Giga-Scale Exhaustive Points-To Analysis for Java in Under a Minute

Jens Dietrich
Massey University, New Zealand
j.b.dietrich@massey.ac.nz

Nicholas Hollingum
The University of Sydney, Australia
nhol8058@uni.sydney.edu.au

Bernhard Scholz
Oracle Labs, Australia
Bernhard.Scholz@oracle.com

Abstract

Computing a precise points-to analysis for very large Java programs remains challenging despite the large body of research on points-to analysis. Any approach must solve an underlying dynamic graph reachability problem, for which the best algorithms have near-cubic worst-case runtime complexity, and, hence, previous work does not scale to programs with millions of lines of code.

In this work, we present a novel approach for solving the field-sensitive points-to problem for Java with the means of (1) a transitive-closure data-structure, and (2) a pre-computed set of potentially matching load/store pairs to accelerate the fix-point calculation. Experimentation on Java benchmarks validates the superior performance of our approach over the standard context-free language reachability implementations. Our approach computes a points-to index for the OpenJDK with over 1.5 billion tuples in under a minute.

Categories and Subject Descriptors F.3.2 [*Theory of Computation*]: Semantics of Programming Languages—Program analysis

Keywords Context-free Language, Transitive Closure, Java, Points-to Analysis

1. Introduction

Points-to analysis is a well-researched computational problem, with important applications to compiler optimisations and productivity tools. Much progress has been made for the C programming language [2, 34, 49], which establishes a hierarchy of analyses that trade-off speed for precision. In the Java[1] context, the main research focus is on *field-sensitive* analysis, which tracks the dataflow between heap objects that are stored in and loaded from fields. Points-to analyses for Java vary in precision, and can be context sensitive [25, 30, 44, 45], or insensitive [32, 33], but usually necessitate field-sensitivity to improve precision.

The points-to problem for Java specifies the task of determining which heap objects a program variable may reference during runtime. Computing the points-to analysis statically is intricate in the presence of recursive methods, dynamic dispatch, and other complex control-flow constructs. A light-weight points-to analysis abstracts the statements of a program in a context-insensitive and flow-insensitive fashion to obtain an over-approximation of the program's runtime behaviour.

There are many applications for points-to analysis including its use in compilers, integrated development environments, bug checking and security tools. Most of these applications require a whole program analysis for points-to, i.e., for any program variable in the input program, the points-to set is to be computed. For example, this whole program analysis requirement is necessary for security analysis (as reported in [18]) for which the whole state of the input program has to be explored. This necessitates exhaustive analysis [30, 32, 48, 49], as opposed to the often faster on-demand analysis [31, 33, 44, 45, 45]. Whole program analysis is particularly challenging for cloning-techniques used for context-sensitivity and for large code bases such as OpenJDK.

The points-to problem is commonly encoded as a *context-free language reachability* (CFLR) problem [23]. In a CFLR encoding, program variables and heap objects form the vertices of the directed *points-to* graph, whose edges are labelled with the relational semantics for object creation, variable assignment, and field load/store accesses. An instance of the points-to problem is solved by searching for paths whose edge labels form sentences that are in the language of a context-free grammar, that expresses the semantics of the points-to analysis. Unfortunately, CFLR algorithms are

OOPSLA'15, October 25–30, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3689-5/15/10...$15.00
http://dx.doi.org/10.1145/2814270.2814307

[1] Java and JDK are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

well-known to have a practical complexity of $\mathcal{O}(n^3)$ [19], and no algorithm faster than $\mathcal{O}(\frac{n^3}{\log n})$ [7] has been discovered, for general graphs with n vertices and general context-free grammars. Cubic time-complexity is the performance bottleneck of many static program analyses in practice, due to their underlying connection to CFLR. Hence, there is need to optimise the performance of concrete instances of CFLR, including points-to.

To overcome the performance bottleneck of a field-sensitive Java points-to analysis, we employ a *transitive closure*[2] data-structure. Transitive closures have more in common with regular languages than their context-free counterparts. Indeed, Yannakakis' foundational work on CFLR [46] expresses the regular language reachability problem via transitive closure on an expanded graph. Importantly, the CFLR formulation exposes transitive properties of points-to, hence we are able to adapt very fast transitive closure algorithms (e.g. [21]) to solve points-to. Specifically, our contributions are:

- We introduce a novel algorithm for field-sensitive points-to analysis using a transitive closure data-structure. We use a set of potentially matching load/store pairs pre-computed by an *over-approximated bridge oracle* to accelerate the fix-point computation. We discuss soundness and completeness, and detail its implementation.
- We provide a parameterised worst-case runtime complexity analysis for our points-to algorithm, that might exhibit a complexity of $\mathcal{O}(n^4)$ assuming the worst-case for the parameters. We provide an empirical analysis showing that, for real-world problems including DaCapo and OpenJDK, some parameters are nearly constant.
- We conduct experiments with our points-to algorithm on Java benchmarks, including the very large OpenJDK dataset, whose points-to relation has over 1.5 billion tuples for which a points-to index is computed in under a minute. We compare the performance of our algorithm with the standard CFLR algorithm by Melski and Reps [19], the fast difference propagation-style algorithm by Sridharan and Fink [32] and a Datalog implementation using the LogicBlox engine. Our experiments show that precision is unrecoverable when top-down refinement techniques are used, such as in [33], therefore bottom-up refinement is necessary.

This paper is structured as follows: Section 2 presents a running example for this paper. Section 3 introduces the formulation of points-to as transitive closure, including proofs on the soundness and completeness of the transformation, and a presentation of the algorithm skeleton (including a time-complexity analysis). The specific details of how to implement a fast solver are explained in Section 4. We verify the correctness and performance claims for our approach experimentally in Section 5, and revisit the practical complexity discussion in response to experimental statistics in Section 6. Finally, we give a brief survey of the literature in Section 7, and our concluding remarks and a brief discussion of future work in Section 8.

