SOFTWARE—PRACTICE AND EXPERIENCE
*Softw. Pract. Exper.* 2012; **00**:1–19
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

# Speed and Precision in Range Analysis

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

*UFMG – 6627 Antônio Carlos Av, 31.270-010, Belo Horizonte, Brazil*

SUMMARY

Range analysis estimates statically the lower and upper values that the variables may assume during a program's execution. This analysis is used to detect program vulnerabilities, and to enable compiler optimizations such as dead and redundant code elimination. This paper describes an inter-procedural range analysis of integer variables that scales up to programs with millions of assembly instructions. Contrary to previous techniques, we handle comparisons between variables without resorting to relational lattices or expensive algorithms. We gain precision from conditional tests by using Bodik's Extended Static Single Assignment intermediate representation, and we obtain partial context sensitiveness via function inlining. We have implemented our technique in LLVM and have been able to process programs totaling 2.72 million lines of C in a few seconds. During the effort to produce an industrial-quality implementation of our algorithm, we had to face a constant tension between precision and speed. In this paper we 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. Copyright © 2012 John Wiley & Sons, Ltd.

Received . . .

## 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 [1]. 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 [2] and unnecessary array bound checks [3, 4]. Furthermore, range analysis is essential not only to the bitwidth aware register allocator [5, 6], but also to more traditional allocators that handle registers of different sizes [7, 8, 9]. Additionally, range analysis has also been used to statically predict the outcome of branches [10], to detect buffer overflow vulnerabilities [11, 12], to find the trip count of loops [13] and even to synthesize hardware [14, 15, 16].

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 [17, 16, 18, 19]. 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 [10, 20, 21]; 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.* [22], who introduced an abstract interpretation

*Correspondence to: fernando@dcc.ufmg.br

framework which processes programs with hundreds of thousands of lines of code. Nevertheless, Oh *et al.* have designed a simple range analysis, which does not handle comparisons between variables, for instance. They do not discuss the precision of their implementation, but only its runtime and memory consumption. In this paper we claim to push this discussion considerably further.

In this paper we provide a complete description of a range analysis algorithm, and show extensive experimental data that justifies our engineering choices. Our first algorithmic contribution on top of previous works is a three-phases approach to handle comparisons between variables without resorting to any exponential time technique. The few publicly available implementations of range analyses that we are aware of, such as those in FLEX [†] and gcc [‡] only deal with comparisons between variables and constants. Even theoretical works, such as Su and Wagner's [19] or Gawlitza *et al.*'s [17] suffer from this limitation. This deficiency is one of the reasons explaining why none of these works has made their way into industrial-strength compilers. Two other insights allow our implementation to scale up to very large programs. We use Bodik's Extended Static Single Assignment (e-SSA) form [3] to perform path-sensitive range analysis sparsely. This program representation ensures that the interval associated with a variable is constant along its entire live range. Finally, we process the strongly connected components that underline our constraint system in topological order. It is well-known that this technique is essential to speedup constraint solving algorithms [23, Sec 6.3]; however, due to our three-phases approach, a careful propagation of information along strong components not only gives us speed, but also improves the precision of our results.

We have implemented our algorithm in the LLVM compiler [24], and have used it to process a test suite with 2.72 million lines of C code. As we show in Section 4.2, our implementation, which is publicly available, is able to analyze programs with over one million assembly instructions within fifteen seconds. And our implementation is not a straw-man: it produces very precise results. We have compared the ranges that our implementation finds with the results obtained via a dynamic profiler, which we have also implemented. As we show in Section 4.2, when analyzing well-known numeric benchmarks we are able to estimate tight ranges for almost half of all the integer variables present in these programs. Our results are similar to Stephenson *et al*'s [18], 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.* [25] and Lakhdar *et al.* [26], who rely on more costly methods.

While designing and implementing our algorithm we had to face several important engineering decisions. Many approaches that we have used to increase the precision of our results would result in runtime slowdowns. We cannot determine the optimum spot in this design space, given the vast number of possibilities that it contains, but we discuss our most important implementation choices in Section 5. Section 5.1 shows how we could improve runtime and precision substantially by processing data-flow information in the strongly connected components that underly our constraint system. Section 5.2 discuss the importance of choosing a suitable intermediate representation when implementing a sparse data-flow framework. Section 5.3 compares the intra-procedural and the inter-procedural versions of our algorithm. The role of context sensitiveness is discussed in Section 5.4. Finally, Section 5.5 discusses the different widening strategies that we have experimented with.

This work concludes a two years long effort to produce a solid and scalable implementation of range analysis. Our first endeavor to implement such an algorithm was based on Su and Wagner's constraint system, which can be solved exactly in polynomial time [19, 27]. However, although we could use their formulation to handle a subset of C-like constructs, their description of how to deal with loops was not very explicit. Thus, in order to solve loops we adopted Gawlitza *et al.*'s [17] approach. This technique uses the Bellman-Ford algorithm to detect increasing or decreasing cycles

[†]The MIT's FLEX/Harpoon compiler provides an implementation of Stephenson's algorithm [18], and is available at `http://flex.cscott.net/Harpoon/`.

[‡]Gcc's VRP pass (at `http://gcc.gnu.org/svn/gcc/trunk/gcc/tree-vrp.c`) implements a variant of Patterson's algorithm [10].



DOI: 10.1002/spe

in the constraint system, and then saturates these cycles via a simple widening operator. A detailed description of our implementation has been published by Couto and Pereira [28]. Nevertheless, the inability to handle comparisons between variables, and the cubic complexity of the Bellman-Ford method eventually led us to seek alternative solutions to range analysis. This quest reached a pinnacle in the present work.

## 2. BACKGROUND

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:

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

From the lattice $\mathcal{Z}$ we define the product lattice $\mathcal{Z}^2$, which is defined as follows:

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

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

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

Given an interval $\iota = [l, u]$, we let $\iota_\downarrow = l$, and $\iota_\uparrow = u$. We let $\mathcal{V}$ be a set of constraint variables, and $I : \mathcal{V} \mapsto \mathcal{Z}^2$ a mapping from these variables to intervals in $\mathcal{Z}^2$. Our objective is to solve a constraint system $C$, formed by constraints such as those seen in Figure 1(left). We let the $\phi$-functions be as defined by Cytron *et al.* [29]: they join different variable names into a single definition. Figure 1(right) defines a valuation function $e$ on the interval domain. Armed with these concepts, we define the range analysis problem as follows:

*Definition 3*
RANGE ANALYSIS PROBLEM
**Input:** a set $C$ of constraints ranging over a set $\mathcal{V}$ of variables.
**Output:** a mapping I such that, for any variable $V \in \mathcal{V}$, e(V) = I[V].

We will use the program in Figure 2(a) as the running example to illustrate our range analysis. Figure 2(b) shows the same program in SSA form [29], and Figure 2(c) outlines the constraints that we extract from this program. There is a clear correspondence between instructions and constraints. A possible solution to the range analysis problem, as obtained via the techniques that we will introduce in Section 4, is given in Figure 2(d). The SSA form, so common in modern compilers, leads to a very conservative solution. As we will see shortly, we can improve this solution substantially by using a more sophisticated program representation – the e-SSA form – which gives us path-sensitiveness.

## 4. OUR DESIGN OF A RANGE ANALYSIS ALGORITHM

In this section we explain the algorithm that we use to solve the range analysis problem. This algorithm involves 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 3 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.



DOI: 10.1002/spe

$$Y = [l, u] \qquad e(Y) = [l, u]$$

$$Y = \phi(X_1, X_2) \qquad \frac{I[X_1] = [l_1, u_1] \qquad I[X_2] = [l_2, u_2]}{e(Y) = [\min(l_1, l_2), \max(u_1, u_2)]}$$

$$Y = X_1 + X_2 \qquad \frac{I[X_1] = [l_1, u_1] \qquad I[X_2] = [l_2, u_2]}{e(Y) = [l_1 + l_2, u_1 + u_2]}$$

$$Y = X_1 \times X_2 \qquad \frac{I[X_1] = [l_1, u_1] \qquad I[X_2] = [l_2, u_2] \qquad L = \{l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2\}}{e(Y) = [\min(L), \max(L)]}$$

$$Y = aX + b \qquad \frac{I[X] = [l, u] \qquad k_l = al + b \qquad k_u = au + b}{e(Y) = [\min(k_l, k_u), \max(k_l, k_u)]}$$

$$Y = X \sqcap [l', u'] \qquad \frac{I[X] = [l, u]}{e(Y) \leftarrow [\max(l, l'), \min(u, u')]}$$

Figure 1. A suite of constraints that produce an instance of the range analysis problem.

