Toward Full Elasticity in Distributed Static Analysis

MSR-TR-2015-88

Diego Garbervetsky
Edgardo Zoppi
Univesidad de Buenos Aires

Thomas Ball Benjamin Livshits
Microsoft Research

Abstract

In this paper we present the design and implementation of a distributed static analysis framework that is designed to scale with the size of the input. Our approach is based on the actor programming model and is deployed in the cloud. Our reliance on a cloud cluster provides a degree of elasticity for CPU, memory, and storage resources. To demonstrate the potential of our technique, we show how a typical call graph analysis can be implemented in a distributed setting. The vision that motivates this work is that every large-scale software repository such as GitHub, BitBucket, or Visual Studio Online will be able to perform static analysis on a very large scale.

We experimentally validate our distributed analysis approach using a combination of both synthetic and real benchmarks. To show scalability, we demonstrate how the analysis presented in this paper is able to handle inputs that are almost 10 million LOC in size, without running out of memory. Our results show that the analysis scales well in terms of memory pressure independently of the input size, as we add more VMs. As the number of analysis VMs increases, we observe that the analysis time generally improves as well. Lastly, we demonstrate that querying the results can be performed with a median latency of 15 ms.

1. Introduction

In the last decade, we have seen a number of attempts to build increasingly more scalable whole program analysis tools. Advances in scalability have often come from improvements in underlying solvers such as SAT and Datalog solvers as well as sometimes improvements to the data representation in the analysis itself; we have seen much of this progress in the space of pointer analysis [5, 19, 20, 28, 30, 31, 44].

Limits of scalability: A typical whole-program analysis is designed to run on a single machine, primarily storing its data structures in memory. Despite the intentions of the analysis designer, this approach ultimately leads to scalability issues as the input program size increases, with even the most lightweight of analyses. Indeed, if the analysis is stateful, i.e. it needs to store data about the program as it progresses, typically, in memory, eventually this approach ceases to scale to very large inputs. Memory is frequently a bottleneck even if the processing time is tolerable. We believe that the need to develop scalable program analyses is now greater than ever. This is because we see a shift toward developing large projects in centralized source repositories such as GitHub, which opens up opportunities for creating powerful and scalable *analysis backends* that go beyond what any developer's machine may be able to accomplish.

Distributed analysis: In this paper we explore an alternative approach to build distributed static analysis tools, designed to scale with the input size, with the goal of achieving full elasticity. In other words, no matter how big the input program is, given enough *computing resources*, i.e. machines to execute on, the analysis will complete in a reasonable time. Our analysis architecture assumes that the static analysis runs in the cloud, which gives us elasticity for CPU and memory resources, as well as storage. More specifically, in the context of large-scale code repositories, even code understanding and code browsing tasks are made challenging by the size of the code base. We have seen the emergence of scalable online code browsers such as Mozilla's LXR [33]. These tools often operate in batch mode, and thus have a hard time keeping up with a rapidly changing code repository in real time, especially for repositories with many simultaneous contributors. In this paper we aim to show how a more nimble system can be designed, where analysis results are largely stored in memory, spread across multiple machines. This design results in more responsive queries to obtain analysis results.

1.1 Motivation: Static Analysis Backend

Imagine a large project hosted within a centralized source repository such as GitHub or Visual Studio Online[1]. We see a clear emerging opportunity to perform server-side analysis in such a setting. Indeed, the backends of many such repositories consists of a large collection of machines, not all of which are fully utilized at any given time. During the downtime, some of the available cycles could be used to do static analysis of the code base. This can help developers with both program understanding tasks such as code browsing as well as other static analysis applications such as finding bugs.

The ever-changing code base: As shown in Figure 1, multiple developers may constantly update the code base, so it is imperative that the server-side analysis be both responsive to read-only user queries and propagate code updates fast. At the same time, within a large code base, many parts of the code, often entire directories remain unchanged for days or months at a time. Often, there is no reason to access these for analysis purposes. Therefore, to ensure that we do not run out of memory, it is important to have a system that is able to bring analysis nodes into memory on demand and persist them to disk (put them to sleep) when they are no longer needed.

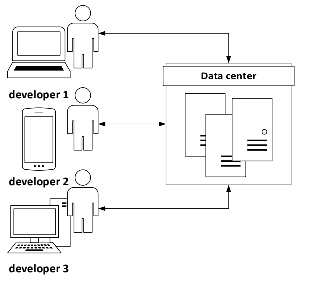


Figure 1: Analysis architecture: the analysis is performed using a cloud backend, using multiple machines, with developers both querying the results and sending updates.

1.2 Call Graph Computation

In this paper we advocate the use of the *actor model* as a building block of typical worklist-based analysis approaches. More specifically, we use this approach to implement a typical call graph construction algorithm. While the algorithm itself is quite well-known and is not a contribution of this paper, the way it is implemented in a distributed setting is.

Call graph construction is a fundamental step of most whole-program analysis techniques. However, most of the time, call graph analysis computation is a *batch* process: starting with one or more entry points such as `Main`, the call graph is iteratively updated until no more methods are discovered.

Interactive analysis: Our setting in this paper is a little different. Our goal is to answer *interactive* user queries quickly. Our queries are the kind that are most frequently posed in the context of code browsing and debugging, and are already supported on a syntactic level by many IDEs. Specifically, our analysis in this paper has been developed to provide semantic, analysis-backed answers for the following IDE-based tasks:

- **Go to definition.** Given a symbol in the program, find its possible definitions[2].
- **Who calls me.** Given a method definition, find all of its callers.
- **Auto-complete.** Auto-completion, invoked when the developer presses a dot is one of the most common and well-studied tasks within an IDE [11, 24, 29, 34–36]. If the variable or expressions on the left-hand side of the dot is of a generic interface type, completion suggestions be not particularly useful or too general. It is therefore helpful to know which concrete type flow to a given abstract location.

We have architected our analysis backend to respond to REST calls [1] that correspond to the queries above (we show some examples of such calls in Figure 15 in the Appendix). These queries constitute an important part of what is collectively known as *language services* and can be issued by both online IDEs, sophisticated code editors such as SublimeText, and full-fledged IDEs such as Eclipse and Visual Studio. Figure 14 in the Appendix shows some examples of an IDE in action, responding to user interactions.

Soundness: Given the nature of such tasks that focus on *program understanding*, the goal is not to always be absolutely precise, but to be both useful to the end user and responsive. Our analysis judiciously cuts corners in the spirit of soundness [27]. As the analysis results are used in an advisory role in the context of program understanding in an interactive setting, complete soundness is not the goal. While we focus on C# as the input language, our work should apply equally well to analyzing large projects in Java and other similar object-oriented languages. It is not, however, our goal to faithfully handle all the tricky language features such as reflection, runtime code generation, and `pinvoke`-based native calls.

1.3 Contributions

This paper makes the following contributions:

[1] More about Visual Studio Online is available from `https://www.visualstudio.com/en-us/products/visual-studio-team-services-vs.aspx`.

[2] Note that this process is complicated by the presence of polymorphism, common in object-oriented languages. Given a call site, it is not always possible to determine which is the actual method implementation being invoked. This problem known as *call site devirtualization* is well-studied in the literature. Therefore, a static analysis can only over approximate the target method definitions for a virtual method invocation.

- We propose a distributed static analysis approach and show how to apply it to call graph construction for answering program understanding and code browsing queries.
- We describe how our analysis framework is implemented on top of the Orleans distributed programming platform and is deployed on legacy hardware in the cloud using Microsoft Azure.
- We experimentally demonstrate the scalability of our technique using a range of synthetic and real benchmarks. The results show that our analysis scales well in terms of memory pressure independently of the input size, as we add more machines. Elapsed analysis times can vary based on project complexity.

  Despite using stock hardware and incurring a non-trivial communication overhead, we scale to inputs containing 10 million LOC, and our processing time for some benchmarks of close to 1 million LOC is about 5 minutes, excluding compilation time. While the communication overhead can become a bottleneck, we show that as the number of machines increases (up to 64), the analysis time generally drops. Lastly, we demonstrate that querying the results can be performed with an acceptable median latency of 15 ms.

1.4 Paper Organization

The rest of the paper is organized as follows. Section 2 provides a high-level overview of our distributed analysis approach. Section 3 describes the specifics of the call graph construction algorithm. Section 4 discusses some of the implementation details of our system on top of the Orleans distributed programming framework. Section 5 presents our experimental evaluation. Finally, Sections 6 and 7 describe related work and conclude. The Appendix contains extra screen-shots and figures.

2. Overview

Given the architecture shown in Figure 1, our goal is to have the analysis backend respond to queries quickly, independently of the input size. Of course, we also need to make sure that the backend does not run out of memory or timeout in some unpredictable way. Our requirements force us to rethink some of the typical assumptions of whole-program analysis.

