Speed and Precision in Range Analysis

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

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

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

Abstract. *Range analysis is a compiler technique that determines statically the lower and upper values that each integer variable from a target program may assume during this program's execution. This type of inference is very important, because it enables several compiler optimizations, such as dead and redundant code elimination, bitwidth aware register allocation, and detection of program vulnerabilities. In this paper we describe an inter-procedural, context-sensitive range analysis algorithm that we have implemented in the LLVM compiler. During the effort to produce an industrial-quality implementation of our algorithm, we had to face a constant tension between precision and speed. The foremost goal of this paper is to discuss the many engineering choices that, due to this tension, have shaped our implementation. Given the breath of our evaluation, we believe that this paper contains the most comprehensive empirical study of a range analysis algorithm ever presented in the compiler related literature.*

1. Introduction

Range analysis is a compiler technique whose objective is to determine statically, for each program variable, limits for the minimum and maximum values that this variable might assume during the program execution. Range analysis is important because it enables many compiler optimizations. Among these optimizations, the most well-known are dead and redundant code elimination. Examples of redundant code elimination include the removal of array bounds checks [3, 13, 25] and overflow checks [20]. Additionally, range analysis is also used in bitwidth aware register allocation [1, 18, 24], branch prediction [17] and synthesis of hardware for specific applications [4, 12, 14, 21]. Because of this importance, the programming language community has put much effort in the design and implementation of efficient and precise range analysis algorithms.

However, the compiler related literature does not contain a comprehensive evaluation of range analysis algorithms that scale up to entire programs. Many works on this subject are limited to very small programs [14, 19, 21], or, given their theoretic perspective, have never been implemented in production compilers [5, 9, 10, 22, 23]. There are implementations of range analysis that deal with very large programs [2, 7, 13, 15]; nevertheless, because these papers focus on applications of range analysis, and not on its implementation, they do not provide a thorough discussion about their engineering decisions. A noticeable exception is the recent work of Oh *et al.* [16], which discusses a range analysis algorithm developed for C programs that can handle very large benchmarks. Oh *et al.* present an evaluation of the speed and memory consumption of their implementation. In this paper we claim to push this discussion considerably further.

We have implemented an industrial-quality range analysis algorithm in the LLVM compiler [11]. As we show in Section 2.1, our implementation, which is publicly avail-

able, is able to analyze programs with over one million assembly instructions in ten seconds. And our implementation is not a straw-man: it produces very precise results. We have compared the results produced by our implementation with the results obtained via a dynamic profiler, which we have also implemented. As we show in Section 2.1, 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.

While designing and implementing our algorithm we had to face several important engineering choices. Many approaches that we have used in an attempt to increase the precision of our implementation would result in runtime slowdowns. Although we cannot determine the optimum spot in this design space, given the vast number of possibilities, we discuss our most important implementation decisions in Section 3. Section 3.1 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 3.4 discuss the importance of choosing a suitable intermediate representation when implementing a sparse data-flow framework. Section 3.2 compares the intra-procedural and the inter-procedural versions of our algorithm. The role of context sensitiveness is discussed in Section 3.3. Finally, Section 3.5 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 [22, 23]. 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 [9] approach. This technique uses the Bellman-Ford algorithm to detect increasing or decreasing cycles 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 [8]. 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, which we summarize in this paper.

2. Brief Description of our Range Analysis Algorithm

The Interval Lattice. Following Gawlitza *et al.*'s notation, we shall be performing arithmetic operations over the complete lattice $\mathcal{Z} = \mathbb{Z} \cup \{-\infty, +\infty\}$, where the ordering is naturally given by $-\infty < \ldots < -2 < -1 < 0 < 1 < 2 < \ldots + \infty$. For any $x > -\infty$ we define:

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

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

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

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

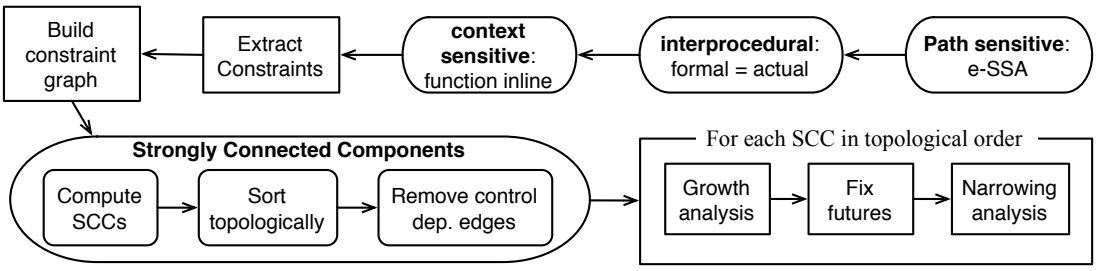

Fig. 1. Our implementation of range analysis. Rounded boxes are optional steps.

the, during the execution of the target program, any valued i assigned to v is such that $l \leq i \leq u$.

