# Inclusion-based pointer analysis using actors

Cosmin Radoi
University of Illinois
cos@illinois.edu

Semih Okur
University of Illinois
okur2@illinois.edu

## Abstract

Ubiquitous multicore computers and affordable cloud computation services provides a great opportunity for increasing the speed or precision of program analysis algorithms. We express an context-sensitive inclusion-based (Andersen-style) pointer analysis with on-the-fly call graph construction as an actor system. We implement the algorithm in Scala as a drop-in replacement of WALA's (T.J.Watson Libraries for Analysis) pointer analysis and allow for all the flexibility of the original library, i.e., highly configurable context sensitivity, pointer and instance abstraction, etc.

## 1. Introduction

Pointer analysis determines which memory locations are referenced by the pointers in the program, information fundamental to a large variety of program verification, optimization, and comprehension techniques [**?** ]. Precise pointer analysis is $\mathcal{NP}$-hard [**?** ] but there are well-researched algorithms that trade precision for better performance [**? ?** ].

Inclusion-based pointer analysis, introduced by Andersen [**?** ], is on the more precise end of the scale. It computes pointer information by solving a constraint graph with nodes being pointers in the program and edges inclusion relations between the sets of memory locations (i.e., objects in OOP languages) referenced to by the pointers. An initial constraint graph is constructed by parsing the program, with assignments generating inclusion relations and pointer dereferences generating special indirect constraints that can only be solved once the points-to information is known. Thus, pointer information is computed by iteratively solving the constraint graph and adding new inclusion relations from indirect references.

Subsequently, there has been a great deal of work improving the precision of the original algorithm by adding different types of context sensitivity [**? ?** ] or varying flow sensitivity [**? ?** ].

Pointer information is also fundamental to computing a precise call graph in languages with dynamic dispatch, as the set of possible target methods can be refined by knowing the possible instances pointed to by the a method's receiver. But pointer analysis can also gain precision and speed from a more precise call graph by not considering the effects of unreachable methods. This mutual dependency led to the development of on-the-fly pointer analysis algorithms [**?** ], which compute the call graph and the points-to graph simultaneously. This approach also allows for context-sensitive call graphs where each node is the abstraction of a method as it is invoked in a specific context.

All these improvements in precision come with a significant performance cost, which has been partially alleviated by a long series of significant enhancement to the original algorithm [**? ? ? ?** ]. . Still, flow-insensitive, context-sensitive analyses still fail to scale to programs larger than a few hundred thousands lines of code.

An alternate, less explored, path to better performance is parallelization. . . The first parallel implementation of an inclusion-based pointer analysis was presented by Méndez-Lojo, Mathew, and Pingali [**?** ]. They express the constraint solving problem as a graph rewrite problem and integrate in their solution the offline stage and the Hybrid Cycle Detection technique proposed by Hardekopf and Lin [**?** ]. The solution relies on a precomputed fixed call graph and is context-insensitive.They implemented their system using Galois [**?** ] and achieved a speedup varying between 1x and 3x when compared to an optimized sequential implementation.

We propose an alternate approach to parallelization that takes advantage of the affinity between graph rewriting and actor systems. We express the constraint graph as a graph of actors, with both pointers and instances (memory locations) represented as actors, and set inclusion and dereferencing edges represented by the "knows" relations between actors. Pointer dereferencing is solved by a two-step messaging scheme propagating instance actor addresses from the dereferenced pointer to the receiving one. Except for the "finalization" actor, the implementation does not rely on a cen-

$$
\begin{array}{rr}
\text{(variable)} & a, b, c \in \mathbb{V} \\
\text{(field)} & f \in \mathbb{F} \\
\text{(method identifier)} & m, n \in \mathbb{M} \\
\text{(class identifier)} & k \in \mathbb{K}
\end{array}
$$

$$
\begin{array}{rcl}
program & ::= & program \mid class; program \\
class & ::= & k' \; extends \; k \; \{ \; methods \; \} \mid k \; \{ \; methods \; \} \\
methods & ::= & m \; \{s\} \mid methods \; ; m \; \{s\} \\
s & ::= & a = \text{ new } k \mid a = b \\
& & \mid a = b.f \mid a.f = b \\
& & \mid a = b_0.m(b_1, b_2, ...) \\
& & \mid s; s
\end{array}
$$

Figure 1: Language syntax

tral actor so it could easily be distributed. Also, the solution is context-sensitive and computes the call graph on-the-fly.