2.1 Analysis Design Principles

We use a *distributed actor model* [4] as the basis of our distributed static analysis engine. For a program written in an object-oriented language such as Java or C#, a natural fit is to have an actor per *method* within the program[3]. These actors are responsible for receiving messages from other actors, processing them using local state (a representation of the method body, for instance), and sending information to other methods that depend on it. For example, for a call graph construction analysis, actors representing individual methods may send messages to actors for their callers and callees. Our analysis design adhere to the following distilled principles.

```
while $|MQ| > 0$ do
    $\langle a, m \rangle := MQ.choose()$
    $v := UNPACK(m) \sqcup VALUE[a]$
    if $v \sqsubseteq VALUE[a]$ then
        continue
    end if
    $v' := TF[a](v)$
    if $v \sqsubseteq v'$ then
        $U := DELTA(v, v')$
        for each $u$ in $U$ do
            $MQ := MQ \cup PACK(a, u)$
        end for
        $VALUE[a] := v'$
    end if
end while
```

Figure 2: Distributed worklist algorithm.

- **Minimal in-memory state per actor.** We want to "pack" as many actors per machine as possible without creating undue memory pressure, which could lead to swapping, etc.
- **Design for lightweight serialization.** We have designed our analysis so that the updates sent from one actor to another are generally small and easily serialized. There is minimal sharing among actors, as actor holds on to its local state and occasionally sends small updates to others. The same principle applies to persistent per-actor state as well. Even if the in-memory state for an active actor is sizeable, we only serialize the bare minimum to disk, before the actor is put to sleep. This can happen when the actor runtime decides to page an actor out due to memory pressure or lack of recent use.
- **State can be recomputed on demand.** In a distributed setting, we have to face the reality that processes may die due to hardware and/or software faults. It is therefore imperative to be able to recover in case of state loss. While it is possible to commit local state to persistent store, we eschew the overhead of such an approach and instead choose to recompute per-node state on demand.
- **Locality optimizations to minimize communication.** We attempt to place related actors together on the same machine. In the case of a call graph analysis, this often means that entire strongly connected components co-exist on the same physical box, which minimizes the number of messages that we actually need to dispatch across the network.

[3] We could choose to have an actor per *class* in a program, or other well-defined program entity.

2.2 Distributed Worklist Algorithm

We now present a high-level view of a distributed analysis problem as a pair $\langle A, L \rangle$ where:

- A is a set of actors distributed in a network.
- $\langle L, \sqsubseteq, \sqcup \rangle$ is a complete semi-lattice of finite height.

Each actor $a \in A$ has the following associated functions:

- $VALUE[a] = v \in L$ is the local state of actor a;
- $TF[a](v) = v' \in L$ is the transfer function for the local computation performed within actor a. We assume TF is monotone;

The following helper functions are for communicating state changes among actors:

- $DELTA(v, v')$ computes a set U of (global) updates required when switching from local state v to v';
- $PACK(a, u)$ is a function that given an update at actor a produces one or several messages to communicate to other actors.
- $UNPACK(m)$ is a function that unpacks a message and returns a value in L.

Figure 2 shows the pseudocode for a distributed worklist algorithm. The algorithm makes use of a global message queue, denoted as MQ[4]. The queue is initialized with a set of starting messages that will depend on the actual analysis instance.

2.3 Termination and Non-Determinism

Let H denote the (finite) height of semi-lattice L and let $N = |A|$. Consider iterations through the loop on line 1. Let's consider two sets of sequences of iterations, I_1 are iterations that lead to a value increase on line 7 and I_2 are those that do not.

We can have at most $H \times N$ iterations in I_1 given the finite size of the lattice. For iterations in I_2, the size of MQ decreases because at least one message is consumed but it does not generate other messages. We consider two possibilities:

- Starting from some iteration i, we only have iterations in I_2. This, however, means that on every iteration the size of MQ decreases, until it eventually becomes empty.
- The other possibility is that we will have an infinite number of iterations in I_1 . This is clearly impossible because the size of I_1 is bounded by $H \times N$.

It is important to emphasize the difference between this distributed algorithm and a single-node worklist approach. If a message is in flight, we do not wish the program analysis to terminate. However, detecting the emptiness of MQ is not trivial, so in practice we must have an effective means for detecting termination. We make use of an *orchestrator* mechanism for termination detection, as described in Section 4.5.

While the algorithm in Figure 2 reaches a fixpoint independently of the arrival order of messages, it is natural to ask whether that is the only fixpoint that can be reached. Given that $TF[a]$ is monotone and L is of finite height the uniqueness of least fixpoint is guaranteed [13, 23].

[4] Note that MQ is a *mathematical abstraction*: we do not actually use a global message queue in our implementation. Conceptually, we can think of a (local) worklist maintained on a per-actor basis. Termination is achieved when all the worklists are empty.

3. Call Graph Analysis

In this section we present an instantiation of the general framework described in the previous section for computing call graphs. Our analysis is a distributed interprocedural and incremental inclusion-based static analysis inspired by the Variable Type Analysis (VTA) presented in [41]. This flow-insensitive analysis computes the set of potential types for each *object reference* (variable, field, etc.) by solving a system of inclusion constraints. Because it propagates type constraints from object allocation sites to their uses, this kind of analysis is sometimes referred to as *concrete type* analysis.

3.1 Program Representation

Propagation graphs: At the method level, the inclusion-based analysis is implemented using a data structure we call a *propagation graph* (PG) [41]. A PG is a directed graph used to "push" type information to follow data flow in the program, as described by analysis rules. Our analysis naturally lands itself to incrementality. A typical change in the program would require often minimal recomputation within the modified code fragment as well as propagation of that information to its "neighbors". Propagation graphs support incremental updates since the propagation of information is triggered only when a new type reaches a node.

Terminology: More formally, let $PG = \langle R, E \rangle$ where R denotes a set of *abstract locations* in the method (such as variables and fields) and E refers to a set of edges between them.

An edge $e = (v_1, v_2) \in E$ connects nodes in the PG to model the potential flow of type information from v_1 to v_2. Essentially, an edge represents a rule stating that $\texttt{Types}(v_2) \supseteq \texttt{Types}(v_1)$ (e.g, $v_2 = v_1$). To model interprocedural interaction, the PG also includes special nodes for representing method invocations and return values (rv). Finally, $I \subseteq R$ denotes the set of invocations. Let T be the set of all possible types, `dType` contains declared types (compile-time types) for abstract locations and `Types` denotes concrete types inferred by our analysis.

$$\begin{array}{rcl}
v_1 = v_2 & \Longrightarrow & \texttt{Types}(v_1) \supseteq \texttt{Types}(v_2) \\
v_1 = v_2.f & \Longrightarrow & \texttt{Types}(v_1) \supseteq \texttt{Types}(\texttt{dType}(v_1).f) \\
v_1.f = v_2 & \Longrightarrow & \texttt{Types}(\texttt{dType}(v_1).f) \supseteq \texttt{Types}(v_2) \\
v = \texttt{new}\ C() & \Longrightarrow & C \in \texttt{Types}(v) \\
\texttt{return}\ v & \Longrightarrow & \texttt{Types}(rv) \supseteq \texttt{Types}(v) \\
loc : v = v_0.m(v_1 \ldots v_n) & \Longrightarrow & \texttt{Types}(inv_{loc}) \supseteq \bigcup_{j=0..n} \texttt{Types}(v_j)
\end{array}$$

Figure 3: VTA analysis rules.

3.2 Analysis Phases

In the actor model, the choice of granularity is key for performance. We decided to use one actor per method. Each method-level actor contains a PG that captures type information that propagates through the method.

The analysis starts by analyzing an initial set of root methods M_0. We describe both intra- and interprocedural processing below.

3.2.1 Intraprocedural Analysis

Instantiating the problem: The lattice L for our analysis consists of a mapping from abstract locations to sets of possible types and is defined as

$$L = \langle \texttt{Types} : R \mapsto 2^T, \sqsubseteq_{type}, \sqcup_{type} \rangle$$

with $\sqsubseteq_{type}$ defined as

$$l_1 \sqsubseteq_{type} l_2 \text{ iff } l_1.\texttt{Types}(r) \subseteq l_2.\texttt{Types}(r), \forall r \in R$$

and $\sqcup$ defined as

$$l_1 \sqcup_{type} l_2 = l_3 \text{ where}$$

$$l_3.\texttt{Types}(r) = l_1.\texttt{Types}(r) \cup l_2.\texttt{Types}(r), \forall r \in R.$$

Analysis rules that compute $TF[a]$ are summarized in Figure 3 and cover the typical statement types such as loads, stores, allocations, etc. Object dereferences (i.e., $v.f$) are represented by using the name of the class defining the field. That is, the analysis is field-sensitive but not object-sensitive. In the case of invocations there is an inclusion relation to model the flow of all the arguments to the invocation abstract location $inv_{loc} \in I \subseteq R$. Note that the left-hand side v of the invocation is not updated by the rule since it depends on the result of the invoked method. This will be handled by interprocedural analysis.

