The Design and Implementation of a Non-Iterative Range Analysis Algorithm on a Production Compiler

Douglas do Couto Teixeira and Fernando Magno Quintão Pereira

Departamento de Ciência da Computação – UFMG
Av. Antônio Carlos, 6627 – 31.270-010 – Belo Horizonte – MG – Brazil
{douglas,fpereira}@dcc.ufmg.br

Abstract. This paper presents the first implementation of a non-iterative range analysis algorithm in a production compiler. Discrete range analyses try to discover the intervals of values that may be bound to the integer variables during program execution. This information enables compiler optimizations such as dead code elimination and the detection of bugs such as buffer overflow vulnerabilities. So far, non-iterative range analysis algorithms have been constrained to theoretical works – actual implementations never reaching the boundaries of industrial strength compilers. In this paper we fix this omission by implementing in the LLVM compiler the constraint system that Su and Wagner designed in 2005. In the effort to implement this method in an actual compiler we had to modify Su's algorithm in many ways. In particular, we use Gawlitza's algorithm to handle program loops, and Bodik's Extended Static Single Assignment form to add flow sensitiveness to our analysis. We have tested this analysis with a compiler optimization that converts 32-bit variables to either 8-bit or 16-bit variables whenever possible. Our implementation of range analysis has been able to process over 4 million assembly instructions in 223 seconds, yielding results on par with previous works. For instance, we have reduced by 39.4% on average the bit size of the integer variables in the bitwise benchmark suite.

1 Introduction

Compilers use integer range analysis to infer the possible values that discrete variables may assume during the execution of programs. Such analysis has many uses. For instance, it enables very extensive forms of dead code elimination, allowing the optimizing compiler to remove from the program text redundant overflow tests [18] and array bound checks [4]. Additionally, range analysis is essential not only to the bitwidth aware register allocator [2, 21], but also to more traditional allocators that handle registers of different sizes [10, 15, 16]. Finally, range analysis has also seen use in the static prediction of branches [14], to detect buffer overflow vulnerabilities [17, 22], to find the trip count of loops [12] and even in the synthesis of hardware [5, 13].

Given this great importance, it comes as no surprise that the compiler literature is rich in works describing many variations of range analyses [9, 13, 19, 20]. These works follow two main avenues, which, although bearing similar objectives, constitute very different approaches. The most well known techniques, such as Stephenson's [19] or Mahlke [13], are iterative. That is, these algorithms rely on the monotone set of dataflow

equations traditionally used in program analyses in order to find a fix point that satisfies a collection of range constraints. In order to stop, iterative methods recourse to widening and narrowing operators, as proposed by Cousot and Cousot [7]. There exist implementations of iterative range analyses in compilers such as Open64 and STMicroelectronic's LAO. The second group of works brings in non-iterative algorithms, which can be solved in polynomial time in the program size. Among these algorithms we cite Su and Wagner's [20] and Gawlitza *et al.*'s [9] methods. While elegant and supposedly efficient, none of these approaches has been put to use in a production compiler to the best of our – and the the original authors's [1] – knowledge.

In this paper we side with the advocates of the non-iterative approach, and provide the first implementation of such an algorithm on an industrial strength compiler. We have implemented a variation of Su and Wagner's contraint system [20] in the Low Level Virtual Machine (LLVM) [11] compiler. In the effort to craft a fully functional implementation of Su and Wagner's algorithm we had to make many changes into the original technique, in order to accommodate the complexities of actual programs. Furthermore, we had to write a new algorithm to deal with program loops, as the original exposition, in the words of Gawlitza *et al.* [9, p.422], "is not very explicit". In the end, we designed an algorithm that not only solves range analysis with impressive speed and acceptable precision, but also subsumes the conditional constant propagation technique of Wegman and Zadeck [23]. Therefore, we claim the following contributions:

- the first implementation of a non-iterative range analysis algorithm in a production compiler, as well as the first extensive use of the technique in large program bodies, such as SPEC CPU 2006;
- a different algorithm to deal with program loops, which relies on an adaptation of the Bellman-Ford maximum flow algorithm by Gawlitza *et al.* [9];
- a suite of extensions on Su and Wagner's constraints, in order to deal with arithmetic shifts and bitwise integer operations. Moreover, we use the type information from the compiler's intermediate representation to restrict even more the range of values that can be assigned to variables;
- the use of Extended Static Single Assignment form (e-SSA) [4] to add flow sensitiveness to our implementation of range analysis. By deriving constraints from e-SSA form programs we can use conditional tests to put tighter bounds on variables, without having to modify Su and Wagner's constraints.

We have run our implementation on over 2 million lines of C programs, processing over 4 million assembly instructions. Our test set include traditional C benchmarks such as Shootout, MediaBench and Touchstone. In this paper we present results for SPEC CPU 2006 and bitwise, which is a collection of programs used in previous works to test range analyses [2, 19]. In order to test the correctness of our implementation we have also developed a compiler transformation that does a limited form of type specialization. Thus, we convert 32-bit integer variables to either 8-bit bytes or 16-bit shorts, whenever the range analysis indicates that such conversion is safe. This pass is useful on its own, because it enables bit-width aware peephole optimizations. Even more important, this

[1] Personal communication with Zhendong and Wagner (February 2011) and Gawlitza (April 2011)

transformation frees developers from the very mundane concern of finding economical representation for program variables. Our implementation has been able to reduce by 39.4% the amount of bits necessary to hold discrete variables in the bitwise benchmark. Our type specialization pass has changed 4.6% of the integer variables into shorts, and 26% of these variables into bytes.

