Pointer Disambiguation via Strict Inequalities

Maroua Maalej
University of Lyon, France & LIP
(UMR CNRS/ENS Lyon/
UCB Lyon1/INRIA)
Maroua.Maalej@ens-lyon.fr

Vitor Paisante
Department of Computer Science
UFMG, Brazil
paisante@dcc.ufmg.br

Pedro Ramos
Department of Computer Science
UFMG, Brazil
pedroramos@dcc.ufmg.br

Laure Gonnord
Univ. Lyon1, France & LIP
(UMR CNRS/ENS Lyon/
UCB Lyon1/INRIA)
Laure.Gonnord@ens-lyon.fr

Fernando Magno Quintão Pereira
Department of Computer Science
UFMG, Brazil
fernando@dcc.ufmg.br

Abstract

The design and implementation of static analyses that disambiguate pointers has been a focus of research since the early days of compiler construction. One of the challenges that arise in this context is the analysis of languages that support pointer arithmetics, such as C, C++ and assembly dialects. This paper contributes to solve this challenge. We start from an obvious, yet unexplored, observation: if a pointer is strictly less than another, they cannot alias. Motivated by this remark, we use the abstract interpretation framework to build strict less-than relations between pointers. To this end, we construct a program representation that bestows the Static Single Information (SSI) property onto our dataflow analysis. SSI gives us an efficient sparse algorithm, which, once seen as a form of abstract interpretation, is correct by construction. We have implemented our static analysis in LLVM. It runs in time linear on the number of program variables, and, depending on the benchmark, it can be as much as six times more precise than the pointer disambiguation techniques already in place in that compiler.

Categories and Subject Descriptors D - Software [*D.3 Programming Languages*]: D.3.4 Processors - Compilers

General Terms Languages, Experimentation

CGO, February 4–8, 2017, Austen, US.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

Keywords Alias analysis, range analysis, speed, precision

1. Introduction

Pointer disambiguation is the problem of determining if two pointers, p_1 and p_2, can refer to the same memory location. If this overlapping happens, then p_1 and p_2 are said to *alias*. Pointer disambiguation has been a focus of intense research [3, 15, 17, 34], since the debut of the first alias analysis approaches [6, 33]. Today, state-of-the-art algorithms have acceptable speed [15, 28], good precision [39] or meet each other halfway [16]. However, understanding the relationships between memory references in programming languages that support pointer arithmetics remains challenging.

Pointer arithmetics come from the ability to associate pointers with offsets. Much of the work on automatic parallelization consists in the design of techniques to distinguish offsets from the same base pointer. Michael Wolfe [41, Ch.7] and Aho et al. [1, Ch.11] have entire chapters devoted to this issue. State-of-the-art approaches perform such distinction by solving diophantine equations, be it via the greatest common divisor test, be it via integer linear programming, as Rugina and Rinard do [31]. Other techniques try to associate intervals, numeric or symbolic, with pointers [4, 26, 27, 43], presenting different ways to build Balakrishnan and Reps' notion of *value sets*. And yet, as expressive and powerful as such approaches are, they fail to disambiguate locations that are obviously different, as v[i] and v[j], in the loop:

```
for(i = 0, j = N; i < j; i++, j--) v[i] = v[j];
```

In this paper, we present a simple and efficient solution to this shortcoming. We say that v[i] and v[j] are obviously different locations because $i < j$. There are techniques to compute *less-than* relations between integer variables in programs [7, 20, 21, 23]. Nevertheless, so far, they have not been used to disambiguate pointer locations. The insight that such techniques are effective and useful to such purpose is the key contribution of this paper. However, we go beyond: we rely on recent advances on the construction of sparse dataflow analyses [37] to design an efficient way to solve less-than inequalities. The sparse implementation lets us view this problem as an instance of the abstract interpretation framework; hence, we get correctness for free. The end result of our tool is a less-than analysis that can be augmented to handle different program representations, and that can increase in non-trivial ways the ability of compilers to distinguish pointers.

To demonstrate this last statement, we have implemented our static analysis in the LLVM compiler [19]. We show empirically that industrial-quality alias analyses still leave unresolved pointers which our simple technique can disambiguate. As an example, we distinguish 11,881 pairs of pointers in SPEC's `lbm`, whereas LLVM's analyses distinguish 1,888. Furthermore, by combining our approach with basic heuristics, we obtain even more impressive results. For instance, our less-than check increases the success rate of LLVM's basic disambiguation heuristic from 48.12% (1,705,559 queries) to 64.19% (2,274,936) in SPEC's `gobmk`.

2. Overview

To motivate the need for a new points-to analysis we show its application on the programs seen in Figure 1. The figure displays the C implementation of two sorting routines that make heavy use of pointers. In both cases, we know that memory positions `v[i]` and `v[j]` can never alias within the same loop iteration. However, traditional points-to analyses cannot prove this fact. Typical implementations of these analyses, built on top of the work of Andersen [3] or Steensgaard [34], can distinguish pointers that dereference different memory blocks; however, they do not say much about references ranging on the same array.

There are points-to analyses designed specifically to deal with pointer arithmetics [2, 4, 24, 27, 31, 32, 35, 38, 40]. Still, none of them works satisfactorily for the two examples seen in Figure 1. The reason for this ineffectiveness lays on the fact that these analyses use range intervals to disambiguate pointers. In our examples, the ranges of integer variables `i` and `j` overlap. Consequently, any conservative range analysis, à la Cousot [10], once applied on Figure 1 (a), will conclude that `i` exists on the interval $[0, \mathtt{N} - 2]$, and that `j` exists on the interval $[1, \mathtt{N} - 1]$. Because these two intervals have non-empty intersection, points-to analyses based on the interval lattice will not be able to disambiguate the memory accesses at lines 6-8 of Figure 1 (a). The same holds true for the memory accesses in Figure 1 (b).

The technique that we introduce in this paper can disambiguate every use of `v[i]` and `v[j]` in both examples. The key to this success is the observation that $\mathtt{i} < \mathtt{j}$ at every program point where we have an access to `v`. We conclude that $\mathtt{i} < \mathtt{j}$ by means of a "less-than check". A less-than check is a relationship between two variables that is true whenever we can prove – statically – that one holds a value lesser than the value stored in the other. In Figure 1 (a), we know that $\mathtt{i} < \mathtt{j}$ because of the way that `j` is initialized, within the for statement at line 4. In Figure 1 (b), we know that $\mathtt{i} < \mathtt{j}$ due to the conditional check at line 7.

A more precise alias analysis brings many advantages to compilers. One of such benefits is optimizations: the extra precision gives compilers information to carry out more extensive transformations in programs. Figure 2 illustrates this benefit. The figure shows the result of applying Surendran's [36] inter-iteration scalar replacement on the insertion sort algorithm seen in Figure 1. Scalar replacement is a compiler optimization that consists in moving memory locations to registers as much as possible. This optimization tends to speed programs up because it removes memory accesses from their source code. In this example, if we can prove that `v[i]` and `v[j]` do not reference overlapping memory locations, we can move these locations to temporary variables. For instance, we have loaded location `v[j]` into `tmp_j` at line 6. We update the value of `v[i]` at line 13.