We implemented the algorithm as a drop-in replacement of WALA's (T.J. Watson Libraries for Analysis) [**?** ] internal sequential pointer analysis and allow for all the flexibility present in the original implementation, i.e., n-CFA, n-object sensitivity [**?** ], configurable pointer and instance abstractions, etc. The implementation is in Scala and relies on the Akka actor framework [**?** ].

This paper makes the following contributions:

- an actor-model algorithm for computing flow-insensitive, context-sensitive, pointer information with on-the-fly call graph refinement. To the best of our knowledge, it is the first actor-model approach to pointer analysis, the first parallel approach that is capable of constructing the call graph on-the-fly, and the first approach that parallelizes the call graph construction and constraint generation
- a reasonably efficient and highly configurable implementation available as a drop-in replacement for the pointer analysis in WALA, a widely used analysis framework
- a preliminary performance evaluation

## 2. Background

In this section we first present how constraints are generated in an inclusion-based pointer analysis with on-the-fly call graph construction. For presentation purposes, we use the simple language shown in Figure 1 with its intuitive semantics - our implementation does handle the full Java language. The program starts from a predefined object and method.

In an inclusion-based pointer analysis, the instructions in the program generate a set-constraint graph with *base*, *simple*, and *complex* constraints [**? ?** ]. Method calls generate *interprocedural* constraints which our analysis handles in a context-sensitive manner using a variation of the *lam* constructor [**? ?** ] (Figure 3).

The graph's nodes represent pointers and are points-to sets, i.e. sets of abstract objects. The edges are different types of set constraint relations between the pointers. Each variable in the program has an associated pointer and an instantiated object belongs to the points-to set to which it is assigned (*base* constraint). Assignment between two pointers generates a *simple* constraint: the set of the righthand-side pointer is a subset of the lefthand-side pointer - hence the name of this particular pointer analysis.

Reading or writing a field generates a *complex* constraint. When reading field, a subset relation is added between the pointer representing the field of each object referenced by the read pointer and the assigned pointer. Similarly for writing a field. This means that complex constraints can only be solved after propagating objects through simple constraints and they, in turn, generate new simple constraints.

Method calls generate *interprocedural* constraints which add inclusion relations between the actual and formal parameters and between method formal return pointers and the assigned pointers. Contexts are understood in the pointer analysis sense, e.g., whether to clone a method on application is determined by its calling context. We abstract away context selection details in order to allow any kind of calling context sensitivity.

## 3. Actor-model algorithm

In this section we explain how the constraint generation and constraint solving can be modeled as an actor system. There are two main sources of parallelism that can be exploited:

1. Constraint solving, which has been explored by Méndez-Lojo et al [**?** ]. We give an alternate, actor-model solution.
2. Call graph construction and constraint generation. This source is, to the best of our knowledge, novel in both goals and approach.

The system is comprised of the following types of actors (symbol in parenthesis):

- pointer actor (circle)
- object actor (square)
- call graph node actor (pentagon)
- a singleton call graph actor (hexagon)

### 3.1 Constraint generation

Figure 4 illustrates the constraint generation process. A call graph node represents the evaluation of a method in a particular execution context. The call graph node actor interprets the methods' instructions by creating new actors and sending them messages about their constraints.

An object instantiation generates a points-to (p) relation, i.e. base constraint, between a pointer actor and a newly

| | |
|---|---|
| (variable) | $a, b, c \in \mathbb{V}$ |
| (pointer) | $\Gamma(a) \in \mathbb{P} = 2^{\mathbb{O}}$ |
| (ordered set of pointers) | $l_x \in List[\mathbb{P}]$ |
| (object) | $o \in \mathbb{O}$ |
| (field of an object) | $f : \mathbb{O} \to \mathbb{P}$ |
| (method identifier) | $m, n \in \mathbb{M}$ |
| (class identifier) | $k \in \mathbb{K}$ |
| (context) | $C \in \mathbb{C}$ |
| (context selector) | $call : \mathbb{C} \times \mathbb{O} \times \mathbb{M} \to \mathbb{C}$ |
| (instructions for a call) | $instrFor : \mathbb{C} \to$ "$methods$" |
| (start environment for a call) | $start : \mathbb{C} \to (\mathbb{V} \to \mathbb{P})$ |
| (formal parameter) | $param_\Gamma : \mathbb{C} \to List[\mathbb{P}]$ |
| (formal return) | $return_\Gamma : \mathbb{C} \to \mathbb{P}$ |
| (pointer environment) | $\Gamma \in \mathbb{V} \to \mathbb{P}$ |
| (subset) | $l_a \subseteq l_b \equiv \forall i = \overline{0...length(l_a)} \;\; l_a(i) \subseteq l_b(i)$ |