Notice that $TF[a]$ is monotone because the propagation of types never removes a type and L satisfies the finite-height condition because it is a finite lattice.

3.2.2 Interprocedural Analysis

Once the intraprocedural phase finishes, relevant updates must be communicated to the corresponding methods (callees and callers). As mentioned, the analysis considers invocations using the set $I \subseteq R$. To handle callers' updates, we need to extend the lattice to include the caller's information for the current method. This has the form $\langle m, lhs \rangle$, where $m \in A$ denotes the caller's name and $lhs \in R$ represents the left-hand side of the invocation made by the caller. The extended lattice is shown below.

$$\begin{array}{rcl}
L & = & \langle \texttt{Types} : R \mapsto 2^T \times \texttt{Callers} : 2^{A \times R}, \sqsubseteq, \sqcup \rangle \\
 & & \\
l_1 \sqsubseteq l_2 & \text{iff} & l_1 \sqsubseteq_{type} l_2 \ \wedge \\
 & & l_1.\texttt{Callers}(r) \subseteq l_2.\texttt{Callers}(r), \forall r \in R \\
 & & \\
l_1 \sqcup l_2 & = & (ts, cs) \text{ where} \\
 & & ts = l_1 \sqcup_{type} l_2 \ \wedge \\
 & & cs = l_1.\texttt{Callers}(r) \cup l_2.\texttt{Callers}(r), \forall r \in R
\end{array}$$

A message m has the form $\langle kind, d, data \rangle$, where $kind \in \{\text{callMsg}, \text{retMsg}\}$ is the kind of message, $d \in A$ is the destination actor and $data$ is a tuple.

Instantiating ***DELTA*****:** In Figure 4a we show the definition of the *DELTA* operation described in Section 2. It computes the set of invocations that were affected by the propagation. An invocation is affected if the set of types flowing to any of its parameters grew. Additionally, we also must consider changes in types that the return value may correspond to, since they need to be communicated to the callers.

Instantiating ***PACK*****:** Figure 4b shows a definition of *PACK*. This function is in charge of converting local updates to messages that can be serialized and sent to other actors. For each invocation, the analysis uses the computed type information of the receiver argument to resolve potential callees.

Then, it builds a caller message including the potential types for each argument. Those types will be added to the set of types of the parameters on the caller actor. In case of an update in return value it builds a message to inform the caller about changes to the return value's types. This message includes the (original) caller's left-hand side, so that the caller can update its types.

Instantiating ***UNPACK*****:** Function *UNPACK* in Figure 4c is responsible for processing messages received by an actor. This function converts a message into a value in the lattice of the local analysis that will be then joined into the local state. A message can be either a *call message* (i.e., an invocation made by a caller) or a *return message* (i.e., to inform a change in the callee's return value). For call messages we produce an element that incorporates the types for each call argument into the method parameters. We also update the set of callers. For return messages we need to update the left-hand side of the invocation with the potential types of the return value.

Example 1 This example illustrates the advantage of using concrete types as opposed to declared types to obtain more precision. Consider the small program in Figure 5a. In Figure 5b we show the propagation graphs for both methods. As the analysis starts, only the left-hand sides of allocations (lines 2 and 11) contain types.

During propagation, type `B` flows from variable `x` into an invocation of `M` as an argument. This triggers a message to the actor for method `B.M`. The flow through parameter `p` and `w` makes the return value of `B.M` to contain type `B`. This in turn triggers a return message that adds `B` to the types of `y`. This propagates to `z`.

$$
\begin{aligned}
\texttt{let } d(v,v')(r) &:= v'.\texttt{Types}(r) - v.\texttt{Types}(r)\\
\texttt{let } \mathit{Inv}(v,v') &:= \{\mathit{inv} \mid \mathit{inv} \in I \wedge\ d(v,v')(\mathit{inv}) \neq \varnothing\}\\
\texttt{let } \mathit{Rv}(v,v') &:= \begin{cases} \{\mathit{rv}\} & \text{if } d(v,v')(\mathit{rv}) \neq \varnothing \\ \varnothing & \text{otherwise} \end{cases}\\
\mathit{DELTA}(v,v') &\overset{\text{def}}{=} \mathit{Inv}(v,v') \cup \mathit{Rv}(v,v')
\end{aligned}
$$

(a) Definition of $\mathit{DELTA}(v,v')$

$$
\begin{aligned}
\texttt{let } \mathit{callees}(\mathit{inv}) &:= \{C.m \mid C \in l.\texttt{Types}(\mathit{args}(\mathit{inv})_0)\}\\
\texttt{let } \mathit{callMsg}(a,\mathit{inv}) &:= \langle a, \mathit{lhs}(\mathit{inv}), l.\texttt{Types}(\overline{\mathit{args}(\mathit{inv})})\rangle\\
\texttt{let } \mathit{callMsgs}(a,\mathit{inv}) &:= \{\langle \text{callMsg}, d, \mathit{callMsg}(\mathit{inv})\rangle\\
&\qquad \mid\ d \in \mathit{callees}(\mathit{inv})\}\\
\texttt{let } \mathit{returnMsg}(a,c) &:= \langle a, \mathit{lhs}(c), l.\texttt{Types}(\mathit{rv})\rangle\\
\texttt{let } \mathit{retMsgs}(a) &:= \{\langle \text{retMsg}, \mathit{method}(c), \mathit{returnMsg}(a,c)\rangle\\
&\qquad \mid\ c \in l.\texttt{Callers}\}\\
\mathit{PACK}(a,u) &\overset{\text{def}}{=} \begin{cases} \mathit{callMsgs}(a,u) & \text{if } u \in I \\ \mathit{retMsgs}(a) & \text{if } u = \mathit{rv} \end{cases}
\end{aligned}
$$

(b) Definition of $\mathit{PACK}(a,u)$. $l.\texttt{Types}(\overline{\mathit{args}})$ is the lifting of $l.\texttt{Types}$ to the list of arguments, it returns a lists of set of types. Given $\mathit{inv} = \langle v = v_0.m(v_1 \ldots v_n)\rangle$, $\mathit{args}(\mathit{inv}) = [v_0, v_1, \ldots, v_n]$, $\mathit{lhs}(\mathit{inv}) = v$. For a caller $c = (m, \mathit{lhs}) \in l.\texttt{Callers}$, $\mathit{method}(c) = m$, the caller's name and $\mathit{lhs}(c) = \mathit{lhs}$, the left-hand side of the original invocation made by the caller.

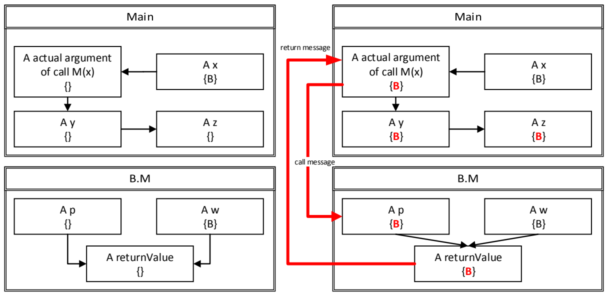
$$
\begin{aligned}
\texttt{let } l_1.\texttt{Types}(r) &= \begin{cases} \mathit{argTypes}(m)_i \text{ if } r = p_i \\ \varnothing \text{ otherwise} \end{cases}\\
\texttt{let } l_1.\texttt{Callers} &= \{(\mathit{sender}(m), \mathit{lhs}(m))\}\\
\texttt{let } l_2.\texttt{Types}(r) &= \begin{cases} \mathit{retTypes}(m) \text{ if } r = \mathit{lhs}(m) \\ \varnothing \text{ otherwise} \end{cases}\\
\mathit{UNPACK}(m) &\overset{\text{def}}{=} \begin{cases} l_1 & \text{if } \mathit{kind}(m) = \text{callMsg} \\ l_2 & \text{if } \mathit{kind}(m) = \text{retMsg} \end{cases}
\end{aligned}
$$

(c) Definition of $\mathit{UNPACK}(m)$. For a message $m = \langle \text{callMsg}, d, \langle a, \mathit{lhs}, [\mathit{ts}_0, \mathit{ts}_1, \ldots, \mathit{ts}_n]\rangle\rangle$ $\mathit{argTypes}(m)_i = \mathit{ts}_i$, the set of potential types for the i^{th} argument p_i. $\mathit{lhs}(m) = \mathit{lhs}$, $\mathit{sender}(m) = a$. For a return message $m' = \langle \text{retMsg}, d, \langle a, \mathit{lhs}, \mathit{ts}\rangle\rangle$, $\mathit{retTypes}(m') = \mathit{ts}$ is the set of potential types of the method's return value.

Figure 4: Defining *DELTA*, *UNPACK*, and *PACK*.

```
 1  public static void Main() {
 2    A x = new B();  // allocation
 3    A y = x.M(x);
 4    A z = y;
 5  }
 6  public class A {
 7    public abstract A M(A p);
 8  }
 9  public class B : A  {
10    public override A M(A p)  {
11      A w = new B();  // allocation
12      return (p != null) ? p : w;
13    }
14  }
```

(a) Code example for interprocedural propagation.