[2] More accurately: reflexive transitive closure, though in the literature the formulation is typically used without reflexivity.

```
v1 = new Obj(); // h1  | v12 = v8;    | v3 = v2.g;
v2 = new Obj(); // h2  | v1.h = v1;   | v8 = v7.g;
v4 = new Obj(); // h3  | v2.g = v1;   | v10 = v9.g;
v6 = new Obj(); // h4  | v4.f = v2;   | v10 = v10.h;
v5 = v4;               | v7 = v5.f;   |
v5 = v6;               | v9 = v6.f;   |
```

Figure 1: Running example: Java fragment. Class `Obj` has fields: `public Obj f, g, h;`

2. Points-To via Graph Reachability

Consider the program listed in Figure 1. The dataflow of the program is constructed from object creations, variable assignments, and load/store operations on fields. The interplay between object creations, variables, and operations may be expressed as a collection of binary relations that capture sufficient semantics for an input program to compute flow-insensitive points-to analysis. Given an arbitrary Java program with a set of heap allocation sites *Objects*, program variables *Vars*, and fields *Fields*, the semantics is given by the following binary relations:

- Relation *assign* $\subseteq$ *Vars* $\times$ *Vars* records the assignment of program variables to other program variables. For simplicity, casting, variable returning, and the passing of actual parameters to formal parameters, are all considered to be assignments.
- Relation *alloc* $\subseteq$ *Vars* $\times$ *Objects* records the allocation of heap objects and the variables they are assigned to, i.e. `x = new Obj();`. The Java semantics allows for a variable to allocate many heap objects, but each object is allocated by at most one variable.
- Relations $load_f \subseteq Vars \times Vars$, for all $f \in fields$, record reading from a field of another variable. A distinct relation exists for every field that the Java program reads to or writes from. For the statement `x = y.f;`, we refer to variable y as the *base variable* of the load operation.
- Relations $store_f \subseteq Vars \times Vars$, for all $f \in fields$, record writing to a field. The statement `x.f = y;` has variable x as the base variable of the store operation.

Since our analysis is flow-insensitive and context-insensitive, we omit the notion of order between statements, and method invocations are modelled as a series of assignments in *assign*: (1) the transfer of data from actual parameters to the formal parameters of the called method, (2) object bases between caller and callee, and (2) return values.

A graphical representation of the binary relations is an edge-labelled graph. The edge-labelled graph of the code

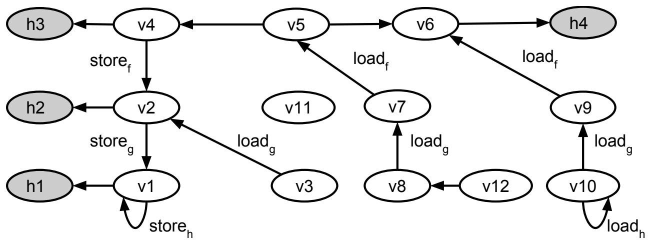

Figure 2: Input Graph of Example in Figure 1

snippet from Figure 1 is depicted in Figure 2. Nodes in the graph $V = V_{Vars} \cup V_{Objects}$ are either program variables (`v1` - `v12`) denoted by V_{Vars}, or heap-object creation sites (`h1` - `h4`) denoted by $V_{Objects}$. Edges are labelled according to the relation name from which they were drawn. In Figure 2, we have omitted the labels for assignment and allocation edges, as their meanings are clear in context, i.e., *assign* edges flow from a variable to a variable, and *alloc* edges from a variable to an object-creation, respectively.

The main difficulty in finding the points-to relation is evidenced in Figure 2, namely tracking dataflow information. Deducing, for example, that variable $v12$ may point to $h1$, requires tracking two principle flows that may happen. First, a flow may be realised between a variable and an object-creation via edges in the *assign* relation, a sequence of assignment statements, transferring the object from one variable to the next. Second, a flow may be realised with one or more load/store pairs. A load/store pair may introduce an artificial assignment-like edge which we refer to as a *bridge*. A bridge connects a stored variable, and the result of a load statement, if the receiver of the load/store pair reach a common object-creation, i.e., the base variables may point to the same object. The bridges of load/store pairs capture the flow of information through the heap. The tracking of bridges become a recursive problem and can be expressed concisely with an instance of the *context-free language reachability* (CFLR) problem [23]. The instance consists of an input graph whose edges are labelled by terminal symbols *alloc*, *assign*, $load_f$, and $store_f$ and a grammar given below,

$$\begin{aligned} \textit{alias} &\to \textit{pointsTo}\ \overline{\textit{pointsTo}} \\ \textit{bridge} &\to \textit{load}_f\ \textit{alias}\ \textit{store}_f \qquad \forall f \in \textit{Fields} \qquad (1) \\ \textit{pointsTo} &\to (\textit{assign}|\textit{bridge})^*\ \textit{alloc} \end{aligned}$$

with non-terminals *pointsTo*, *bridge*, and *alias*. As a result of "parsing" the input-graph, relations for the non-terminal symbols are produced. The non-terminal *pointsTo* represents paths between variables and object-creations. An *alias* path exists between two variables when we can walk forwards to a heap object the first points to, and $\overline{\text{backwards}}$ from that object to the second variable (i.e. they both point to the same object)[3]. A *bridge* path acts like an assignment edge for information flowing across a load edge, between two aliasing nodes, then down a store of the same field. The grammar is based on Sridharan et al.'s work [33], with a superficial difference that our *pointsTo* reverses the *flowsTo* $\subseteq$ *Objects* $\times$ *Vars* in their work. The change of directionality in the grammar facilitates the description of our new algorithm.