The rest of this paper is divided as follows: in Section 2 we define range analysis and explain Su and Wagner's constraint system. In Section 3 we describe our implementation of the algorithm, giving special emphasis on those aspects of our implementation that differ from the original technique. Section 4 shows our experimental results, and Section 5 concludes this work.

2 Background

2.1 Theoretical Definitions

In this paper we use Gawlitza *et al.*'s definition of range analysis [9]. We shall be performing arithmetic operations over the complete lattice $Z = \mathbb{Z} \cup \{-\infty, +\infty\}$, where the ordering is naturally given by $-\infty < \ldots < -2 < -1 < 0 < 1 < 2 < \ldots + \infty$. Z is equipped with addition and multiplication with nonnegative constants; these operations have the semantics traditionally seen in ordinary math. Additionally, for any $x > -\infty$ we define:

$x + \infty = \infty, x \neq -\infty$ $\quad$ $x - \infty = -\infty, x \neq +\infty$
$x \times \infty = \infty$ if $x > 0$ $\quad$ $x \times \infty = -\infty$ if $x < 0$
$0 \times \infty = 0$ $\quad$ $(-\infty) \times \infty =$ not defined

From the lattice Z we define the interval lattice I as follows:

$$I = \emptyset \cup \{[z_1, z_2] \in Z^2 \mid z_1 \leq z_2\} \tag{1}$$

This interval lattice is partially ordered by the subset relation, which we denote by "$\sqsubseteq$". The meet operator "$\sqcap$" is defined by:

$$[a_1, a_2] \sqcap [b_1, b_2] = [\max(a_1, b_1), \min(a_2, b_2)], \text{ if } a_1 \leq b_1 \leq a_2 \text{ or } b_1 \leq a_1 \leq b_2$$

$$[a_1, a_2] \sqcap [b_1, b_2] = \emptyset, \text{ otherwise}$$

whereas the join operator "$\sqcup$" is defined by:

$$[a_1, a_2] \sqcup [b_1, b_2] = [\min(a_1, b_1), \max(a_2, b_2)]$$

Given $\{a, b\} \subset \mathbb{Z}$, $\{c, d\} \subset Z$ and $\{X, Y\} \subset \mathcal{V}$, where $\mathcal{V}$ is a set of variables, we consider a system with the constraints defined by Equation 2, as identified by Su and Wagner [20].

$$aX + b \sqcap [c, d] \sqsubseteq Y \tag{2}$$

Quoting Su and Wagner, we define a *valuation* ρ as a mapping $\mathcal{V} \mapsto \mathcal{I}$, such that:

- $\rho([c, d]) = [c, d]$
- $\rho(n \times X) = n \times \rho(X)$, where $n \times [c, d] = [\min(nc, nd), \max(nc, nd)]$

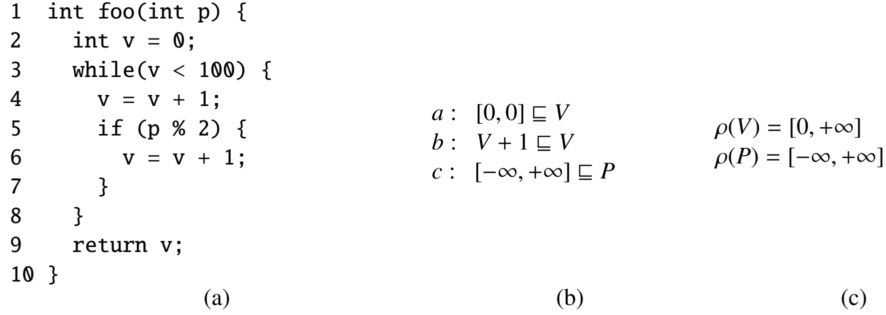

Fig. 1. (a) An example program written in C. (b) The constraint system derived from the program. (c) A solution to the constraints.

– $\rho(E_1 + E_2) = \rho(E_1) + \rho(E_2)$, where $[l_1, u_1] + [l_2, u_2] = [l_1 + l_2, u_1 + u_2]$

We say that ρ satisfies a constraint $E \sqcap r \sqsubseteq X$ if $\rho(E) \sqcap r \sqsubseteq \rho(X)$. Armed with these concepts, we define the range constraints problem as follows:

Definition 1. Range Constraints Problem
Input: *a set S of constraints following Equation 2.*
Output: *a valuation ρ that satisfies every constraint in S.*

We shall use the example in Figure 1 to illustrate the notions previously introduced. The program in Figure 1(a) increments variable v, either 50 or 100 times, depending on the parity of the parameter p. From line 2 in the program we infer the constraint a in Figure 1(b). From line 6 we infer constraint b. We adopt the convention that lower case letters denote names of program variables, and upper case letters denote variables in $\mathcal{V}$. The program does not offer any hint to narrow the interval bound to p, so we let $\rho(P) = [-\infty, +\infty]$. Notice that a valuation ρ that assigns $[-\infty, +\infty]$, the largest element of our lattice, to every range variable satisfies any constraint system. However, this solution is too imprecise to have any use, and generally we can do better. Figure 1(c) shows a possible solution to this constraint system. In Section 3 we shall explain the algorithm to find this solution, and shall discuss a different program representation that allows us to infer much more precise intervals.