3. The Less-Than Check

This section introduces a dataflow analysis whose goal is to construct a "less-than" set for each variable x (pointer or numeric, as we will discuss in Section 3.6). We denote

```
1  void ins_sort(int* v, int N) {
2    int i, j;
3    for (i = 0; i < N - 1; i++) {
4      for (j = i + 1; j < N; j++) {
5        if (v[i] > v[j]) {
6          int tmp = v[i];
7          v[i] = v[j];
8          v[j] = tmp;
9        }
10     }
11   }
12 }
```
(a)

```
1  void partition(int *v, int N) {
2    int i, j, p, tmp;
3    p = v[N/2];
4    for (i = 0, j = N - 1;; i++, j--) {
5      while (v[i] < p) i++;
6      while (p < v[j]) j--;
7      if (i >= j)
8        break;
9      tmp = v[i];
10     v[i] = v[j];
11     v[j] = tmp;
12   }
13 }
```
(b)

Figure 1. Two snippets of C code that challenge typical pointer disambiguation approaches.

```c
void ins_sort_opt(int* v, int N) {
  int i, j, tmp_i, tmp_j, tmp;
  for (i = 0; i < N - 1; i++) {
    tmp_i = v[i];
    for (j = i + 1; j < N; j++) {
      tmp_j = v[j];
      if (tmp_i > tmp_j) {
        tmp = tmp_i;
        tmp_i = tmp_j;
        v[j] = tmp;
      }
    }
    v[i] = tmp_i;
  }
}
```

Figure 2. Implementation of insertion sort, seen in Figure 1 (a), after scalar replacement [36].

Integer constants	::=	$\{c_1, c_2, \ldots\}$
Variables	::=	$\{x_1, x_2, \ldots\}$
Program (P)	::=	$\{\ell_1 : I_1; \ldots, \ell_n : I_n;\}$
Instructions (I)	::=	
– Addition	|	$x_0 = x_1 + x_2$
– ϕ-function	|	$x_0 = \phi(x_1 : \ell_1, \ldots, x_n : \ell_n)$
– Comparison	|	$(x_1 < x_2)$? goto ℓ_t : goto ℓ_f

Figure 3. The syntax of our language. Variables have scalar type, e.g., either integer or pointer.

such an object by $\mathsf{LT}(x)$. As we prove in Section 3.5, the important invariant that this static analysis guarantees is that if $x' \in \mathsf{LT}x$, then $x' < x$ at every program point where both variables are alive. Our ultimate goal is to use this invariant to disambiguate pointers, as we explain in Section 3.6.

3.1 The Core Language

We use a core language to formalize the developments that we present in this paper. Figure 3 shows the syntax of this language. Our core language contains only those instructions that are essential to describe our static analysis. The reader can augment it with other assembly instructions, to make it as expressive as any industrial-strength program representation. As a testimony to this fact, the implementation that we describe in Section 4 comprises the entire LLVM intermediate representation. Figure 3 describe programs in Static Single Assignment form [12]; therefore, it contains ϕ-functions. Additionally, it contains arithmetic instructions and conditional branches. These two kinds of instructions feed our static analysis with new information.

EXAMPLE 3.1. *Figure 4 describes a program in our core language. This is an artificial example, whose semantics is immaterial. Figure 4 illustrates a few key properties of the strict SSA representation: (i) the definition point of a variable dominates all its uses, and (ii) if two variables interfere, one of them is alive at the definition point of the other. Such properties will be useful in Section 3.5.*

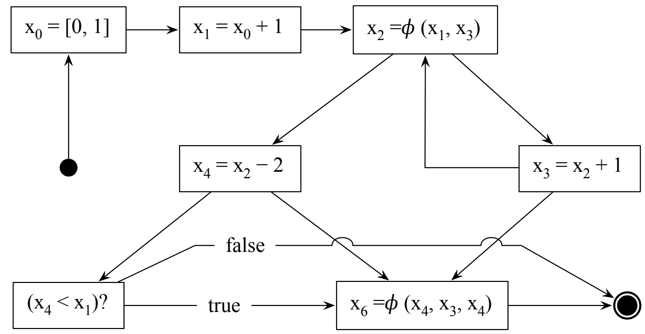

Figure 4. Program written in our core language.

3.2 Program Representation

For efficiency reasons, we want to implement a *sparse dataflow analysis*. A dataflow analysis is said to be sparse if runs on a program representation that ensures the *Static Single Information* (SSI) Property [37]. To keep this paper self-contained, we quote Tavares *et al.*'s notion of single information property:

DEFINITION 3.2 (Static Single Information Property). *A dataflow analysis bears the static single information property if it always associates a variable with the same abstract state at every program point where that variable is alive.*

Following Tavares *et al.* [37], to ensure the SSI property, we split the live range of every variable x at each program point where new information about x can appear. The live range of a variable v is the collection of program points where v is alive. To split the live range of x at a point ℓ, we create a copy $x' = x$ at ℓ, and rename uses of x at every program point dominated by ℓ. We shall write ℓ dom ℓ' to indicate that ℓ dominates ℓ', meaning that any path from the beginning of the control flow graph to ℓ' must cross ℓ. There are three situations that create new less-than information about a variable x:

1. x is defined. For instance, if $x = x' + 1$, then we know that $x' < x$;
2. x is used in a subtraction, e.g,. $x' = x + n, n < 0$. In this case, we know that $x' < x$;
3. x is used in a conditional, e.g., $x < x'$. In this case, we know that $x < x'$ at the true branch, and $x' \le x$ at the false branch.

The Support of Range Analysis on Integer Intervals. The SSA representation ensures that a new name is created at each program point where a variable is defined. To meet the

Instructions (I)	::=	
– Addition	|	$x_0 = x_1 + x_2$
– Subtraction	|	$x_0 = x_1 - n \parallel \langle x_2 = x_1 \rangle$
– ϕ-function	|	$x_0 = \phi(x_1 : \ell_1, \ldots, x_n : \ell_n)$
– Comparison	|	$(x_1 < x_2)? \begin{cases} \ell_t : \langle x_{1t}, x_{2t} \rangle \\ \ell_f : \langle x_{1f}, x_{2f} \rangle \end{cases}$

Figure 5. The syntax of our intermediate language.

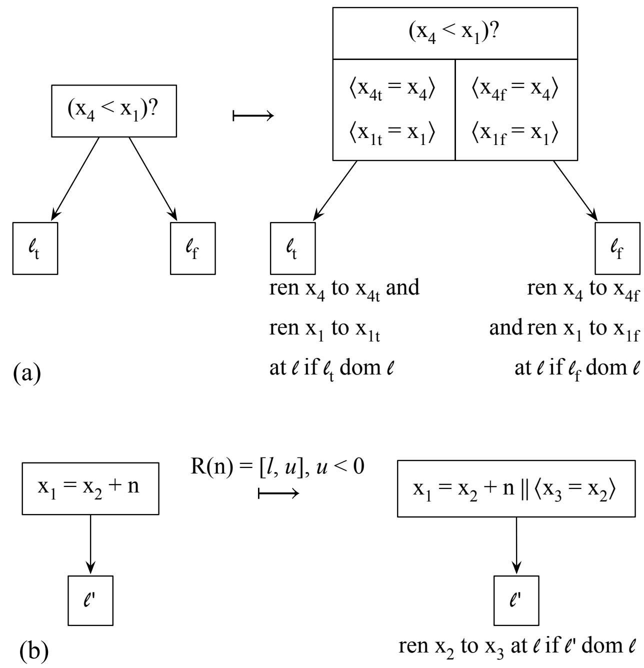

Figure 6. Transformation rules used to convert the syntax in Figure 3 into the syntax in Figure 5 (ren stands for "rename").

other two requirements, we split live ranges at subtractions and after conditionals. Going back to Figure 3, we see that our core language contains only syntax for arithmetic additions. However, we can use range analysis to know that one, or the two, terms of an addition are negative. Range analysis [10] is a static dataflow analysis that associates each variable x to an interval $R(x) = [l, u], \{l, u\} \subset \mathbb{N}, l \leq u$. The efficient and precise computation of range analysis has been researched extensively in the literature, and we shall not discuss it further. In our experiments, we have used the implementation of Rodrigues *et al.* [30]. Given $x_1 = x_2 + x_3$, where $R(x_2) = [l_2, u_2]$ and $R(x_3) = [l_3, u_3]$, we have a subtraction if $u_3 < 0$ or $u_2 < 0$. If both variables have positive ranges, then we have an addition. Otherwise, we have an *unknown* instruction, which shall not generate constraints.