In this work we focus on a core high-performance building block of field-sensitive points-to analysis for Java. We have chosen an Andersen-style analysis [2], which is precise enough for many applications, but makes minimal treatment of Java semantics. This style of analysis has been implemented by the Java static optimiser SPARK [16][4]. Extending our algorithm to further improve precision, such as with call-graph construction on the fly, context-sensitivity, or reasoning about types and reflection, is out of scope, and left as future work.

[3] To simplify, we do not list the rules for $\overline{\text{backwards paths}}$, which can be deduced mechanically (cf. [23]).

[4] Specifically the type-unaware version: `ignore-types:true`

3. Points-To via Transitive Closure

In this section we introduce the algorithmic aspects of our new points-to analysis. First, we present means to solve the points-to problem using a transitive closure operation on a binary relation under the assumption that there exists an oracle, which decides the load/store pairs that form *bridges* – inferred assign edges representing flow through matching loads/stores. Second, given that pre-computing the oracle is expensive, we detail how to solve points-to without an oracle. We assume that a *bridge-finder* over-approximates the oracle, i.e., the bridge-finder determines a set of potentially matching load/store pairs, and the algorithm refines the bridges employing a transitive closure data-structure.

3.1 Points-To with Transitive Closure Using an Oracle

The most obvious difficulty in computing the points-to relation is in correctly identifying how objects are passed on using object fields of heap objects. We want to find the bridges, between load and store, along which the references flow. A bridge edge acts as a pseudo-assign edge between the source and the sink of matching load and store edges, so that reachability can be computed using this bridge edge instead of the store and load. This is similar to the concept of match edges used by Sridharan et. al. [33]. Figure 3 depicts our running example with the bridge edges, $\{(v7, v2), (v3, v1), (v8, v1)\}$, as dashed arrows.

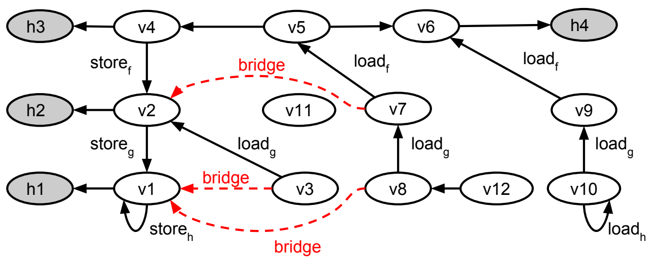

Figure 3: Example in Figure 1 with bridge edges

Let's assume that we know the relation *bridge* ahead of time from an oracle. The grammar (cf. Eq. (1)) would reduce to

$$\textit{pointsTo} \rightarrow (\textit{assign}|\textit{bridge})^* \; \textit{alloc} \tag{2}$$

Instead of describing the problem as a CFLR problem, we can express the points-to computation using relations. The relationship between CFLR problems and relational algebra has been studied in the seminal work of Yannakakis [46]. For computing points-to as a relation, we introduce a new relation T that is defined as

$$T = (\textit{assign} \cup \textit{bridge})^*$$

representing the grammar expression $(\textit{assign}|\textit{bridge})^*$. I.e. the relation T is the reflexive transitive closure of the binary relation $\textit{assign} \subseteq \textit{Vars} \times \textit{Vars}$ unified with relation $\textit{bridge} \subseteq \textit{Vars} \times \textit{Vars}$.

Semantically, the relation T expresses the value transfer between two variables. If there is a pair $(u, v) \in T$, then variable u receives the objects that v may point to either because u is equal to v, or u and v are connected by a path consisting of $E_{\textit{bridge}} \cup E_{\textit{assign}}$. Hence, the path indicates a data flow from variable v to u. In the running example, $(v12, v1) \in T$, since $(v12, v8) \in \textit{assign}$, and $(v8, v1) \in \textit{bridge}$.

To obtain the relation $\textit{pointsTo} \subseteq \textit{Vars} \times \textit{Objects}$, relations T and *alloc* are composed, i.e.,

$$\textit{pointsTo} = T \circ \textit{alloc}$$

where operation $\circ$ is the composition of two binary relations (i.e. $A \circ B = \{(u, w) \mid (u, v) \in A, (v, w) \in B\}$) and corresponds to the concatenation of symbols on the right-hand side of production rules in the grammar. Hence, the computation of *pointsTo* requires the computation of the reflexive transitive closure and the binary relation composition. The fastest known transitive closure algorithms employ a fast matrix multiplication algorithm and divide-and-conquer techniques. Their worst-case complexity is in the order of a Boolean matrix multiplication [21]. Hence, *pointsTo* is computable in $\mathcal{O}(|\textit{Vars}|^{2.37})$ with the fastest known matrix-multiplication [10, 11] assuming we have an oracle that provides the relation *bridge*. However, this approach is not viable since *bridge* is not known ahead-of-time and computing it would be at least as hard as computing the entire CFLR problem.

To overcome this issue, we find an over-approximation of the relation *bridge*, and devise a new algorithm based on transitive closure and a certificate for bridge edges. Note that an over-approximation of relation *bridge* can always be found. For example, the all-pair solution $\textit{load} \times \textit{store}$ would be a valid, alas a very imprecise over-approximation of the oracle.

3.2 Points-To with a Transitive Closure Data-Structure

The fundamental idea of our new points-to algorithm is to use a reflexive transitive closure data-structure T, to refine a sound over-approximation of the bridge oracle denoted by $\widetilde{\textit{bridge}}$, produced by a *bridge-finder*. The data-structure has an underlying carrier relation S for which it answers queries about its reflexive transitive closure S^*. Initially, the carrier relation is empty. The carrier relation S can be extended by operation $\textit{add}(X)$, which adds set X to the carrier set (i.e., $S' \leftarrow S \cup X$). The operation $\textit{query}(u)$ returns the reflexive and transitive elements of u, i.e., $\textit{query}(u) = \{v | (u, v) \in S^*\}$. Efficient transitive closure implementations keep track of the previously computed results in some intermediate form to facilitate fast query time [21]. A more detailed discussion about our transitive closure implementation is found in Section 4.3.