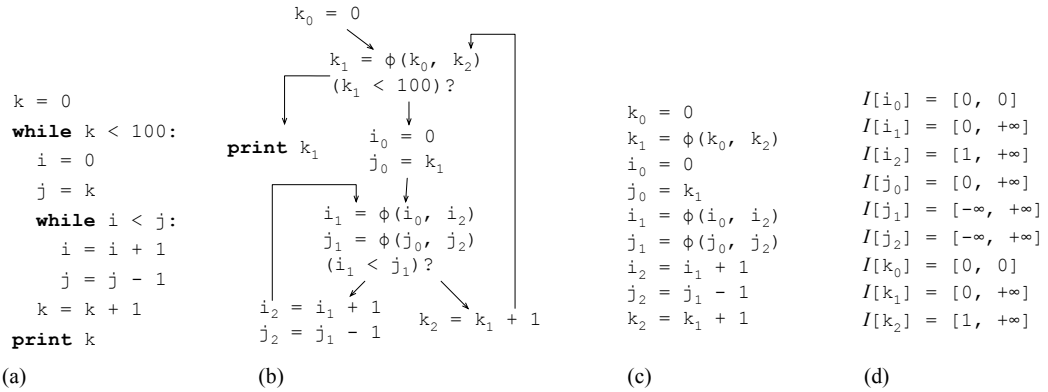



Figure 2. (a) Example program. (b) Control Flow Graph in SSA form. (c) Constraints that we extract from the program. (d) Possible solution to the range analysis problem.

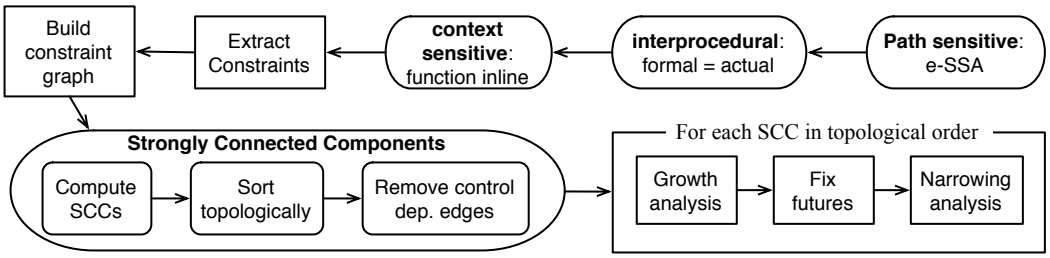



Figure 3. Our implementation of range analysis. Rounded boxes are optional steps.

**Choosing a Program Representation.** The solution to the range analysis problem in Figure 2 is imprecise because we did not take conditional tests into considerations. Branches give us information about the ranges that some variables assume, but only at *specific* program points. For



DOI: 10.1002/spe

instance, given a test such as $(k_1 < 100)?$ in Figure 2(b), we know that $I[k_1] \sqsubseteq [-\infty, 99]$ whenever the condition is true. In order to encode this information, we might split the live range of $k_1$ right after the branching point; thus, creating two new variables, one at the path where the condition is true, and another where it is false. There is a program representation, introduced by Bodik *et al.* [3], that performs this live range splitting: the *Extended Static Single Assignment* form, or e-SSA for short.

Given that the exact rules to convert a program to e-SSA form have never been explicitly stated in the literature, we describe our rules as follows. Let $(v < c)?$ be a conditional test, and let $l_t$ and $l_f$ be labels in the program, such that $l_t$ is the target of the test if the condition is true, and $l_f$ is the target when the condition is false. We split the live range of $v$ at any of these points if at least one of two conditions is true: (i) $l_f$ or $l_t$ dominate any use of $v$; (ii) there exist a use of $v$ at the dominance frontier of $l_f$ or $l_t$. For the notions of dominance and dominance-frontier, see Aho *et al.* [30, p.656]. To split the live range of $v$ at $l_f$ we insert at this program point a copy $v_f = v \sqcap [c, +\infty]$, where $v_f$ is a fresh name. We then rename every use of $v$ that is dominated by $l_f$ to $v_f$. Dually, if we must split at $l_t$, then we create at this point a copy $v_t = v \sqcap [-\infty, c-1]$, and rename variables accordingly. If the conditional uses two variables, e.g., $(v_1 < v_2)?$, then we create intersections bound to *futures*. We insert, at $l_f$, $v_{1f} = v_1 \sqcap [\mathbf{ft}(v_2), +\infty]$, and $v_{2f} = v_2 \sqcap [-\infty, \mathbf{ft}(v_1)]$. Similarly, at $l_t$ we insert $v_{1v} = v_1 \sqcap [-\infty, \mathbf{ft}(v_2) - 1]$ and $v_{2f} = v_2 \sqcap [\mathbf{ft}(v_1) + 1, +\infty]$. Notice that a variable $v$ can never be associated with a future bound to itself, e.g., $\mathbf{ft}(v)$. This invariant holds because whenever the e-SSA conversion associates a variable $u$ with $\mathbf{ft}(v)$, then $u$ is a fresh name created to split the live range of $v$, given that $v$ was used in a conditional.

We use the notation $\mathbf{ft}(v)$ to denote the *future* bounds of a variable. As we will show in Section 4.1, once the growth pattern of $v$ is known, we can replace $\mathbf{ft}(v)$ by an actual value. After splitting the live ranges according to the rules stated above, we might have to insert $\phi$-functions into the transformed program to re-convert it to SSA form. This last step avoids that two different names given to the same original variable be simultaneously *alive* at the program code. A variable $v$ is alive at a program point $p$ if the program's control flow graph contains a path from $p$ to a site where $v$ is used, that does not go across any re-definition of $v$. Figure 4(a) shows our running example changed into standard e-SSA form. We have not created variable names for $i_f$ and $j_f$, because neither $i_1$ nor $i_j$ are dominated by the target of the conditional's else case. In this example, new $\phi$-functions are not necessary: new variable names are not alive together with the original variables. The part (b) of this figure shows the solution that we get to this new program. The e-SSA form allows us to bind interval information directly to the live ranges of variables; thus, giving us the opportunity to solve range analysis sparsely. More traditional approaches, which we call *dense analyses*, bind interval information to pairs formed by variables and program points.

**Extracting Constraints.** Our implementation handles 18 different assembly instructions. The constraints in Figure 1 show only a few examples. Instructions that we did not show include, for instance, the multiplicative operators `div` and `modulus`, the bitwise operators `and`, `or`, `xor` and `neg`, the different types of shifts, and the logical operators `andalso`, `orelse` and `not`. Most of these instructions are sign-agnostic; that is, given that numbers are internally represented in 2's complement, the same implementation of a constraint handles positive and negative numbers. However, there are instructions that require different constraints, depending on the input being signed or not. Examples include `modulus` and `div`. We also handle different kinds of type conversion, e.g., converting 8-bit characters to 32-bit integers and vice-versa. In addition to constraints that represent actual assembly instructions, we have constraints to represent $\phi$-functions, and intersections, as seen in Figure 1. The growth analysis that we will introduce in Section 4.1 require monotonic transfer functions. Many assembly operations, such as modulus or division, do not afford us this monotonicity. However, these non-monotonic instructions have conservative approximations [31].

**The Constraint Graph.** The main data structure that we use to solve the range analysis problem is a variation of Ferrante *et al.*'s *program dependence graph* [32]. For each constraint variable $V$ we create a variable node $N_v$. For each constraint $C$ we create a constraint node $N_c$. We add an edge from $N_v$ to $N_c$ if the name $V$ is used in $C$. We add an edge from $N_c$ to $N_v$ if the constraint $C$ defines



DOI: 10.1002/spe

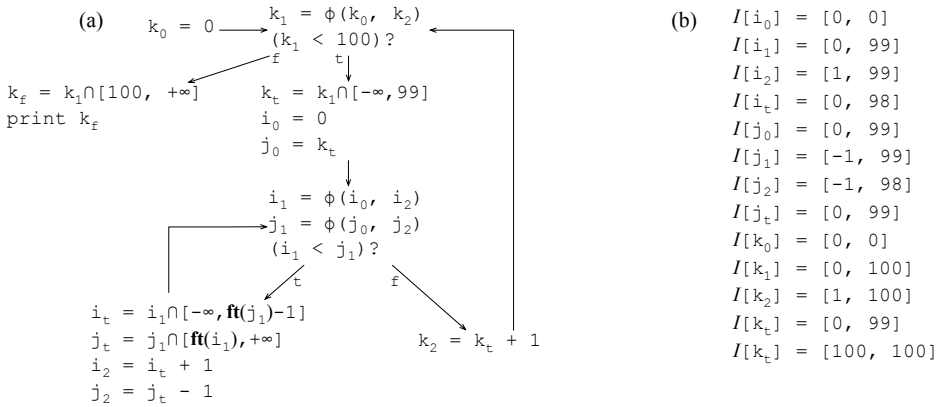



Figure 4. (a) The control flow graph from Figure 2(b) converted to standard e-SSA form. (b) A solution to the range analysis problem

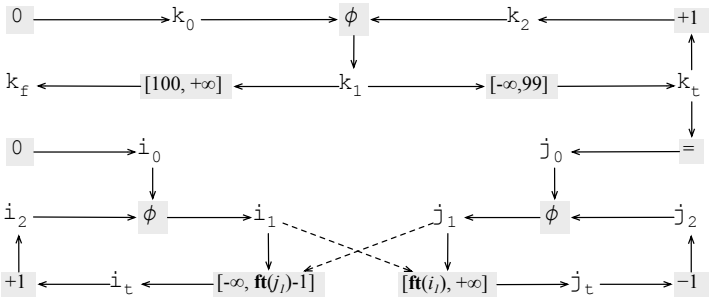



Figure 5. The dependence graph that we build to the program in Figure 4.