A Holistic View of our Range Analysis Algorithm. We perform range analysis in a number of steps. First, we convert the program to a suitable intermediate representation that makes it easier to extract constraints. From these constraints, we build a dependence graph that allows us to do range analysis sparsely. Finally, we solve the constraints applying different fix-point iterators on this dependence graph. Figure 1 gives a global view of this algorithm. Some of the steps in the algorithm are optional. They improve the precision of the range analysis, at the expense of a longer running time. In Section 3 we discuss in more details these tradeoffs. The last phase, which we call the *micro algorithm*, happens per strong component; however, if we opted for not building these components, then it happens once for the entire constraint graph. Nevertheless, the use of strongly connected components is so essential for performance and precision, as we show in Section 3.1, that it is considered optional only because we can easily build our implementation without this module.

We will illustrate the mandatory parts of the algorithm via the example program in Figure 2. More details about each phase of the algorithm will be introduced in Section 3, when we discuss our engineering decisions. Figure 2(a) shows an example program taken from the partition function of the quicksort algorithm used by Bodik *et al.* [3]. We have removed the code that performs array manipulation from this program, as it plays no role in our explanation. Figure 2(b) shows one possible way to represent this program internally. As we explain in Section 3.4, a good program representation allows us to find more precise results. In this example we chose a program representation called Extended Static Single Assignment form, which lets us to solve range analysis via a path sensitive algorithm. Figure 2(c) shows the constraints that we extract from the intermediate representation seen in part (b) of this figure. From these constraints we build the *constraint graph* in Figure 2(d). This graph is the main data-structure that we use to solve range analysis. For each variable v in the constraint system, the constraint graph has a node n_v. Similarly, for each constraint $v = f(\ldots, u, \ldots)$ in the constraint system, the graph has an *operation node* n_f. For each constraint $v = f(\ldots, u, \ldots)$ we add two edges to the graph: $\overrightarrow{n_u n_f}$ and $\overrightarrow{n_f n_v}$. Some edges in the constraint graph are dashed. These are called *control dependence edges*. If a constraint $v = f(\ldots, \mathbf{ft}(u), \ldots)$ uses a *future* bound from a variable u, then we add to the constraint graph a control dependence edge $\overrightarrow{n_u n_f}$. The final solution to this instance of the range analysis problem is given in Figure 2(e).