Figure 2: Domains

$\boxed{C, \Gamma \rhd s : \Gamma', K}$

$(base)$ $\quad C, \Gamma \rhd a = \text{ new } k : \Gamma[a \to \{\text{new } k\}], \emptyset$

$(simple)$ $\quad C, \Gamma \rhd a = b : \Gamma, \{\Gamma(a) \subseteq \Gamma(b)\}$

$(complex)$ $\quad C, \Gamma \rhd a = b.f : \Gamma, \{\forall o \in \Gamma(b) \;\; \Gamma(f(o)) \subseteq \Gamma(a)\}$
$\quad C, \Gamma \rhd a.f = b : \Gamma, \{\forall o \in \Gamma(a) \;\; \Gamma(a) \subseteq \Gamma(f(o))\}$

$(interprocedural)$
$$\frac{o \in \Gamma(b_0) \quad C' = call(C, o, m) \quad C', start(C') \rhd instrFor(C') : \Gamma', K \quad b^f = param(C') \quad r^f = return(C')}{C, \Gamma \rhd a = b_0.m(b_1, ...) : \Gamma', K \cup \{b \subseteq b^f\} \cup \{r^f \subseteq \Gamma(a)\}}$$

$(other)$
$$\frac{C, \Gamma \rhd s_1 : \Gamma', K_1 \quad C, \Gamma \rhd s_2 : \Gamma'', K_2}{C, \Gamma \rhd s_1; s_2 : \Gamma' \cup \Gamma'', K_1 \cup K_2}$$

Figure 3: Effects

created abstract object instance actor. An assignment between two pointers generates a subset (s) relation, i.e. a simple constraint, between the righthand-side and the lefthand-side operands. A field read generates a read-field relation, i.e. a type of complex constraint, from the dereferenced object (righthand-side) to the receiving pointer (lefthand-side). Similarly for a field write.

In this stage, a method call only informs the actual parameters that they are part of a particular method call in this node. Why this helps and how the actual invocation is realized is discussed in detail in Section 3.3.

### 3.2 Constraint solving

We solve the set constraint problem by computing a dynamic transitive closure over the constraint graph using actors. Figure 5 illustrates the process that is triggered when a pointer actor $b$ gains a points-to relation to an object $o$. For each subset-of constraint, $b$ propagates the new points-to constraint to the superset actor, $a$. When $b$ is the source of a read-field constraint, i.e. a field is read from the pointer, it notifies $o$ that its field needs to be a subset of the assigned pointer, $a$. When $b$ is the source of a write-field constraint, i.e. a field is written through the pointer, it notifies $o$ which, in turn, lets $a$ know the identity, address, of its field, $f$. Making objects first-class citizens increases the number of messages but it also increases the available parallelism in the system, while also conforming to the actor model restriction of having only immutable shared data.

### 3.3 On-the-fly call graph construction

A context sensitive pointer analysis that compute the call graph on the fly *clones* the target method for each distinct discovered context. Figure 6 illustrates this process. Each clone interprets all instructions in the method independently, allowing a great amount of parallelism.

What constitutes a distinct context can be configured by specifying a *call* method (see Fig. 2) that takes as arguments the current context, the receiver object and the called method. The context distinctiveness is a global property so parallelizing this process involves either performing redundant computation or having a single decision actor. We choose the latter route by creating a specialized actor, the "CG" actor, that has managing the call graph as a sole purpose.

Knowing the pointer to a method's actual parameters is not enough for determining its context. The context is directly dependent on the receiver object, both for determining the target of dynamic dispatch and for object-sensitive analyses. But the receiver abstract object (or objects) may only be available after constraint solving. We solved this problem by not trying to determine the target method and context when encountering a method call, but delay it to the pointer the required information is available. Thus, on interpreting a method call, the call graph node actor only notifies the receiver pointer that it is part of a particular method invocation in this particular node. Then, when the pointer discovers a new pointer object, it notifies the call graph actor. The call graph actor then determines whether the new object generates a new distinct context, and, if so, it spawns a new call graph node. Also, it sends the caller call graph node the pointers for the formal parameters and return pointers of the callee.