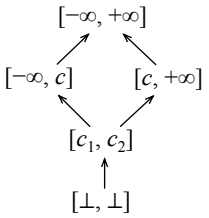
the name $V$. Figure 5 shows the dependence graph that we build for the e-SSA form program given in Figure 4(a). If $V$ is used by $C$ as the input of a future, then the edge from $N_v$ to $N_c$ represents what Ferrante *et al.* call a *control dependence* [32, p.323]. We use dashed lines to represent these edges. All the other edges denote *data dependences* [32, p.322].

**Strongly Connected Components.** To solve range analysis we find all the strongly connected components (SCCs) of the dependence graph and collapse them into single nodes, obtaining a directed acyclic graph. We then sort the resulting DAG topologically, and apply the analyses from Section 4.1 on every SCC in topological order. Once we solve the range analysis problem for a SCC, we propagate the intervals that we found to the variable nodes at the *frontier* of this SCC. A variable node $N_v$ is said to be in the frontier of a strongly connected component $S$ if: (i) $N_v \notin S$; and (ii) there exists a variable node $N'_v \in S$, and a constraint node $N_c$, such that $N_v \leftarrow N_c$, and $N_c \leftarrow N'_v$. This propagation ensures that when analyzing a strongly connected component $S$ any influence that $S$ might suffer from nodes outside it has already been considered.

When finding strongly connected components, we take control dependence edges into consideration. For instance, in Figure 5 the nodes that correspond to the variables $i_1$, $i_2$, $i_t$, $j_1$, $j_2$ and $j_t$ form a single component. The dashed edges, which represent control dependences, keep all these variables connected. In this way, we ensure that, upon stumbling upon an interval associated with future bounds, e.g., $\mathbf{ft}(v)$, either variable $v$ has been solved in a previous component, or it belongs to the current component. In the latter case, as we will see soon, we can still take $v$'s interval into consideration. This flexibility is possible because we first discover how each variable in a strong



DOI: 10.1002/spe




$$\frac{I[Y] = [\bot, \bot]}{I[Y] \leftarrow e(Y)} \qquad \frac{e(Y)_\downarrow < I[Y]_\downarrow \qquad e(Y)_\uparrow > I[Y]_\uparrow}{I[Y] \leftarrow [-\infty, +\infty]}$$

$$\frac{e(Y)_\downarrow < I[Y]_\downarrow}{I[Y] \leftarrow [-\infty, I[Y]_\uparrow]} \qquad \frac{e(Y)_\uparrow > I[Y]_\uparrow}{I[Y] \leftarrow [I[Y]_\downarrow, +\infty]}$$

Figure 6. (Left) The lattice of the growth analysis. (Right) Cousot and Cousot's widening operator. We evaluate the rules from left-to-right, top-to-bottom, and stop upon finding a pattern matching. Again: given an interval $\iota = [l, u]$, we let $\iota_\downarrow = l$, and $\iota_\uparrow = u$

$$\frac{Y = X \sqcap [l, \mathbf{ft}(V) + c] \qquad I[V]_\uparrow = u}{Y = X \sqcap [l, u + c]} \quad u, c \in \mathbb{Z} \cup \{-\infty, +\infty\}$$

$$\frac{Y = X \sqcap [\mathbf{ft}(V) + c, u] \qquad I[V]_\downarrow = l}{Y = X \sqcap [l + c, u]} \quad l, c \in \mathbb{Z} \cup \{-\infty, +\infty\}$$

Figure 7. Rules to replace futures by actual bounds.

component grows, before resolving future bounds. As we show in Section 4.2, most of the strong components in actual programs are singletons. Furthermore, the composite components tend to be small. These two facts ensure that the more costly parts of our algorithm only have to handle small inputs.

### *4.1. Finding Ranges in Strongly Connected Components*

Given a strongly connected component of the dependence graph with $N$ nodes, we solve the range analysis problem in three-steps:

1. Run growth analysis: $O(N)$.
2. Fix intersections: $O(N)$.
3. Apply the narrowing operator: $O(N^2)$.

However, before we start, we remove control dependence edges from the strongly connected component, as they have no semantics to our transfer functions.

**Growth Analysis.** The first step of our algorithm consists in determining how the interval bound to each variable grows. The possible behaviors of an interval are: (i) does not change; (ii) grows towards $+\infty$; (iii) grows towards $-\infty$; and (iv) grows in both directions. We ensure termination of this phase via a *widening operator*. We have experimented with four different widening strategies, which we discuss in Section 5.5. One of these strategies is based on Cousot and Cousot's widening operator [1, p.247]. The lattice of abstract states, plus a constraint system representing this operator is given in Figure 6. Because the lattice has height three, the intervals bound to each variable can change at most three times.

**Fixing futures.** The ranges found by the growth analysis tells us which variables have fixed bounds, independent on the intersections in the constraint system. Thus, we can use actual limits to replace intersections bounded by futures. Figure 7 shows the rules to perform these substitutions. In order to correctly replace a future $\mathbf{ft}(v)$ that limits a variable $v'$, we need to have already applied the growth analysis onto $v$. Had we considered only data dependence edges, then it would be possible that $N'_v$ be analyzed before $N_v$. However, because of control dependence edges, this case cannot happen. The control dependence edges ensure that any topological ordering of the constraint graph either places




DOI: 10.1002/spe

$$\frac{I[Y]_\downarrow = -\infty \qquad e(Y)_\downarrow > -\infty}{I[Y] \leftarrow [e(Y)_\downarrow, I[Y]_\uparrow]} \qquad \frac{I[Y]_\downarrow > e(Y)_\downarrow}{I[Y] \leftarrow [e(Y)_\downarrow, I[Y]_\uparrow]}$$

$$\frac{I[Y]_\uparrow = +\infty \qquad e(Y)_\uparrow < +\infty}{I[Y] \leftarrow [I[Y]_\downarrow, e(Y)_\uparrow]} \qquad \frac{I[Y]_\uparrow < e(Y)_\uparrow}{I[Y] \leftarrow [I[Y]_\downarrow, e(Y)_\uparrow]}$$

Figure 8. Cousot and Cousot's narrowing operator.

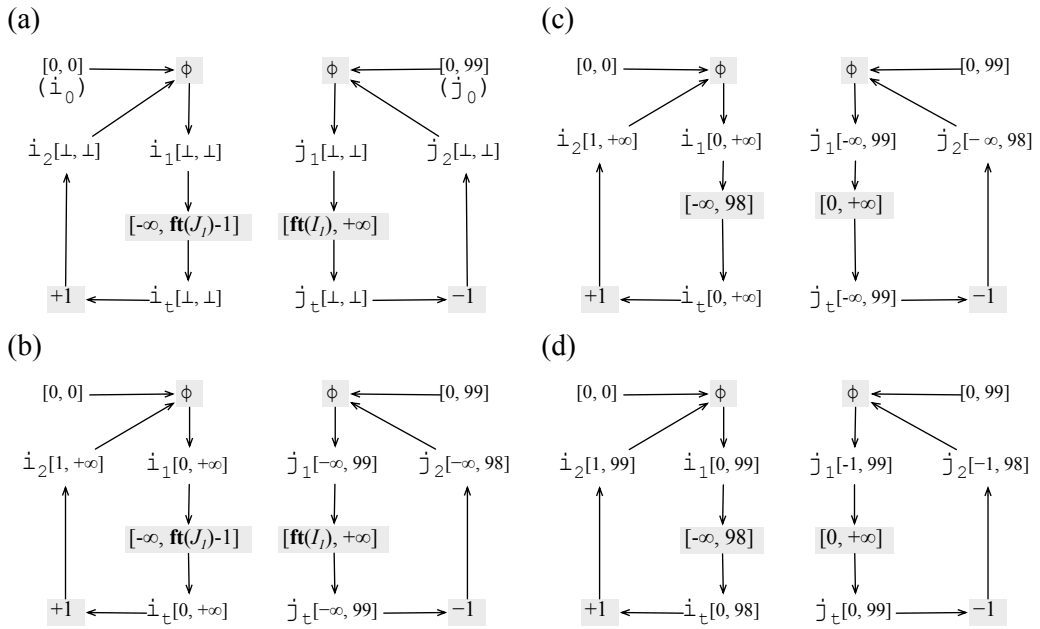



Figure 9. Four snapshots of the last SCC of Figure 4. (a) After removing control dependence edges. (b) After running the growth analysis. (c) After fixing the intersections bound to futures. (d) After running the narrowing analysis.

$N_v$ before $N_{v'}$, or places these nodes in the same strongly connected component. For instance, in Figure 5, variables $j_1$ and $i_t$ are in the same SCC only because of the control dependence edges.

**Narrowing Analysis.** The growth analysis associates very conservative bounds to each variable. Thus, the last step of our algorithm consists in narrowing these intervals. We accomplish this step via Cousot and Cousot's classic narrowing operator [1, p.248], which we show in Figure 8.

**Example:** Continuing with our example, Figure 9 shows the application of our algorithm on the last strong component of Figure 5. We are not guaranteed to find the least fixed point of a constraint system. However, in this example we did it. Figure 4(b) shows our final solution for this example. This solution is very precise, in the sense that it is the least fixed point of the constraint system given in Figure 1. However, the solution is still an over approximation of the dynamic behavior of the program in Figure 2(a). For instance, we have found that variable $i$ could reach the upper value of 99. In any actual run of the program, $i$ could be at most 50. Analyses on relational lattices, such as the polyhedron [33] or the Octagon [34] domains, can infer such tighter bounds, as shown by Lakhdar-Chaouch *et al.* [26]. However, analyses on these higher dimensional domains are much more computationally expensive than analyses on the interval domain [22].