2.2 Applications of Range Analysis

Range analysis has many applications, mostly in program optimizations, but also in bug detection and program understanding. In this section we shall discuss some of these applications, starting with code optimization, which we explain via the programs in Figure 2. In this case, the first transformation that yields good results are peephole optimizations that consist in removing bit masks and redundant truncation from the program text. For instance, the program in Figure 2(a) uses a mask to zero the upper three bytes of an integer. However, the conditional test guarantees that these bytes are already zeros, and the mask is unnecessary.

```
int foo(int v) {
  if (v < 100) {
    int u = v &
      0x000000FF;
    return u;
  }
}
```

(a)

```
int[] a =
  new int[100];
int i = 0;
while (i < 100) {
  if (i < 100) {
    a[i] = 0;
  } else
    throw ArrayEx
  i++;
}
```

(b)

```
int i = 0;
while (i < 100) {
  if (i > 255) {
    throw new
      OvfEx();
  } else {
    i++;
  }
}
```

(c)

Fig. 2. (a) Peephole optimization. (b) Array bound checking in strongly typed languages. (c) Overflow elimination in script languages.

Range analysis is also used in strongly typed languages, such as Java, to eliminate safety checks related to array accesses [4]. In such languages it is a programming error to read or write a position outside the boundaries of an array, and an exception must be raised whenever such attempt happens. The compiler normally checks, via conditional tests, every array access. However, programmers many times also do it, in order to explicitly control the exceptional behavior. In this case, the compiler test is unnecessary, and can be removed, like in the example in Figure 2(b), where the branch inside the loop is redundant.

Another situation in which range analysis is useful is in the elimination of overflow tests in scripting languages such as Lua and JavaScript [18]. In these languages, numbers are all represented as floating point values. A common optimization in this case is to convert these numbers to integers, whenever the compiler infers, due to the program syntax, that such conversion is in line with the programmer's intentions. However, in order to preserve the semantics of the original program, an overflow test must precede every arithmetic operation that can go beyond the range of an integer. Many of these overflow tests are redundant, and, as in the example of Figure 2(c), can be eliminated.

Range analysis is also important to detect program bugs, mainly those related to security vulnerabilities, such as buffer overflow attacks [17, 22]. As an example, Figure 3(a) shows a typical program that is vulnerable to a buffer overflow attack. The size s of the input array a may be larger than the size of the buffer, i.e, 100. Given that the C programming language does not enforce that arrays are only used inside declared bounds, the memory past the limits of the buffer may be compromised by the copy. Range analysis, in this case, may detect that the interval bound to s is larger than the interval bound to i; hence, flagging an error.

Finally, range analysis is also useful to improve automatic program understanding [14]. As an example, a static branch predictor may assign a 0.99 probability of the inner branch being taken in Figure 3(b). Furthermore, a compiler supplied by the range analysis can easily infer that the loop in this example has a trip count of 100, and so should (or should not) be unrolled.

```
int* copy(int* a, int s) {
  int int* buffer =
    (int*)malloc(100)
  while (i < s) {
    buffer[i] = a[i];
  }
  return buffer;
}
```
(a)

```
int i = 0;
int N = 100;
while (i < N) {
  if (i < 99) {
    ...
  }
}
```
(b)

Fig. 3. (a) A program that is vulnerable to a buffer overflow attack. (b) Example in which range analysis improves static branch prediction.

3 Our Implementation

Our range analysis algorithms consists of three main steps, which we explain in the rest of this section:

1. convert the program to e-SSA form [4];
2. extract constraints from the program, and build a constraint graph;
3. solve the constraint graph using the algorithm of Bellman-Ford [3] to handle cycles.

3.1 Obtaining flow sensitiveness via e-SSA form

We use the *Extended Static Single Assignment* (e-SSA) representation to improve the precision of our range analysis. The e-SSA program representation is a superset of the well known *Static Single Assignment* (SSA) form [8]. This representation was first used by Bodik *et al.* [4] to eliminate array bound checks. Its main advantage, in our case, is the possibility of acquiring useful information from the outcome of conditional tests, and then binding this information directly to variables. The e-SSA form guarantees this property via special instructions called ϕ-functions and σ-functions. The latter splits the *live ranges* of variables used in conditionals, and the former, already present in SSA-form programs, joins the live ranges of names that denote the same variable in the original program. Ananian gives a detailed explanation about the semantics of these instructions [1]. The live range of a variable v is the collection of program points where v is alive, that is, from where it is possible to reach a statement where v is used. We shall illustrate these concepts on the Program in Figure 1(a); however, instead of relying on its source code, we will use that program's *control flow graph* (CFG), which is given in Figure 4(a). Notice how each variable is defined only once, and how ϕ-functions are used to combine different variable names into a single definition.

The constraints that we can derive from Figure 4(a) are given in the part (b) of this Figure, and a valuation of this constraint system is given in Figure 4(c). We have been able to infer, for instance, that this program does not contain negative values. We could add the unsigned modifier to the integer types, in case the compiler deems this specialization worthwhile. However, although we have been able to infer non-trivial ranges for the program variables, we are far from the tightest possible bounds. In particular,

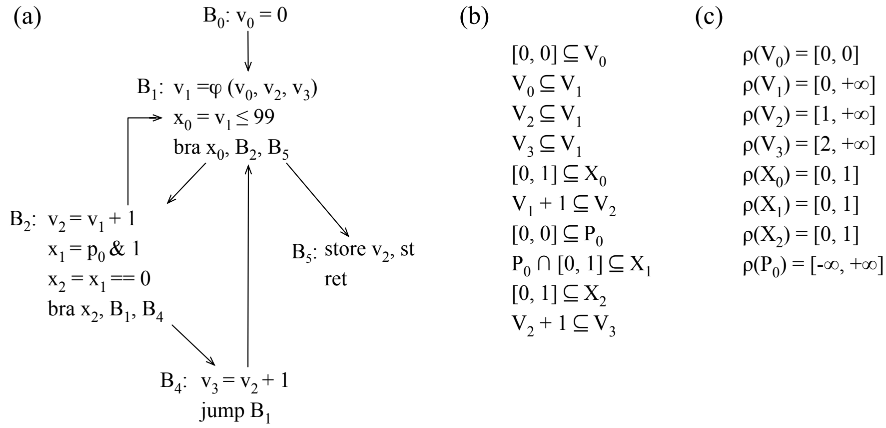

Fig. 4. (a) The control flow graph of the program in Figure 1. (b) The constraints that we derive from this program. (c) A valuation of this constraint system.