Our live range splitting strategy leads to the creation of a different program representation. Figure 5 shows the instructions that constitute the new language. Figure 6 shows the two syntactic transformations that convert a program written in the syntax of Figure 3 into a program written in the syntax of Figure 5. We let $x_0 = x_1 - n \parallel \langle x_2 = x_1 \rangle$ denote a composition of two statements, $x_0 = x_1 - n$ and $x_2 = x_1$. The second instruction splits the live range of x_1. Both statements happen in parallel. Thus, $x_0 = x_1 - n \parallel \langle x_2 = x_1 \rangle$ does not represent an actual assembly instruction; it is only used for notational convenience. Similarly, when transforming conditional tests, we let $\langle x_{1t} = x_1, x_{2t} = x_2 \rangle$ denote two copies that happen in parallel: $x_{1t} = x_1$, and $x_{2t} = x_2$. Whenever there is no risk of ambiguity, we write simply $\langle x_{1t}, x_{2t} \rangle$, as in Figure 5. Parallel copies and ϕ-functions are removed before code generation, after the analyses that require them have already run. This step is typically called *SSA-Elimination phase*.

EXAMPLE 3.3. *Figure 7 shows the result of applying the rules seen in Figure 6 onto the program in Figure 4.*

3.3 Constraint Generation

Once we have a suitable program representation, we use the rules in Figure 8 to generate constraints. These constraints determine the less-than set of variables. Constraint generation is $O(|\mathcal{V}|)$, where $\mathcal{V}$ is the set of variables in the target program. We have four kinds of constraints:

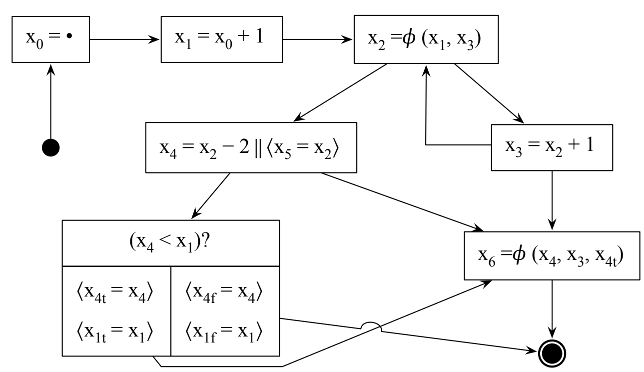

Figure 7. Figure 4 transformed by the rules in Figure 6.

init: Set the less-than set of a variable to empty, e.g., $\mathsf{LT}(x) = \emptyset$.

union: Set the less-than set of a variable to be the union of another less-than set and a single element, e.g., $\mathsf{LT}(x_3) = \{x_1\} \cup \mathsf{LT}(x_2)$.

inter: Set the less-than set of a variable to be the intersection of multiple less-than sets, e.g., $\mathsf{LT}(x) = \mathsf{LT}(x_1) \cap \mathsf{LT}(x_2)$.

copy: Sets the less-than set of a variable to be the less-than set of another variable, e.g., $\mathsf{LT}(x) = \mathsf{LT}(x')$.

EXAMPLE 3.4. *The rules in Figure 8 produce the following constraints for the program in Figure 7: $LT(x_0) = \emptyset$, $LT(x_1) = \{x_0\} \cup LT(x_0)$, $LT(x_2) = LT(x_1) \cap LT(x_3)$,*

$$x = \bullet \rightsquigarrow_1 \mathsf{LT}(x) = \emptyset$$

$$x_1 = x_2 + n \rightsquigarrow_2 \mathsf{LT}(x_1) = \{x_2\} \cup \mathsf{LT}(x_2)$$

$$x_1 = x_2 - n \parallel \langle x_3 = x_2 \rangle \rightsquigarrow_3 \begin{cases} \mathsf{LT}(x_3) = \{x_1\} \cup \mathsf{LT}(x_2) \\ \mathsf{LT}(x_1) = \emptyset \end{cases}$$

$$x = \phi(x_1, \ldots, x_n) \rightsquigarrow_4 \mathsf{LT}(x) = \mathsf{LT}(x_1) \cap \ldots \cap \mathsf{LT}(x_n)$$

$$(x_1 < x_2)? \begin{cases} \ell_t : \langle x_{1t}, x_{2t} \rangle \\ \ell_f : \langle x_{1f}, x_{2f} \rangle \end{cases} \rightsquigarrow_5 \begin{cases} \mathsf{LT}(x_{2t}) = \\ \qquad \{x_{1t}\} \cup \mathsf{LT}(x_1) \\ \mathsf{LT}(x_{1f}) = \mathsf{LT}(x_2) \\ \mathsf{LT}(x_{1t}) = \mathsf{LT}(x_1) \\ \mathsf{LT}(x_{2f}) = \mathsf{LT}(x_2) \end{cases}$$

Figure 8. Constraint generation rules. Numbers labelling each rule are used in the proofs of Section 3.5. n is a variable such that $R(n) = [l, u], l > 0$.

$LT(x_3) = \{x_2\} \cup LT(x_2)$, $LT(x_4) = \emptyset$, $LT(x_5) = \{x_4\} \cup LT(x_2)$, $LT(x_{1t}) = \{x_{4t}\} \cup LT(x_{4t})$, $LT(x_{1f}) = LT(x_1)$, $LT(x_{4f}) = LT(x_1)$, $LT(x_{4t}) = LT(x_4)$, $LT(x_6) = LT(x_3) \cap LT(x_{4f}) \cap LT(x_5)$.

3.4 Constraint Solving

Constraints are solved via a worklist algorithm. We initialize $\mathsf{LT}(x)$ to $\mathcal{V}$, for every variable x. During the resolution process, elements are removed from each LT, until a fixed point is achieved. Theorem 3.7, in Section 3.5, guarantees that this process terminates. Constraint solving is equivalent to finding transitive closures; thus, it is $O(|\mathcal{V}|^3)$. In practice, we have observed an $O(|\mathcal{V}|)$ behavior, as we show in Section 4.

EXAMPLE 3.5. *To solve the constraints in Example 3.4, we initialize every LT set to $\{x_0, x_1, x_2, x_3, x_4, x_6, x_{1f}, x_{1t}, x_{4f}, x_{4t}\}$, i.e., the set of program variables. Chaotic iterations on those constraints achieves the following fixed point: $LT(x_0) = LT(x_4) = LT(x_{4t}) = \emptyset$; $LT(x_1) = LT(x_2) = LT(x_{4f}) = LT(x_{1f}) = LT(x_6) = \{x_0\}$; $LT(x_3) = \{x_0, x_2\}$; $LT(x_5) = \{x_0, x_4\}$; and $LT(x_{1t}) = \{x_{4t}\}$.*

3.5 Properties

There are a number of properties that we can prove about our dataflow analysis. In this section we focus on two core properties: *termination* and *adequacy*. Termination ensures that the constraint solving approach of Section 3.4 always reaches a fixed point. Adequacy ensures that our analysis conforms to the semantics of programs. To show termination, we start by proving Lemma 3.6.