To use the reflexive transitive closure algorithm as a vehicle to solve points-to, we rewrite the input grammar of Eq. (1) to capture the semantics of the transitive closure relation T:

$$\begin{aligned} \textit{pointsTo} &\rightarrow T \; \textit{alloc} \\ T &\rightarrow (\textit{assign}|\textit{bridge})^* \\ \textit{bridge} &\rightarrow \textit{load}_f \; T \; \textit{alloc} \; \overline{\textit{alloc}} \; \overline{T} \; \textit{store}_f \quad \forall f \in \textit{Fields} \end{aligned} \tag{3}$$

Algorithm 1 computes points-to using the modified grammar in Eq. (3), specifically the relations for the non-terminals T and *bridge*. In the first step, candidate list W is initialised by the bridge-finder's output. The task of the bridge-finder is to exclude as many potential matching load/store pairs as possible by still providing an over-approximation for relation *bridge*. An efficient and effective bridge-finder is described in Section 4 based on Dyck-reachability [48].

After initialising the candidate list W, the algorithm iterates over the candidate list until no new bridge edges can be added to the carrier relation of T. Inside the loop, we certify each potential bridge (u, v) with the IS-BRIDGE procedure. If $(u, v) \in W$ is a bridge, we add the pair (u, v) to the set N. After the traversal over W, set N captures all newly certified bridges and is added to the carrier relation of T, and removed from the candidate list. After termination, the candidate list may still contain pairs, caused by the over-approximation of the bridge-finder. Composing the computed transitive closure T, after the fix-point iteration, with relation *alloc*, produces the points-to relation.

Algorithm 1 Points-To Computation: requires $\widetilde{bridge}$ as an over-approximation of the bridge oracle; T is the incremental transitive closure with *add* and *query* operations.

```
function POINTS-TO($\widetilde{bridge}$)
    $W \leftarrow \widetilde{bridge}$
    $T.add(assign)$
    repeat
        $N \leftarrow \emptyset$
        for all $(u, v) \in W$ do
            if IS-BRIDGE$(u, v)$ then
                $N \leftarrow N \cup \{(u, v)\}$
            end if
        end for
        $T.add(N)$
        $W \leftarrow W \setminus N$
    until $N = \emptyset$
    $pointsTo \leftarrow T \circ alloc$
end function
function IS-BRIDGE$(u, v)$
    for all $(b_u, b_v) \in lsb(u,v)$ do
        if $(T.query(b_u) \cap T.query(b_v)) \diamond alloc \neq \emptyset$ then
            return true
        end if
    end for
    return false
end function
```

The certifier IS-BRIDGE(u, v) holds if the pair (u, v) constitutes a bridge edge for the current version of relation T. In order to construct a bridge edge, we need at least a load/store pair for the same field, whose destination is u and source v, respectively, and their base variables b_u and b_v, may point to a common object. Here $\diamond$ is shorthand for $S \diamond R \Leftrightarrow \{o \mid \exists(e, o) \in R : e \in S\}$. The function $lsb \in Vars \times Vars \to 2^{Vars\times Vars}$ retrieves all load/stores base variables that may emanate from a load/store pair with variables (u, v), i.e.,

$$\bigcup_{f \in Fields} \{b_u \mid (u, b_u) \in load_f\} \times \{b_v \mid (v, b_v) \in store_f\}$$

To check whether both base variables b_u and b_v share a common object, we query the transitive closure for both base variables. For both queries we obtain a set of variables that pass on their points-to information to the base variables, respectively. If both sets have at least one variable in common which was used in an object-creation site, we have a transfer of data from the store operation to the load operation, and (u, v) becomes a bridge edge.

3.2.1 Soundness and Completeness

The soundness and completeness of our algorithm is shown by expressing the computations as a least fix-point of a system of simultaneous equations over relation T and *bridge*. The systems of simultaneous equations is directly deduced from the CFLR grammar in Eq. (1) based on Yannakakis' work [46], and is given as follows,

$$\begin{aligned} T &= (assign \cup bridge)^* \ alloc \\ bridge &= \bigcup_{f \in Fields} load_f \circ T \circ alloc \circ \overline{alloc} \circ \overline{T} \circ store_f \end{aligned} \quad (4)$$

where $\overline{alloc}$ and $\overline{T}$ denote the reverse relations[5] of relation *alloc* and T, respectively. The simultaneous system of equations has two unknowns, which are relations T and *bridge*. A solution is a pair of relations $(T, bridge)$ that satisfies the equations of Eq. (4). Solutions are ordered by a partial order denoted by symbol $\preceq$. For two solutions $(T, bridge) \preceq (T', bridge')$ holds, iff $T \subseteq T'$ and $bridge \subseteq bridge'$. The partial order induces a finite subset lattice, as well.

For our simultaneous system of equations, we seek the smallest solution. The smallest solution gives a sound solution for points-to but has the smallest numbers of points-to relations, which makes it the most precise solution. Tarski-Knaster's fix-point theorem states the existence of the smallest solution, also known as the least fix-point solution. In the following, we outline that our algorithm is computing the least fix-point solution for the simultaneous equation system of Eq. (4).