(b) PGs for methods `Main` and `B.M` before (left) and after (right) the propagation for the code in Figure 5a.

Figure 5: Code and propagation graph for Example 1.

Concrete type analysis produces results that are more accurate for `y`, `z`, etc. than what we can obtain from their declared types. ■

Type approximation: In the interprocedural stage, our analysis sends information about concrete parameter types to its callees. However, when it comes to complex, nested objects, this information is potentially insufficient, as it only concerns one level of the object hierarchy. Consider the following example:

```
void Main {                        void M(A p) {
    A x = new B();                     A z = p.f;
    x.f = new B();                     return z;
    y = M(x)                       }
}
```

Function *PACK* will create a message that propagates the type of `x` into `M` and *UNPACK* will discover the type of `p` to be `B`. However, no information is given for the type of `p.f`, potentially leading to unsoundness.

We could include information about the first (or k^{th}) level of references for parameter types. This may possibly lead to larger message sizes, with many more types included for complex objects, with limited benefit. This size explosion could be mitigated by pruning these messages if callees' method signatures include information about the fields they read. We take an alternative approach and rely on Rapid Type Analysis (RTA) to maintain soundness at the cost of loosing some precision [8].

We use the type of `p.f` given by RTA; when RTA provides no useful information, we fall back on declared types. As usual, this kind of relaxation can lead to imprecision that may be undesirable, i.e. too many spurious suggestions may appear for auto-complete. We did not observe it, an acceptable tradeoff would be to choose an under-approximation instead of an over-approximation here.

Other uses of the analysis framework: The distributed algorithm in Figure 2 can be instantiated for other program analyses that follow the same design principle. For instance, consider an inclusion-based analysis like Andersen's points-to [7]. A possible instantiation may be as follows: (1) Each node represents a method; (2) The transfer function implements Andersen's inclusion rules locally and, in case there is a change in an argument of a method invocation, produces an update message to be sent to the potential callees; (3) Similarly, by just replacing the inclusion rules with unification rules in the transfer function, we can turn it into a unification based points-to analysis like Steensgaard's [40]. We envision future work where our distributed backend would be combined with a natural front-end for this kind of analysis that uses Datalog, as previously proposed for single-machine analysis [22].

4. Implementation

We implemented a prototype of our distributed approach[5] to analyze large-scale projects written in C#. This prototype relies on Roslyn [32], a compiler framework for analyzing C# code and the Orleans framework [10], an implementation of a distributed actor model that can be deployed in the cloud. Although other deployment options such AWS are possible, for this paper we used Azure as a platform for running our experiments.

4.1 Orleans and the Actor Model

Orleans [10] is a framework designed to simplify the development of distributed applications. It is based on the abstraction of virtual *actors*. In Orleans terminology, these actors are called *grains*. Orleans solves a number of the complex distributed systems problems, such as deciding where — on which machine — to allocate a given actor, sending messages across machines, etc., largely liberating developers from dealing with those concerns. At the same time, the Orleans runtime is designed to enable applications that have high degrees of responsiveness and scalability.

Grains are the basic building blocks of Orleans applications and are the units of isolation and distribution.

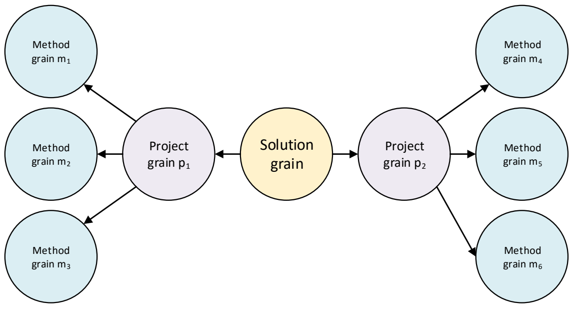


Figure 6: Logical organization of grains. The arrows show how grains create each other: solution grains create project grains; project grains create method grains, etc.

Every grain has a unique global identity that allows the underlying runtime to dispatch messages between actors. An actor encapsulates both behavior and mutable local state. State updates across grains can be initiated using messages.

The runtime decides which physical machine (*silo* in Orleans terminology) a given grain should execute on, given concerns such as memory pressure, amount of communication between individual grains, etc. This mechanism is designed to optimize for communication locality because even within the same cluster cross-machine messages are considerably smaller than local messages within the same machine.

4.2 Grain Organization

We follow a specific strategy in organizing grains at runtime. This strategy is driven by the input structure. The input consists of an MSBuild *solution*, a `.sln` file that can be opened in Visual Studio. Each solution consists of a set of *project files*, `*.csproj`, which may depend on each other. Roslyn allows us to enumerate all project files withing a solution, source files within a project, classes within a file, methods within a class, etc. Furthermore, Roslyn can use its build-in C# compiler to compile sources on the fly. In Figure 6 we show how grains are organized to follow this logical hierarchy.

We define grains for solutions, projects and methods. While we initially considered more levels in this hierarchy, we do not find it necessary to provide grains for classes and other higher-level code artifacts such as documents or `namespaces`.

- A solution grain is a singleton responsible for maintaining the list of projects and providing functionality to find methods within projects.
- A project grain contains the source code of all files for that project and provides functionality to compute the information required by method grains (e.g., to build propagation graphs by parsing the method code) as well as type resolution (e.g., method lockup, subtyping queries, etc).

[5] Source code implementing the analysis in this paper is available on GitHub. We are planning to provide a link in the final, de-anonymized version of this paper.

- The method grain is responsible for computing the local type propagation and resolve caller/callees queries. It stores type information for the abstract locations in the method.

The hierarchical grain organization in Figure 6 allows us to minimize the amount of unnecessary IO. The solution grain reads the $*$`.sln` file from cloud storage; in our implementation we used Azure Files, but other forms of input that support file-like APIs such as GitHub or Dropbox are also possible. Project grains read $*$`.csproj` files and also proceed to compile the sources contained in the project to get a Roslyn `Compilation` object. This information is only contained in the project grain to minimize duplication. To obtain information about the rest of the project, method grains can consult the project grain. We use caching to reduce the number of messages between grains.

Example 2 To illustrate persistent state for a typical method grain, consider the example in Figure 5a. The state of both methods is as follows.

Method `Main`:

```
Callers = {}
Types   = {(x,{B}), (y,{B}), (z,{B}), (3,{B})}
```

Method `B.M`:

```
Callers = {(A.Main, y)}
Types   = {(p,{B}), (w,{B}), (returnValue,{B})}
```

This minimal state is easily serialized to disk if the grains are ever deactivated by the Orleans runtime. ■

4.3 Orleans-based Implementation

Orleans is built on a cooperative multitasking model. A grain activation operates in discrete units of work called *turns* and finishes each execution unit before moving on to the next. A turn executes the computation to handle requests from other grains or external clients and to run closures at the resolution of a promise. While a system may execute many turns belonging to different activations in parallel, each activation always executes its turns sequentially. Therefore, execution in an activation is always logically single-threaded.

Asynchronous programming: Orleans is deeply integrated with asynchronous programming in .NET, which involves the `async/await` programming paradigm. Orleans takes care of serializing messages and deserializing them on the side of the received. This is achieved by Orleans rewriting calls such as

```
Task<Effects> effects = await GetEffects(method);
```

to perform cross-machine calls: `method` is serialized and sent to the grain that executes `GetEffects`. The return value `effects` is a *promise* for an eventual result. Note, however, that serializing a complex object such as a method may be taxing for the network and is largely unnecessary. We take care to ensure that we avoid doing so. A better approach here would be to call

```
string methodName = method.GetUniqueMethodName();
Effects effects = await GetEffects(methodName);
```

where `GetUniqueMethodName` returns a short string. Generally, we mostly pass around strings or lists of strings; other complex in-memory types are "reduced" to this simplified representation. Under the covers, the transport layer may use JSON or another similar encoding to transmit the data.

Fault tolerance: In addition to helping with data seamless passing across machine boundaries, Orleans will automatically attempt message delivery several times before throwing an exception. Moreover, Orleans provides APIs for recovery of a grain in case of a failure. This allows us to find and request information from the corresponding project grain to build the method-level propagation graph. Similarly, project and solution grains can recover information from the file storage in case of a crash.

4.4 Distributed Analysis Challenges

Implementing a distributed system like ours is fraught with three fundamental challenges.

Reentrancy: Since the callgraph can have cycles, a grain can start a propagation which will in turn eventually propagate to the original method. However, since Orleans uses turn-based concurrency this will create a deadlock. Even without recursion it is possible for a method grain that is currently being processed to receive another message (i.e. a return message).

Termination: In a distributed setting, detecting when we achieve termination is not so easy. This is in part because even if all the local worklists are empty, we may have messages that are in flight or those that have been delayed.