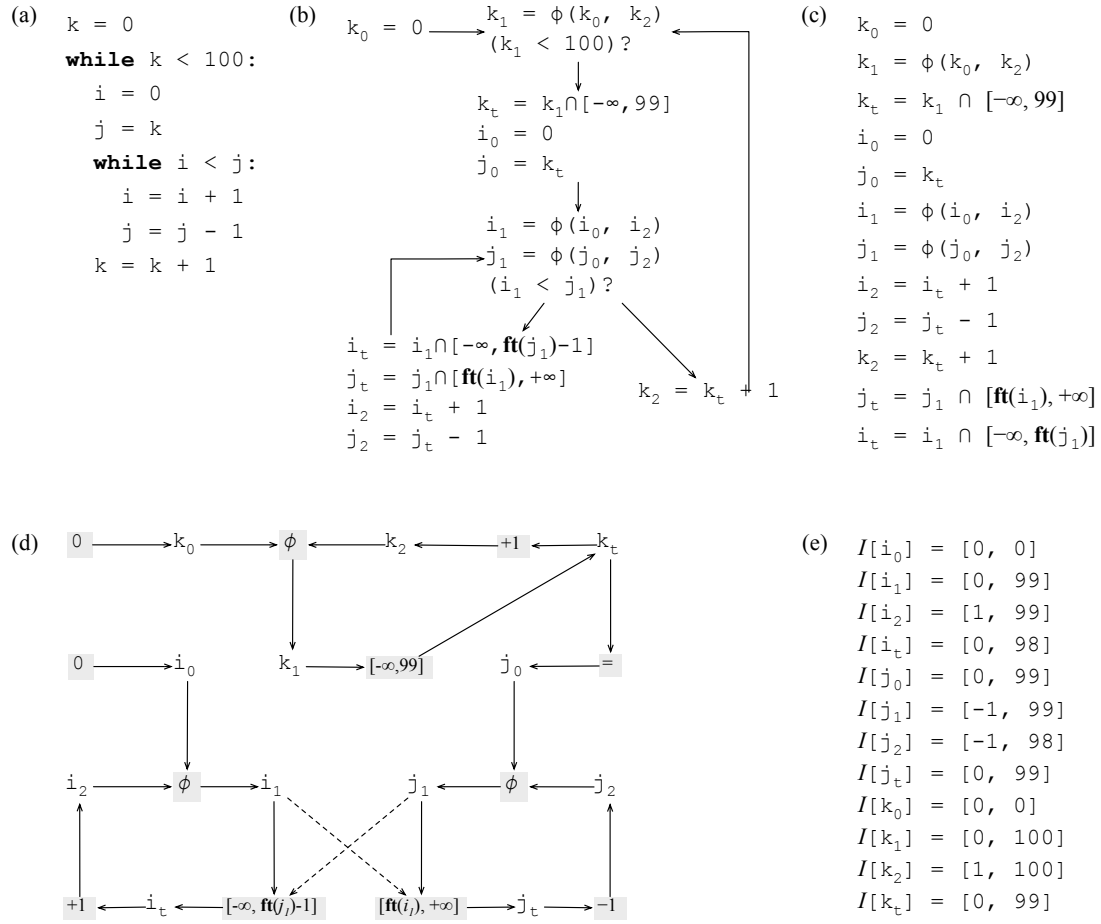

Fig. 2. Range analysis by example. (a) Input program. (b) Internal compiler representation. (c) Constraints of the range analysis problem. (d) The constraint graph. (e) The final solution.

The Micro Algorithm. We find the solution given in Figure 2(e) in a process that we call the micro algorithm. This phase is further divided into three sub-steps: (i) growth analysis; (ii) future resolution and (iii) narrowing analysis.

Growth analysis: The objective of growth analysis is to determine the growth behavior of each program variable. There are four possible behaviors: (a) the variable is bound to a constant interval, such as k_0 in Figure 2(b). (b) The variable is bound to a decreasing interval, i.e., an interval whose lower bound decreases. This is the case of j_1 in our example. (c) The variable is bound to an increasing interval, i.e., its upper bound increases. This is the case of i_1 in the example. (d) The variable is bound to an interval that expands in both directions. The growth analysis uses an infinite lattice, i.e., $\mathcal{Z}^2$. Thus, a careless implementation of an algorithm that infers growth patterns might not terminate. In order to ensure termination, we must reckon on a technique called *widening*, first introduced by Cousot and Cousot as a key component of abstract interpretation [6]. There are many different widening strategies. We discuss some of them in Section 3.5.

Future resolution: In order to learn information from comparisons between variables, such as `i < j` in Figure 2(a), we bind some intervals to *futures*. Futures are symbolic limits, which will be replaced by actual numbers once we finish the growth analysis. The

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

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

Fig. 3. Rules to replace futures by actual bounds. S is the interval bound to each variable after the widening analysis.

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

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

Fig. 4. Cousot and Cousot's narrowing operator.

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 3 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 V' be analyzed before 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 N_v before $N_{v'}$, or places these nodes in the same strongly connected component. For instance, in Figure 2(b), 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 [6, 248], which we show in Figure 4.

Example: Continuing with our example, Figure 5 shows the application of our algorithm on the last strong component of Figure 2(d). Upon meeting this SCC, we have already determined that the interval $[0, 0]$ is bound to i_0 and that the interval $[100, 100]$ is bound to j_0. We are not guaranteed to find the least fix point of a constraint system. However, in this example we did it. We emphasize that finding this tight solution was only possible because of the topological ordering of the constraint graph in Figure 2(d). Had we applied the widening operator onto the whole graph, then we would have found out that variable j_0 is bound to $[-\infty, +\infty]$, because (i) it receives its interval directly from variable k_t, which is upper bounded by $+\infty$, and (ii) it is part of a negative cycle. On the other hand, by only analyzing j's SCC after we have analyzed k's, k only contribute the constant range $[0, 99]$ to j_0.