## 4. Evaluation

The primary purpose of the evaluation is to determine whether our approach is scalable. We implemented the analysis in Scala using the Akka actor framework and we evaluate using a machine with 10 Intel Xeon E7-4860 processors and 132GB of RAM.

For our preliminary evaluation we use five Java applications ranging from 1k to 300k source lines of code (Fig. 7). We ran the pointer analysis in 0-CFA mode for all applications and in 1-CFA mode for two of them (there was no particular reason for this choice - it is just where we left off). Our preliminary results are shown in Figure 8.

## 5. Future work

Our preliminary evaluation indicates the approach is feasible. The current implementation, when run on a single processor, is still roughly one order of magnitude slower than the reference WALA implementation. We believe there are two sources for this difference. First, we haven't taken great care in the choices of data structures. There might be many small changes (e.g., using lists instead of sets when insertion of new elements in more common than addition, bundling messages, etc.) that might make a big difference in results. Second, we have not yet implemented any cycle detection algorithm for the constraint solving part. A primary candidate would be Hybrid Cycle Detection, which is shown to be effective [**?** ] and highly parallelizable [**?** ].

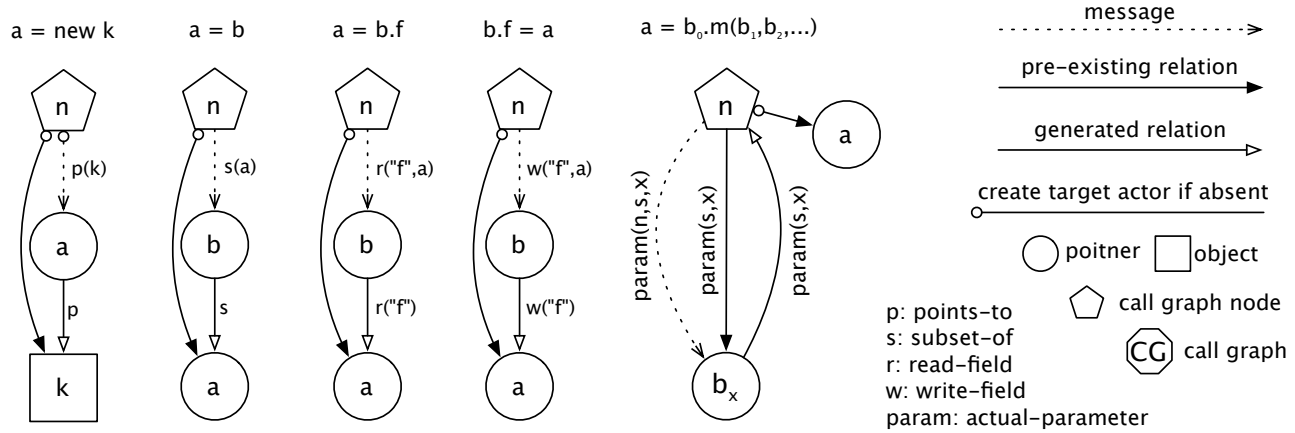


Figure 4: Constraint generation.