Timeouts: In a manner similar to other turn-based concurrency systems (for instance, JavaScript in the browser), in order to detect potential failures and deadlocks, Orleans monitors the duration of calls to other grains and terminates calls that it deems to be timeouts. This has a number of undesirable consequences such as exceptions that propagate throughout the system. Some program analysis tasks, such as compiling a project or creating a propagation graph for a long method, may exceed the timeout that Orleans imposes.

4.5 Addressing Analysis Challenges

Naïve solution #1: Given the built-in powerful features of Orleans, it is tempting to implement the main worklist algorithm *recursively*, as shown in Figure 7. Note that Orleans will rewrite `async` calls such as `callee.ProcessMethodAsync()` to communicate across

```
 1 foreach (var r in roots) {
 2  var effects = await r.ProcessMethodAsync();
 3  // ...
 4 }
 5 class MethodGrain {
 6  async Task<Effect> ProcessMethodAsync() {
 7   // process local state
 8   // ...
 9  foreach (var inv in this.Invocations) {
10   var callees = this.ResolveCallees(inv);
11   foreach (var c in callees) {
12    var effects = [await] c.ProcessMethodAsync();
13    // ...
14   }}}}
```

Figure 7: Naïve attempts #1 and #2: the keyword `await` of line 12 is included in attempt #1, but not in attempt #2.

machines. Unfortunately, this naïve approach is not going to work well because of reentrancy issues: the call on line 12 may affect the current grain directly (self-recursion) or after a series of calls.

Naïve solution #2: A possible solution to the issue of reentrancy is to not `await` the completion of calls on line 12 in Figure 7. Doing so will allow other grains to do their processing in parallel. This is sometimes called "send-and-forget" style of asynchronous processing. While this approach is likely to increase the level of parallelism, it provides no easy way to detect global termination. Indeed, the top-level loop will return almost immediately without waiting for the results to propagate back. Besides, not awaiting the result has as a consequence missing the exceptions may be thrown by the callee.

Orchestrator: A more radical option is to use of an *orchestrator* to establish some degree of centralized control over the propagation process. Grains communicate with the orchestrator exclusively, instead of communicating with each other peer-to-peer. This avoids the issue of reentrancy by construction; only the orchestrator can send messages to grains via a single *message queue*.

The orchestrator keeps track of the outstanding tasks and can therefore detect both termination and prevent reentrant calls from taking place. To overcome the issue of timeouts, the orchestrator catches the timeout exceptions thrown by the grains. This is possible because the orchestrator initiates all the chain of grain calls. The key disadvantage of this design is that it is possible to have a great deal of *contention* for access to the orchestrator. We observed this in practice, suggesting a different variant of this idea.

Reducing contention with multiple queues: Instead of having an unique centralized queue, we have *a collection* of queues distributes across the distributed system. Each method grain is a potential producer of *effects* to be processed by other method grains. To avoid reentrancy, this information is not sent directly to the target method grain but it is enqueued in one of the queues in the round robin fashion. The information is then consumed by *dispatchers grains* that pull the data from the queues and deliver it to the corresponding method grains; this is illustrated in Figure 8.

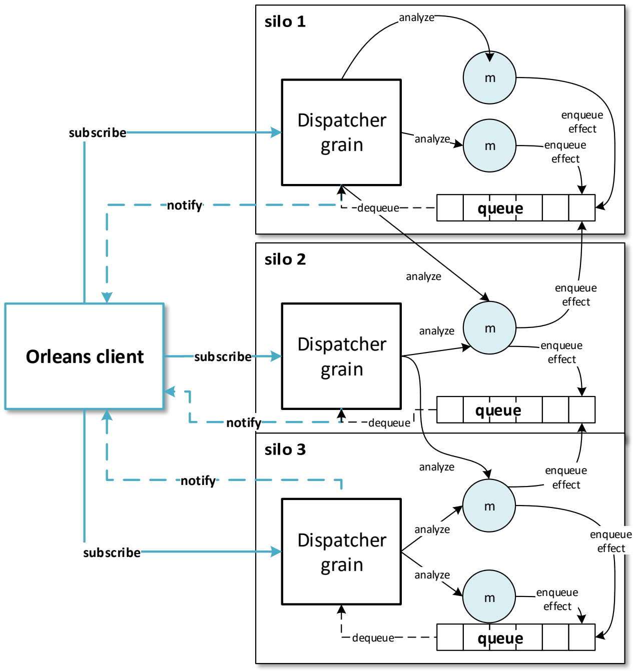


Figure 8: The multi-queue approach, illustrated. Method grains are circles shown in light blue. Solid and dashed arrows represent standard invocations and callbacks respectively.

Using this mechanism we avoid both reentrancy, bottlenecks and single points of failure. The drawback is that detecting termination is more complex. For that, we use timers to determine when a dispatcher becomes idle (i.e., inactive longer than a predetermined threshold), at which point we notify the client. The analysis finishes when the client is sure that all dispatchers are idle[6].

In practice, we set the number of queues to be four times higher than the number of worker VMs (for example, 128 queues for 32 worker VMs) and set the termination threshold to 10 seconds.

4.6 Azure Deployment

Our analysis is deployed in Azure as illustrated in Figure 13 in the Appendix. On the left, there is the analysis client such as an IDE or a code editor like SublimeText. The cluster we used consists on one *front-end VM* and a number of worker VMs. The client used REST requests to communicate to the front-end VM. The job of the front-end VM is to (1) accept and process external analysis client requests; (2) dispatch jobs to the worker

[6] We have a mechanism to detect the case when an idle dispatcher becomes active again.

VMs and process the results; and (3) provide a Web UI with analysis results and statistics.

In Figure 14 in the Appendix we show two screenshots of an experimental IDE prototype that uses the API exposed by our analysis to resolve caller/callees queries[7]. In Figure 15 in the Appendix, we show several typical REST requests for common IDE navigation tasks. The API is designed for use with a variety of clients; for a task such as getting all references to a symbol, we simply package up the name of the symbol into a string and dispatch the request.

5. Evaluation

In our evaluation, we aim to answer the following three research questions.

RQ1: Is our analysis capable of handling arbitrary amounts of input (i.e., more lines of code, files, projects, etc.) by increasing the number of worker VMs, without running out of memory?

RQ2: While the communication overhead can become significant, as more worker VMs are added, does an increase in the number of worker VMs significantly increase the overall analysis times?

RQ3: Is the analysis query latency small enough to allow for interactive use[8]?

5.1 Experimental Setup

All the experiments presented in this paper were executed in the cloud, on a commercially available Azure cluster. We could also have used an AWS cluster, as our dependency on Azure is small. The Azure cluster we used for the experiments consists on one front-end VM and up to 64 worker role VMs. The front-end VM is an Azure VM with 14 GB of RAM (this is an $A4\backslash ExtraLarge$ VM in Azure parlance[9]). Each worker role is an Azure VM with 7 GB of RAM (called $A3\backslash Large$ in Azure). For benchmarking purposes, we run our analysis with configurations that include 1, 2, 4, 8, 16, 32, and 64 worker VMs. Note that the orchestrator always resides in a single VM.

To collect numbers for this paper, we used a custom-written experimental controller as our analysis client throughout this section; this setup is illustrated in Figure 13 in the Appendix. The controller is scripted to issue commands to analyze the next `.sln` file, collect timings, etc.

[7] Another example of such an IDE can be found at `http://source.roslyn.io/`

[8] Generally, query latencies of 10 to 20 ms are considered to be acceptable.

[9] Up-to-date VM specifications are available at `https://azure.microsoft.com/en-us/documentation/articles/virtual-workerVMs-size-specs/`.

We heavily instrumented our analysis to collect a set of relevant metrics. We instrumented our analysis code to measure the analysis elapsed time. We introduced wrappers around our grains (solution, project, and method grains) to distinguish between local messages (within the same VM) and network messages. Using Orleans-provided statistics, we measured the maximum memory consumption per VM. Lastly, we also have added instrumentation to measure query response times. While these measurements are collected at the level of an individual grain, we generally wanted to report aggregates. In order to collect these, we posted grain-level statistics to a special auxiliary grain.

5.2 Benchmarks

For our inputs, we have used two categories of benchmarks, *synthetic* benchmarks we have generated specifically to test the scalability of our call graph analysis and a set of 3 real applications written in C# that push our analysis implementation to be as complete as possible, in terms of handling tricky language features such as `delegate`, `lambdas`, etc. and see the impact of dealing with polymorphic method invocations. In all cases, we start with a solution file (`.sln`) which references several project files (`.csproj`), each of which in turn references a number of C# source files (`.cs`).

Synthetic benchmarks: We designed a set of synthetic benchmarks to test the scalability of our analysis approach. These are solution files generated to have the requisite number of methods (for the experiments, we ranged that number between 1,000 and 1,000,000).

These methods are broken into classes, one class per input file. The pattern of calls within this synthetic code is set up to maintain a fixed average number of callees per invocation and also to force all methods to be reachable from root methods. The figure below summarizes some statistics about the synthetic projects we have used for this evaluation.