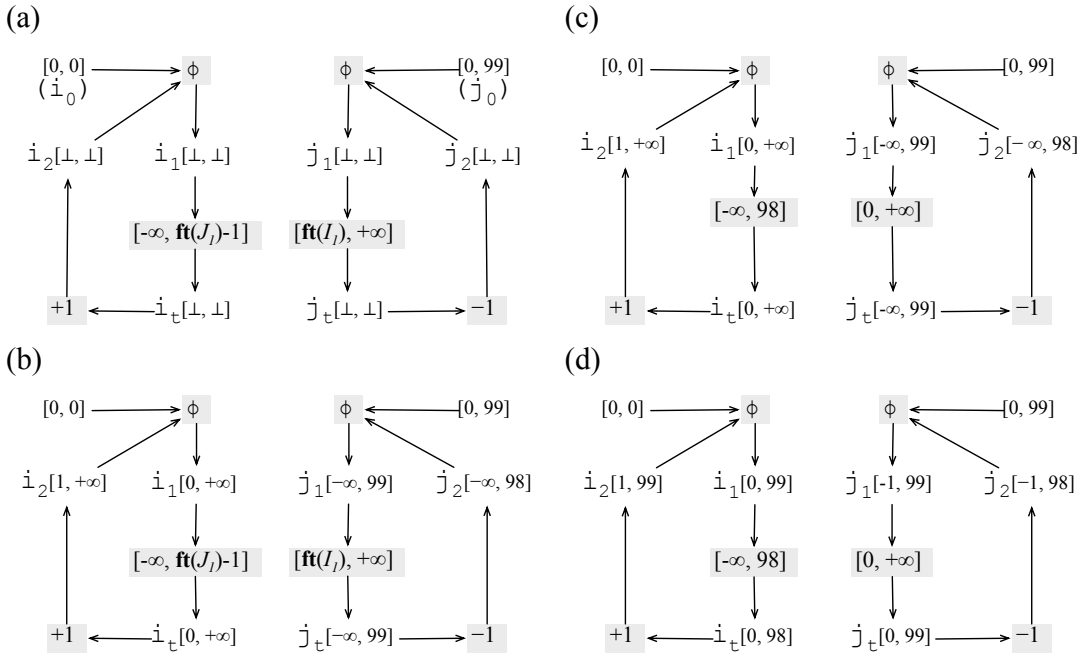

Fig. 5. Four snapshots of the last SCC of Figure 2(d). (a) After removing control dependence edges. (b) After running the growth analysis. (c) After fixing the intersections bound to futures. (d) After running the narrowing analysis.

2.1. 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 6 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, to increase program sizes. 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 15,91secs 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.

Memory complexity. The experiments also reveal that the memory consumption of our implementation is linear with the program size. Figure 7 plots these two quantities together. The linear correlation, in this case, is even stronger than that found in Figure 6, 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 benchmarks, requiring 265,588KB to run. Memory includes stack, heap and the

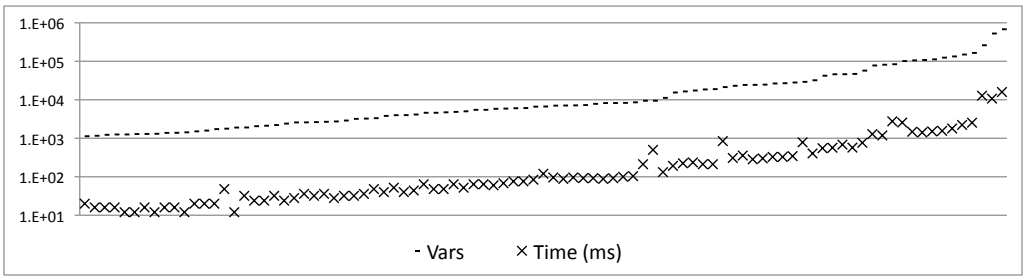

Fig. 6. Correlation between program size (number of var nodes in constraint graphs after inlining) and analysis runtime (ms). Coefficient of determination = 0.967.

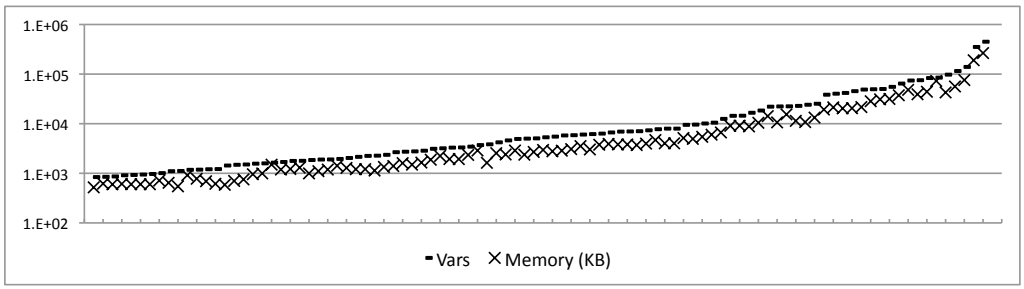

Fig. 7. Comparison between program size (number of var nodes in constraint graphs after inlining) and memory consumption (KB). Coefficient of determination = 0.9947.

executable program code.

