The Complexity of Andersen's Analysis in Practice

Manu Sridharan and Stephen J. Fink

IBM T.J. Watson Research Center
{msridhar,sjfink}@us.ibm.com

Abstract. While the tightest proven worst-case complexity for Andersen's points-to analysis is nearly cubic, the analysis seems to scale better on real-world codes. We examine algorithmic factors that help account for this gap. In particular, we show that a simple algorithm can compute Andersen's analysis in worst-case quadratic time as long as the input program is *k-sparse*, *i.e.*, it has at most k statements dereferencing each variable and a sparse flow graph. We then argue that for strongly-typed languages like Java, typical structure makes programs likely to be k-sparse, and we give empirical measurements across a suite of Java programs that confirm this hypothesis. We also discuss how various standard implementation techniques yield further constant-factor speedups.

1 Introduction

The scalability of Andersen's points-to analysis [1] has received much attention, as scalable points-to analysis lies on the critical path for many static analyses. Andersen's analysis implementations [6, 9, 13, 16, 20, 22, 27, 29] have scaled to increasingly large programs through clever algorithms and careful engineering. Despite this progress, the best known worst-case complexity bound for Andersen's analysis remains (nearly) cubic.[1]

If Andersen's analysis required time cubic in program size for *typical* inputs, implementations would not scale as well as the literature reports. On the contrary, experiences from real implementations suggest that the worst-case bound rarely governs performance in practice. For example, Heintze and McAllester suggested that a standard algorithm for 0-CFA [23], similar to Andersen's analysis, "rarely exhibit[s] cubic behavior" [10]. Similarly, Goldsmith et al. [8] observed no worse than quadratic scaling for an implementation of Andersen's analysis for C [14].

In this paper, we show that (1) Andersen's analysis can be computed in worst-case quadratic time for a restricted class of inputs and that (2) realistic Java programs usually belong to this class.

Andersen's analysis can be formulated as a *dynamic transitive closure* problem over a *flow graph* representing the flow of pointers in the program, where the

[1] One can reduce Andersen's analysis to CFL-reachability [21, 25], for which Chaudhuri's algorithm is slightly sub-cubic [5].

flow graph grows dynamically to capture value flow through pointer dereferences (details in §3). We show that when the input program is *k-sparse*, Andersen's analysis can be computed in quadratic time. A k-sparse program must have (1) at most k statements dereferencing each variable and (2) a sparse flow graph *at analysis termination*, *i.e.*, including all dynamically inserted edges. The key insight behind the bound is that *difference propagation* [7, 20] limits both the closure work and edge insertion work to quadratic time for k-sparse programs (details in §4).

For strongly-typed languages like Java, program structure typically constrains programs to be k-sparse and hence analyzable in quadratic time. In particular, types associated with heap allocated data in Java limit flow analysis to a small number of named instance fields per object. Furthermore, modularity of typical Java programs—in particular, the common use of "getter" and "setter" methods to encapsulate field accesses—limits the number of statements accessing any particular field. We show that programs with no arrays or dynamic dispatch that use fields in this manner must be k-sparse, and we argue that dispatch and arrays usually do not materially compromise k-sparseness in practice. Furthermore, we present empirical evidence of k-sparseness across a suite of large Java programs.

Perhaps surprisingly, our quadratic complexity bound does not rely on many engineering details that have been shown to have a significant impact on the real-world performance of Andersen's analysis. We include a discussion of many of these issues (set implementation, worklist ordering, cycle elimination, etc.), describing potential constant-factor speedups, space consumption, and the relative importance of the techniques for strongly-typed languages like Java vs. weakly-typed languages like C.

Contributions This paper makes the following contributions:

- We show that a simple algorithm for Andersen's analysis runs in worst-case quadratic time for k-sparse programs.
- We show that Java's type system and typical program structure suggest that realistic programs will be k-sparse.
- We present measurements across a suite of large Java programs that show they are k-sparse, with the number of flow graph edges being at most 4.5X the number of nodes. We also show that the implementation we tested [28] scales roughly quadratically.

2 Andersen's Analysis for Java

Here, we briefly review Andersen's points-to analysis [1] for Java, as treated in detail in previous work [16, 22, 25, 29]. A points-to analysis computes an overapproximation of the heap locations that program variables may point to, where a finite heap abstraction represents configurations of the runtime heap. The analysis result is typically represented as a *points-to set* $pt(x)$ for each variable x. Andersen's points-to analysis has the following properties:

Statement	Constraint	
`i: x = new T()`	$\{o_i\} \subseteq pt(x)$	[New]
`x = y`	$pt(y) \subseteq pt(x)$	[Assign]
`x = y.f`	$\dfrac{o_i \in pt(y)}{pt(o_i.f) \subseteq pt(x)}$	[Load]
`x.f = y`	$\dfrac{o_i \in pt(x)}{pt(y) \subseteq pt(o_i.f)}$	[Store]

Table 1. Canonical statements for Java points-to analysis and the corresponding points-to set constraints.