Benchmark	LOC	Projects	Classes	Methods
X1,000	9,196	10	10	1,000
X10,000	92,157	50	50	10,000
X100,000	904,854	100	100	100,000
X1,000,000	9,005,368	100	100	1,000,000

Real-world benchmarks: We have selected several large open-source projects from GitHub for our analysis. A summary of information about these programs in shown in Figure 9. We tried to focus on projects that are under active development. To illustrate, one of our benchmarks, Azure Powershell is one of the most popular projects written in C# on GitHub. According to the project statistics, over a period of one month in October–November 2015, 51 authors have pushed 280 commits to the main branch and 369 commits to all branches. There have been 342,796 additions and 195,366 deletions. We picked solution `ResourceManager.ForRefactoringOnly.sln` from Azure Powershell because it is the only one that contains all the projects. Generally, discovering good root methods to serve as starting points for the call graph analysis is not trivial. Because there is no natural `Main` method in several of these projects, we have decided to use as entry points the included *unit tests*, *event handlers*, and `public` methods within the project to increase the number of methods our analysis reaches[10].

5.3 [RQ1]: Scales with Input Size

To answer RQ1, we measured the memory consumption of each VM and computed the average and maximum memory consumption across all VMs. Figure 10a shows the *average* memory consumption for each benchmark during the run, for each experimental configuration, i.e. number of worker VMs used. To give an aggregate perspective of the effect that adding more VMs to the mix has on memory pressure, Figure 10b shows the memory consumption *averaged* across all benchmarks shown for every cloud configuration.

As can be observed from the chart, the memory consumption decreases steadily as the number of worker VMs increases. Recall that worker VMs come equipped with 7 GB of memory, so these memory consumption numbers are nowhere near that limit. Looking at Figure 10a, we can see peaks of about 3.2 GB for a single worker VM while analyzing X1,000,000[11].

These experiments naturally highlight the notion of analysis *elasticity*. While we run the analysis with different number of VMs set for the sake of measurement, in reality, more machines would be added (or removed) due to memory pressure (or lack thereof) or to respond to how full analysis processing queues get. We can similarly choose to increase (or decrease) the number of queues and dispatchers involved in effect propagation. It is the job of the Orleans runtime to redistribute the grains to update the system with the new configuration.

RQ1: capable of handling input size?

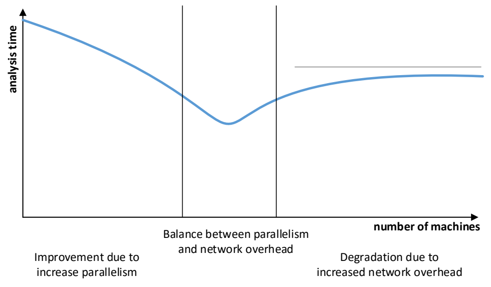
The memory consumption per worker VMs steadily decreases as the number of worker VMs increases.

5.4 [RQ2]: Scales with the # of Worker VMs

To answer RQ2, we proceeded to measure the total elapsed analysis time for each benchmark on all the configurations. Figure 16 in the Appendix shows the overall analysis and compilation times; the latter can be quite substantial (i.e., about 3 minutes to compile the larger benchmarks such as X100,000 and Azure-PW). Analysis time is compared to compilation time in Figure 17. Clearly, the analysis time is highly input size-specific. It is therefore instructive to normalize the analysis time by the input size. Figure 11 shows the elapsed analysis time *normalized* by the number of methods in the input.

Note that the real-world benchmarks shown on the right-hand side of the chart, despite containing fewer methods, require more time than the synthetic benchmarks with 100,000 methods. This is simply because of the analysis time that goes into analyzing more complex method bodies.

Real-world benchmarks allocate more objects per method, involving more type propagation time, and perform more virtual invocations, adding to the method resolution time, while the synthetic benchmarks only perform static invocations and allocate relatively few objects. As the number of worker VMs increases, we see a consistent drop in the normalized analysis times. However, this effect generally diminishes after 16 VMs. This has to do with the tension between more parallel processing power of more machines and the increase in the network overhead, as shown below.



It is instructive to focus on the average number of (unprocessed) messages in the analysis queues. If the queues are *too full*, adding more machines will increase the number of queues, reducing the size of each one. More machines will increase the parallelism because of more dispatchers to process the messages in the new queues. As we add more resources, however, when the queues become *mostly empty*, their associated dispatchers will be mostly idle. So the cluster as a whole will have more computing resources than needed. Additionally, if more machines are added, the probability of sending a message to a grain on the same machine as the sender will be reduced, leading to more network overhead. So after reaching a certain cut-off point, adding more machines is not only not helping the analysis, but starts to degrade its performance.

RQ2: does adding more worker VMs increase analysis time?

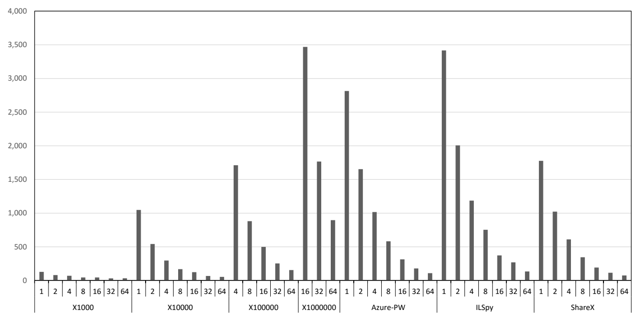
Normalized analysis time generally *decreases*, as the number of worker VMs increases, up to a point, where the law of diminishing returns kicks in.

[10] Note that we do not analyze libraries provided as DLLs; our analysis works at the source level only.

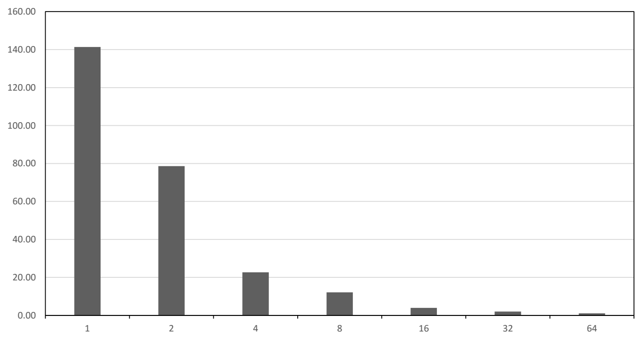
[11] Note also that for that benchmark, we needed to use at least 16 worker VMs to fit all the methods into (their shared) memory. We needed at least 4 worker VMs for X100,000.

Benchmark	URL	LOC	Projects	Classes	Methods	Main	Test	Event handlers	Public	Total	Reachable methods
Azure-PW	`https://github.com/Azure/azure-powershell`	416,833	60	2,618	23,617	0	997	1	18,747	18,759	23,663
ShareX	`https://github.com/ShareX/ShareX`	110,038	11	827	10,177	2	0	1,122	6,257	7,377	10,411
ILSpy	`https://github.com/icsharpcode/ILSpy`	300,426	14	2,606	25,098	1	0	119	14,343	14,498	21,944

Figure 9: Summary of information about real-world projects from GitHub. The number of reachable methods include also library methods invoked by the application methods. Note that some application methods might not be reachable.



(a) Average memory consumption in MB, for each benchmark as a function of the number of worker VMs. We see a steady decrease across the board.



(b) Average memory consumption in KB/method, as a function of the number of worker VMs used, averaged over all the synthetic benchmarks, normalized by the number of reachable methods. We similarly observe a steady decrease as the number of worker VMs goes up. Fitting an exponential trend line to this data gives us the following formula: $M = 169.09/e^{0.728 \cdot m}$ with $R^2 = 0.99545$.

Figure 10: Average memory consumption.

5.5 [RQ3]: Fast Enough for Interactive Queries

One of the goals of our approach is to enable interactive queries submitted by an analysis client such as an IDE or a sophisticated code editor. In such a setting, responsiveness of such queries is paramount [36]. The user is unlikely to be happy with an IDE that takes several seconds to populate a list of auto-complete suggestions. We want to make sure that as the query times remain tolerable (under 20 ms) even as the size of input increases and the number of VMs goes up.

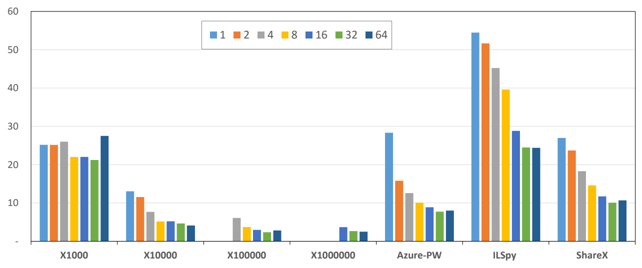


Figure 11: Elapsed analysis time in *ms*, as a function of the number of worker VMs per test, normalized by the number of reachable methods. The number of worker VMs is indicated in color in the legend above the figure.