LEMMA 3.6 (Decreasing). *If constraint resolution starts with $LT(x) = \mathcal{V}$ for every x, then $LT(x)$ is monotonically decreasing or stationary.*

Proof: The proof follows from a case analysis on each constraint produced in Figure 8, plus induction on the number of elements in LT. Figure 8 reveals that we have only three kinds of constraints:

- $\mathsf{LT}(x) = \emptyset$: in this case, $\mathsf{LT}(x)$ is stationary;
- $\mathsf{LT}(x) = \{x'\} \cup \mathsf{LT}(x'')$: by induction, $\mathsf{LT}(x'')$ is decreasing or stationary, and $\mathsf{LT}(x)$ always contains $\{x'\}$;
- $\mathsf{LT}(x) = \mathsf{LT}(x_1) \cap \ldots \cap \mathsf{LT}(x_n)$: we apply induction on each $\mathsf{LT}(x_i), 1 \leq i \leq n$. $\square$

To prove Theorem 3.7, which states termination, we need to recall $\mathcal{P}^{\mathcal{V}}$, the semi-lattice that underlines our less-than analysis. $\mathcal{P}^{\mathcal{V}} = \{\mathcal{V}, \cap, \bot = \emptyset, \top = \mathcal{V}, \subseteq\}$ is the lattice formed by the partially ordered set of program variables. Ordering is given by subset inclusion $\subseteq$. The meet operator (greatest lower bound) is set intersection $\cap$. The lowest element in this lattice is the empty set, and the highest is $\mathcal{V}$.

THEOREM 3.7 (Termination). *The constraint resolution process terminates.*

Proof: The proof of this theorem is the conjunction of two facts: (i) Constraint sets are monotonically decreasing; and (ii) they range on a finite lattice. Fact (i) follows from Lemma 3.6. Fact (ii) follows from the definition of $\mathcal{P}^{\mathcal{V}}$. $\square$

We followed the framework of Tavares *et al.* [37] to build our intermediate program representation. Thus, we get correctness for free, as we are splitting live ranges at every program point where new information can appear. Lemma 3.8 formalizes this notion.

LEMMA 3.8 (Sparcity). *$LT(x)$ is invariant along the live range of x.*

Proof: The proof follows from case analysis on the constraint generation rules in Figure 8. By matching constraints with the syntax that produce them, we find that the abstract state of a variable can only change at its definition point. This property is ensured by the live range splitting strategy that produces the program representation that we use. $\square$

We close this section showing adequacy. In a nutshell, we want to show that if our constraint system determines that a variable x belongs into the less-than set of another variable x', then we know that $x < x'$. This is a static notion: we consider the values of x and x' that exist at the same moment during the execution of a program. Theorem 3.9 formalizes this observation. The proof of that theorem requires the semantics of the transformed language, whose syntax appears in Figure 5. However, for the sake of space, we omit this semantics. We assume the existence of a small-step transition rule $\rightarrow$, which receives an instruction ι, plus a store environment Σ, and produces a new environment Σ'. The store maps variables to integer values. Analogously, the constraints of Figure 8 show how instructions change the abstract state LT. We write $\iota \vdash \mathsf{LT} \triangleright \mathsf{LT}'$ to denote an abstract transition. We let $\models$ model the following relation: if $x' \in \mathsf{LT}(x)$, then

$\Sigma(x') < \Sigma(x)$. If this relation is true for every element in the domain of Σ, then we write $\mathsf{LT} \models \Sigma$.

THEOREM 3.9 (Adequacy). *If $LT_1 \models \Sigma_1 \wedge \iota \vdash \Sigma_1 \rightarrow \Sigma_2 \wedge \iota \vdash LT_1 \triangleright LT_2$, then $LT_2 \models \Sigma_2$.*

Proof: The proof happens by case analysis on the five instructions in Figure 8. For brevity, we show two cases: First case: ι is $x_1 = x_2 + n$. Notice that $n > 0$ because otherwise our range analysis would have led us to transform that instruction into a subtraction, or would have produced no constraint at all. Henceforth, we shall write "$f \setminus a \mapsto b$" to denote function update, i.e.: "$\lambda x.$if $x = a$ then b else $f(x)$" We have that $x_1 = x_2 + n \vdash \Sigma_1 \rightarrow (\Sigma_1 \setminus x_1 \mapsto \Sigma_1(x_2) + n)$, and $x_1 = x_2 + n \vdash \mathsf{LT}_1 \triangleright (\mathsf{LT}_1 \setminus x_1 \mapsto \mathsf{LT}_1(x_2) \cup \{x_2\})$. We look into possible variables $x \in \mathsf{LT}_1 \setminus x_1 \mapsto \mathsf{LT}_1(x_2) \cup \{x_2\}$:

- $x = x_2$: the theorem is true because $\Sigma_1(x_2) + n > \Sigma_1(x_2)$;
- $x \in \mathsf{LT}(x_2)$: From the hypothesis $\mathsf{LT}_1 \models \Sigma_1$, we have that $x < x_2$. We know that $x_2 < x_1$; thus, by transitivity, $x < x_1$.

Second case: ι is a ϕ-function. Then, $x = \phi(x_1, \ldots, x_n) \vdash \Sigma_1 \rightarrow \Sigma_1 \setminus x \mapsto \Sigma_1(x_i)$, for some i, $1 \leq i \leq n$, depending on the program's dynamic control flow. From Figure 8, we have that $\mathsf{LT}_2 = \mathsf{LT}_1 \setminus x \mapsto \mathsf{LT}_1(x_1) \cap \ldots \cap \mathsf{LT}_1(x_n)$. Thus, any $x' \in \mathsf{LT}_2(x)$ is such that $x' \in LT_1(x_i)$. By the hypothesis, $x' < x_i$. By the semantics of ϕ-functions, $x = x_i$; hence, $x' < x$. $\square$

COROLLARY 3.10 (Invariance). *Let x_i and x_j be two variables simultaneously alive. If $x_i \in LT(x_j)$, then $x_i < x_j$.*

Proof: In a SSA-form program, if two variables interfere, then one is alive at the definition point of the other [8, 14, 45]. From Lemma 3.8, we know that $\mathsf{LT}(x)$ is constant along the live range of x. Theorem 3.9 gives us that this property holds at the definition point of the variables. $\square$

3.6 Pointer Disambiguation

Pointers, in low-level languages, are used in conjunction with integer offsets to refer to specific memory locations. The combination of a base pointer plus an offset produces what we call a *derived pointer*. The less-than check that we have discussed in this paper lets us compare pointers directly, if they are bound to a less-than relation, or indirectly, if they are derived from a common base. This observation lets us state the disambiguation criteria below:

DEFINITION 3.11 (Pointer Disambiguation Criteria). *Let p, p_1 and p_2 be variables of pointer type, and x_1 and x_2 be variables of arithmetic type. We consider two disambiguation criteria:*

1. *Memory locations p_1 and p_2 will not alias if $p_1 \in LT(p_2)$ or $p_2 \in LT(p_1)$.*
2. *Memory locations $p_1 = p + x_1$ and $p_2 = p + x_2$ will not alias if $x_1 \in LT(x_2)$ or $x_2 \in LT(x_1)$.*

The C standard refers to arithmetic types and pointer types collectively as scalar types [18]{§6.2.5.21}. Notice that the less-than analysis that we have discussed thus far works seamlessly for scalars; thus, it also builds relations between pointers. For instance, the common idiom "for(int* pi = p; pi < pe; pi++);" gives us that pi < pe inside the loop. This fact justifies Definition 3.11(1). Along similar lines, if $p_1 = p + x_1$, we have that $p \in \mathsf{LT}(p_1)$; thus, Definition 3.11(2) lets us disambiguate a base pointer from its non-null offsets, e.g., $p \neq p + n$, if we know that $n \neq 0$.