Precision Our implementation of range analysis is remarkably precise, considering its runtime. Lakhdar *et al.*'s relational analysis [10], for instance, takes about 25 minutes to go over a program with almost 900 basic blocks. We analyze programs of similar size 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 [10]. 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 upper and lower limits, given an execution of the target program. Figure 8 compares our results with those measured dynamically for the Stanford benchmark suite, which is publicly available [1].

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 estimated 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 $-\infty$.

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

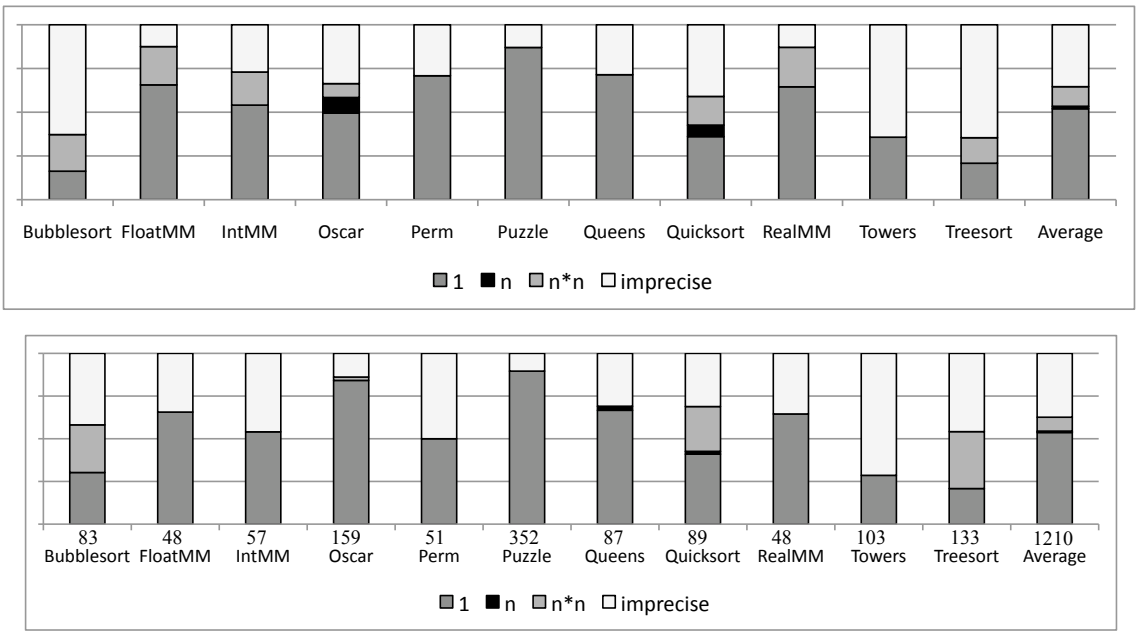

Fig. 8. (Upper) Comparison between static range analysis and dynamic profiler for upper bounds. (Lower) Comparison between static range analysis and dynamic profiler for lower bounds. The numbers above the benchmark names give the number of variables in each program.

As we see in Figure 8, 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 [21]. 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 large programs.

3. Design Space

As we see from a cursory glance at Figure 1, our range analysis algorithm has many optional modules. These modules provide the user with the possibility 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 9 plots, for the integer programs in the SPEC CPU 2006 benchmark suite, precision versus speed for different configurations of our implementation.

3.1. 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 [**?**, Sec 6.3]. In our case, the results are dramatic,