DOI: 10.1002/spe

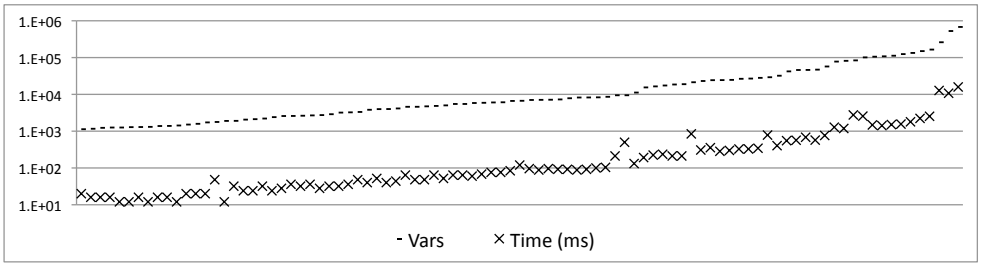



Figure 10. Correlation between program size (number of var nodes in constraint graphs after inlining) and analysis runtime (ms). Coefficient of determination = 0.967.

We emphasize that finding this tight solution was only possible because of the topological ordering of the constraint graph in Figure 5. Upon meeting the constraint graph's last SCC, shown in Figure 9, we had already determined that the interval $[0, 0]$ is bound to $i_0$ and that the interval $[0, 99]$ is bound to $j_0$, as we show in Figure 9(a). Had we applied the widening operator onto the whole graph, then we would have found out that variable $j_1$ is bound to $[-\infty, +\infty]$. This imprecision happens because, on one hand $j_1$'s interval is influenced by $k_t$'s, which is upper bounded by $+\infty$. On the other hand $j_1$ is part of a decreasing cycle of dependences formed by variables $j_t$ and $j_2$ in addition to itself. Therefore, if we had applied the widening phase over the entire program followed by a global narrowing phase, then we would not be able to recover some of widening's precision loss. However, because in this example we only analyze $j$'s SCC after we have analyzed $k$'s, $k$ only contribute the constant range $[0, 99]$ to $j_0$.

### *4.2. Range Analysis Showdown*

The objective of this section is to show, via experimental numbers, that our implementation of range analysis is fast, economic and effective. We have used it to analyze a test suite with 2.72 million lines of C code, which includes, in addition to all the benchmarks distributed with LLVM, the programs in the SPEC CPU 2006 collection.

**Time and Memory Complexity.** Figure 10 provides a visual comparison between the time to run our algorithm and the size of the input programs. We show data for the 100 largest benchmarks in our test suite, in number of variable nodes in the constraint graph. We perform function inlining before running our analysis. Each point in the X line corresponds to a benchmark. We analyze the smallest benchmark in this set, `Prolangs-C/deriv2`, which has 1,131 variable nodes in the constraint graph, in 20ms. We take 15.91 sec to analyze our largest benchmark, `403.gcc`, which, after function inlining, has 1,266,273 assembly instructions, and a constraint graph with 679,652 variable nodes. For this data set, the coefficient of determination ($R^2$) is 0.967, which provides very strong evidence about the linear asymptotic complexity of our implementation.

The experiments also reveal that the memory consumption of our implementation is linear with the program size. Figure 11 plots these two quantities together. The linear correlation, in this case, is even stronger than that found in Figure 10, which compares runtime and program size: the coefficient of determination is 0.9947. The figure only shows our 100 largest benchmarks. Again, SPEC `403.gcc` is the heaviest benchmark, requiring 265,588KB to run. Memory includes stack, heap and the executable program code.

**Precision.** Our implementation of range analysis is remarkably precise, considering its runtime. Lakhdar *et al.*'s relational analysis [26], for instance, takes about 25 minutes to go over a program with almost 900 basic blocks. We analyze programs of similar size in less than one second. We do not claim our approach is as precise as such algorithms, even though we are able to find exact bounds to 4/5 of the examples presented in [26]. On the contrary, this paper presents a compromise between precision and speed that scales to very large programs. Nevertheless, our results are far from being trivial. We have implemented a dynamic profiler that measures, for each variable, its



DOI: 10.1002/spe

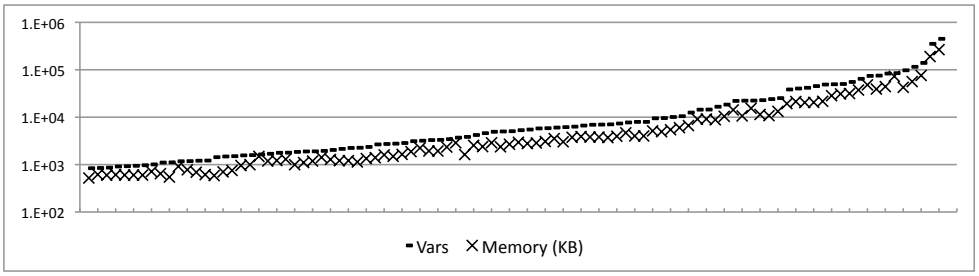



Figure 11. Comparison between program size (number of var nodes in constraint graphs) and memory consumption (KB). Coefficient of determination = 0.9947.

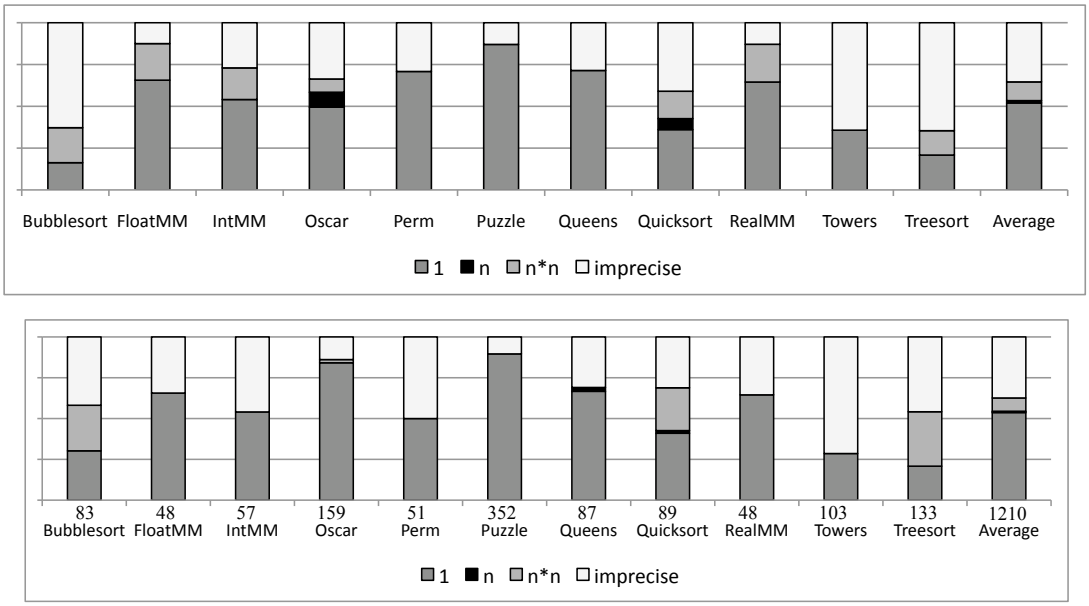



Figure 12. (Upper) Comparison between static range analysis and dynamic profiler for upper bounds. (Lower) Comparison between static range analysis and dynamic profiler for lower bounds. The numbers above the benchmark names give the number of variables in each program.

upper and lower limits, given an execution of the target program. Figure 12 compares our results with those measured dynamically for the Stanford benchmark suite, which is publicly available [§].

We have classified the bounds estimated by the static analysis into four categories. The first category, which we call 1, contains those bounds that are tight: during the execution of the program, the variable has been assigned an upper, or lower limit, that equals the limit inferred statically. The second category, which we call $n$, contains the bounds that are within twice the value inferred statically. For instance, if the range analysis estimates that a variable $v$ is in the range $[0, 100]$, and during the execution the dynamic profiler finds that its maximum value is $51$, then $v$ falls into this category. The third category, $n^2$, contains variables whose actual value is within a quadratic factor from the estimated value. In our example, $v$'s upper bound would have to be at most $10$ for it to be in this category. Finally, the fourth category contains variables whose estimated value lays outside a quadratic factor of the actual value. We call this category *imprecise*, and it contains mostly the limits that our static analysis has marked as either $+\infty$ or $-\infty$.