Definition 3.11 provides one, among several criteria, that can be used to disambiguate pointers. For instance, the C standard says that pointers of different types cannot alias. Aliasing is also impossible in well-defined programs between references derived from non-aliased base pointers. Additionally, derived pointers whose offsets have non-overlapping ranges cannot alias, as discussed in previous work [4, 27, 31]. Thus, our analysis says nothing about `p1` and `p2` in scenarios as simple as: `p1 = malloc(); p2 = malloc()`, or `p1 = p + 1; p2 = p + 2`. Previous work is already able to disambiguate `p1` and `p2` in both cases. Our pointer disambiguation criterion does not compete against these other approaches. Rather, as we further explain in Section 5, it complements them.

4. Evaluation

To demonstrate that a "less-than" check can be effective and useful to disambiguate pointers, we have implemented an inter-procedural, context insensitive version of the analysis described in this paper in LLVM version 3.7. In this section we shall answer three research questions to evaluate the precision, the scalability and the applicability of our approach:

Precision: how effective are strict inequalities to disambiguate pairs of pointers?

Scalability: can the analysis described in this paper scale up to handle very large programs?

Applicability: can our pointer disambiguation method increase the effectiveness of existing program analyses?

In the rest of his section we provide answers to these questions. Our discussion starts with precision.

4.1 Precision

The precision of an alias analysis method is usually measured as the capacity of said method to indicate that two given pairs of pointers do not alias each other. To measure the precision of our method, we compare it against the techniques already in place in the LLVM compiler. Our metric is the percentage of improvement our algorithm adds on top of of LLVM's basic disambiguation technique, the `basic-aa` algorithm. Henceforth, we shall refer to it as **BA**. This analysis uses several heuristics to disambiguate pointers, relying mostly on the fact that pointers derived from different allocation sites cannot alias in well-formed programs.

In addition to the basic algorithm, LLVM contains three other alias analysis, whose results we shall not use, for they have been able to resolve a very low number of queries in our experiments. These algorithms are `tb-aa`, which reports as non-aliases pointers of different types, `globalsmodref-aa`, which reports as non-aliases global variables whose address have never been taken, and `scev-aa`, which uses a form of value sets to disambiguate pointers [4]. We chose to omit `scev-aa` because we have not been able to get results for all our benchmarks using the implementation available in LLVM 3.7. Paisante *et al.* have evaluated it before, and, on average, it solves less than 5% of all the queries [27].

Figure 9 shows the results of the three alias analyses when applied on the 100 largest benchmarks in the LLVM test suite. We have removed the benchmark TSVC from this lot, because its 36 programs were giving us the same numbers. This fact occurs because they use a common code base. Our method rarely disambiguates more pairs of pointers than **BA**. Such result is expected: most of the queries consist of pairs of pointers derived from different memory allocation sites, which **BA** disambiguates, and we do not analyze. The ISO C Standard prohibits comparisons between two references to separately allocated objects [18]{§6.5.8p5}, even though they are used in practice [22, p.4].

Nevertheless, Figure 9 still lets us draw encouraging conclusions. There exist many queries that we can solve, but **BA** cannot. For the entire LLVM test suite, our analysis that we shall refer to it as **LT**, increases the precision of **BA** by 9.49% (56,192,064 vs 59,184,181 no-alias responses). Yet, in programs that make heavy use of pointer arithmetics, our results are even more impressive. For instance, in SPEC's `lbm` we disambiguate 11,881 pairs of pointers, whereas **BA** provides precise answers to only 1,888. And, even in situations where **LT** falls behind **BA**, the former can increase the precision of the latter non-trivially. As an example, in SPEC's `gobmk`, **LT** returns 852,368 "no-alias" answers and **BA** 1,705,559. Yet, these sets are mostly disjoint: the combination of both analyses solves 2,274,936 queries: an increase of 34% over **BA**. Figure 10 summarizes these results for SPEC CINT 2006.

How do we compare against Andersen's analysis? Andersen's [3] inclusion-based alias analysis is the quintessential pointer disambiguation technique. At the time of this writing, the most up-to-date version of LLVM did not contain an implementation of such technique. However, there exists one algorithm available in LLVM 4.0, which is still experimental. We shall call it **CF**, because it uses context free languages (CFL) to model the inclusion-based resolution of constraints, as proposed by Zheng and Rugina [46], and by Zhang *et al.* [44]. Jia Chen [9] provides a detailed description of the LLVM implementation[1].

[1] Available at https://github.com/grievejia/andersen

Figure 11 compares our analysis and Andersen's. Our numbers have been obtained in LLVM 3.7, whereas **CF**'s has been produced via LLVM 4.0. We emphasize that both versions of this compiler produce exactly the same number of alias queries, and, more importantly, **BA** outputs exactly the same answers in both cases. This experiment reveals that there is no clear winner in this alias analysis context. **BA+LT** is more than 20% more precise than **BA+CF** in three benchmarks: `lbm`, `milc` and `gobmk`. **BA+CF**, in turn, is three times more precise in `omnetpp`. The main conclusions that we draw from this comparison are the following: (i) these analysis are complementary; and (ii) mainstream compilers still miss opportunities to disambiguate alias queries.

4.2 Scalability

We claim that the less-than analysis that we introduce in this paper presents – in practice – linear complexity on the size of the target program. Size is measured as the number of instructions present in the intermediate representation of said program. In this section we provide evidence that corroborates this claim. Figure 12 relates the number of constraints that we produce for a program, using the rules in Figure 8, with the number of instructions in that program. The strong linear relation between these two quantities is visually apparent in Figure 12. And, going beyond visual clues, the coefficient of determination (R^2) between constraints and instructions is 0.992. The closer to 1.0 is R^2, the stronger the evidence of a linear behavior.

As Figure 12 shows, the number of constraints that we produce is linearly proportional to the number of instructions that these constraints represent. But, what about the time to solve such constraints – is it also linear on the number of instructions? To solve constraints, we compute the transitive closure of the "less-than" relation between program variables. We use a cubic algorithm to build the transitive closure [25]. When fed with our benchmarks, this algorithm is likely to show linear behavior: the coefficient of determination between the number of constraints for all our benchmarks, and the runtime of our analysis is 0.988. This linearity surfaces in practice because most of the constraints enter the worklist at most twice. For instance, SPEC CPU 2006, plus the 308 programs that are part of the LLVM test suite give us 8,392,822 constraints to solve. For this lot, we pop the worklist 17,800,102 times: a ratio that indicates that each constraint is visited 2.12 times until a fixed point is achieved.