- *Abstract location per allocation*: The heap abstraction represents all objects potentially allocated by a statement with a single abstract location.
- *Flow insensitive*: The analysis assumes statements in a procedure can execute in any order and any number of times.
- *Subset based*: The analysis models directionality of assignments; *i.e.*, a statement `x = y` implies $pt(y) \subseteq pt(x)$. In contrast, an equality-based analysis (*e.g.*, [26]) would require $pt(y) = pt(x)$ for the same statement, a coarser approximation.

As is typical for Java points-to analyses, we also desire *field sensitivity*, which requires separate reasoning about each abstract location instance field. Finally, we restrict our attention to *context-insensitive* analysis, which computes a single result for each procedure, merging the behaviors from all call sites.

Table 1 presents the canonical statements for Java points-to analysis and the corresponding points-to set constraints, as seen previously (*e.g.*, in [16]). More complex statements (*e.g.*, `x.f = y.g.h`) are handled through suitable introduction of temporary variables. We assume that all accesses to global variables (static fields in Java) occur via a copy assignment to a local, implying that load and store statements do not directly dereference globals; this model matches Java `putstatic` and `getstatic` bytecodes. Since we focus on context-insensitive analysis, we elide method calls and assume copy statements for parameter passing and return values via some precomputed call graph. (§5.2 discusses on-the-fly call graph construction.) Arrays are modeled with a single field representing the contents of all array indices. For simplicity, we ignore the effects of reflection and native code in this exposition.

3 Algorithm

Here we present an algorithm for Andersen's analysis for Java, as specified in §2. The algorithm is most similar to Pearce et al.'s algorithm for C [20] and also resembles existing algorithms for Java (*e.g.*, [16]).

The algorithm constructs a flow graph G representing the pointer flow for a program and computes its (partial) transitive closure, a standard points-to analysis technique (*e.g.*, see [6, 10, 13]). G has nodes for variables, abstract locations, and fields of abstract locations. At algorithm termination, G has an edge $n \to n'$ iff one of the following two conditions holds:

1. n is an abstract location o_i representing a statement `x = new T()`, and n' is `x`.
2. $pt(n) \subseteq pt(n')$ according to some rule in Table 1.

Given a graph G satisfying these conditions, it is clear that $o_i \in pt(x)$ iff x is reachable from o_i in G. Hence, the transitive closure of G—where only abstract location nodes are considered sources—yields the desired points-to analysis result.

Since flow relationships for abstract location fields depend on the points-to sets of base pointers for field accesses (see [Load] and [Store] rules referencing $pt(o_i.f)$ in Table 1), certain edges in G can only be inserted after some reachability has been determined, yielding a *dynamic transitive closure* (DTC) problem. Note that Andersen's analysis differs from a general DTC problem in two key ways. First, unlike general DTC, edge deletions from G need not be handled. Second, as observed in [10], adding new edges to G is part of the points-to analysis work—edge insertion work is typically not considered in discussions of DTC algorithms. The second point is important, as we must consider edge insertion work when reasoning about analysis complexity.

Pseudocode for the analysis algorithm appears in Figure 1. The DoAnalysis routine takes a set of program statements of the forms shown in Table 1 as input. (We assume suitable data structures that, given a variable `x`, yield all load statements `y = x.f` and store statements `x.f = y` in constant time per statement.) The algorithm maintains a flow graph G as just described and computes a points-to set $pt(x)$ for each variable x, representing the transitive closure in G from abstract locations. Note that abstract location nodes are eschewed, and instead the relevant points-to sets are initialized appropriately (line 2).

The algorithm employs *difference propagation* [7, 16, 20] to reduce the work of propagating reachability facts. For each node n in G, $pt_\Delta(n)$ holds those abstract locations o_i such that (1) the algorithm has discovered that n is reachable from o_i and (2) this reachability information has not yet propagated to n's successors in G. $pt(n)$ holds those abstract locations for which (1) holds and propagation to successors of n is complete. The DiffProp routine updates a difference set $pt_\Delta(n)$ with those values from *srcSet* not already contained in $pt(n)$. After a node n has been removed from the worklist and processed, all current reachability information has been propagated to n's successors, so $pt_\Delta(n)$ is added to $pt(n)$ and emptied (lines 21 and 22).

```
DoAnalysis()
 1  for each statement i: x = new T() do
 2          pt_Δ(x) ← pt_Δ(x) ∪ {o_i}, o_i fresh
 3          add x to worklist
 4  for each statement x = y do
 5          add edge y → x to G
 6  while worklist ≠ ∅ do
 7          remove n from worklist
 8          for each edge n → n' ∈ G do
 9                  DiffProp(pt_Δ(n), n')
10          if n represents a local x
11            then for each statement x.f = y do
12                          for each o_i ∈ pt_Δ(n) do
13                                  if y → o_i.f ∉ G
14                                    then add edge y → o_i.f to G
15                                          DiffProp(pt(y), o_i.f)
16                  for each statement y = x.f do
17                          for each o_i ∈ pt_Δ(n) do
18                                  if o_i.f → y ∉ G
19                                    then add edge o_i.f → y to G
20                                          DiffProp(pt(o_i.f), y)
21          pt(n) ← pt(n) ∪ pt_Δ(n)
22          pt_Δ(n) ← ∅

DiffProp(srcSet, n)
1  pt_Δ(n) ← pt_Δ(n) ∪ (srcSet − pt(n))
2  if pt_Δ(n) changed then add n to worklist
```

Fig. 1. Pseudocode for the points-to analysis algorithm.

