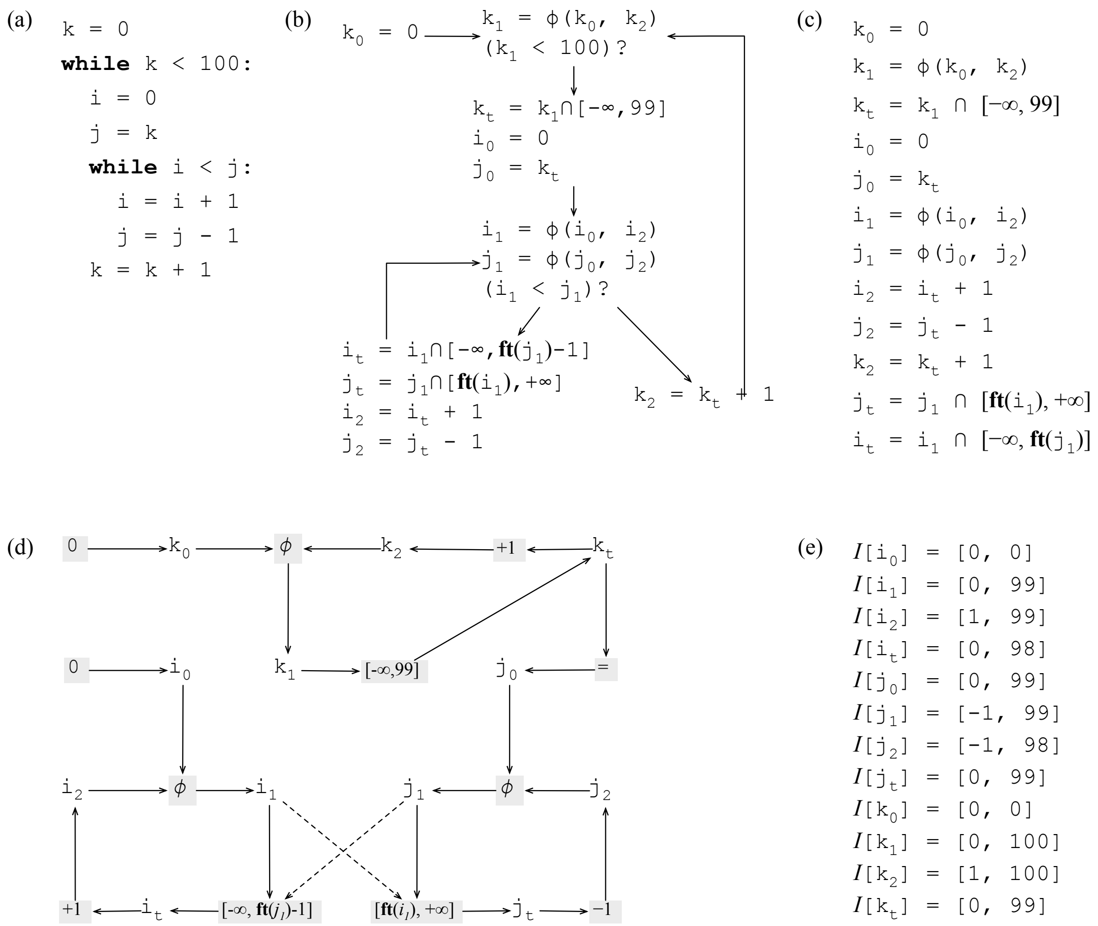

Fig. 2. Range analysis by example. (a) Input program. (b) Internal compiler representation. (c) Constraints of the range analysis problem. (d) The constraint graph. (e) The final solution.

```
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 we are using the intra-procedural version using Cousot Cousot's original narrowing operator.

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.

```
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. This will make the dot files that we produce to look nicer. We only use this pass for aesthetic reasons. *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 will increase the precision of our range analysis (just a tiny bit though), because the e-SSA transformation will be able to insert more sigma-functions into the code.

Now, we can run our example client. We can do this with the code below:

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

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[2] 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.

[3] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *I@A*, pages 1–38. AIAA, 2010.

[4] B. Blanchet, P Cousot, R. Cousot, J. Feret, L. Mauborgne, A Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207. ACM, 2003.

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

[6] Jason Cong, Yiping Fan, Guoling Han, Yizhou Lin, Junjuan Xu, Zhiru Zhang, and Xu Cheng. Bitwidth-aware scheduling and binding in high-level synthesis. *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, 2:856–861, 18-21 Jan. 2005.

[7] 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.

[8] 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.

[9] P. Cousot and N.. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM, 1978.

[10] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does astrée scale up? *Form. Methods Syst. Des.*, 35(3):229–264, 2009.

[11] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

[12] Douglas do Couto Teixeira and Fernando Magno Quintao Pereira. The design and implementation of a non-iterative range analysis algorithm on a production compiler. In *SBLP*, pages 45–59. SBC, 2011.

[13] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319–349, 1987.

[14] Thomas Gawlitza, Jerome Leroux, Jan Reineke, Helmut Seidl, Gregoire Sutre, and Reinhard Wilhelm. Polynomial precise interval analysis revisited. *Efficient Algorithms*, 1:422 – 437, 2009.

[15] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. Taming false alarms from a domain-unaware c analyzer by a bayesian statistical post analysis. In *SAS*, pages 203–217, 2005.

[16] Timothy Kong and Kent D Wilken. Precise register allocation for irregular architectures. In *MICRO*, pages 297–307. IEEE, 1998.

[17] 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.

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

[19] G Lhairech-Lebreton, P Coussy, D. Heller, and E. Martin. Bitwidth-aware high-level synthesis for designing low-power dsp applications. In *ICECS*, pages 531–534. IEEE, 2010.

[20] 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.

[21] 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.

[22] Antoine Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19:31–100, 2006.

[23] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[24] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for c-like languages. In *PLDI*, page to appear. ACM, 2012.

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

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

[27] Bernhard Scholz and Erik Eckstein. Register allocation for irregular architectures. In *LCTES/SCOPES*, pages 139–148. ACM, 2002.

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

[29] 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.

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

[31] Zhendong Su and David Wagner. A class of polynomially solvable range constraints for interval analysis without widenings and narrowings. In *TACAS*, pages 280–295, 2004.

[32] Zhendong Su and David Wagner. A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computeter Science*, 345(1):122–138, 2005.

[33] Sriraman Tallam and Rajiv Gupta. Bitwidth aware global register allocation. In *POPL*, pages 85–96, New York, NY, USA, 2003. ACM.

[34] Andre L. C. Tavares, Benoit Boissinot, Mariza A. S. Bigonha, Roberto Bigonha, Fernando M. Q. Pereira, and Fabrice Rastello. A program representation for sparse dataflow analyses. *Science of Computer Programming*, X(X):2–25, 201X. Invited paper with publication expected for 2012.

[35] Arnaud Venet and Guillaume Brat. Precise and efficient static array bound checking for large embedded c programs. *SIGPLAN Not.*, 39:231–242, 2004.

[36] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, pages 3–17. ACM, 2000.

[37] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., 2002.