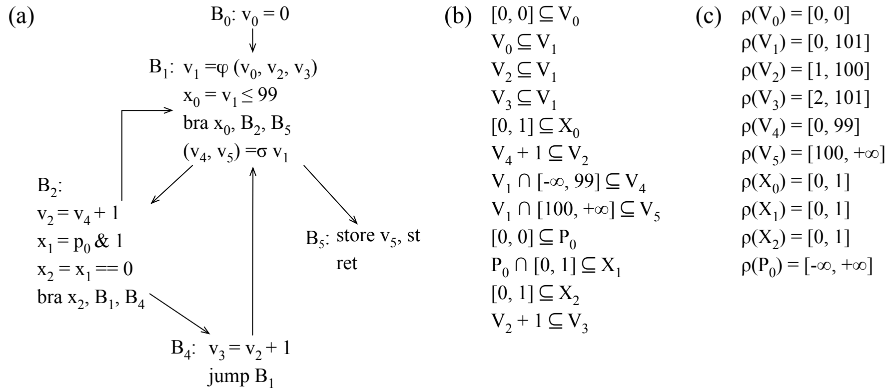

Fig. 5. (a) The CFG from Figure 4 converted into e-SSA form. (b) The constraints that we derive from the e-SSA form program. (c) A valuation of this constraint system. Notice the tighter bounds on v_1.

a visual inspection on the program in Figure 1(a) reveals that the variables v defined inside the loop cannot be larger than 101, due to the conditional test at the beginning of the loop. By converting the SSA-form program into the e-SSA representation, we have a natural way to grab such knowledge.

We convert a program into e-SSA form by redefining the variables that are used in conditional tests, right past these branches. Figure 5(a) shows our example program

Description	Operation	Constraint
Constant assignment	$v = c$	$[c, c] \sqsubseteq v$
Variable assignment	$v_1 = v_2$	$v_2 \sqsubseteq v_1$
Addition	$v_1 = v_2 + c$	$v_2 + c \sqsubseteq v_1$
Multiplication	$v_1 = v_2 * c$	$cv_2 \sqsubseteq v_1$
Integer division	$v_1 = c \text{ div } v_2$	$[-c, c] \sqsubseteq v_1$
Modulus	$v_1 = v_2 \bmod c$	$[0, c-1] \sqsubseteq v_1$
Bitwise and	$v_1 = v_2 \& c$	$v_2 \sqcap [0, c] \sqsubseteq v_2, -v_2 - 1 \sqcap [0, c] \sqsubseteq v_1$
Bitwise or	$v_1 = v_2 \mid c$	$v_2 + c \sqsubseteq v_1, [0, c] \sqsubseteq v_1, v_2 \sqsubseteq v1$
Left shift	$v_1 = v_2 \ll c$	$2^c v_2 \sqsubseteq v_1$
Less than	$(v < c) \rightarrow (v_1, v_2) = \sigma(v)$	$v \sqcap [-\infty, c-1] \sqsubseteq v_1$
		$v \sqcap [c, +\infty] \sqsubseteq v_2$
Less than or equal	$(v \leq c) \rightarrow (v_1, v_2) = \sigma(v)$	$v \sqcap [-\infty, c] \sqsubseteq v_1$
		$v \sqcap [c+1, +\infty] \sqsubseteq v_2$
ϕ-function	$v = \phi(v_1, \ldots, v_n)$	$\forall i, 1 \leq i \leq n, v_i \sqsubseteq v$

Table 1. Constraint derivation rules for integers. We let c denote a positive constant, and $\{v, v_1, v_2, \ldots, \} \subset \mathcal{V}$.

converted into e-SSA form. The only differences in this new program are the two redefinitions of variable v_1, which now gives origin to v_4 and v_5. We know that v_4 is necessarily less than 100, because it only exists in the path where $v_1 \leq 99$ evaluates to true. Similarly, we know that v_5 is necessarily greater than or equal 100. This knowledge, that we can associate directly to the variables created by σ-functions, gives the constraints in Figure 5(b). From this constraints we derive the valuation in the part (c) of this figure. This result is much more precise than the previous result, given in Figure 4(c). In particular, we know that every variable created inside the loop is less than or equal 101. An optimizing compiler could store all these variables into byte-sized words.

3.2 Extracting constraints from the program text

After converting the source program into e-SSA form, the next step is to extract constraints from the program text. We perform this step guided by the rules in Table 1. The table shows only the constraints that we derive for instructions having at most one variable on the right side and using positive constants. Extending the rules to deal with negative constants is trivial, and we will omit it from this presentation. However, dealing with instructions that apply an operation on two variables, such as $v = v_1 + v_2$ is not straightforward. For each such an operation, we create a constraint $v_1 + v_2 \sqsubseteq v$. These types of constraints do not fit on Su/Wagner's framework, and we conservatively let $[-\infty, +\infty] \sqsubseteq v$, if there exists a cycle of constraints that lead from v to either v_1 or v_2.