Theorem 1 DoAnalysis *terminates and computes the points-to analysis result specified in Table 1.*

Proof. (*Sketch*) DoAnalysis terminates since (1) the constructed graph is finite and (2) a node n is only added to the worklist when $pt_\Delta(n)$ changes (line 2 of DiffProp), which can only occur a finite number of times. For the most part, the correspondence of the computed result to the rules of Table 1 is straightforward. One subtlety is the handling of the addition of new graph edges due to field accesses. When an edge $y \rightarrow o_i.f$ is added to G to handle a putfield statement (line 14), only $pt(y)$ is propagated across the edge, not $pt_\Delta(y)$ (line 15). This operation is correct because if $pt_\Delta(y) \neq \emptyset$, then y must be on the worklist, and hence $pt_\Delta(y)$ will be propagated across the edge when y is removed from the worklist. A similar argument holds for the propagation of $pt(o_i.f)$ at line 20. □

4 Complexity for k-Sparse Programs

Here, we show that the algorithm of Figure 1 has quadratic worst-case time complexity for k-sparse input programs (§4.1). We then argue that, due to strong types and typical program structure, realistic Java programs are likely to be k-sparse (§4.2).

4.1 Quadratic Bound

Let N be the number of variables in an input program plus the number of `new` statements in the program (*i.e.*, the number of abstract locations in the heap abstraction). [2] Also, let $D(x)$ be the number of statements dereferencing variable x, and let E be the number of edges in G at analysis termination. We show that DoAnalysis from Figure 1 runs in worst-case $O(N^2 \max_x D(x) + NE)$ time.

Definition 1 *A program is k-sparse if (1) $\max_x D(x) \leq k$ and (2) $E \leq kN$.*

For k-sparse programs with k being constant, DoAnalysis runs in worst-case $O(N^2)$ time.

Note that our definition of k-sparsity depends on both the input program *and* the analysis computing its flow graph. In particular, variants of Andersen's analysis with different levels of context sensitivity may compute different flow graphs for the same input program, yielding different values of k. When discussing k-sparsity in this paper, we assume flow graphs are constructed with the context-insensitive analysis described in §2.[3]

We begin with a key lemma characterizing the effect of difference propagation.

Lemma 1 *For each abstract location o_i and node p in G, there is at most one execution of the loop at lines 6-22 of* DoAnalysis *for which $n = p \wedge o_i \in pt_\Delta(p)$.*

Proof. If there exists a loop execution where $n = p \wedge o_i \in pt_\Delta(p)$, line 21 of DoAnalysis adds o_i to $pt(p)$ and line 22 removes o_i from $pt_\Delta(p)$. Subsequently, $pt_\Delta(p)$ may only be modified by line 1 of DiffProp, which ensures that elements of $pt(p)$ cannot be re-added to $pt_\Delta(p)$. □

DoAnalysis does four kinds of work:

1. *Initialization*: Lines 1 through 5, which handle `new` statements and add the initial edges to G.
2. *Edge Adding*: Lines 13, 14, 18, and 19, which add new edges to or from abstract location field nodes in G as needed.

[2] Note that N must be no greater than the number of statements in a program, since each statement can introduce at most one variable.

[3] Alternately, k-sparsity could be defined in terms of the possible value flow in the input program with a dynamic semantics that matches the pointer analysis model; we include the flow graph in the definition for clarity.

3. *Propagation*: All calls to the DIFFPROP routine.
4. *Flushing Difference Sets*: Lines 21 and 22.

The cost of the algorithm is the sum of the costs of these four types of work, which we analyze in turn. In this sub-section, we assume a points-to set data structure which allows for (1) constant time membership checks, (2) constant time addition of a single element, and (3) iteration in constant time per set element, *e.g.*, an array of bits (for (1) and (2)) combined with a linked list (for (3)). (Note that this is not a space-efficient data structure; we discuss space / time tradeoffs in §5.1 and §5.4.) We also assume a worklist data structure that prevents duplicate worklist entries and allows for constant-time removal of a node.