[§] `http://classes.engineering.wustl.edu/cse465/docs/BCCExamples/stanford.c`



DOI: 10.1002/spe

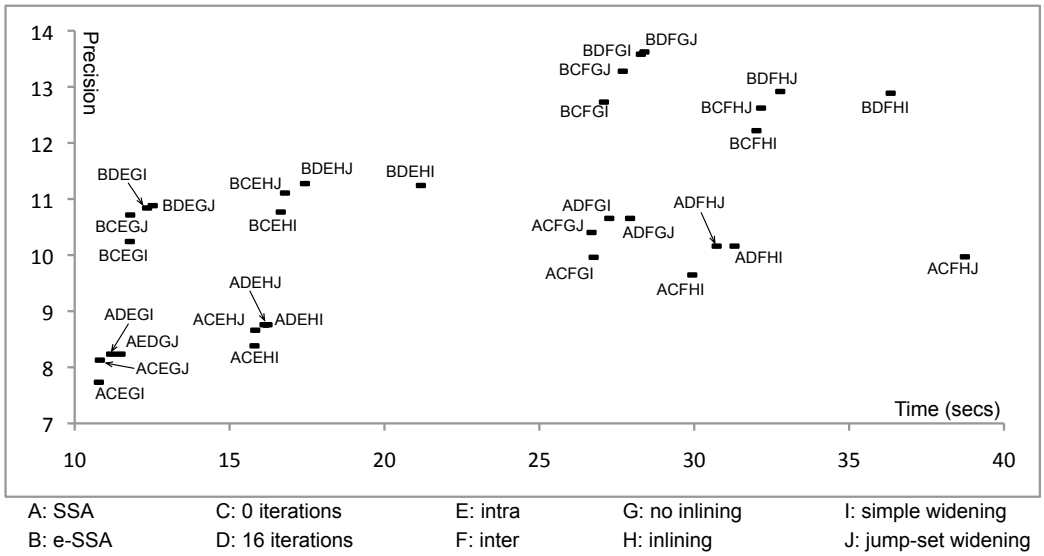



Figure 13. Design space exploration: precision (percentage of bitwidth reduction) versus speed (secs) for different configurations of our algorithm analyzing the SPEC CPU 2006 integer benchmarks.

As we see in Figure 12, 54.11% of the lower limits that we have estimated statically are exact. Similarly, 51.99% of our upper bounds are also tight. The figure also shows that, on average, 37.39% of our lower limits are imprecise, and 35.40% of our upper limits are imprecise. This result is on pair with those obtained by more costly analysis, such as Stephenson *et al.*'s [18]. However, whereas that approach have not been used with programs larger than the Stanford benchmark suite, we, as shown before, have been able to deal with remarkably larger programs.

## 5. DESIGN SPACE

As we see from a cursory glance at Figure 3, our range analysis algorithm has many optional modules. These modules give the user the chance to choose between more precise results, or a faster analysis. Given the number of options, the design space of a range analysis algorithm is vast. In this section we try to cover some of the most important tradeoffs. Figure 13 plots, for the integer programs in the SPEC CPU 2006 benchmark suite, precision versus speed for different configurations of our implementation. Our initial goal when developing this analysis was to support a bitwidth-aware register allocator. Thus, we measure precision by the average number of bits that our analysis allows us to save per program variable. It is very important to notice that we do not consider constants in our statistics of precision. In other words, we only measure bitwidth reduction in variables that a constant propagation step could not remove.

### *5.1. Strongly Connected Components*

The greatest source of improvement in our implementation is the use of strongly connected components. In order to propagate ranges across the constraint graph, we fragment it into strongly connected components, collapse each of these components into single nodes, and sort the resulting directed acyclic graph topologically. We then solve the range analysis problem for each component individually. Once we have solved a component, we propagate its ranges to the next components, and repeat the process until we walk over the entire constraint graph. It is well-known that this technique is essential to speedup constraint solving algorithms [23, Sec 6.3]. In our case, the results are dramatic, mostly in terms of speed, but also in terms of precision. Figure 14 shows the speedup



*Prepared using* speauth.cls
DOI: 10.1002/spe

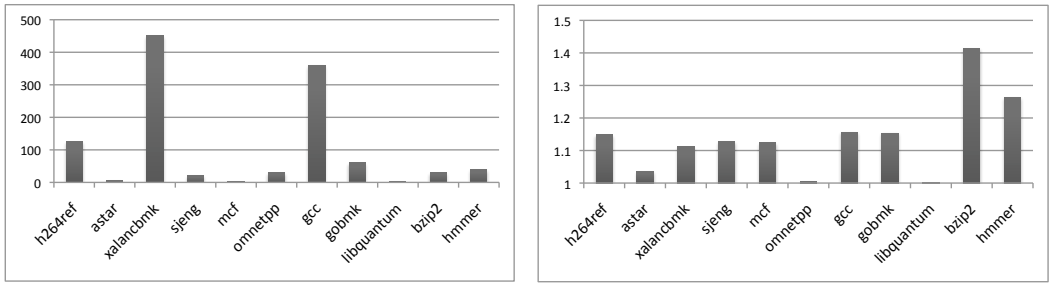



Figure 14. (Left) Time to run our analysis without building strong components divided by time to run the analysis on strongly connected components. (Right) Precision, in bitwidth reduction, that we obtain with strong components, divided by the precision that we obtain without them.

that we gain by using strong components. We show results for the integer programs in the SPEC CPU 2006 benchmark suite. In some cases, as in `xalancbmk` the analysis on strong components is 450x faster.

The strong components improve the precision of our growth analysis. According to Figure 14, in some cases, as in `bzip2`, strong components increase our precision by 40%. The gains in precision happen because, by completely resolving a component, we are able to propagate constant intervals to the next components, instead of propagating intervals that can grow in both directions. An an example, in Figure 9 we pass the range $[0, 99]$ from variable $k$ to the component that contains variable $j$. Had we run the analysis in the entire constraint graph, by the time we applied the growth analysis on $j$ we would still find $k$ bound to $0, +\infty$.

### 5.2. *The choice of a program representation*

If strong components account for the largest gains in speed, the choice of a suitable program representation is responsible for the largest gains in precision. However, here we no longer have a win-win condition: a more expressive program representation decreases our speed, because it increases the size of the target program. We have tried our analysis in two different program representations: the Static Single Assignment (SSA) Form [29], and the Extended Static Single Assignment (e-SSA) form [3]. The SSA form gives us a faster, albeit more imprecise, analysis. Any program in e-SSA form has also the SSA core property: any variable name has at most one definition site. The contrary is not true: SSA form programs do not have the core e-SSA property: any use site of a variable that appears in a conditional test post-dominates its definition. The program in Figure 2(b) is in e-SSA form. The live ranges of variables $i_1$ and $j_1$ have been split right after the conditional test via the assertions that creates variables $i_t$ and $j_t$. The e-SSA format serves well analyses that extract information from definition sites and conditional tests, and propagate this information forwardly. Examples include, in addition to range analysis, tainted flow analysis [35] and array bounds checks elimination [3].

Figure 15 compares these two program representations in terms of runtime. As we see in Figure 15(Left), the e-SSA form slows down our analysis. In some cases, as in `xalancbmk`, this slowdown increases execution time by 71%. Runtime increases because the e-SSA form programs are larger than the SSA form programs, as we show in Figure 15(Right). The increase in size is due to the extra instructions used to partition the live ranges of variables after conditionals. However, this growth is small: in none of the integer programs in SPEC CPU 2006 we verified an increase in code size of more than 9%. But, if the e-SSA form slowdowns the analysis runtime, its gains in precision are remarkable, as seen in Figure 16. These gains happen because the e-SSA format lets the analysis to use the results of conditional tests to narrow the ranges of variables.



DOI: 10.1002/spe

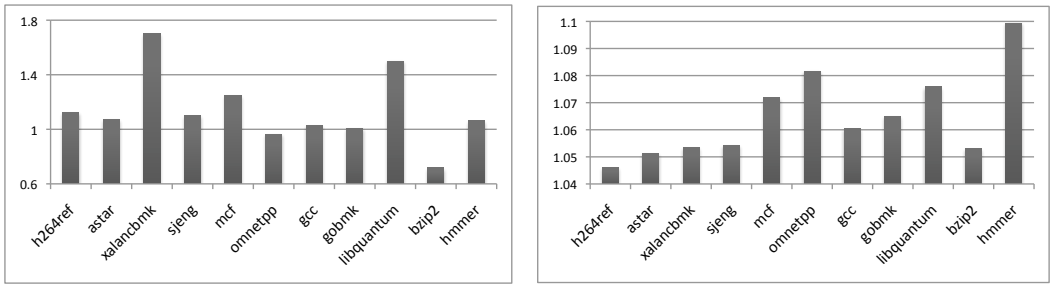



Figure 15. (Left) Bars give the time to run analysis on e-SSA form programs divided by the time to run analysis on SSA form programs. (Right) Bars give the size of the e-SSA form program, in number of assembly instructions, divided by the size of the SSA form program.

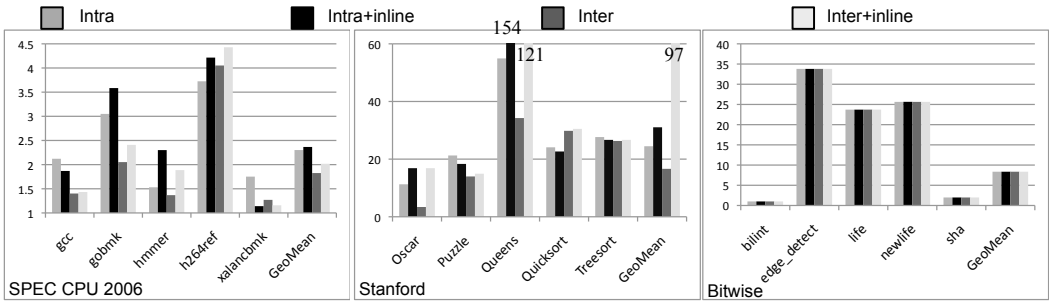



Figure 16. The impact of the e-SSA transformation on precision for three different benchmark suites. Bars give the ratio of precision (in bitwidth reduction), obtained with e-SSA form conversion divided by precision without e-SSA form conversion.