To evaluate query performance, we automatically generated sequences of 100 random queries, by repeating the following process. We would first pick a random method name from the list of all methods. Then we would (1) Request the solution grain for the corresponding method grain; (2) Select a random invocation from method and request the set of potential callees. In Figure 12 we show the mean and median query times (the latency of the two steps above) for each benchmark and worker VM configuration.

RQ3: is response latency small enough?

The query median response time is consistently between 10 and 20 ms. Increasing the number of worker VMs and the input size does not negatively affect the query response times.

6. Related Work

There exists a wealth of related work on traditional static analysis algorithms such as call graph construction. While we have seen dedicated attempts to scale up important analyses such as points-to in the literature, we are unaware of projects that aim to bring analysis to the cloud.

6.1 Analysis

While our approach is general, the two types of analysis below are most relevant for our effort. Note, however, that given that our approach is designed for program

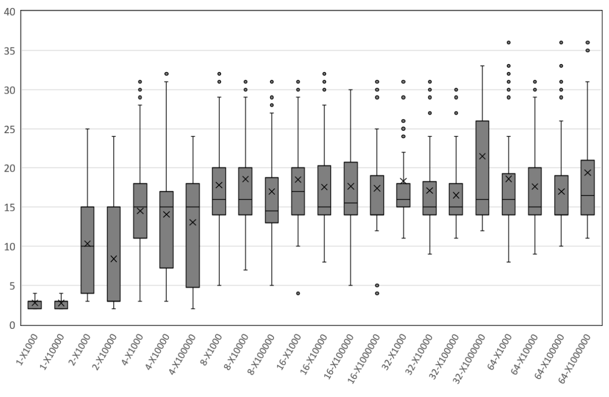


Figure 12: Mean and median query time in *ms* as a function of the number of worker VMs for each of the synthetic tests.

understanding, we are satisfied with soundiness [27] and not full soundness.

Concrete types: Most of the work in concrete type inference for object-oriented programs goes back to the early 1990s [2, 3, 12, 37, 39]. Many of these techniques are now considered standard. Concrete type inference generally does not require the same complexity as a content-sensitive points-to analysis [25] would and scales better as a result.

Call graph construction: Call graph construction for object-oriented code was explored in the 1990s, with standard algorithms such CHA and VTA proposed at that time [17, 18, 42]. A comparison of analysis precision is presented in Lhoták *et al.* [26]. Some of the recent work focuses on call graph construction in the presence of frameworks and libraries [6, 29]. We largely skirt that issue in this paper, focusing on input being provided to us in the form of source code.

6.2 Scaling up Static Analysis

Points-to analysis: Hardekopf *et al.* [20] show how to scale up a flow-sensitive points-to analysis of LLVM code using a staged approach. Their flow-sensitive algorithm is based on a sparse representation of program code created by a staged, flow-insensitive pointer analysis. They are able to analyze 1.9M LOC programs in under 14 minutes. Their largest benchmark, however, required a machine with 100 GB of memory, which is generally beyond the reach of most people. Our focus, in contrast, is on using legacy, low-cost hardware.

Hardekopf *et al.* [19] introduce and evaluate two novel techniques for inclusion-based pointer analysis that significantly improve scalability without negatively impacting precision. These techniques focus on the problem of online cycle detection, a critical optimization for scaling such analyses. The combination of their techniques is on average 3.2× faster than Heintze and Tardieu's algorithm [21], and 6.4× faster than Pearce *et al.*'s algorithm [38], and 20.6× faster than Berndl [9].

Yu *et al.* [44] propose a method for analyzing pointers in a program level by level in terms of their points-to levels. This strategy enhances the scalability of a context- and flow-sensitive pointer analysis. They demonstrate that their analysis can handle some programs with over a million lines of C code in minutes.

Mendez-Lojo *et al.* [31] propose a parallel analysis algorithm for inclusion-based pointer analysis and show a speed up of up to 3× on an 8-core machine on code bases with size varying from 53 KLOC to 0.5 MLOC. Our focus is on bringing our approach to the cloud and going beyond multicore.

Voung *et al.* [43] propose a technique that uses the notion of a *relative lockset*, which allows functions to be summarized independent of the calling context. This, in turn, allows them to perform a modular, bottom-up analysis that is easy to parallelize. They have analyzed 4.5 million lines of C code in 5 hours, and after applying some simple filters, found a total of 53 races.

Frameworks: Albarghouthi *et al.* [5] present a generic framework to distribute top-down algorithms using a map-reduce strategy. Their focus is in obtaining speed ups in analysis elapsed times. Even tough they report some potential improvements they admit that one important limiting scaling factor is memory consumption and propose distributing their algorithm as future work. McPeak *et al.* [30] propose a multicore analysis that allows them to handle millions of lines of code in several hours on an 8-core machine. In contrast, our approach focuses on analysis within a cloud cluster on often less powerful hardware.

Boa (Dyer *et al.* [14–16]) is a domain-specific language for mining large code repositories like GitHub to answer questions such as "how many Java projects use SVN?" or "how many projects use a specific Java language feature over the years". Boa runs these queries on a map-reduce cluster. However, while it uses a distributed backend, Boa is not a static analysis.

7. Conclusions

As modern development is increasingly moving to large online cloud-backed repositories such as GitHub, BitBucket, and Visual Studio Online, there is a natural tendency to wonder what kind of analysis can be performed on large bodies of code. In this paper, we explore an analysis architecture in which static analysis is executed on a *distributed cluster* composed of legacy VMs available from a commercial cloud provider.

We present the design and implementation of the first static analysis approach designed for *elasticity*, i.e. to scale gracefully with the size of the input. Our static analysis is based on the actor programming model build on top of the Orleans framework and deployed in Microsoft Azure. To demonstrate the potential of our techniques for static analysis, we show how a typical call graph analysis can be implemented.

The analysis presented in this paper is able to handle inputs that are almost 10 million LOC in size. Our results show that our analysis scales well in terms of memory pressure independent of the input size, as we add more VMs. Despite using stock hardware and incurring a non-trivial communication overhead, our processing time for some of the benchmarks of close to 1 million LOC can be about 5 minutes, excluding compilation time. As the number of analysis VMs increases, we show that the analysis time does not suffer. Lastly, we demonstrate that querying the results can be performed with a median latency of 15 ms.

In our future work we plan to investigate how to increase the analysis throughput. We also want to understand how to combine distributed processing with incremental analysis: we are ultimately interested in designing a distributed analysis that can respond quickly to frequent updates in the code repository. We plan to incorporate the analysis into an IDE and to also perform user studies.

References

[1] Representational state transfer. `https://en.wikipedia.org/wiki/Representational_state_transfer`, 2015.

[2] O. Agesen. *Concrete type inference: delivering object-oriented applications.* PhD thesis, Stanford University, 1996.

[3] O. Agesen and U. Hölzle. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. *ACM SIGPLAN Notices*, 1995.

[4] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.

[5] A. Albarghouthi, R. Kumar, A. V. Nori, and S. K. Rajamani. Parallelizing top-down interprocedural analyses. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2012.

[6] K. Ali and O. Lhoták. Application-only call graph construction. In *Proceedings of the European Conference on Object-Oriented Programming*, 2012.

[7] L. O. Andersen. *Program analysis and specialization for the C programming language.* PhD thesis, University of Cophenhagen, 1994.

[8] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, 1996.

[9] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using bdds. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2003.

[10] P. A. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, Microsoft Research, 2014.

[11] J. Bornholt and E. Torlak. Scaling program synthesis by exploiting existing code. *Machine Learning for Programming Languages*, 2015.

[12] R. Chatterjee, B. G. Ryder, and W. A. Landi. Complexity of concrete type-inference in the presence of exceptions. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, 1998.

[13] P. Cousot. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Technical report, Laboratoire IMAG, Université scientifique et médicale de Grenoble, 1977.

[14] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the International Conference on Software Engineering.* IEEE Press, 2013.

[15] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen. Mining billions of ast nodes to study actual and potential usage of Java language features. In *Proceedings of the International Conference on Software Engineering.* ACM, 2014.

[16] R. Dyer, H. Rajan, and T. N. Nguyen. Declarative visitors to ease fine-grained source code mining with full history on billions of ast nodes. In *ACM SIGPLAN Notices.* ACM, 2013.

[17] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 2001.

[18] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. *ACM SIGPLAN Notices*, 1997.

[19] B. Hardekopf and C. Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2007.

[20] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2011.

[21] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2001.

[22] M. S. Lam, J. Whaley, B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the Symposium on Principles of Database Systems*, June 2005.