Initialization The loop from lines 1 to 3 clearly takes $O(N)$ time. The loop on lines 4 and 5 takes time proportional to the number of copy assignments in the program, which could be $O(N^2)$ in the worst case. Hence, we have an $O(N^2)$ bound for initialization.[4]

Edge Adding We assume a suitable graph data structure so that lines 13, 14, 18, and 19 each execute in constant time. For each statement dereferencing a given variable x, the algorithm performs at most $|pt(x)|$ edge adding work, since by Lemma 1 the loops headed at lines 12 and 17 can execute at most $|pt(x)|$ times per such statement. Since $|pt(x)|$ is $O(N)$ and we have $O(N)$ variables, the edge adding work is bounded by $O(N^2 \max_x D(x))$.

Propagation To reason about propagation work, we first prove the following lemmas.

Lemma 2 *For each graph node p, at any time during execution of* DOANALYSIS *except between lines 21 and 22, $pt_\Delta(p) \cap pt(p) = \emptyset$.*

Proof. The condition clearly holds before the first iteration of the loop starting at line 6. Afterward, for any node p, $pt_\Delta(p)$ can only be changed by line 22 or by a call to DIFFPROP. The condition clearly holds after line 22 since $pt_\Delta(n)$ is emptied. DIFFPROP also preserves the condition, since it only adds abstract locations to $pt_\Delta(n)$ that are not contained in $pt(n)$. □

Lemma 3 *For each abstract location o_i and each edge $e = n \to n'$ in G, o_i is propagated across e at most once during execution of* DOANALYSIS.

Proof. Propagation across edges occurs via the DIFFPROP calls at lines 9, 15, and 20. By Lemma 1, line 9 can propagate o_i across e at most once. By Lemma 2, we know that $pt_\Delta(y) \cap pt(y) = \emptyset$ at line 15 and that elements from $pt(y)$ cannot later be re-added to $pt_\Delta(y)$. Hence, if an abstract location is propagated across

[4] A tighter bound would be the number of statements, which we expect to be $O(N)$ in practice. This is irrelevant to our proof since initialization costs are dominated by edge adding.

$y \rightarrow o_i.f$ at line 15, it cannot again be propagated across the same edge by line 9 in a later loop iteration. Similar reasoning holds for the propagation at line 20. □

By Lemma 3, propagation work is bounded by $O(NE)$, where E is the final number of edges in G.

Flushing Difference Sets By Lemma 1, each abstract location can be flushed from a difference set at most once per variable, immediately yielding an $O(N^2)$ bound for this work.

In the worst case, the work of initialization and flushing difference sets will be dominated by edge adding and propagation. So, we have a worst-case bound of $O(N^2 \max_x D(x) + NE)$ for the algorithm, or $O(N^2)$ for k-sparse programs (see Definition 1), as desired.

We note that if the average points-to set size is $O(N)$, the $O(N^2)$ bound for k-sparse programs is tight, as the result itself would be quadratic in the size of the program. In §6, we give evidence that average points-to set size grows with the size of the program (see Figure 3(a)). Sub-quadratic bounds may be possible in practice for clients that do not require all points-to sets [10].

4.2 Realistic Java Programs

Here, we argue that the Java type system and typical program structure imply that realistic Java programs are likely to be k-sparse. In particular, we show that method size limits imply that $\max_x D(x)$ is bounded and that the type system and modular programming imply that G will most likely be sparse.

The structure of Java methods ensures that $\max_x D(x)$ does not grow with program size. For $\max_x D(x)$ to grow with program size, methods would have to become larger in bigger programs.[5] In practice, Java programs tend to have many small methods, and in fact the Java virtual machine enforces a fixed bound on method size.

Java programs must have $E = O(N)$ if they exclusively use "getters" and "setters" to access fields and do not use arrays or dynamic dispatch. Edges in G correspond to either (1) copy assignments or (2) flow through abstract location fields. For (1), the number of intraprocedural copy assignments in a program can only grow linearly (due to limited method size), and without dynamic dispatch, the number of interprocedural copies must grow linearly as well. For (2), we can bound the number of inserted edges for abstract location fields as follows. Let K be the maximum number of accesses of any field, and let F be the maximum number of instance fields in any class. The maximum number of field edges is $O(NFK)$: there are at most NF abstract location field nodes, and each such node can have at most K incident edges. Java enforces a bound on class size, which yields a constant limit for F. Furthermore, if all fields are only accessed via

[5] This assumes that static fields cannot be directly dereferenced, which holds for Java bytecode.

"getter" and "setter" methods, then $K = 2$ for all fields.[6] Hence, the number of edge insertions for abstract location fields must be $O(N)$ for this class of programs.

In practice, the number of statements accessing any given field tends to be small due to encapsulation. However, the synthetic field *arr* used to model array contents is an exception: the number of array access statements (*i.e.*, accesses of *arr*) increases with the size of the program, potentially leading to a quadratic number of flow graph edges. If the base pointers of array accesses have points-to sets of bounded size, then only a linear number of inserted edges will be required for these accesses, maintaining flow graph sparseness. While array base pointers usually have small points-to sets (again due to encapsulation), exceptions can occur due to context-insensitive analysis of frequently used library methods; we discuss this issue further in §6.