Proposition 1. *For a given T, certificate* IS-BRIDGE(u, v) *holds iff*

$$(u, v) \in \bigcup_{f \in Fields} load_f \circ T \circ alloc \circ \overline{alloc} \circ \overline{T} \circ store_f \quad (5)$$

We observe that the condition, $T.query(b_u) \cap T.\underline{query}(b_v) \diamond alloc \neq \emptyset$, is equivalent to $(b_u, b_v) \in T \circ alloc \circ \overline{alloc} \circ \overline{T}$ assuming that the relation *alloc* has for each object-creation site a single variable only, which in general holds for Java, because `new()` cannot assign a newly created instance to more than one variable. By the definition of $lsb(u, v)$, we connect the term with all load/store pairs over all fields, and conclude that (u, v) is a member.

Lemma 1. *In the i-th iteration, the transitive closure represents the set*

$$\begin{aligned} T_0 &= assign^* \\ T_i &= (assign \cup bridge_i)^* \qquad \forall i > 0 \end{aligned}$$

where

$$bridge_i = \bigcup_{f \in Fields} load_f \circ T_{i-1} \circ alloc \circ \overline{alloc} \circ \overline{T_{i-1}} \circ store_f$$

From Proposition 1 we have the certified bridges, and procedure IS-BRIDGE certifies according to the "given T" from the $(i-1)$-th iteration, initialised with the *assign* edges.

[5] Iff for a pair a, b, relation aRb holds, then the reverse relation $b\overline{R}a$ holds.

In each iteration new bridges are added that could be certified and that have not been added yet. We can show by induction that the above lemma holds. Since the number of pairs in relations T and *bridge* is finite, and $T_{i+1} \supseteq T_i$, $\mathit{bridge}_{i+1} \supseteq \mathit{bridge}_i$ holds, after a finite number of iterations, the fix-point is obtained. We denote with T_* and bridge_* relation T_i and bridge_i, respectively, for the smallest i, such that $T_{i+1} = T_i$, and $\mathit{bridge}_i = \mathit{bridge}_{i+1}$.

Theorem 1. *(T_*, bridge_*) is the least-fix-point solution.*

The theorem can be shown by Kleene's fix-point theorem using the monotonicity of the fix-point equations.

3.2.2 Complexity

We present the algorithm's worst-case runtime complexity in parametric form, and as a function of vertex-set size only, which we refer as "absolute form". The parametric form provides some insights into the nature of the program's execution, and why this approach may be faster than the standard method in practice. The absolute form relates the running time to the size of the input, noted n, measured as the number of program variables and object-creation sites. Converting the parametric form to the absolute form produces a conservative over-estimate of the runtime complexity which may be unrealistic for concrete instances.

Let the cost of updating a transitive closure over n vertices when adding m new edges be noted $\mathbf{T_U}(n, m)$. Similarly, let $\mathbf{T_Q}(n)$ note the time a transitive closure for n variables takes to query all reachable nodes from a single source. Finally, we shall call k the *stratification depth* which is the number of iterations required in the outer-loop of Algorithm 1.

Lemma 2. *For input problems with $n = |V|$, $l = |\widetilde{bridge}|$, $|N|$-sized updates to the incremental transitive closure, and a stratification depth of k, the time complexity is asymptotically bounded to $\mathcal{O}(k(\mathbf{T_U}(n, |N|) + \mathbf{T_Q}(n)l))$.*

The lemma is immediate from the definition of Algorithms 1. The parametric complexity can be converted to an absolute complexity bounds for inputs with n vertices as follows:

Lemma 3. *For input problems with $n = |V|$, the time complexity is asymptotically bounded to $\mathcal{O}(n^4)$.*

The lemma can be shown by substituting conservative values in Lemma 2. We make the following assumptions: First, the stratification depth k can not exceed n, since each level of depth requires writing to and reading from at least one variable each, so the depth is at most $\frac{n}{2}$. Second, the bridge-finder may return up to n^2 pairs of load/store edges. Third, updating the incremental transitive closure can be no worse than performing the entire closure from-scratch, which takes $\mathcal{O}(n^{2.37})$. Fourth, querying the incremental transitive closure takes $\mathcal{O}(n)$ time (cf. Section 4.3). When the transitive closure is represented by a boolean adjacency matrix, we iterate over the n other variables to check if both base variables reach it. For any variable that both base variables reflexively and transitively reach, checking that this variable is an allocation site is a constant-time operation. Therefore the complexity is $\mathcal{O}(n(n^{2.37} + n^2 n)) = \mathcal{O}(n^4)$.

Based on Lemma 3, we deduce that the basic CFLR algorithm is superior to our algorithm assuming worst-case parameters. However, we justify the use of this technique by experimental validation (see Section 5), which shows that in practice the performance is significantly better, since some parameters do not exhibit their worst-case behaviour (see Section 6).

4. Implementation

This section describes the implementation of our fast points-to analysis. Engineering effort was necessary to create an efficient implementation using new data-structures for transitive closures. In particular, the algorithm from Section 3 requires a bridge-finder to over-approximate the relation *bridge*, and a transitive closure structure to certify the bridges. The algorithm is implemented in three stages. The first stage is the partitioning of input vertices into an initial set of weakly-connected points-to components (aka. *componentisation*, cf. Sec 4.1). Next, the bridge-finder simultaneously merges components and computes the oracle (aka. *folding*, cf. Section 4.2). Finally, we incrementally certify the correct edges (aka. *refinement*, cf. Section 4.3), as shown in Algorithm 1.

4.1 Componentisation

Our approach to computing points-to information relies on an incrementally updated transitive closure along assignment edges in the program graph. In programs containing no loads or stores, this can be achieved trivially with the grammar $\mathit{pointsTo} = \mathit{assign}^* \circ \mathit{alloc}$ (cf. Eq. 1). In the presence of loads and stores, the relation becomes an under-approximation of the solution. We can compute the under-approximation of the points-to relation, by first partitioning the graph into subsets of vertices that are weakly connected amongst $E_{\mathit{alloc}} \cup E_{\mathit{assign}}$. We call this computation the **componentisation** stage.