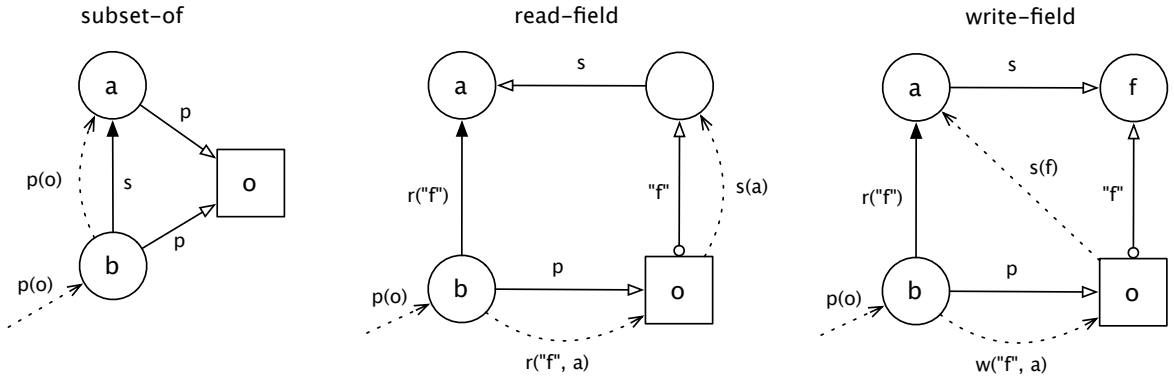


Figure 5: Constraint solving

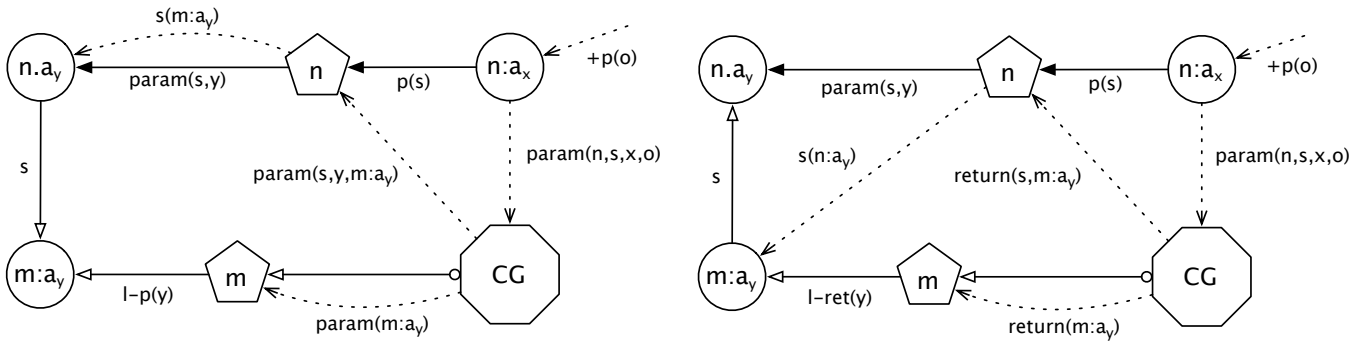


Figure 6: On-the-fly call graph construction. Adding subset constraints from actual to formal parameters (left). Adding constraints from formal return to the assigned variable (right).

| Project | Description | SLOC (app+lib) |
|---|---|---|
| BH | Barnes-Hut simulation | 899+220k |
| Coref | NLP coreference finder | 41k+225k |
| Weka | data mining software | 301k+253k |
| LuSearch | Lucene search benchmark | 48k+220k |
| jUnit | testing framework | 15.6k+220k |

Figure 7: Evaluation suite. The size of library code varies as some programs use extra libraries besides JDK.

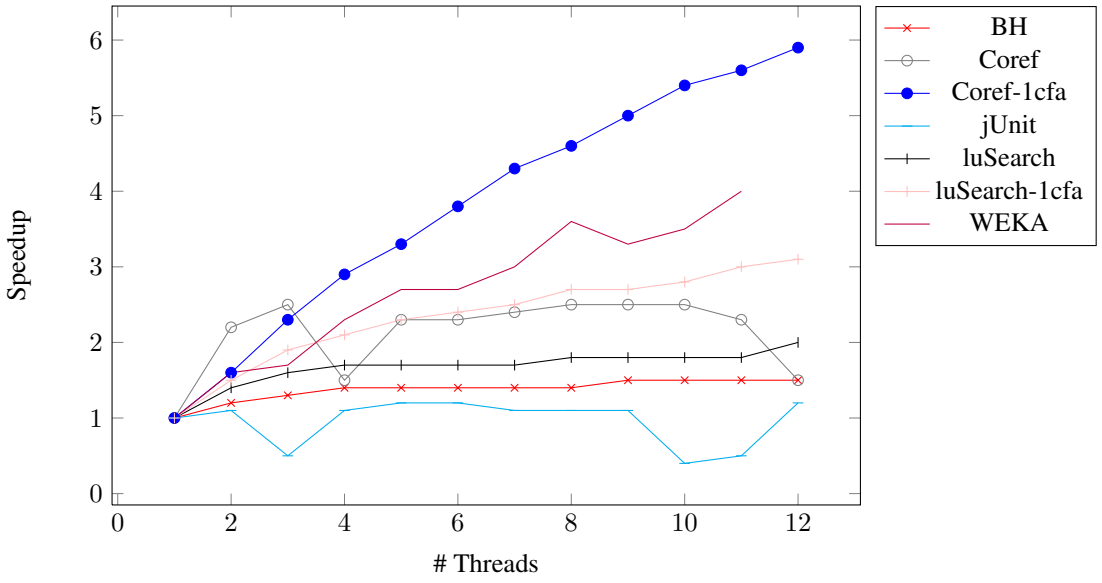


Figure 8: Results