Hypothetically, dynamic dispatch could also cause a quadratic number of flow graph edges, in the case where there were $O(N)$ call sites of a method, each of which could dispatch to $O(N)$ possible targets. In practice, this phenomenon could only occur for methods defined in the root `java.lang.Object` class like `toString()`, and its likelihood is mitigated by on-the-fly call graph construction (see §5.2); we have not observed such a blowup in practice.

5 Other Factors

The literature presents myriad other implementation techniques that, at the least, yield significant constant-factor time improvements. Furthermore, when performing points-to analysis on large programs, space concerns often dominate, necessitating space-saving techniques that complicate analysis of running time. In this section, we briefly discuss several other factors relevant to Andersen's analysis performance and relate them to our complexity result.

5.1 Bit-Vector Parallelism and Worklist Ordering

The use of bit-wise operations for propagation can yield significant constant-factor speedups in practice. The complexity proof of §4 assumes that abstract locations are propagated across edges one at a time. With an appropriate set representation, bit-wise operators can effectively propagate up to k abstract locations across an edge in constant time, where k is the machine word size (*e.g.*, 64). When using such operations, our proof of a quadratic bound no longer applies, since propagation across an edge becomes proportional to the total number of abstract locations instead of the size of the source set. Nevertheless, using bit-wise operations usually improves performance in practice, since the cost model on real machines usually depends more on cache locality than the number of register-level arithmetic instructions.

[6] Note that this bound relies on the context-*insensitive* analysis of the these methods, *i.e.*, it is independent of the number of calls to the methods.

The speedup due to bit-wise operations depends on an effective worklist ordering [19]. For a node n, the analysis would ideally complete all propagation to n before removing n from the worklist, since this maximizes the benefits of using bit-wise operations when propagating to n's successors. If the analysis only required computing standard transitive closure over a DAG, the best worklist ordering would be topological, in which case the algorithm would propagate across each edge at most once.

With dynamic transitive closure, even if the final flow graph G is a DAG, it may not be possible to do propagation in topological order due to cyclic data dependences. Consider the following example program:

```
x = new Obj(); // o1
z = new Obj(); // o2
y = x; y.f = z; x = y.f;
```

Note that `y = x` and `x = y.f` are cyclically data dependent on each other. Initially, G only contains the edge $e = x \rightarrow y$. After o_1 is propagated across e, the analysis can add incident edges for $o_1.f$, yielding the graph $z \rightarrow o_1.f \rightarrow x \rightarrow y$. After these edge insertions, the analysis must repeat propagation across e to add o_2 to $pt(y)$. Hence, repeated work across edges may be required even when G is a DAG. It would be interesting to characterize how much cyclic data dependences affect performance in practice.

In real programs, cycles in the flow graph G further complicate matters. Online cycle elimination can lessen the impact of flow graph cycles, as we shall discuss further in §5.5. In WALA [28], the analysis implementation used for our measurements, worklist order is determined by a pseudo-topological ordering of the flow graph, periodically updated as edges are added. Further discussion of worklist ordering heuristics appears in [19].

5.2 Function Calls

Direct handling of higher-order functions, *i.e.*, on-the-fly call graph construction, does not affect the $O(N^2)$ bound for k-sparse programs. For Java, on-the-fly call graph construction requires (1) reasoning about possible virtual call targets using receiver points-to sets and (2) incorporating constraints for discovered call targets, as described previously [16, 22, 29]. Both of these operations can be performed for all relevant call sites in quadratic time, and the core propagation and edge adding operations of the analysis are unaffected.

Though it does not affect our worst-case bound, on-the-fly call graph building has a significant impact on real-world performance. If constraint generation costs are ignored, on-the-fly call graph reasoning can slow down analysis, as more iterations are required to reach a fixed point [29]. However, if the costs of constraint generation are considered (which we believe is a more realistic model), on-the-fly call graph building improves performance, since constraints need not be generated for unreachable library code. Also, as suggested in §4.2, on-the-fly call graph reasoning can make the flow graph for a program more sparse, improving performance.

Much recent work on Java points-to analysis employs some context-sensitive handling of method calls [17, 18, 29]. Cloning-based context sensitivity causes a blowup in input size; an m-limited call-string or m-object-sensitive analysis may require $O(N^m)$ clones. Our bound implies that Andersen's analysis can run in time quadratic in the size of the program after cloning, assuming the program with clones is still k-sparse. In some cases selective cloning can yield a significantly sparser flow graph, thereby improving both precision and running time (further discussion in §6). BDDs have been employed to make the space explosion from cloning more manageable, as we shall discuss in §5.4.