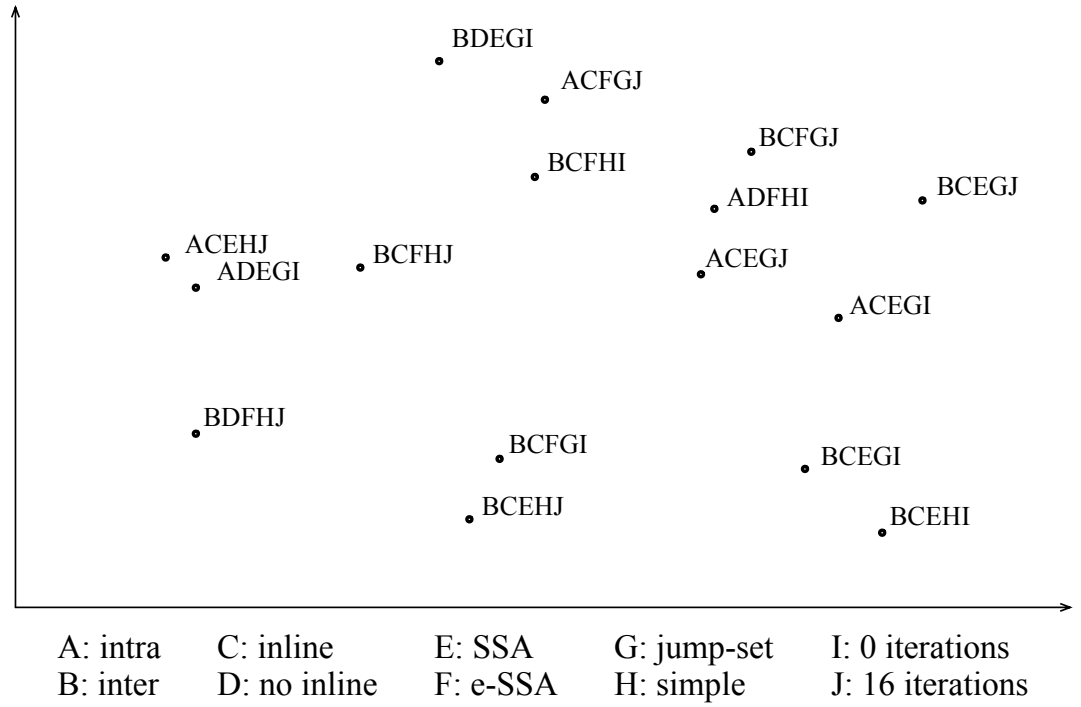

Fig. 9. Design space exploration: precision versus speed for different configurations of our algorithm.

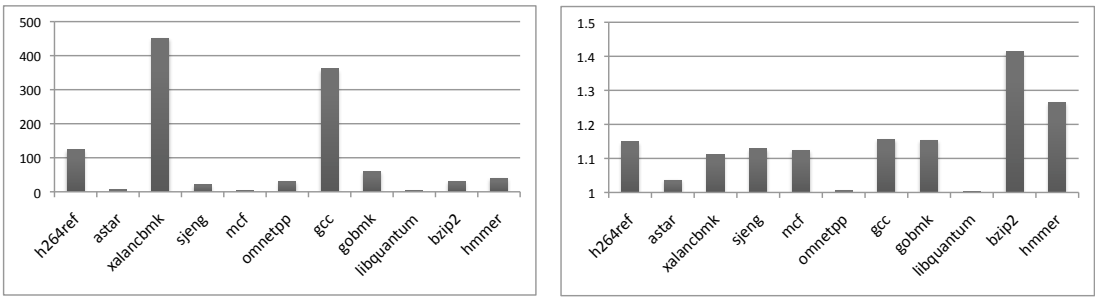

Fig. 10. (Left) Bars give time to run our analysis without building strong components divided by time to run the analysis on strongly connected components. (Right) Bars give precision, in bitwidth reduction, that we obtain with strong components, dividid by the precision that we obtain without them.

mostly in terms of speed, but also in terms of precision. Figure 10 shows the speed up 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. As we see in Figure 10, in some cases, as in `bzip2`, it is 40% more precise. 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 5 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,

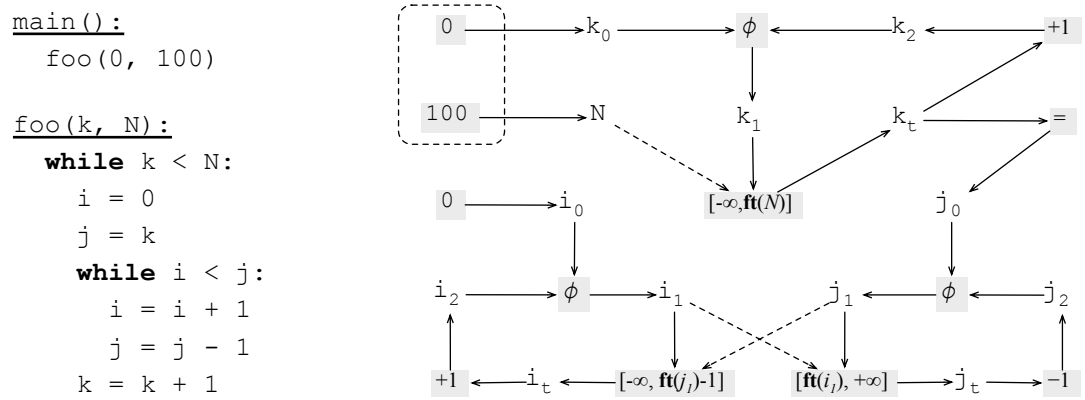

Fig. 11. Example where an intra-procedural implementation would lead to imprecise results.

by the time we applied the growth analysis on j we would still find k bound to $0, +\infty$.