Components are computed by traversing the graph following reversed assign edges starting at heap object vertices. For each heap object vertex, a new component is created, and all visited vertices are associated with this component. If a vertex is visited that has already been assigned to a component then the respective components are merged. Finally, the sources of load edges with sinks already assigned to components are considered as "pseudo-heap objects", and new traversals are started there. We can think of load edges as providing pseudo-allocations to the heap objects that had to be allocated for the base variable of the load.

The result of this step is a *component cover* - some (but not necessarily all) vertices are associated with exactly one component. The vertices not associated with components represent vertices that can not reach any heap object vertex, and will therefore always have empty points-to sets. These vertices can be excluded from further analysis.

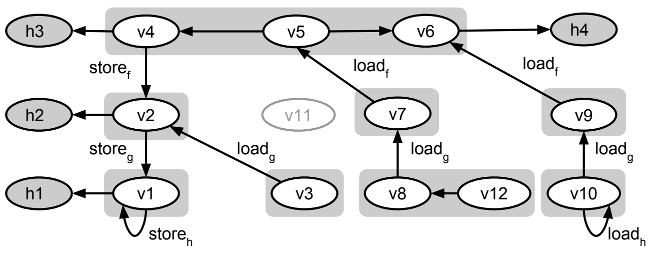

Figure 4: The running example, broken up into its initial components (grey boxes)

Figure 4 depicts the component cover generated for the running example. The component containing $v8$ and $v12$ is an example of a component computed using a virtual pseudo heap object vertex induced by the $load_g$ edge. The vertex $v11$ cannot be associated with a component and can be safely removed as discussed above.

4.2 Folding

According to the abstract implementation from Section 3 we require a *bridge-finder* to over-approximate the indirect assignment across load/store pairs. From an engineering perspective, several options are available to us, with different trade-offs. From least accurate to most, some options are: (1) all store-load pairs might bridge, (2) all store-load pairs of the same type might bridge, (3) all store-load pairs for the same field might bridge [31], (4) all store-load pairs whose base-variables alias might bridge, and (5) all store-load pairs for the same field whose base-variables alias might bridge. Note that the last of these options is equivalent to computing the exact solution (so a perfect oracle), but the fastest known algorithms for this are of near-cubic complexity [7].

For our algorithm we choose the fast bi-directed Dyck-reachability relationship by Zhang et al. [48], due to the speed of evaluation and the acceptably high precision for alias. This phase of the algorithm is therefore called *Dyck-folding*. Informally, this mechanism treats allocation and assignment edges similar to Steensgaard [34] equivalence-based approach, and collapses equivalent sets when the load/store paths between them form properly-balanced parenthesis structures over their fields. The over-approximation of the points-to relation when using this method derives from the lack of directionality information, which is lost when the input is converted to a bi-directed Dyck-reachability problem.

The component $\{v4, v5, v6\}$ in the running example (Figure 4) shows how bi-directionality produces false positives. Normally, $v4$ cannot reach $h4$ and $v6$ cannot reach $h3$. During the folding stage, all nodes in the same component are viewed as inter-reachable, hence, since $v4$,$v5$ and $v6$ are in the same component, $v4$ and $v6$ are falsely determined to point-to $h4$ and $h3$ respectively. The underlying problem is that the equivalence relationship used to build components is transitive, while alias is not.

We use the initial component cover generated in the componentisation stage as equivalence classes for the folding stage followed by running the fast bi-directed Dyck-reachability algorithm over the component graph. Bridges are created when matching load-store pairs, those with load sinks and store sources in the same component, are encountered. If the load sources and the store sinks are in different components, the respective components are merged (folding). Book-keeping for the later refinement stage requires modifying the fast-Dyck algorithm [48].

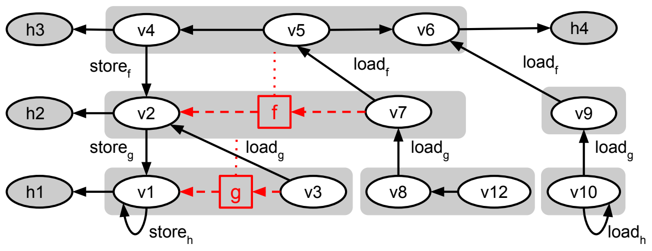

Figure 5: Applying a Dyck fold

Whereas Zhang et al. sought to compute which vertices belong to the same equivalence class, we also need to record additional information on **why** two components were collapsed. For this purpose, we create symbolic *bridge vertices* in the collapsed components to store this additional information. Bridge vertices indicate which field was collapsed in order to record which component the base variables were members of. Figure 5 illustrates this approach. The squares represent the newly introduced bridge vertices. Bridge vertices are labelled with the field of the respective load and store edges, and the dashed line is used to link the bridge vertices to the respective components they were derived from. The dotted lines do not represent edges in the graph.