5.3 Exploiting Types

Type filters [3, 16], which ensure that points-to sets for variables are consistent with their declared types, are critical to good performance for Java points-to analysis. Type filters do not affect our worst-case bound for sparse programs: a quadratic pre-processing step can create an appropriate mask for each type to use during propagation. The filters improve performance by dramatically reducing the size of points-to sets and hence the amount of propagation work [16].

We remark that only applying type filters at propagation for downcasting operations, whether explicit (a JVM `checkcast` bytecode) or implicit (passing the receiver parameter at a virtual call site), yields the same precision benefit as applying them for all propagation operations (as was formulated in some previous work [3, 29]). Reducing the use of type filters without affecting precision can be a significant performance win, since they make propagation more expensive.

5.4 Space

Space usage often presents a bigger bottleneck for points-to analysis than running time, especially for context-sensitive analyses. Here, we discuss some space optimizations performed by points-to analyses and their effects on running time.

Employing difference propagation exhaustively as in Figure 1 may double space requirements and hence represent an unattractive space-time tradeoff. Our complexity proof relies on exhaustive use of difference propagation since it assumes that propagating a single abstract location requires one unit of work. A set implementation that enables propagation of abstract locations in parallel (see §5.1) lessens the need for exhaustive difference propagation in practice. In our experience, the key benefit of difference propagation lies in operations performed for each abstract location in a points-to set, *e.g.*, edge adding (see lines 12 and 17 in Figure 1). To save space, WALA [28] only uses difference propagation for edge adding and for handling virtual call receivers (since with on-the-fly call graph construction, each receiver abstract location may yield a new call target). Also note that the best data structure for the $pt_\Delta(x)$ sets may differ from the $pt(x)$ sets to support smaller sets and iteration efficiently; see [16, 19] for further discussion.

Many points-to analyses use set data structures that exploit redundancy between points-to sets, like shared bit sets [13] or BDDs [3, 29, 30], to dramatically

reduce space requirements. Their effects on running time are difficult to analyze, since the propagation cost model is very different: running times for BDD operations are highly dependent on variable orderings, and shared bit set operations depend on the current bit set cache state. Further understanding of the use of these data structures for analyzing k-sparse programs is a topic for future work.

5.5 Other Languages

Our main result of quadratic time complexity for points-to analysis of k-sparse programs can be adapted to other languages fairly easily. We believe the algorithm of Pearce et al. for C points-to analysis with difference propagation [20, Figure 7], very similar to the algorithm of Figure 1, would run in quadratic time for k-sparse C programs. The Figure 1 algorithm could also be adapted to perform control-flow analysis for functional languages [23] (formulated as dynamic transitive closure in [10]). For this case the notion of k-sparseness (see Definition 1) would be slightly transformed: rather than counting the number of dereferences of a variable $D(x)$, one would count the number of function applications of a variable / expression.

It is an open question as to whether typical programs in other languages are k-sparse. Pearce et al. [20] present some evidence that for C, the number of flow graph edges increases much more quickly with program size than for Java. They present a benchmark `gawk` with less than 20,000 LOC where the number of flow graph edges added during analysis is over 40X the number of variables; in contrast, our measurements in the next section never saw a factor more than 4.5X. This increased edge density in C may be due to the use of the `*` operator rather than named fields and the weaker type system. It may also explain the greater importance of projection merging [27] and online cycle elimination [6, 9, 13] for C.[7] In our experience, when respecting declared types in Java, relatively few cycles are discovered (also observed in [16]).

6 Measurements

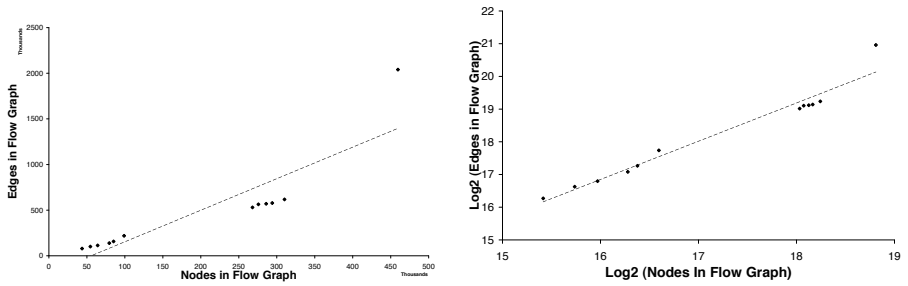
We used the Watson Libraries for Analysis (WALA) [28] for our measurements. The WALA implementation of Andersen's analysis differs from the algorithm of Figure 1 in a few ways. WALA employs on-the-fly call graph construction (§5.2) and type filters (§5.3), both of which reduce the size of the constraint graph without increasing worst-case complexity. WALA also models some Java reflective methods (*e.g.*, `Class.newInstance()`) and native methods with synthetic code generated during analysis.[8] Program sizes for the presented programs may differ from other published numbers due to variations in library versions and handling of reflection; we analyzed the IBM Java 1.6.0 libraries.