We emphasize that our implementation still bears the status of a research prototype. Its runtime is far from being competitive, because, currently, it relies heavily on C++'s standard data-structures, instead of using data-types more customized to do static analyses. We use `std::set` to represent LT sets, `std::map` to bind LT sets to variables, and `std::vector` to implement the worklist. Therefore, our implementation still has much room to improve in terms of runtime. For instance, we took two hours and twelve minutes to solve all the less-than relations between all the scalar

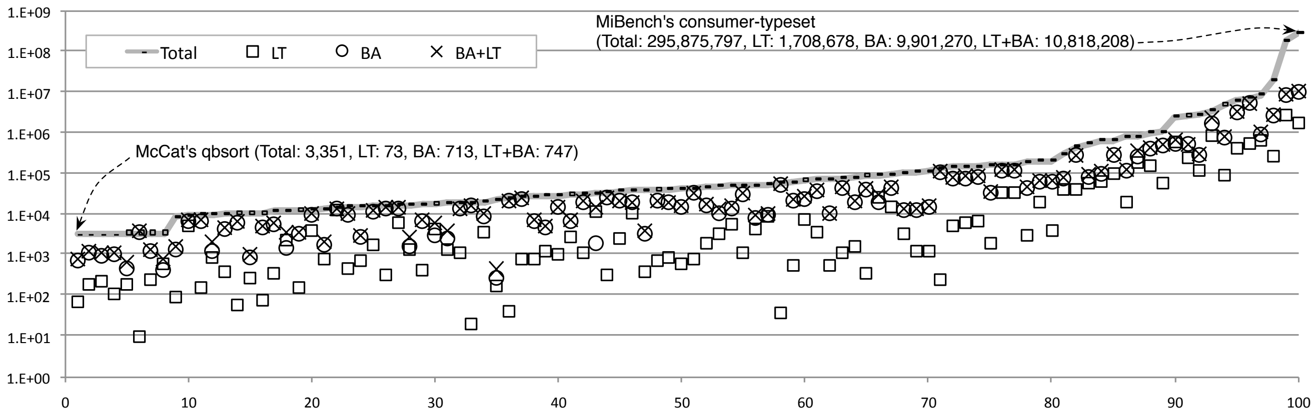

Figure 9. Effectiveness of our alias analysis (**LT**), when compared to LLVM's basic alias analysis on the 100 largest benchmarks in the LLVM test suite. Each tick in the X-axis represents one benchmark. The Y-axis represents total number of queries (one query per pair of pointers), and number of queries in which each algorithm got a "no-alias" response.

Benchmark	# Queries	BA	LT	BA + LT
lbm	31,944	5.90%	10.15%	**15.74%**
mcf	49,133	15.28%	8.95%	16.52%
astar	95,098	45.54%	16.05%	47.66%
libq	146,301	51.64%	3.45%	52.67%
sjeng	428,082	70.64%	2.03%	71.64%
milc	808,471	31.05%	23.90%	**43.88%**
soplex	1,787,190	21.43%	12.48%	23.53%
bzip2	2,472,234	21.48%	23.09%	**26.70%**
hmmer	2,574,217	8.79%	4.48%	**9.38%**
gobmk	3,492,577	48.49%	22.91%	**63.33%**
namd	3,685,838	22.59%	0.93%	22.76%
omnetpp	12,943,554	18.71%	0.46%	18.81%
h264ref	20,068,605	12.86%	1.29%	13.16%
perl	23,849,576	9.92%	3.87%	10.19%
dealII	37,779,455	75.05%	20.21%	75.46%
gcc	186,008,992	4.26%	1.47%	4.65%

Figure 10. Comparison between three alias analyses in SPEC CINT 2006. "# Queries" is the total number of queries performed when testing a given benchmark. Percentages show the ratio of queries that yield"no-alias", given a certain alias analysis. The higher the percentage, the more precise is the pointer disambiguation method. We have highlighted the cases in which our less-than check has increased by 10% or higher the precision of LLVM's basic alias analysis.

variables found in the 16 programs of SPEC CPU that the LLVM's C frontend compiles on an 2.4GHz Intel Core i7. We have already observed that most of the LT sets end up empty, and that the vast majority of them, over 95%, contain only two or less elements. We intend to use such observations to improve the runtime of our analysis as future work.

4.3 Applicability

One way to measure the applicability of an alias analysis is to probe how it improves the quality of some compiler optimization, or the precision of other static analyses. In this work, we have opted to follow the second road, and show how our new alias analysis improves the construction of the Program Dependence Graph (PDG), a classic data structure introduced by Ferrante *et al.* [13]. We use the implementation of PDGs available in the FlowTracker system [29], which has a distribution for LLVM 3.7[2]. The PDG is a graph whose vertices represent program variables and memory locations, and the edges represent dependences between these entities. An instruction such as $a[i] = b$ creates a data dependence edge from b to the memory node $a[i]$. The more memory nodes the PDG contains, the more precise it is, because if two locations alias, they fall into the same node. In the absence of any alias information, the PDG contains at most one memory node; perfect alias information yields one memory node for each independent location in the program.

It is not straightforward to compare LLVM's basic alias analysis against our less-than-based analysis, because the former is intra-procedural, whereas the latter is inter-procedural. Therefore, **BA** ends up creating at least one memory node per function that contains a load or store operation present in the target program. **LT**, on the contrary, joins nodes that exist in the scope of different functions if it cannot prove that they do not overlap. In order to circumvent this shortcoming, we decided to use Csmith [42][3]. Csmith produces random C programs that conform to the C99 standard, using an assortment of techniques, with the goal to find bugs in compilers. Csmith has one important advantage to us: we can tune it to produce programs with a single function, in

[2] Available at http://cuda.dcc.ufmg.br/flowtracker/

[3] Available at https://embed.cs.utah.edu/csmith/

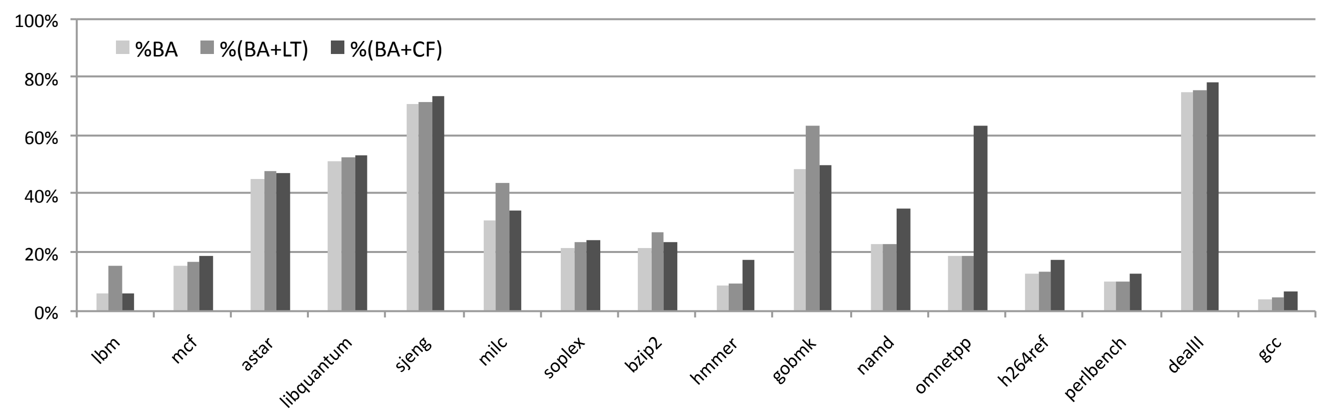

Figure 11. How two different alias analysis (**LT** and **CF**) increase the capacity of LLVM's basic alias analysis (**BA**) to disambiguate pointers. The Y-axis shows the percentage of no-alias responses. The higher the bar, the better.

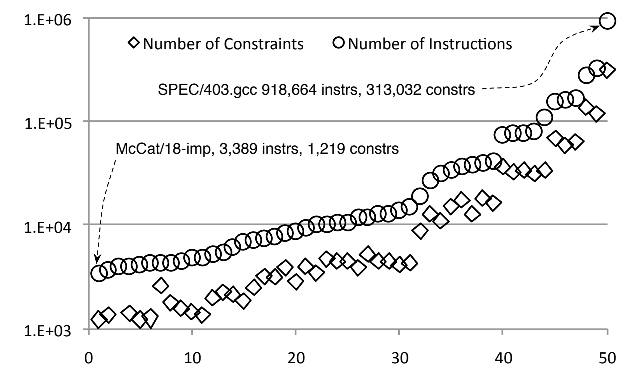

Figure 12. Comparison between the number of instructions and the number of constraints that we produce (using rules in Figure 8) per benchmark. X-axis represents benchmarks, sorted by number of instructions. The coefficient of determination (R^2) between these two metrics is 0.992, indicating a strong linear correlation.