### *5.3. Intra versus Inter-procedural Analysis*

A naive implementation of range analysis would be intra-procedural; that is, would solve the range analysis problem once per each function. However, we can gain in precision by performing it inter-procedurally. An inter-procedural implementation allows the results found for a function $f$ to flow into other functions that $f$ calls. Figure 17 illustrates the inter-procedural analysis for the program seen in Figure 2(a). The trivial way to produce an inter-procedural implementation is to insert into the constraint system assignments from the actual parameter names to the formal parameter names. In our example of Figure 17, our constraint graph contains a flow of information from 0, the actual parameter, to $k_0$, the formal parameter of function `foo`.

Figure 19 compares the precision of the intra and inter-procedural analyses for the five largest programs in three different categories of benchmarks: SPEC CPU 2006, the Stanford Suite [¶] and Bitwise [18]. Our results for the SPEC programs were disappointingly: on the average for the five largest programs, the intra-procedural version of our analysis saves 5.23% of bits per variable. The inter-procedural version increases this number to 8.89%. A manual inspection of the SPEC programs reveals that this result is expected: these programs manipulate files, and their source codes do no provide enough explicit constants to power our analysis up. However, with numerical benchmarks we fare much better. On the average our inter-procedural algorithm reduces the bitwidth of the Stanford benchmarks by 36.24%. For Bitwise we obtain a bitwidth reduction of 12.27%. However, this average is lowered by two outliers: `edge_detect` and `sha`, which have been purposely

[¶] `http://classes.engineering.wustl.edu/cse465/docs/BCCExamples/stanford.c`



DOI: 10.1002/spe

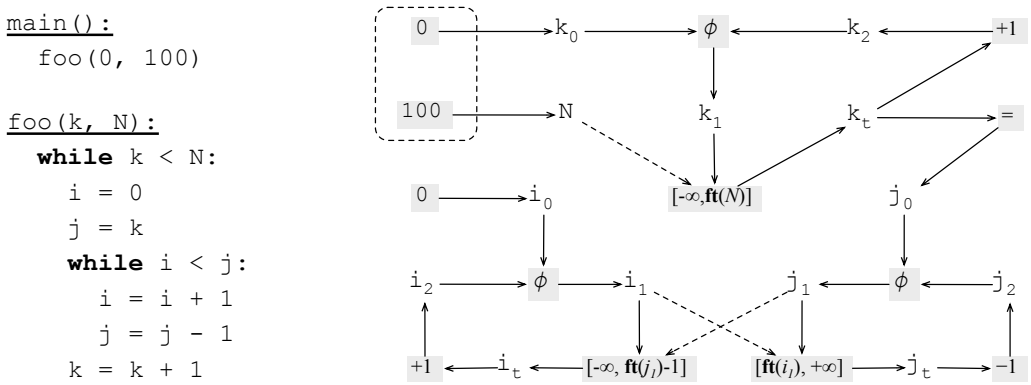



Figure 17. Example where an intra-procedural implementation would lead to imprecise results.

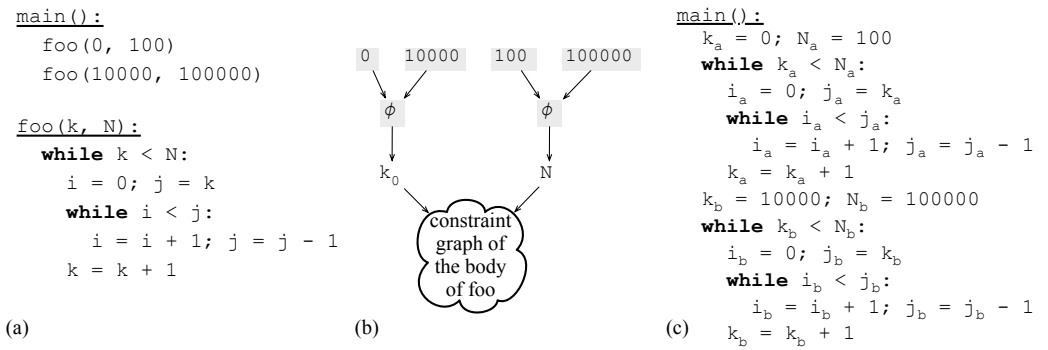



Figure 18. Example where a context-sensitive implementation improves the results of range analysis.

engineered to be resilient against range analyses [18]. The bitwise benchmarks were implemented by Stephenson *et al.* [18] to validate their intra-procedural bitwidth analysis. Our results are on par with those found by the original authors. The bitwise programs contain only the `main` function; thus, different versions of our algorithm find the same results when applied onto these programs.

### *5.4. Achieving Partial Context-Sensitiveness via Function Inlining*

Another way to increase the precision of range analysis is via a context-sensitive implementation. Context-sensitiveness allows us to distinguish different calling sites of the same function. Figure 18 shows why the ability to make this distinction is important for precision. In Figure 18(a) we have two different calls of function `foo`. If we apply the trivial intra-procedural approach of Section 5.3, then we get the graph shown in Figure 18(b). In other words, if a function is called more than once, then its formal parameters will receive information from many actual parameters. We use $\phi$-functions to bind this information together into a single flow. However, in this case the multiple assignment of values to parameters makes the ranges of these parameters very large, whereas in reality they are not. A way to circumvent this source of imprecision is via function inlining, as we show in Figure 18(c). The results that we can derive for the transformed program are more precise, as each input parameter is assigned a single value.

Figure 19 also shows how function inlining modifies the precision of our results. It is difficult to find an adequate way to compare the precision of our analysis with, and without inlining. This



DOI: 10.1002/spe

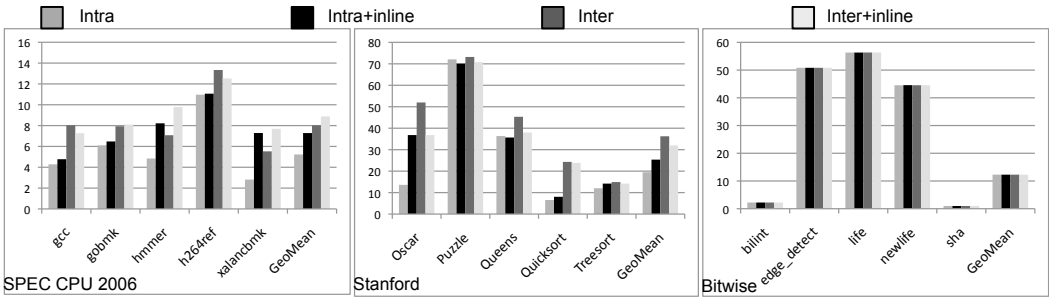



Figure 19. The impact of whole program analysis on precision. Each bar shows precision in %bitwidth reduction.

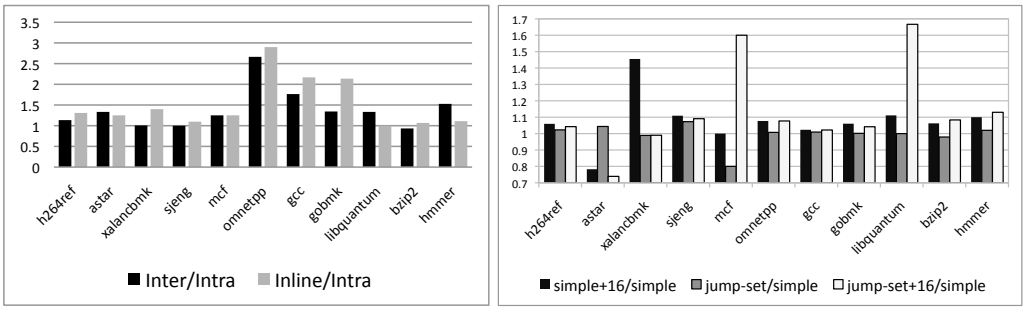



Figure 20. (Left) Runtime comparison between intra, inter and inter+inline versions of our algorithm. (Right) Runtime comparison between different widening operators. The bars are normalized to the time to run the intra-procedural analysis.

difficulty stems from the fact that this transformation tends to change the target program too much. In absolute numbers, we always reduce the bitwidth of more variables after function inlining. However, proportionally function inlining leads to a smaller percentage of bitwidth reduction for many benchmarks. In the Stanford Collection, for instance, where most of the functions are called in only one location, inlining leads to worst precision results. On the other hand, for the SPEC programs, inlining, even in terms of percentage of reduction, tends to increase our measure of precision.

**Intra vs Inter-procedural runtimes.** Figure 20(Right) compares three different execution modes. Bars are normalized to the time to run the intra-procedural analysis without inlining. On the average, the intra-procedural mode is 28.92% faster than the inter-procedural mode. If we perform function inlining, then this difference is 45.87%. These numbers are close because our runtime is bound to the size of the strong components. Although function inlining can increase the number of strongly connected components in the constraint graph, it cannot increase the size of the largest component, when compared to the simple inter-procedural analysis of Section 5.3.