[7] Note that conceptually, cycle elimination is orthogonal to difference propagation; the former eliminates redundancy across variables with provably equivalent points-to sets, while the latter prevents redundant propagation of abstract locations.

[8] Reflection and native code may still cause the analysis to be unsound.

Benchmark	Methods	Bytecodes (KB)	Flow Graph Nodes (K)	Flow Graph Edges (K)	Edges / Nodes	Runtime (s)
antlr	3381	238	54	101	1.87	20
bloat	6438	456	99	218	2.20	43
chart	19089	1359	310	617	1.99	414
eclipse	17021	1169	275	563	2.05	359
fop	25542	2225	459	2039	4.44	2920
hsqldb	4600	330	79	138	1.75	25
jython	5291	386	85	157	1.85	35
luindex	4114	296	64	113	1.77	15
lusearch	16826	1160	268	530	1.98	312
pmd	18125	1248	286	569	1.99	361
xalan	2691	176	43	79	1.84	9
apache-ant	18404	1449	294	577	1.96	378

Table 2. Characteristics of programs analyzed.



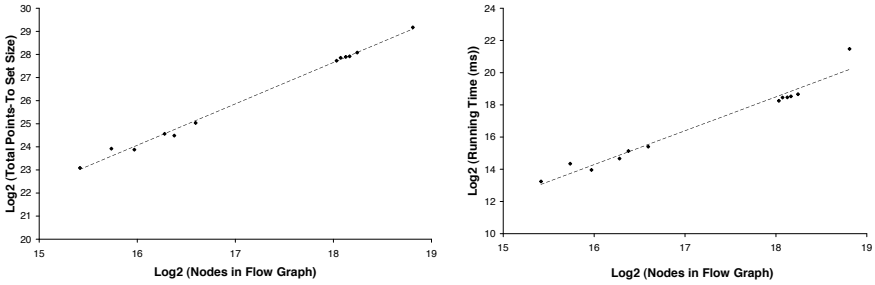
(a) (b)

Fig. 2. Sparsity measure for flow graphs. Each data point represents one benchmark analyzed. We show a standard scale in (a) and a log scale in (b). The lines show the best fit via linear regression, with a slope of 3.46 in (a) and 1.17 in (b).

Table 2 lists the programs used in this study. They include all of the DaCapo 2006-10-MR2 benchmarks [4] and Apache Ant 1.7.1 [2], another large Java program. The table reports program sizes as determined by the methods discovered during on-the-fly call graph construction.

Density of Flow Graphs The second-to-last column of Table 2 reports the density of the final flow graph constructed during pointer analysis. The Table shows that for all programs, the number of edges (E) per node (N) is less than 4.5. The `fop` program seems to be an outlier; all other programs have less than 2.2 edges per node in the flow graph.

Figure 2(a) displays the edges in flow graphs as a function of the number of nodes. The figure shows the best linear fit, which would have E grow as $3.46N$. Figure 2(b) shows the same data on a log-log scale. The best linear fit on a log-log scale has slope 1.17, indicating that the best polynomial fit to the data has E growing as $N^{1.17}$. If we exclude `fop` as an outlier, the best polynomial fit indicates E grows as $N^{1.05}$. We conclude the flow graphs are mostly sparse. As



(a) (b)

Fig. 3. Total size of computed points-to sets in (a) and pointer analysis running time in (b) on log scales, both as functions of nodes in flow graph. The lines show the best fit via linear regression, with a slope of 1.79 in (a) and 2.10 in (b).

discussed in §4.2, we expect $E = O(N)$, with exceptions arising from arrays and dispatch, and the data support this conclusion.

We examined `fop` in detail. Most of the edges in the `fop` flow graph result from the failure of the pointer analysis to adequately disambiguate the contents of object arrays passed to library routines, due to context insensitivity. In a practical pointer analysis client, we would recommend a context-sensitivity policy designed to clone common library routines that manipulate arrays, such as the `java.util.Arrays` utilities. As suggested in §5.2, though such cloning increases the size of the program, it can make the flow graph significantly sparser (since many infeasible flows are ruled out), improving both performance and precision. Further study of increased sparsity via context sensitivity would be an interesting topic for future work.

Size of pointer analysis result Figure 3(a) shows the total size of the computed points-to sets, as a function of the node count in the flow graph on a log scale. The figure shows the best linear fit, which has a slope of 1.79, indicating that the points-to solution size grows roughly as $N^{1.79}$. As we have defined the pointer analysis problem, this factor represents a lower bound on complexity of the Andersen's analysis in practice, since any algorithm will take at least $O(N^{1.79})$ to output the solution. In practice, many clients do not demand the complete analysis result, instead issuing a targeted set of alias queries. For these clients, demand-driven pointer analysis [12, 24, 25] may offer a better fit.