Appendix

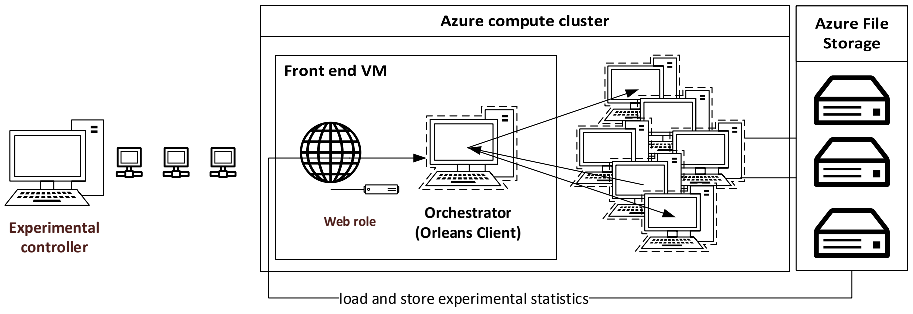
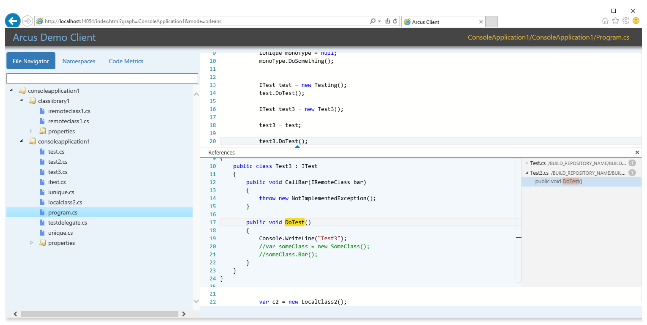


Figure 13: Azure-based deployment of our analysis. The majority of work happens within an Azure cluster, within worker VMs. The analysis client interacts with the cluster via a front-end VM.

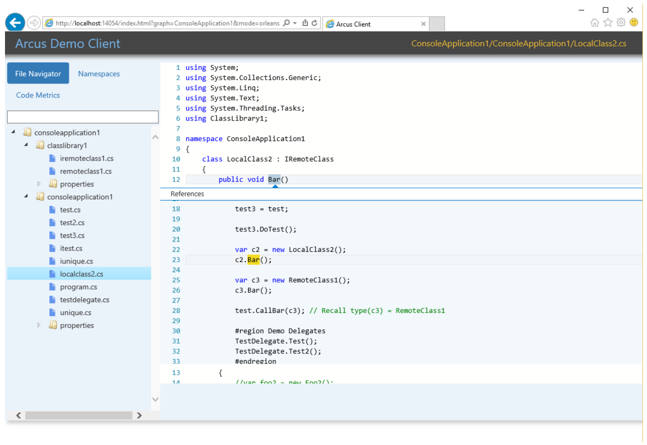


```
9           IUnique monoType = null;
10          monoType.DoSomething();
11
12
13          ITest test = new Testing();
14          test.DoTest();
15
16          ITest test3 = new Test3();
17
18          test3 = test;
19
20          test3.DoTest();
```

```
10      public class Test3 : ITest
11      {
12          public void CallBar(IRemoteClass bar)
13          {
14              throw new NotImplementedException();
15          }
16
17          public void DoTest()
18          {
19              Console.WriteLine("Test3");
20              //var someClass = new SomeClass();
21              //someClass.Bar();
22          }
23      }
24  }
```

```
21
22          var c2 = new LocalClass2();
```

(a) Visualizing callees: call site on line 20 invokes function `DoTest` on line 17.



```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6  using ClassLibrary1;
7
8  namespace ConsoleApplication1
9  {
10     class LocalClass2 : IRemoteClass
11     {
12         public void Bar()
```

```
18              test3 = test;
19
20              test3.DoTest();
21
22              var c2 = new LocalClass2();
23              c2.Bar();
24
25              var c3 = new RemoteClass1();
26              c3.Bar();
27
28              test.CallBar(c3); // Recall type(c3) = RemoteClass1
29
30              #region Demo Delegates
31              TestDelegate.Test();
32              TestDelegate.Test2();
33              #endregion
```

```
13         {
14              //var foo2 = new Foo2();
```

(b) Visualizing callers: method `Bar` defined on line 12 is called on line 23.

Figure 14: An experimental online IDE that uses analysis for resolving references for callees and callers.

Task issued by the client	URL for the REST call	Server-side request handler
Get all abstract locations in a source code document	`http://<hostname>:49176/api/Orleans?filePath=program.cs`	`[HttpGet] public async Task<IList<FileResponse>> GetFileEntitiesAsync(string filePath)`
Get symbol references	`http://<hostname>:49176/api/Orleans?ruid=Program.Main`	`[HttpGet] public async Task<IList<SymbolReference>> GetReferencesAsync(string ruid)`
Get symbol definitions	`http://<hostname>:49176/api/Orleans?ruid=Program.Main@2`	`[HttpGet] public async Task<IList<SymbolReference>> GetReferencesAsync(string ruid)`

Figure 15: Examples of interacting with the analysis backend via REST queries.

	X1,000							**X10,000**							**X100,000**					**X1,000,000**		
Machines	1	2	4	8	16	32	64	1	2	4	8	16	32	64	4	8	16	32	64	16	32	64
Compilation	6	11	9	20	28	32	49	48	57	68	58	67	72	88	188	205	204	207	259	281	215	352
Analysis	25	25	26	22	22	21	28	130	115	77	52	52	47	41	609	371	298	237	284	3,704	2,666	2,514

	Azure-PW							**ILSPY**							**ShareX**						
Machines	1	2	4	8	16	32	64	1	2	4	8	16	32	64	1	2	4	8	16	32	64
Compilation	305	266	269	272	276	285	308	121	115	177	216	129	189	165	85	94	88	105	108	121	162
Analysis	670	373	298	238	210	183	190	1,281	1,214	1,063	931	677	576	568	280	246	190	152	122	105	111

Figure 16: Analysis and compilation times for synthetic and real benchmarks (in *seconds*).

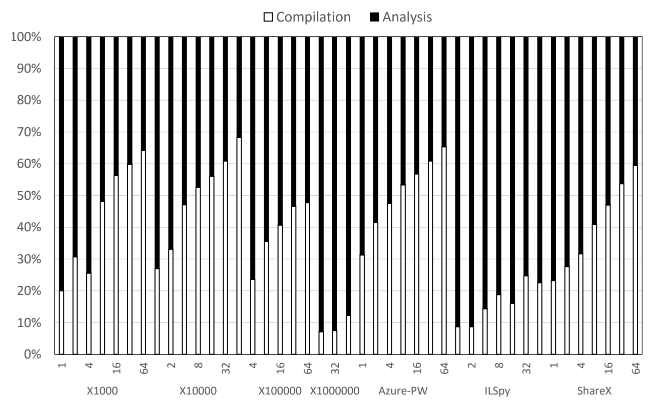


Figure 17: Compilation time compared to analysis time for the benchmarks across a number of VM configurations, in *ms*. Analysis time can be *relatively* large when fewer VMs are involved. For 16 or 64 VMs, the analysis time can often be half of compilation time.

[23] J.-L. Lassez, V. Nguyen, and E. Sonenberg. Fixed point theorems and semantics: A folk tale. *Information Processing Letters*, 1982.

[24] Y. Y. Lee, S. Harwell, S. Khurshid, and D. Marinov. Temporal code completion and navigation. In *Proceedings of the International Conference on Software Engineering*, 2013.

[25] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *Proceedings of the Conference on Compiler Construction*, 2003.

[26] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: Is it worth it? In *Proceedings of the International Conference on Compiler Construction*, 2006.

[27] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 2015.

[28] N. P. Lopes and A. Rybalchenko. Distributed and predictable software model checking. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*, 2011.

[29] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, 2013.

[30] S. McPeak, C.-H. Gros, and M. K. Ramanathan. Scalable and incremental software bug detection. In *Proceedings of the Symposium on the Foundations of Software Engineering*, 2013.

[31] M. Mendez-Lojo, A. Mathew, and K. Pingali. Parallel inclusion-based points-to analysis. In *Conference on Object Oriented Programming Systems Languages and Applications*, 2010.

[32] Microsoft Corporation. .NET Compiler Platform ("Roslyn"). `https://roslyn.codeplex.com/`, 2015.

[33] Mozilla. LXR. `https://en.wikipedia.org/wiki/LXR_Cross_Referencer`, 2015.

[34] K. Murray, J. P. Bigham, et al. Beyond autocomplete: Automatic function definition. In *Proceedings of the Visual Languages and Human-Centric Computing Symposium*, 2011.

[35] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the International Conference on Software Engineering*.

[36] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active code completion. In *Proceedings of the International Conference on Software Engineering*, 2012.

[37] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *Proceedings of the Conference on Object-oriented Languages, Systems, and Applications*, 1991.

[38] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis for C. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, 2004.

[39] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. *ACM SIGPLAN Notices*, 1994.

[40] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM, 1996.

[41] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2000.

[42] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2000.

[43] J. W. Voung, R. Jhala, and S. Lerner. Relay: Static race detection on millions of lines of code. In *Proceedings of the Symposium on the Foundations of Software Engineering*, 2007.

[44] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: Making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2010.