3.2. Intra versus Inter-procedural Analysis

A naive implementation of range analysis would solve it once for each function. However, we can gain in precision by performing it inter-procedurally; that is, by allowing the results found for a function f to flow into other functions that f calls. Figure 11 illustrates the inter-procedural analysis for the program seen in Figure 2(a).

Figure 13 shows that the whole program analysis increases moderately the precision of our analysis. We show results for the five largest programs in three different categories of benchmarks: SPEC CPU 2006, the Stanford Suite [2] and Bitwise [21]. 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 variables. 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. Our results for the SPEC programs were disappointingly: on the average, the intra-procedural version of our analysis saves 5.23% of the total program bitwidth. By using the inter-procedural version we can increase this number to 8.89%. A manual inspection of the SPEC programs reveal that this result is expected: they manipulate files, and their texts 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%. Finally, for Bitwise we obtain a bitwidth reduction of 12.27%. However, this average is brought down by two outliers: `edge_detect` and `sha`, which cannot be reduced. The bitwise benchmarks were implemented by Stephenson *et al.* [21] to validate their 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.

3.3. Context Sensitive versus Context Insensitive Analysis

Another way to increase the precision of range analysis is by using a context-sensitive implementation. Context-sensitiveness allows us to distinguish different calling sites of

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

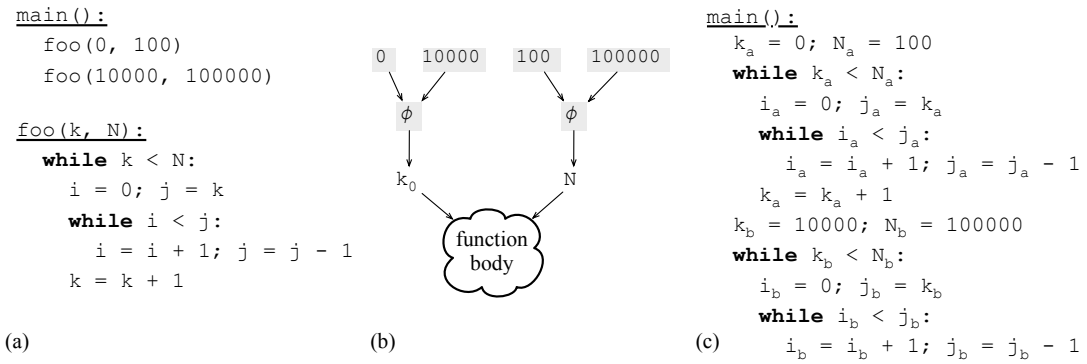

Fig. 12. Example where a context-sensitive implementation improves the results of range analysis.

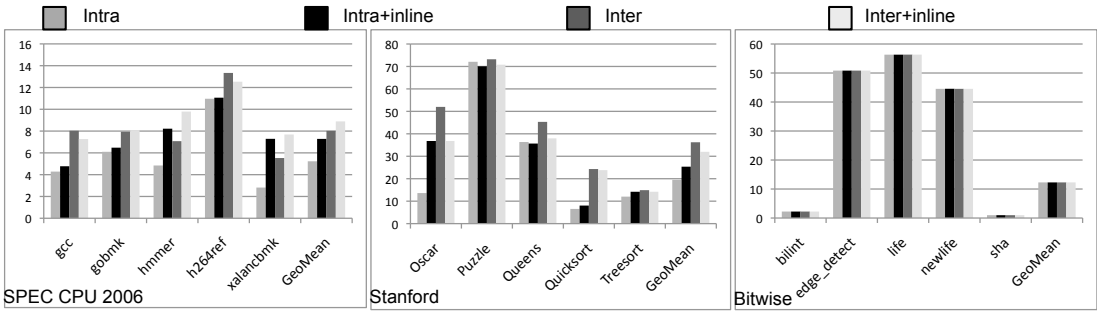

Fig. 13. The impact of whole program analysis on precision. Each bar gives precision in %bitwidth reduciton.

the same function. Figure **??** shows way the ability to make this distinction is important for precision. In Figure 12(a) we have two different calls of function `foo`. A usual way to perform a data-flow analysis inter-procedurally is to create assignments between formal and actual parameters, as we show in Figure 12(b). 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 12(c). The results that we can derive for the transformed program are much more precise, as each input parameter is assigned a single value.