### *5.5. Choosing a Widening Strategy*

We have implemented the widening operator used in the growth analysis in two different ways. The first way, which we call *simple*, is based on Cousot and Cousot's original widening operator [1], and we have shown it in Figure 6(Right). The second widening strategy, which we call *jump-set widening* consists in using the constants that appear in the program text, in sorted order, as the next limits of each interval after widening is applied. This operator is common in implementations of range analysis [23, p.228]. Jump-set widening never produces worse results than the simple operator, and sometimes it does better. Figure 21 shows an example taken from the code of SPEC



DOI: 10.1002/spe

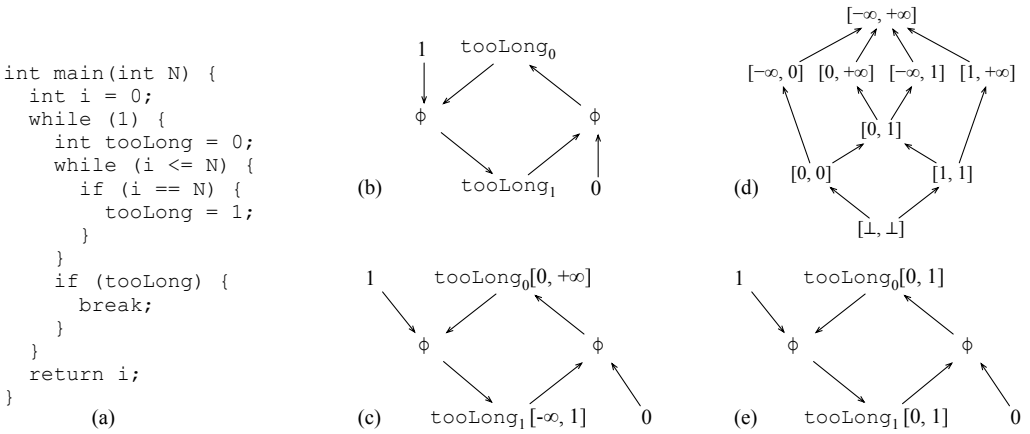



Figure 21. An example where jump-set widening is more precise.

| Benchmark | Size | 0 + Simple | 16 + Simple | 0 + Jump | 16 + Jump |
|---|---|---|---|---|---|
| hmmer | 38,409 | 9.98 | 11.40 (12.45) | 10.98 (9.11) | 11.40 (12.45) |
| gobmk | 84,846 | 8.15 | 9.93 (17.92) | 9.02 (9.64) | 10.13 (19.54) |
| h264ref | 97,494 | 12.58 | 13.11 (4.04) | 13.00 (3.23) | 13.11 (4.04) |
| xalancbmk | 352,423 | 7.71 | 7.98 (3.38) | 7.95 (3.02) | 7.98 (3.38) |
| gcc | 449,442 | 16.09 | 16.63 (3.25) | 16.41 (1.95) | 16.64 (3.31) |

Figure 22. Variation in the precision of our analysis given the widening strategy. The size of each benchmark is given in number of variable nodes in the constraint graph. Precision is given in percentage of bitwidth reduction. Numbers in parenthesis are percentage of gain over 0 + Simple.

CPU `bzip2`. Part of the constraint graph of the program in Figure 21(a) is given in Figure 21(b). The result of applying the simple operator is shown in Figure 21(c). Jump-set widening would use the lattice in Figure 21(d), instead of the lattice in Figure 6(Left). This lattice yields the result given in Figure 21(e), which is more precise.

Another way to improve the precision of growth analysis is to perform a few rounds of abstract interpretation on the constraint graph, and, in case the process does not reach a fixed point, only then to apply the widening operator. Each round of abstract interpretation consists in evaluating all the constraints, and then updating the intervals that change from one evaluation to the other. For instance, in Figure 21 one round of abstract interpretation, coupled with the simple widening operator, would be enough to reach the fixed point of that constraint system. We have experimented with 0 and 16 iterations before doing widening, and the overall result, for the programs in the SPEC CPU 2006 suite is given in Figure 13. Figure 22 shows some of these results in more details for the five largest benchmarks in this collection. In general jump-set widening improves the precision of our results in non-trivial ways. Nevertheless, the simple widening operator preceded by 16 rounds of abstract interpretation in general is more precise than jump-set widening without any cycle of pre-evaluation, as we see in Figure 22.

By combining the different widening operators – simple or jump-set – with the different number of pre-evaluations – in our case 0 or 16, we have four different widening strategies. Figure 20(Right) compares the runtime of all these strategies for the integer programs in the SPEC CPU 2006 collection. We have observed no measurable different between the jump-set and the simple operator: the latter is 0.93% faster, e.g., 1.0093x faster. The strategy that precedes the simple operator with 16 rounds of pre-evaluation is 7% slower than the strategy that does not do any pre-evaluation. Finally, the combination of 16 rounds of pre-evaluation, plus jump-set widening is 13% slower than the simple widening strategy. We observe an anomalous behavior in `astar` and `mcf`: the



DOI: 10.1002/spe

simple strategy results in a small slowdown. These benchmarks have the two fastest runtimes in the benchmark suite; e.g., we analyze `mcf` in 0.02 seconds. Thus, we believe that the unexpected behavior is due to runtime noise that outlives a sequence of 100 executions of each benchmark.

## 6. RELATED WORK

**The Scalability of Range Analysis.** Range analysis is an old ally of compiler writers. The notions of widening and narrowing were introduced by Cousot and Cousot in one of the most cited papers in computer science [1]. Different algorithms for range analysis have been later proposed by Patterson [10], Stephenson *et al.* [18], Mahlke *et al.* [16] and many other researchers. Recently there have been many independent efforts to find exact, polynomial time algorithms to solve constraints on the interval lattice [17, 19, 25, 26, 27]. However, these works are still very theoretical, and have not yet been used to analyze large programs.

There have been many practical approaches to abstract interpretation, with special emphasis on range analysis [4, 20, 36, 37, 38]. Cousot's group, for instance, has been able to globally analyze programs with thousands of lines of code, albeit using domain specific tools. Astrée, for example, analyzes programs that do not contain recursive calls. The work that is the closest to us is Oh *et al.*'s very recent abstract interpretation framework [22]. Oh *et al.* discuss an implementation of range analysis on the interval lattice that scales up to a program with 1,363K LoC (ghostscript-9.00). Because their focus is speed, they do not provide results about precision. We could not find the benchmarks used in those experiments for a direct comparison – the distribution of ghostscript-9.00 available in the LLVM test suite has 27K LoC. On the other hand, we globally analyzed our largest benchmark, SPEC CPU 2006's `gcc`, enabling function inlining, in less than 15 seconds. Oh *et al.*'s implementation took orders of magnitude more time to go over programs of similar size. However, whereas they provide a framework to develop general sparse analyses, we only solve range analysis on the interval lattice.

**Sparse Data-Flow Analyses.** In this paper we provide a *sparse* implementation of range analysis. Sparsity, in our context, means that we associate points in the lattice of interest – intervals in our case – directly to variables. Dense analyses map such information to pairs formed by variables and program points. The compiler related literature contains many descriptions of sparse data-flow analyses. Some among these analyses obtain sparsity by using specific program representations, like we did. Others rely on data-structures. In terms of data-structures, the first, and best known method proposed to support sparse data-flow analyses is Choi *et al.*'s *Sparse Evaluation Graph* (SEG) [39]. The nodes of this graph represent program regions where information produced by the data-flow analysis might change. Choi *et al.*'s ideas have been further expanded, for example, by Johnson *et al.*'s *Quick Propagation Graphs* [40], or Ramalingan's *Compact Evaluation Graphs* [41]. Building upon Choi's pioneering work, researchers have developed many efficient ways to build such graphs [42, 43, 44]. These data-structures have been shown to improve many data-flow analyses in terms of runtime and memory consumption. Nevertheless, the elegance of SEGs and its successors have not, so far, been enough to attract the attention of mainstream compiler writers. Compilers such as gcc, LLVM or Java Hotspot rely, instead, on several types of program representations to provide support to sparse data-flow analyses.

Most eminent among these representations is the Static Single Assignment form [29], which suits well forward flow analyses, such as reaching definitions. Since its debut, the SSA form has been expanded in different ways. For instance, the Gated SSA form allows the static association of logical predicates with data-flow paths [45, 46]. Scott Ananian has introduced in the late nineties the *Static Single Information* (SSI) form, a program representation that supports both forward and backward analyses [47]. This representation was later revisited by Jeremy Singer [48] and, a few years later, by Boissinot *et al.* [49]. Singer provided new algorithms plus examples of applications that benefit from the SSI form, and Boissinot *et al.*, in an effort to clarify some misconceptions about this program representation, introduced the notions of *week* and *strong* SSA form. Another important representation, which supports data-flow analyses that acquire information from uses, is the *Static Single Use* form (SSU). There exists many variants of SSU [50, 51, 52]. For instance, the "strict"



DOI: 10.1002/spe

SSU form enforces that each definition reaches a single use, whereas SSI and other variations of SSU allow two consecutive uses of a variable on the same path. The program representation that we have used in this paper – the *Extended Static Single Assignment* (e-SSA) form – was introduced by Bodik *et al.* [3].