3.3 Building a constraint graph and solving the constraints

We use Su and Wagner's constraint graph as the basic data-structure used to solve the range analysis problem. Given a set S of constraints that use variables from a set $\mathcal{V}$, we

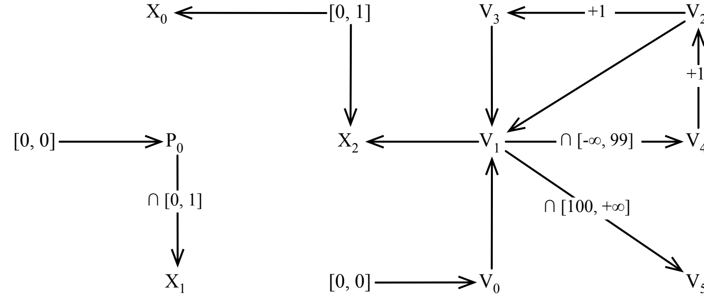

Fig. 6. The constraint graph built after the program in Figure 5(a).

define the constraint graph as a tuple $G = (N, E, L)$. In this case, N is a set of vertices, $E : N \mapsto N$ is a set of edges, and $L : E \mapsto S$ is a function that associates edges to constraints. We build the constraint graph as follows:

- for each variable $V \in \mathcal{V}$ we create a vertex $N_v \in N$;
- for each constraint $f(X) \sqcap r \sqsubseteq Y$, we create an edge $\overrightarrow{N_X N_Y} \in E$, such that $L(\overrightarrow{N_X N_Y}) = f(X) \sqcap r \sqsubseteq Y$;
- for each constraint $[c, d] \sqsubseteq V$, we create a vertex $N_{[c,d]}$, if it does not exist already, and an edge $\overrightarrow{N_{[c,d]} N_v}$. We let $L(\overrightarrow{N_{[c,d]} N_v})$ be $[c, d] \sqsubseteq X$;

As an example, Figure 6 shows the constraint graph that we build from the equations in Figure 5. For the sake of readability, we reuse Su and Wagner's notation, and do not write full constraints on the edges. Rather, we avoid the variable names, because they are implicit in the nodes touched by the edges.

Once we have built the constraint graph $G = (N, E)$, we move on to solve the constraints using the algorithm in Figure 7. This algorithm propagates the intervals along a topological ordering of the constraint graph. If a vertex N_v is a trivial strongly connected component, that is, a SCC that only contains N_v itself, then we find $\rho(v)$ by joining the intervals of every vertex that is a predecessor of N_v, via the *evaluate* procedure. Otherwise, we have a cycle, which we solve via *saturation*. If a cycle contains an intersection edge with a bounded upper limit, i.e., an edge $\overrightarrow{N_x N_y}$ bound to a constraint $aX + b \sqcap [c, d] \sqsubseteq Y$, where $d \neq +\infty$, then variable X can contribute to the range of variable Y at most until d. If these constraints are extracted from an e-SSA form program, then $\rho(Y)$ is upper bounded by d, as we demonstrate in Theorem 1. Showing that Y is lower bounded by c is similar and we omit the proof.

Theorem 1. *Given a constraint system S that we derive from an e-SSA form program, if $aX + b \sqcap [c, d] \sqsubseteq Y \in S$, then $\rho(Y)$ is upper bounded by d.*

Proof: To see that $\rho(Y)$ is upper bounded by d, notice that every range that Y receives from X is upper bounded by d. If the constraints are extracted from an e-SSA form

Solve constraints(G):

1. Sort G topologically
2. For each vertex $N_v \in N$ in topological order:
   (a) If the strongly connected component (SCC) that contains N_v is trivial, then *evaluate* N_v.
   (b) Otherwise, N_v is part of a cycle. Call *cycle resolution* on this SCC.

Evaluate(N_v):

1. If $S = \{L(\overrightarrow{N_u N_v}) \mid \overrightarrow{N_u N_v} \in E\}$ is the set of constraints that come from predecessors of N_v, then $\rho(V) = \bigsqcup L(\overrightarrow{N_u N_v})$.

Cycle resolution(C):

1. If N_s is the entry of C, then evaluate(N_s) and Propagate(N_s).
2. While C contains a non-saturated edge $\overrightarrow{N_x N_y}$ such that $\overrightarrow{L(N_x N_y)} = aX + b \sqcap [c, d]$, call the *Saturate* procedure on $\overrightarrow{N_x N_y}$.

Saturate($\overrightarrow{N_x N_y}$):

1. Let $\overrightarrow{L(N_x N_y)} = aX + b \sqcap [c, d]$, and let $\rho(X) = [l, u]$ be the initial valuation of X.
2. If there exists a positive path linking N_y to N_x, then let $\rho(Y) = [\max(l, c), d]$.
3. If there exists a negative path linking N_y to N_x, then let $\rho(Y) = [c, \min(u, d)]$.
4. Mark $\overrightarrow{N_x N_y}$ as saturated and Propagate(N_y).

Propagate(N_y):

1. Do a depth-first-search on the SCC that contains N_y, and let D be the acyclic graph that results from removing the back edges from the DFS tree.
2. Propagate $\rho(N_y)$ onto a topological ordering of D.
   (a) If $\overrightarrow{N_v N_z} \in D$ and $\overrightarrow{N_w N_z} \in D$, then let $\rho(Z) = \rho(V) \sqcup \rho(W)$.