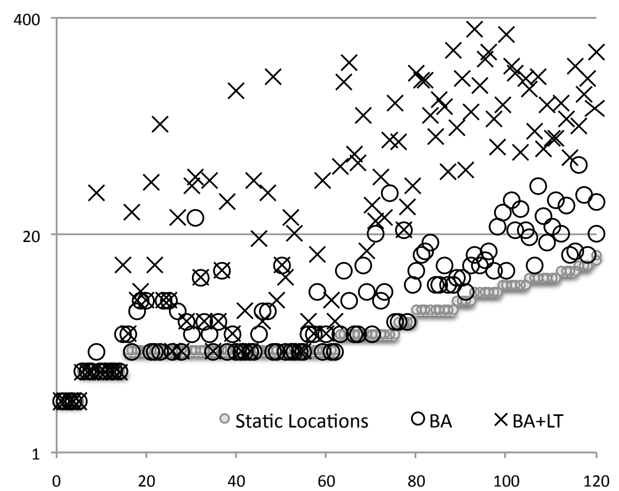

Figure 13. Precision of dependence graph. The X-axis shows benchmarks, sorted by number of static memory references in the source code. Y-axis shows number of memory nodes in the Program Dependence Graph. The more memory nodes the PDG contains, the more precise it is.

addition to the ever present `main` routine. By varying the seed of its random number generator, we obtain programs of various sizes, and by varying the maximum nesting depth of pointers, we obtain a rich diversity of dependence graphs. Figure 13 shows the results that we got in this experiment.

Our alias analysis improves substantially the precision of LLVM's **BA**. We have produced 120 random programs, whose size vary from 50 to 4,030 lines. In total, the 120 PDGs produced with **BA** contain 1,299 memory nodes. When combined, **BA** and **LT** yield 8,114 nodes, an increase of 6.23x. We are much more precise than **BA** because the programs that Csmith produces do not read input values: they use constants instead. Because almost every memory indexing expression is formed by constants known at compilation time, **LT** can distinguish most of them. Although artificial, this experiment reveals a striking inability of LLVM's current alias analyses to deal with pointer arithmetics. None of the other alias analyses available in LLVM 3.7 are able to increase the precision of **BA** – not even marginally. Although we have not used **CF** in this experiment – it is not available for LLVM 3.7 – we speculate, from reading its source code, that it will not be able to change this scenario.

Notice that our results do not depend on the nesting depth of pointers. Our 120 benchmarks contain 6 categories of programs, which we produced by varying the nesting depth of pointers from 2 to 7 levels. Thus, we had 20 programs in each category. A pointer to `int` of nesting depth 3, for instance, is declared as `int***`. All these programs, regardless of their category, present an average of six static memory allocation sites. On average, **BA** produces PDGs with 11 memory nodes, independent on the bucket, and **BA+LT** produce

PDGs with 68. The greater the number of static memory allocation sites, the better the results of both **BA** and **LT**. The largest PDG observed with **BA** only has 52 memory nodes (and 88 nodes if we augment **BA** with **LT**). The largest graph produced by the combination of **BA** and **LT** has 342 nodes (and only 15 nodes if we use **BA** without **LT**).

5. Related Work

The insight of using a less-than dataflow analysis to disambiguate pointers is an original contribution of this paper. However, such a static analysis is not new, having been used before to eliminate array bound checks. We know of two different approaches to build less-than relations: Logozzo's [20, 21] and Bodik's [7]. Additionally, there exist non-relational analyses that produce enough information to solve less-than equations [11, 23]. In the rest of this section we discuss the differences between such work and ours.

The ABCD Algorithm. The work that most closely resembles ours is Bodik *et al.*'s ABCD (short for Array Bounds Checks on Demand) algorithm [7]. Similarities stem from the fact that Bodik *et al.* also build a new program representation to achieve a sparse less-than analysis. However, there are five key differences between that approach and ours. The first difference is a matter of presentation: Bodik *et al.* provide a geometric interpretation to the problem of building less-than relations, whereas we adopt an algebraic formalization. Bodik *et al.* keep track of such relations via a data-structure called the *inequality graph*. This graph is implicit in our approach: it appears if we create a vertex v_i to represent each program variable x_i, and add a weighted edge from v_1 to v_2 if, and only if, $v_1 \in \mathsf{LT}(v_2)$. The weight of an edge is the difference $v_2 - v_1$, whenever known statically. The other four differences are more fundamental.

Bodik *et al.* use a different algorithm to prove that a variable is less than another. In the absence of cycles in the inequality graph, their approach works like ours: a positive path between v_i to v_j indicates that $x_i < x_j$. This path is implicit in the transitive closure that we produce after solving constraints. However, they use an extra step to handle cycles, which, in our opinion, makes their algorithm difficult to reason about. Upon finding a cycle in the inequality graph, Bodik *et al.* try to mark this cycle as *increasing* or *decreasing*. Cycles always exist due to ϕ-functions. Decreasing cycles cause ϕ-functions to be abstractly evaluated with the *minimum* operator applied on the weights of incoming edges; increasing cycles invoke *maximum* instead. Third, Bodik *et al.* do not use range analysis. This is understandable, because ABCD has been designed for just-in-time compilers, where runtime is an issue. Nevertheless, this limitation prevents ABCD from handling instructions such as $x_1 = x_2 + x_3$ if neither x_2 nor x_3 are constants. Fourth, Bodik *et al.*'s program representation does not split the live range of x_3 at an instruction such as $x_1 = x_2 - x_3, x_3 > 0$. This implementation detail lets us know that $x_2 > x_1$. Finally, we chose to compute a transitive closure of less-than relations, whereas ABCD works on demand. This point is a technicality. In our experiments, we had to deal with millions of queries. If we tried to answer them on demand, like ABCD does, then said experiments would take too long. We build the transitive closure to answer queries in $O(1)$.

The Pentagon Lattice. Logozzo and Fähndrich have proposed the Pentagon Lattice to eliminate array bound checks in type safe languages such as C#. This algebraic object is the combination of the lattice of integer intervals and the less-than lattice. Pentagons, like the ABCD algorithm, could be used to disambiguate pointers like we do. Nevertheless, there are differences between our algorithm and Logozzo's. First, the original work on Pentagons describe a dense analysis, whereas we use a different program representation to achieve sparsity. Contrary to ABCD, the Pentagon analysis infers that $x_2 > x_1$ given $x_1 = x_2 - x_3, x_3 > 0$ like we do, albeit on a dense fashion. Second, Logozzo and Fähndrich build less-than and range relations together, whereas our analysis first builds range information, then uses it to compute less-than relations. We have not found thus far examples in which one approach yields better results than the other; however, we believe that, from an engineering point of view, decoupling both analyses leads to simpler implementations.