Observed analysis performance Figure 3(b) shows the running time of the pointer analysis as a function of the node count in the flow graph on a log scale.[9] The figure shows the best linear fit, which has a slope of 2.10, indicating that running time grows roughly as $N^{2.10}$. If we exclude `fop` from the regression, running time grows as $N^{1.92}$. The running of time of a real implementation depends on

[9] We ran the analysis on a Linux machine with an Intel Xeon 3.8GHz CPU and 5GB RAM, using the Sun 1.5.0_06 virtual machine with a 1.8GB heap.

many factors, including those discussed in §4.2 and §5. The results here show that on this benchmark suite, our implementation scales roughly quadratically with program size. It remains for future studies to determine whether quadratic scaling holds for other implementations and benchmarks.

7 Related Work

Our work is most closely related to the studies of points-to analysis complexity of Pearce et al. [19, 20]. Our algorithm is very similar to that of [20, Figure 7], but adapted to Java. Pearce was the first to show difference propagation can affect worst-case complexity, in his case improving a worklist-based algorithm from quartic to cubic time [19, §4.1.3]. We further improve the bound to quadratic time for k-sparse programs, which requires additionally reasoning about the cost of edge-adding work. Difference propagation was first presented by Fecht and Seidl in [7].

Lhoták and Hendren present a Java points-to analysis algorithm with difference propagation (there termed an "incremental worklist" algorithm) and showed its performance benefits [16]. Their algorithm does not fully employ difference propagation for abstract location fields, as it periodically does full propagation for field access statements [15, §4.4.3]. Hence, it is not clear if the quadratic bound for k-sparse programs holds for their algorithm.

Heintze and McAllester present a sub-cubic control-flow analysis algorithm for bounded-typed programs [10]. They formulate the analysis problem as dynamic transitive closure and distinguish edge addition work from closure work (in fact, they occur in separate phases in their algorithm). As their algorithm does not allow for recursive types, it is not immediately applicable to Java.

Various other work studies points-to analysis complexity. Heintze and McAllester relate the difficulty of flow analysis to the 2NPDA complexity class [11]. Melski and Reps formulate Andersen's analysis for C as a CFL-reachability problem, immediately yielding a cubic algorithm [21]. Fändrich et al. use a probability-based analytic model over random graphs to study online cycle elimination for set constraints in inductive form [6]. Chaudhuri presents a slightly sub-cubic algorithm for CFL-reachability, thereby breaking the "cubic bottleneck" for Andersen's analysis [5].

8 Conclusions

We have proven a quadratic worst-case time bound for computing Andersen's analysis for k-sparse input programs, and we have given empirical evidence that Java programs are usually k-sparse. These results help account for the gap between the nearly cubic worst-case complexity of Andersen's analysis and its scalability in practice. The notion of k-sparsity may also be useful in understanding the real-world performance of other program analyses.

Acknowledgments We thank the anonymous reviewers for their helpful comments and Ras Bodík for input on earlier versions of this work.

Bibliography

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

[2] Apache Ant. http://ant.apache.org.

[3] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2006.

[5] S. Chaudhuri. Subcubic algorithms for recursive state machines. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 159–169, New York, NY, USA, 2008. ACM.

[6] M. Fändrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada*, June 1998.

[7] C. Fecht and H. Seidl. Propagating differences: an efficient new fixpoint algorithm for distributive constraint systems. *Nordic J. of Computing*, 5(4):304–329, 1998.

[8] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 395–404, New York, NY, USA, 2007. ACM.

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

[10] N. Heintze and D. McAllester. Linear-time subtransitive control flow analysis. *SIGPLAN Not.*, 32(5):261–272, 1997.

[11] N. Heintze and D. McAllester. On the cubic bottleneck in subtyping and flow analysis. In *LICS '97: Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*, page 342, Washington, DC, USA, 1997. IEEE Computer Society.

[12] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah*, June 2001.

[13] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah*, June 2001.

[14] J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS '05: Proceedings of the 12th International Static Analysis Symposium*. London, United Kingdom, September 2005.

[15] O. Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[16] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction (CC), Warsaw, Poland*, April 2003.

[17] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: Is it worth it? In *International Conference on Compiler Construction (CC)*, 2006.

[18] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[19] D. J. Pearce. *Some directed graph algorithms and their application to pointer analysis*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, 2005.

[20] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Online cycle detection and difference propagation for pointer analysis. In *Proceedings of the third international IEEE Workshop on Source Code Analysis and Manipulation*, 2003.

[21] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, November/December 1998.

[22] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Tampa Bay, Florida*, October 2001.

[23] O. Shivers. Control flow analysis in scheme. In *Conference on Programming Language Design and Implementation (PLDI)*, 1988.

[24] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[25] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.

[26] B. Steensgaard. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1996.

[27] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *ACM Symposium on Principles of Programming Languages (POPL), Boston, Massachusetts*, pages 81–95, January 2000.

[28] T.J. Watson Libraries for Analysis (WALA). http://wala.sf.net.

[29] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[30] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *Conference on Programming Language Design and Implementation (PLDI)*, 2004.