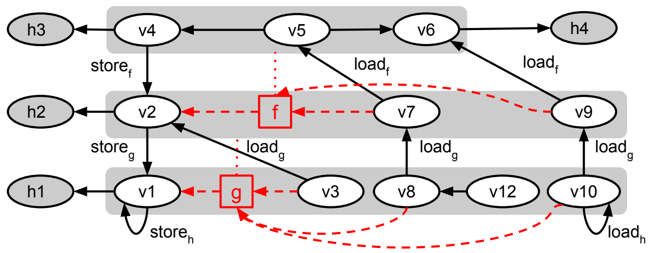

Figure 6: This graph shows the need to connect store/load edges to pre-existing bridge nodes when they are collapsed

Bridge vertices allow for multiple pairs of load/stores to be collapsed into them. Consider the example in Figure 6. Both components of $v7$ and $v9$ must be collapsed to $v2$, but the bridge vertex is created only once (for the first collapse), and re-used when the second fold is applied. During the folding stage, initial components are merged together according to their bi-directed equivalence. Once completed, the resulting components form an over-approximation of the alias relation where comp(u) denotes the component of vertex u:

Lemma 4. $\forall u, v \in V_{Vars} : alias(u, v) \Rightarrow comp(u) = comp(v)$

The lemma holds since the alias relation indicates that two variables transitively reach the same heap object along bridge and assignment edges, finishing with an allocation. This implies that they would end-up in the same component, since vertices sharing an alloc, assign, or bridge edge have been collapsed.

The final result of the folding stage on our running example can be seen in Figure 7. Bridge edges for the oracle can be easily expanded at this stage as the cross product of nodes into and out-of the bridge node. Componentisation information is not needed any more, as its only purpose was to compute the bridge oracle efficiently.

4.3 Refinement

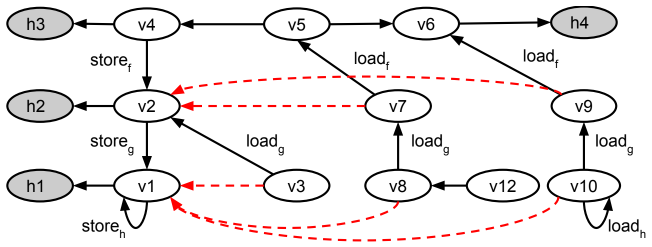

Figure 7: Running example after folding

The refinement for the over-approximated points-to/alias sets can be performed either in bottom-up or top-down fashion. In a top-down approach, the incorrect results of the over-approximation are iteratively pruned if it can be established that the respective store and load end points cannot alias. This strategy was suggested by Sridharan et al. [33]. Top-down approaches have the advantage that they can be terminated in any refinement iteration without producing false negatives, making them appealing for demand driven or time-critical applications.

Unfortunately, top-down techniques can be very imprecise on certain programs, due to the presence of *self-supporting* bridges. Consider the graph depicted in Figure 8. The question is whether $v10$ can point to $h1$. There is clearly a path to support this: $\langle v10, v11, v7, v5, v1, h1\rangle$. But this path contains the bridge edge $(v11, v7)$. To confirm that this is still valid, we have to show that $v8$ (the source of the respective store edge) and $v3$ (the sink of the respective load edge) may alias. Indeed this is the case, as both can point to $h2$. However, for $v3$ to point to $h2$, the bridge edge $(v4, v2)$ is required. To confirm that this is still valid, we have to demonstrate that $v5$ and $v9$ may alias, and this is only the

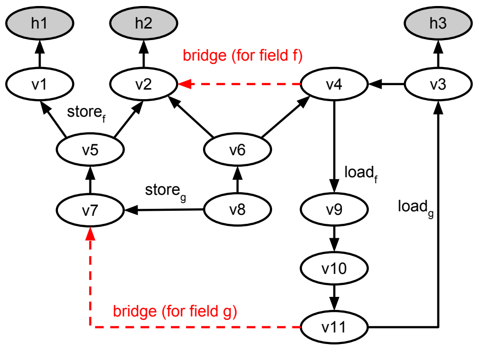

Figure 8: A *self-supporting* bridge pair. This graph is drawn from the `hsqldb` DaCapo 2006 benchmark (with variables renamed and irrelevant edges removed).

case if we can use the bridge edge $(v11, v7)$ (then both can point to $h1$). For this purpose, the argument becomes cyclic.

To overcome the precision limitations of top-down refinement, the bottom-up refinement as described in Section 3 was chosen. Bottom-up evaluation is more expensive than top-down due to the need for iterative re-evaluation, but produces a precise least-fix-point solution rather than an over-approximated solution.

The bottom-up approach starts with a non-solution and needs to be iterated until a fix-point is reached. Hence, terminating the refinement iterations before reaching the fix-point would lead to false negatives, making the analysis unsound. However, our implementation uses a fast transitive closure algorithm that takes full advantage of the sparse nature of the graph. At each iteration, we check the intersection of the points-to sets for the base variables of the bridges edges produced by the bridge-finder. If these sets overlap, the bridge edge is certified. Certified bridge edges are added in bulk at the end of the iteration to minimise re-computation of the transitive closure. This process is repeated until no more bridge edges can be certified.

Figure 9 shows the refinement step performed on the running example. In this case, the fix-point is reached after two iterations, since certifying $(v8, v1)$ depends on $(v7, v2)$ to establish that $v7$ and $v2$ have intersecting points-to sets. The two edges represented by dotted lines cannot be certified. Note that $(v9, v2)$ would also be excluded by top-down refinement. However, top-down refinement would not be able to exclude $(v10, v1)$ as this edge is self-supporting.

Figure 10 shows the final transformed graph. Now load and store edges can be removed and a precise points-to index can be computed from alloc, assign and bridge edges only, via transitive closure.

To achieve high performance, the computation of the transitive-closure and the points-to intersection must be fast. We use the algorithm suggested by Nuutila [21]. This algorithm uses Tarjan's classical algorithm [36] to compute strongly connected components (SCC), and successor sets

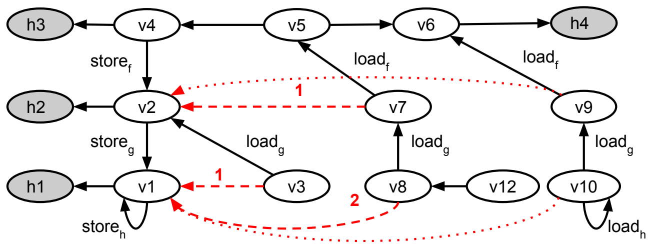

Figure 9: Bottom-up refinement produces the precise set of bridges (dashed lines). A bridge's label indicates which iteration it was certified in.