Fully-Relational Analyses. Our less-than analysis, ABCD and Pentagons are said to be *semi-relational*, meaning that they associate single program variables with sets of other variables. Fully-relational analysis, such as Octogons [23] or Polyhedrons [11], associate tuples of variables with abstract information. For instance, Miné's Octogons build relations such as $x_1 + x_2 \leq 1$, where x_1 and x_2 are variables in the target program. As an example, Polly-LLVM [4] uses fully-relational techniques to analyze loops. Polly's dependence analysis is able to distinguish `v[i]` and `v[j]` in Figure 1 (a), given that $\mathsf{j} - \mathsf{i} \geq 1$; however, it cannot analyze `v[i]` and `v[j]` in Figure 1 (b). These analyses are very powerful; however, they face scalability problems when dealing with large programs. Whereas a semi-relational sparse analysis generates $O(|\mathcal{V}|)$ constraints, $|\mathcal{V}|$ being the number of program variables, a relational one might produce $O(|\mathcal{V}|^k)$, k being the number of variables used in relations.

Range-Based Alias Analyses There exist several different pointer disambiguation strategies that associate ranges with pointers [2, 4, 5, 24, 27, 31, 32, 35, 38, 40]. They all share a common idea: two memory addresses $p_1 + [l_1, u_1]$ and $p_2 + [l_2, u_2]$ do not alias if the intervals $[p_1 + l_1, p_1 + u_1]$ and $[p_2 + l_2, p_2 + u_2]$ do not overlap. These analyses differ in the way they represent intervals, e.g., with holes [4, 35] or contiguously [2, 32]; with symbolic bounds [24, 27, 31] or with numeric bounds [4, 5, 35], etc. None of these previous work is strictly better than ours. For instance, none of them can disambiguate `v[i]` and `v[j]` in Figure 1 (b), because

[4] Available at http://polly.llvm.org/

these locations cover regions that overlap, albeit not at the same time. Nevertheless, range based disambiguation methods can solve queries that our less-than approach cannot. As an example, we are unable to disambiguate p_1 and p_2, given these definitions: $p_1 = p + 1$ and $p_2 = p + 2$. We know that $p < p_1$ and $p < p_2$, but we do not relate p_1 and p_2.

6. Conclusion

This paper has introduced a new technique to disambiguate pointers, which relies on a less-than analysis. Our new alias analysis uses the observation that if p_1 and p_2 are two pointers, such that $p_1 < p_2$, then they cannot alias. Even though this observation is obvious, it has not been used before as the cornerstone of an alias analysis. Testimony of this statement is the fact that our analysis has been able to improve the precision of LLVM's standard suite of pointer disambiguation techniques by a large factor in some benchmarks. There are several ways in which our idea can be further developed. One future avenue that is particularly appealing to us concerns speed. Currently, our research prototype can handle large programs, but its runtime is not practical: it takes several minutes to produce the transitive closure of the less-than relation for our largest benchmarks. We believe that better algorithms can improve this scenario substantially. The design of such algorithms is a problem that we leave open.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[2] P. Alves, F. Gruber, J. Doerfert, A. Lamprineas, T. Grosser, F. Rastello, and F. M. Q. a. Pereira. Runtime pointer disambiguation. In *OOPSLA*, pages 589–606. ACM, 2015.

[3] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[4] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *CC*, pages 5–23. Springer, 2004.

[5] G. Balatsouras and Y. Smaragdakis. Structure-sensitive points-to analysis for C and C++. In *SAS*, pages 84–104. Springer, 2016.

[6] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *POPL*, pages 29–41. ACM, 1979.

[7] R. Bodik, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM, 2000.

[8] Z. Budimlic, K. D. Cooper, T. J. Harvey, K. Kennedy, T. S. Oberg, and S. W. Reeves. Fast copy coalescing and live-range identification. In *PLDI*, pages 25–32. ACM, 2002.

[9] J. Chen. CFL alias analysis, 2016. Google's Summer of Code Report.

[10] 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.

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

[12] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

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

[14] S. Hack, D. Grund, and G. Goos. Register allocation for programs in SSA-form. In *CC*, pages 247–262. Springer-Verlag, 2006.

[15] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, pages 290–299. ACM, 2007.

[16] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *CGO*, pages 265–280, 2011.

[17] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, pages 54–61. ACM, 2001.

[18] ISO-Standard. 9899 - The C programming language, 2011.

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

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

[21] F. Logozzo and M. Fähndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. *Sci. Comput. Program.*, 75(9):796–807, 2010.

[22] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell. Into the depths of c: Elaborating the de facto standards. In *PLDI*, pages 1–15. ACM, 2016.

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

[24] H. Nazaré, I. Maffra, W. Santos, L. Barbosa, L. Gonnord, and F. M. Q. Pereira. Validation of memory accesses through symbolic analyses. In *OOPSLA*, pages 791–809. ACM, 2014.

[25] E. Nuutila. Efficient transitive closure computation in large digraphs. *Acta Polytechnica Scandinavia: Math. Comput. Eng.*, 74:1–124, 1995.

[26] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi. Selective context-sensitivity guided by impact pre-analysis. In *PLDI*, pages 475–484. ACM, 2014.

[27] V. Paisante, M. Maalej, L. Barbosa, L. Gonnord, and F. M. Quintão Pereira. Symbolic range analysis of pointers. In *CGO*, pages 171–181. ACM, 2016.

[28] F. M. Q. Pereira and D. Berlin. Wave propagation and deep propagation for pointer analysis. In *CGO*, pages 126–135. IEEE, 2009.

[29] B. Rodrigues, F. M. Quintão Pereira, and D. F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *CC*, pages 110–120. ACM, 2016.

[30] R. E. Rodrigues, V. H. S. Campos, and F. M. Q. Pereira. A fast and low overhead technique to secure programs against integer overflows. In *CGO*. ACM, 2013.

[31] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *TOPLAS*, 27(2):185–235, 2005.

[32] S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: Static and dynamic memory reference analysis. In *ICS*, pages 251–283. IEEE, 2002.

[33] T. C. Spillman. Exposing side-effects in a PL/i optimizing compiler. In *IFIP*, pages 376–381. Springer, 1971.

[34] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41. ACM, 1996.

[35] Y. Sui, X. Fan, H. Zhou, and J. Xue. Loop-oriented array- and field-sensitive pointer analysis for automatic SIMD vectorization. In *LCTES*, pages 41–51. ACM, 2016.

[36] R. Surendran, R. Barik, J. Zhao, and V. Sarkar. Inter-iteration scalar replacement using array SSA form. In *CC*, pages 40–60. Springer, 2014.

[37] A. L. C. Tavares, B. Boissinot, F. M. Q. Pereira, and F. Rastello. Parameterized construction of program representations for sparse dataflow analyses. In *Compiler Construction*, pages 2–21. Springer, 2014.

[38] R. A. van Engelen, J. Birch, Y. Shou, B. Walsh, and K. A. Gallivan. A unified framework for nonlinear dependence testing and symbolic analysis. In *ICS*, pages 106–115. ACM, 2004.

[39] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144. ACM, 2004.

[40] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for c programs. In *PLDI*, pages 1–12. ACM, 1995.

[41] M. Wolfe. *High Performance Compilers for Parallel Computing*. Adison-Wesley, 1st edition, 1996.

[42] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *PLDI*, pages 283–294. ACM, 2011.

[43] S. H. Yong and S. Horwitz. Pointer-range analysis. In *SAS*, pages 133–148. Springer, 2004.

[44] Q. Zhang, M. R. Lyu, H. Yuan, and Z. Su. Fast algorithms for dyck-cfl-reachability with applications to alias analysis. In *PLDI*, pages 435–446. ACM, 2013.

[45] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formal verification of ssa-based optimizations for llvm. In *PLDI*, pages 175–186. ACM, 2013.

[46] X. Zheng and R. Rugina. Demand-driven alias analysis for c. In *POPL*, pages 197–208. ACM, 2008.