Fig. 7. This algorithm finds a valuation ρ for a set of constraints S represented as a constraint graph $G = (N, E, L)$.

program, then there is no other constraint $a'X' + b' \sqcap [c', d'] \sqsubseteq Y, X' \neq X$. To see this fact, notice that intersection constraints are always produced from σ functions, as given by Figure 1, and any σ function defines only one variable. □

We will be able to achieve the upper limit d if there exists a path from N_y to N_x that increases the upper limit of X. In this case, we can iterate over this path, until assigning d to Y. The same reasoning is valid for negative paths; however, in this case we may assign the lower limit c to Y. We define negative and positive paths as follows:

Definition 2. *A cycle $V_1 V_2 \ldots V_n V_1$ is positive if the propagation of an initial range $[l, u] \sqsubseteq V_1$ through this path produces a final range $[l', u'] \sqsubseteq V_1$, such that $u' > u$. Similarly, the cycle is negative if this propagation produces a final range $[l', u'] \sqsubseteq V_1$, such that $l' < l$.*

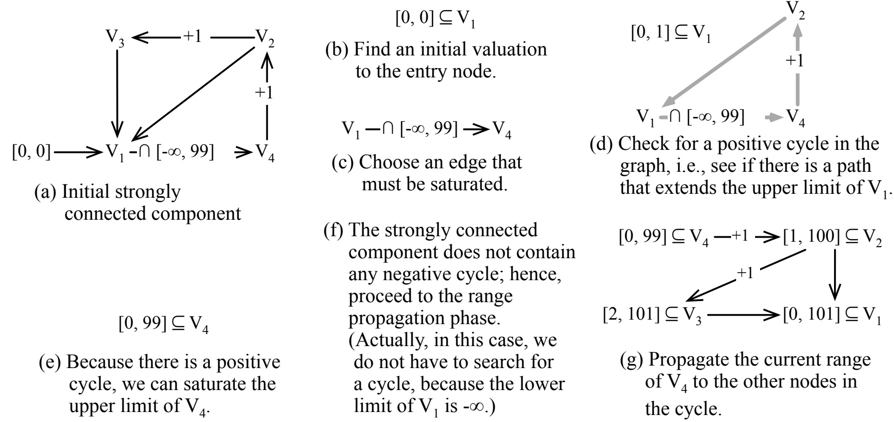

Fig. 8. The resolution of the strongly connected component formed by the set of vertices $\{V_1, V_2, V_3, V_4\}$ in Figure 6.

We find positive or negative paths in strongly connected components using the Bellman-Ford algorithm [6]. This is the same idea adopted by Gawlitza *et al* [9] to solve his constraint system. Figure 8 illustrates the application of our algorithm on the subgraph that the vertices $\{V_1, V_2, V_3, V_4\}$ induce in Figure 6. Our SCC contains two different positive cycles: $\overrightarrow{V_1 V_4 V_2 V_1}$ and $\overrightarrow{V_1 V_4 V_2 V_3 V_1}$. Any of these cycles allows us to increase the initial upper limit of V_1, the entry point of the strongly connected component, and we can use this knowledge to saturate the upper limit of V_4, which is bounded by an intersection with $[-\infty, 99]$. Because there is no negative path linking V_4 to V_1, we know that the lower limit of V_1, and thus V_4, is at most zero. We proceed by propagating the range $[0, 99]$ from V_4 to the other nodes in the strongly connected component.

Complexity Analysis The conversion of a SSA-form program into e-SSA form has worst case complexity quadratic on the size of the program's CFG. Considering assembly codes produced from structured programming languages, this size is proportional to the number of instructions in the source program. Given that almost every instruction in a SSA-form program produces a variable, we say that this conversion is $O(V^2)$, where V is the number of program variables. From Table 1 it is easy to see that we have $O(V)$ constraints per program, and that this is the size of the constraint graph. Sorting this graph topologically is $O(V)$. In order to find a positive or negative path in a strongly connected component we rely on a variation of the Bellman-Ford algorithm, as described by Gawlitza *et al.* [9], which is cubic on the number of vertices in the graph. We may have to saturate a strongly connected component once for each intersection constraint that it has, until we can find the tightest association of intervals to nodes. Thus, the complexity of this final step is $O(V^4)$. Nevertheless, very extensive experiments suggest, as we show in Figure 9, that our implementation is $O(V)$ in practice.

4 Experimental Results

Our analysis, which is intra-procedural, is implemented on top of the Low Level Virtual Machine (LLVM) [11], version 2.7. We run it on an Intel processor with a 2.4GHz clock featuring Linux Kernel 2.6.28. The compilation pipeline adopted by LLVM has two levels: a target independent and a target dependent phases. Our analysis is implemented on the former; thus, we do not benefit from hardware knowledge, such as register sizes, in order to produce constraints.

Benchmark characteristics: We have processed the 78,117 functions from 178 different benchmarks, including the LLVM test suite and SPEC CPU 2006, for a total of over 2 million lines of C code, and 4,169,818 assembly instructions, out of which 2,950,589 instructions gave us some type of constraint to process, for a grand total of 4,038,810 constraints. These instructions are integer operations, conditional tests, ϕ and σ functions in addition to the other operations in Table 1.

Runtime numbers: 99.93%, e.g., 78,065 of the functions were analyzed under 0.001s (1ms), and on the average our processing time was 0.008s (8ms). The longest processing time that we have experienced was 18.45s, for `hypre_StructMatvecCompute`, a function from `SMG2000` (The `ASCI_Purple` benchmark), which had 7,165 instructions and a constraint graph with 22,264 edges. On the average, the constraint graphs had 29.98 vertices, and 51.70 edges. The largest constraint graph that we found, having 22,380 edges and 13,812 vertices, was extracted from the function `c_decode_option` in SPEC CPU 2006's `403.gcc`. Our experiments provide strong evidence that our analysis, although having worst case cubic complexity, is linear in practice, as we show in Figure 9(Top). In this figure we have sorted the 178 benchmarks by size, and plotted them together with the analysis runtime. Multiplying the size by the constant 0.00001, we see a linear correlation between size and time.