3.4. The choice of a program representation

If strong components account for the largest gains in speed, the choice of a suitable program representation are responsible for the largest gains in precision. However, here we no longer have a win-win condition: a more expressive program representation decreases our runtime, because it increases the size of the target program. Nevertheless, the slowdown is inexpressive. We chose to run our analysis in programs in the Extended Static Single Assignment (e-SSA) form [3]. The alternative, which gives us a faster, albeit imprecise, analysis, is to run our algorithm in programs in Static Single Assignment (SSA) form [**?**]. 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

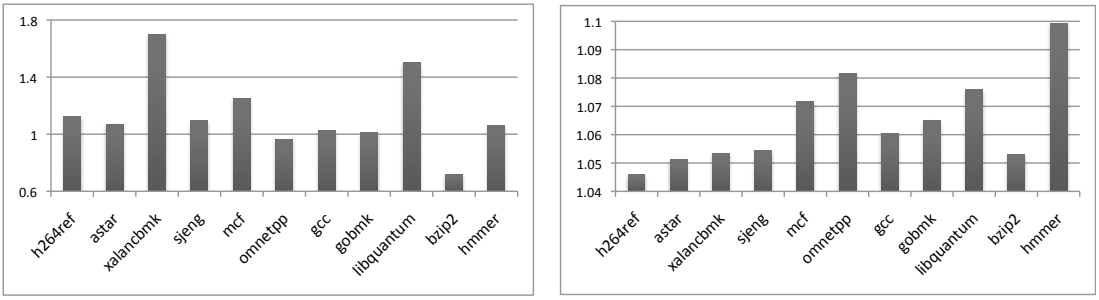

Fig. 14. (Left) Bars give the time to run analysis on e-SSA form programs divided by the time to run analysis on SSA form programs. (Right) Bars give the size of the e-SSA form program, in number of assembly instructions, divided by the size of the SSA form program.

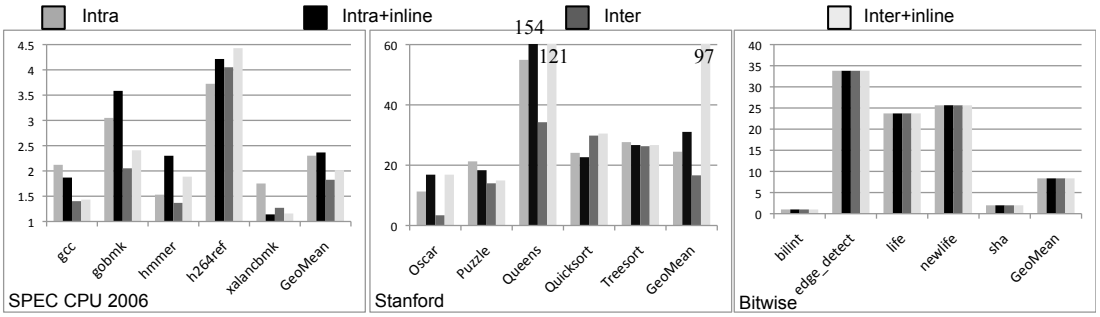

Fig. 15. The impact of the e-SSA transformation on precision for three different benchmark suites. Bars give the ratio of precision (in bitwidth reduction), obtained with e-SSA form conversion divided by precision without e-SSA form conversion.

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 created 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 [**?**] and array bounds checks elimination [3].

Figure 14 compares these two program representations in terms of runtime. As we see in Figure 14(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 14(Right). 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 we show in Figure 15.

3.5. Choosing a Widening Strategy

4. Conclusion

References

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

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

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

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

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

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

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

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

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

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

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

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

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

[14] Fancesco Logozzo and Manuel Fahndrich. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *SAC*, pages 184–188. ACM, 2008.

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

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

[17] Hakjoo Oh, Lucas Brutschy, and Kwangkeun Yi. Access analysis-based tight localization of abstract memories. In *VMCAI*, pages 356–370. Springer, 2011.

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

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

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

[21] Andrei Alves Rimsa, Marcelo D'Amorim, and Fernando M. Q. Pereira. Tainted flow analysis on e-SSA-form programs. In *CC*, pages 124–143. Springer, 2011.

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

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

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

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

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

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

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