There are so many different program representations because they fit specific data-flow problems. Each representation, given a domain of application, provides the following property: the information associated with the live range of a variable is invariant along every program point where this variable is alive. There are two key aspects that distinguish one representation from the others: firstly, where the information about a variable is acquired, and secondly, how this information is propagated. The e-SSA form, for instance, supports flow analyses that obtain information both from variable definitions and conditional tests and propagate this information forwardly. Such analyses are also supported by the SSI form; hence, we could have used this other representation too. However, as we have shown in previous work, the e-SSA form is considerably more economical [53].

## 7. FINAL CONSIDERATIONS

In this presentation we chose to omit correctness proofs for our algorithms. For a proof that the widening and the narrowing operators give origin to approximating sequences, we recommend Cousot and Cousot's work [1]. For a proof that the e-SSA form transformation is semantics preserving, we invite the reader to check a recent report of ours [53]. In that paper we also show that the results that we obtain via the sparse analysis are equivalent to the results provided by a dense analysis. In other words, if the dense analysis tells us that variable $v$ is associated with the interval $[l, u]$ at program point $i$, and $v'$ is the new name of $v$, alive at $i$ in the e-SSA form program, then $v'$ is associated with the interval $[l, u]$ along its entire live range. Additionally, we have used a dynamic profiler, available in our webpage, to empirically validate our results.

Our implementation of the range analysis algorithm described in this paper is publicly available at `http://code.google.com/p/range-analysis/`. This repository contains instructions to deploy and use our implementation. We provide a gallery of examples, including source codes, control flow graphs and constraint graphs that we produce for meaningful programs at `http://code.google.com/p/range-analysis/wiki/gallery`.

## REFERENCES

1. Cousot P, Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *POPL*, ACM, 1977; 238–252.
2. Souza MRS, Guillon C, Pereira FMQ, da Silva Bigonha MA. Dynamic elimination of overflow tests in a trace compiler. *CC*, 2011; 2–21.
3. Bodik R, Gupta R, Sarkar V. ABCD: eliminating array bounds checks on demand. *PLDI*, ACM, 2000; 321–333.
4. Gampe A, von Ronne J, Niedzielski D, Vasek J, Psarris L. Safe, multiphase bounds check elimination in java. *Software – Practice and Experience* 2011; **41**:753–788.
5. Barik R, Grothoff C, Gupta R, Pandit V, Udupa R. Optimal bitwise register allocation using integer linear programming. *LCPC*, *Lecture Notes in Computer Science*, vol. 4382, Springer, 2006; 267–282.
6. Tallam S, Gupta R. Bitwidth aware global register allocation. *POPL*, ACM, 2003; 85–96.
7. Kong T, Wilken KD. Precise register allocation for irregular architectures. *MICRO*, IEEE, 1998; 297–307.
8. Pereira FMQ, Palsberg J. Register allocation by puzzle solving. *PLDI*, ACM, 2008; 216–226.
9. Scholz B, Eckstein E. Register allocation for irregular architectures. *LCTES/SCOPES*, ACM, 2002; 139–148.
10. Patterson JRC. Accurate static branch prediction by value range propagation. *PLDI*, ACM, 1995; 67–78.
11. Simon A. *Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities*. 1th edn., Springer, 2008.
12. Wagner D, Foster JS, Brewer EA, Aiken A. A first step towards automated detection of buffer overrun vulnerabilities. *NDSS*, ACM, 2000; 3–17.
13. Lokuciejewski P, Cordes D, Falk H, Marwedel P. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. *CGO*, IEEE, 2009; 136–146.
14. 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* 18-21 Jan 2005; **2**:856–861.
15. Lhairech-Lebreton G, Coussy P, Heller D, Martin E. Bitwidth-aware high-level synthesis for designing low-power dsp applications. *ICECS*, IEEE, 2010; 531–534.



DOI: 10.1002/spe

16. 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* 2001; **20**(11):1355–1371.
17. Gawlitza T, Leroux J, Reineke J, Seidl H, Sutre G, Wilhelm R. Polynomial precise interval analysis revisited. *Efficient Algorithms* 2009; **1**:422 – 437.
18. Stephenson M, Babb J, Amarasinghe S. Bitwidth analysis with application to silicon compilation. *PLDI*, ACM, 2000; 108–120.
19. Su Z, Wagner D. A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computeter Science* 2005; **345**(1):122–138.
20. Blanchet B, Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X. A static analyzer for large safety-critical software. *PLDI*, ACM, 2003; 196–207.
21. Venet A, Brat G. Precise and efficient static array bound checking for large embedded c programs. *SIGPLAN Not.* 2004; **39**:231–242.
22. Oh H, Heo K, Lee W, Lee W, Yi K. Design and implementation of sparse global analyses for c-like languages. *PLDI*, ACM, 2012; to appear.
23. Nielson F, Nielson HR, Hankin C. *Principles of Program Analysis*. Springer, 1999.
24. Lattner C, Adve VS. LLVM: A compilation framework for lifelong program analysis & transformation. *CGO*, IEEE, 2004; 75–88.
25. Costan A, Gaubert S, Goubault E, Martel M, Putot S. A policy iteration algorithm for computing fixed points in static analysis of programs. *CAV*, 2005; 462–475.
26. Lakhdar-Chaouch L, Jeannet B, Girault A. Widening with thresholds for programs with complex control graphs. *ATVA*, Springer-Verlag, 2011; 492–502.
27. Su Z, Wagner D. A class of polynomially solvable range constraints for interval analysis without widenings and narrowings. *TACAS*, 2004; 280–295.
28. do Couto Teixeira D, Pereira FMQ. The design and implementation of a non-iterative range analysis algorithm on a production compiler. *SBLP*, SBC, 2011; 45–59.
29. Cytron R, Ferrante J, Rosen BK, Wegman MN, Zadeck FK. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS* 1991; **13**(4):451–490.
30. Aho AV, Lam MS, Sethi R, Ullman JD. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.
31. Warren HS. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., 2002.
32. Ferrante J, Ottenstein KJ, Warren JD. The program dependence graph and its use in optimization. *TOPLAS* 1987; **9**(3):319–349.
33. Cousot P, Halbwachs N. Automatic discovery of linear restraints among variables of a program. *POPL*, ACM, 1978; 84–96.
34. Miné A. The octagon abstract domain. *Higher Order Symbol. Comput.* 2006; **19**:31–100.
35. Rimsa AA, D'Amorim M, Pereira FMQ. Tainted flow analysis on e-SSA-form programs. *CC*, Springer, 2011; 124–143.
36. Bertrane J, Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Rival X. Static analysis and verification of aerospace software by abstract interpretation. *I@A*, AIAA, 2010; 1–38.
37. Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Rival X. Why does astrée scale up? *Form. Methods Syst. Des.* 2009; **35**(3):229–264.
38. Jung Y, Kim J, Shin J, Yi K. Taming false alarms from a domain-unaware c analyzer by a bayesian statistical post analysis. *SAS*, 2005; 203–217.
39. Choi JD, Cytron R, Ferrante J. Automatic construction of sparse data flow evaluation graphs. *POPL*, 1991; 55–66.
40. Johnson R, Pingali K. Dependence-based program analysis. *PLDI*, ACM, 1993; 78–89.
41. Ramalingam G. On sparse evaluation representations. *Theoretical Computer Science* 2002; **277**(1-2):119–147.
42. Pingali K, Bilardi G. APT: A data structure for optimal control dependence computation. *PLDI*, ACM, 1995; 211–222.
43. Pingali K, Bilardi G. Optimal control dependence computation and the roman chariots problem. *TOPLAS*, ACM, 1997; 462–491.
44. Johnson R, Pearson D, Pingali K. The program tree structure. *PLDI*, ACM, 1994; 171–185.
45. Ottenstein KJ, Ballance RA, MacCabe AB. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. *PLDI*, ACM, 1990; 257–271.
46. Tu P, Padua D. Efficient building and placing of gating functions. *PLDI*, ACM, 1995; 47–55.
47. Ananian S. The static single information form. Master's Thesis, MIT September 1999.
48. Singer J. Static program analysis based on virtual register renaming. PhD Thesis, University of Cambridge 2006.
49. Boissinot B, Darte A, Rastello F, de Dinechin BD, Guillon C. Revisiting out-of-SSA translation for correctness, code quality, and efficienty. *CGO*, IEEE, 2009; XX–XX.
50. Plevyak JB. Optimization of object-oriented and concurrent programs. PhD Thesis, University of Illinois at Urbana-Champaign 1996.
51. George L, Matthias B. Taming the ixp network processor. *PLDI*, ACM, 2003; 26–37.
52. Lo R, Chow F, Kennedy R, Liu SM, Tu P. Register promotion by sparse partial redundancy elimination of loads and stores. *PLDI*, ACM, 1998; 26–37.
53. Tavares ALC, Boissinot B, Bigonha MAS, Bigonha R, Pereira FMQ, Rastello F. A program representation for sparse dataflow analyses. *Science of Computer Programming* 201X; **X**(X):2–25. Invited paper with publication expected for 2012.



DOI: 10.1002/spe