These runtime numbers are in stark contrast to more precise analyses, that rely on abstract interpretation and the polytope model to find value-ranges. For instance, Lokuciejewski *et al.*'s loop analysis [12] took approximately 4 minutes to analyze `telecom-gsm`, one of the programs from the `MiBench` benchmark suite, whereas we evaluated the 15,866 assembly instructions of this benchmark in 0.48 seconds, obtaining a bitwidth reduction of 43%. These runtime numbers (223s) refer to our value-range analysis only. In order to compute our whole processing time, we must consider also the time to convert the source programs into three-address code, a step that we call *scalarization* (3.51s), and the time to convert the three-address code into e-SSA form (117.9s). Overall, we fully process the whole test suite in 349.7s. The charts in Figure 9 show the runtime distribution, plus absolute runtime numbers for SPEC CPU 2006.

Precision: In order to probe the precision of our analysis, we compare it to the results obtained by Stephenson *et al.* [19]. We have used the exact same benchmarks as Stephenson *et al.* did. They used small programs because, in their words, their baseline compiler was unable to handle recursive functions and more elaborate C data-types. The benchmark suite used in that work is popularly known as *bitwise*, and it has been used also by Barik *et al.* [2]. Figure 10 provides a comparison between our implementation, Stephenson's and the optimal results that Stephenson *et al.* obtained via profiling. We have not implemented Stephenson's analysis or their profiler. Instead, we report their numbers as given in the original work. On the average, the profiler showed that 60.8%

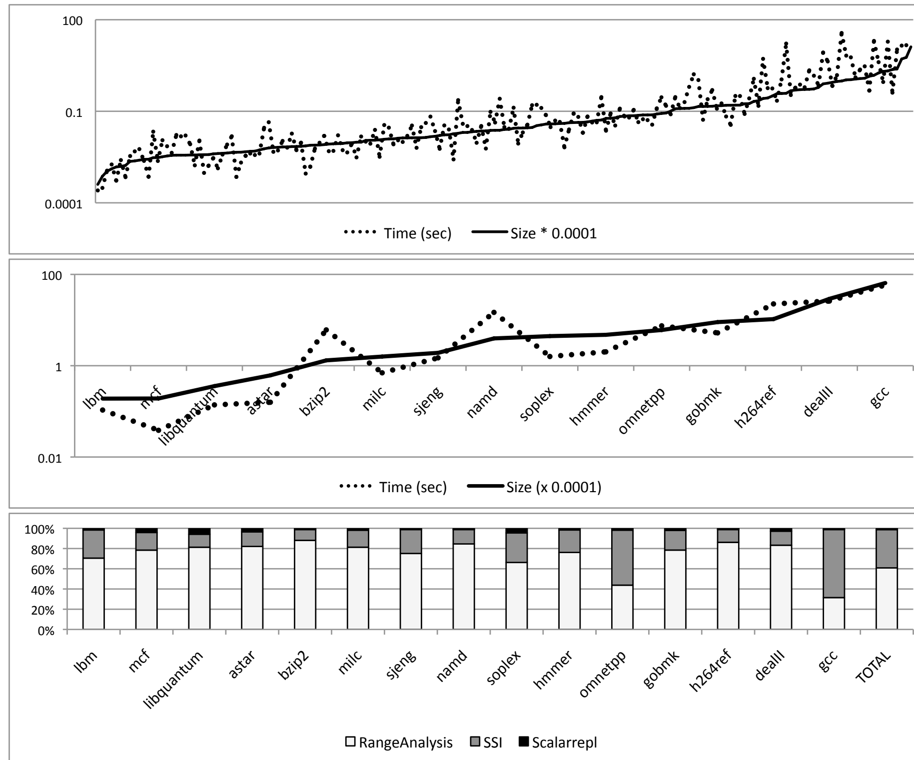

Fig. 9. (**Top**) Absolute runtime for our 178 benchmarks, ordered by size. To show a correlation between time and program size we have adjusted the size with the fitting function $f(x) = 0.0001x$. (**Middle**) Absolute runtime (and program size) for SPEC CPU 2006. (**Bottom**) Time distribution among the different processing phases for SPEC CPU 2006.

of the bits in the program variables are not necessary. Stephenson's analysis reports a 51.5% of unnecessary bits, and we report 39.4%. Our implementation is less precise than Stephenson's because the latter assumes that the source program is correct, and uses this assumption to propagate information backwards. For instance, given an array access such as `a[v]`, this analysis creates the constraint $v < size(a)$. Furthermore, given an assignment such as `char a = b + c`, Stephenson's analysis assumes the absence of an overflow, and creates the constraints $b < 256$ and $c < 256$. These assumptions are not safe in the context of C programs, because neither array accesses nor overflows are explicitly verified in this language. However, these are acceptable speculations in Stepheson's domain: the synthesis of hardware, given that the synthetic program only receives valid input data. Nevertheless, Stephenson's analysis cannot be used in the detection of buffer overflows, for instance.

Effectiveness of type specialization: One of the motivations behind this project is to convert integer variables to narrower types, so that programmers may develop algo-

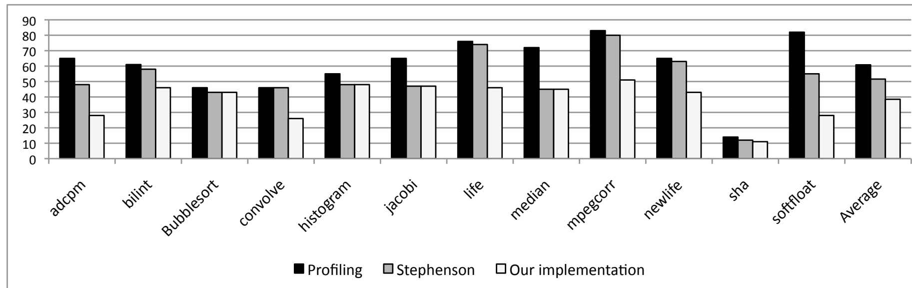

Fig. 10. The precision of our analysis compared to the optimal result of a profiler, and to Stephenson's non-conservative analysis.

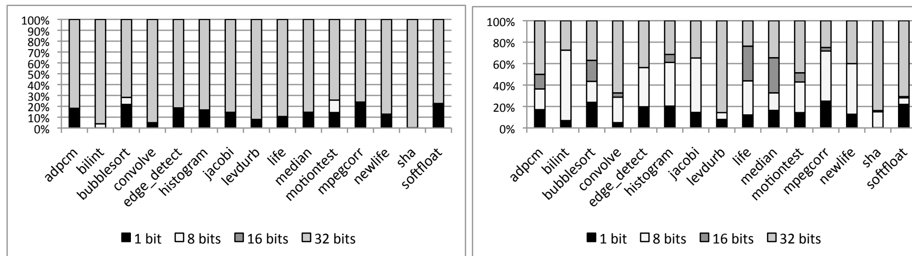

Fig. 11. Type narrowing due to value-range analysis. Distribution of integer types in the original programs (Left) and in the programs after type narrowing (Right).

rithms using the widest integer types, whereas still producing economical assembly programs. Figure 11 shows the results that we obtained for the bitwise benchmark suite. We increased the number of 1-bit variables by 1.04x, 8-bit variables by 52.3x and 16-bit variables by 26.6x. We decreased the number of 32-bit variables by 1.33x.

5 Conclusion

This paper described the first non-iterative implementation of value-range analysis on top of an industrial strength compiler. Our implementation solves the constraint system proposed by Su and Wagner to find the value-ranges of integer variables. This approach differs from previous works in the sense that it is non-iterative. Our analysis departs from Su's description in a few aspects, containing novel elements, such as the way that we handle dependency cycles between program variables using the Bellman-Ford algorithm. We have run our implementation on a vast collection of benchmarks, achieving promising results in terms of time and precision. We are currently working on an interprocedural version of this analysis, and hope to report these results in a future work.
Acknowledgement: Douglas do Couto was partially sponsored by FUMP (Fundação Universitária Mendes Pimentel), under the project *Bolsa de Apoio Socioeducacional*, and by the Google Summer of Code 2010 initiative.

References

1. Ananian, S.: The Static Single Information Form. Master's thesis, MIT (September 1999)
2. Barik, R., Grothoff, C., Gupta, R., Pandit, V., Udupa, R.: Optimal bitwise register allocation using integer linear programming. In: LCPC. Lecture Notes in Computer Science, vol. 4382, pp. 267–282. Springer (2006)
3. Bellman, R.: On a Routing Problem. Quarterly of Applied Mathematics 16, 87–90 (1958)
4. Bodik, R., Gupta, R., Sarkar, V.: ABCD: eliminating array bounds checks on demand. In: PLDI. pp. 321–333. ACM (2000)
5. Cong, J., Fan, Y., Han, G., Lin, Y., Xu, J., Zhang, Z., Cheng, X.: 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)
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. McGraw-Hill, 2nd edn. (2001)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs. In: POPL. pp. 238–252. ACM (1977)
8. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. TOPLAS 13(4), 451–490 (1991)
9. Gawlitza, T., Leroux, J., Reineke, J., Seidl, H., Sutre, G., Wilhelm, R.: Polynomial precise interval analysis revisited. Efficient Algorithms 1, 422 – 437 (2009)
10. Kong, T., Wilken, K.D.: Precise register allocation for irregular architectures. In: MICRO. pp. 297–307. IEEE (1998)
11. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO. pp. 75–88. IEEE (2004)
12. Lokuciejewski, P., Cordes, D., Falk, H., Marwedel, P.: A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In: CGO. pp. 136–146 (2009)
13. Mahlke, S., Ravindran, R., Schlansker, M., Schreiber, R., Sherwood, T.: Bitwidth cognizant architecture synthesis of custom hardware accelerators. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 20(11), 1355–1371 (Nov 2001)
14. Patterson, J.R.C.: Accurate static branch prediction by value range propagation. In: PLDI. pp. 67–78. ACM (1995)
15. Pereira, F.M.Q., Palsberg, J.: Register allocation by puzzle solving. In: PLDI. pp. 216–226. ACM (2008)
16. Scholz, B., Eckstein, E.: Register allocation for irregular architectures. In: LCTES/SCOPES. pp. 139–148. ACM (2002)
17. Simon, A.: Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities. Springer, 1th edn. (2008)
18. Souza, M.R.S., Guillon, C., Pereira, F.M.Q., da Silva Bigonha, M.A.: Dynamic elimination of overflow tests in a trace compiler. In: CC. pp. 1–20 (2011)
19. Stephenson, M., Babb, J., Amarasinghe, S.: Bidwidth analysis with application to silicon compilation. In: PLDI. pp. 108–120. ACM (2000)
20. Su, Z., Wagner, D.: A class of polynomially solvable range constraints for interval analysis without widenings. Theoretical Computeter Science 345(1), 122–138 (2005)
21. Tallam, S., Gupta, R.: Bitwidth aware global register allocation. In: POPL. pp. 85–96. ACM, New York, NY, USA (2003)
22. Wagner, D., Foster, J.S., Brewer, E.A., Aiken, A.: A first step towards automated detection of buffer overrun vulnerabilities. In: NDSS. pp. 3–17. ACM (2000)
23. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. TOPLAS 13(2) (1991)