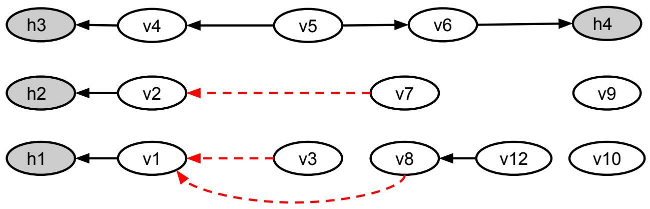

Figure 10: Transformed graph after refinement

are shared amongst the members of the same component. We follow van Schaik and de Moor's [40] suggestion to represent successor sets using compressed sparse bit vectors, and decided to use CONCISE for this purpose [40]. The time required to check points-to intersection is $\mathcal{O}(|V|)$ in the worst case, but in practice we observe significantly better performance due to the small size of the points-to sets. Our use of the reasonably accurate Dyck-reachability oracle is important here to keep the number of alias checks low.

5. Validation

We evaluate the proposed solver on a set of experimental benchmarks drawn from real-world programs.

5.1 Methodology

5.1.1 Implementations

Our experiments verify the viability of the transitive-closure based points-to evaluation strategy via comparison with well-known techniques available both as commodity software and in the research literature. The implementations are:

- **WL**: the worklist algorithm due to Melski and Reps [19, 33].
- **DP**: the difference-propagation algorithm due to Sridharan and Fink [32].
- **LB**: a Datalog implementation[6] using the LogicBlox engine, version 3.9.
- **TC**: our transitive-closure implementation.

All implementations (except the proprietary LogicBlox) are written in Java for JRE version 1.7.0_65. All experiments are run on an Intel® i7-4790 3.6GHz machine, 32GB of RAM, running Lubuntu 14.10. Since the implementations are single-threaded, execution times reported refer to wall-clock time-to-completion.

5.1.2 Datasets

Primarily, our experimental framework uses the DaCapo benchmarks [4] as input programs to generate points-to information. Static information from the `2009-Bach` version of DaCapo is dumped by the Doop framework, version `r-160113`, which makes use of Soot 2.5.0 [39], and Tamiflex [5] 2.0.1. Benchmark information is converted from the output of Doop (`.facts` files) to raw `.csv` files by simple text substitution during pre-processing. We do not include performing this pre-processing step in any algorithm for the purpose of runtime or memory usage experimentations. Pre-processing is the same for all algorithms, and would typically be handled by an analysis driver, which is out-of-scope for this research. The DaCapo benchmarks are supplemented with a very large points-to dataset, the OpenJDK version 1.7.0_b147. Due to its size, the OpenJDK dataset was generated by proprietary systems.

5.1.3 Correctness of Implementation

We run some tests to support our claims on the soundness and completeness of our algorithm (Section 3.2.1) and to ensure that we implemented the other algorithms we use for comparison correctly. To verify the correctness of the various algorithms and their implementations, we first computed the points-to relation using the basic worklist algorithm WL. Due to the size of the OpenJDK dataset, WL is unable to compute the points-to relation. In this case, a proprietary system was used.

We then used these pre-computed points-to relations to check whether all algorithms computed the same result. I.e., we used the pre-computed points-to relation as test oracle, and checked (1) whether the points-to relation computed by the tested solver is a subset of this test oracle (test for false negatives – soundness), and (2) whether the test oracle is a subset of points-to relation computed by the tested solver (test for false positives – precision). We run these tests for all programs in the DaCapo dataset. For the OpenJDK, we run a similar test using a subset of the test oracle.

[6] LogicBlox runs a custom logic program which computes the same analysis that WL, DP, and TC do (as described in Section 2). It is simpler and less precise than the context-insensitive analysis distributed with Doop [28], since the latter performs on-the-fly call-graph construction, improving precision, which our TC implementation does not currently support.

Table 1: Vertex, edge, and solution sizes for the problems used in our evaluation. For this table only, benchmark names are displayed in full.

Bench	$\|V\|$	$\|E\|$	$\|E_{pointsTo}\|$
sunflow	15,464	15,957	16,354
lusearch	15,774	14,994	9,242
luindex	18,532	17,375	9,677
avrora	24,690	25,196	21,532
eclipse	41,383	40,200	21,830
h2	44,717	56,683	92,038
pmd	54,444	59,329	60,518
xalan	58,476	62,758	52,382
batik	60,175	63,089	45,968
fop	86,183	83,016	76,615
tomcat	111,327	110,884	82,424
jython	191,895	260,034	561,720
tradebeans	439,693	466,969	696,316
tradesoap	440,680	468,263	698,567
openjdk	1,621,634	1,964,146	1,570,820,597

Table 2: Solution statistics by benchmark. The average and maximum size of a variable's points-to-set, and the maximum number of loads/stores-per-field.

Bench	Avg.	Max.	Loads/f	Stores/f
sun	1.06	140	530	11
lus	0.59	35	46	21
lui	0.52	35	94	21
avr	0.87	342	104	11
ecl	0.53	88	119	11
h2	2.06	457	520	13
pmd	1.11	221	998	212
xal	0.90	221	998	25
bat	0.76	681	155	20
fop	0.89	285	94	16
tom	0.74	325	512	38
jyt	2.93	1,878	2,118	1,363
trb	1.58	581	517	144
trs	1.59	581	517	144
jdk	968.67	82,665	962	85

5.1.4 Experiments

Our evaluation focuses on the runtime performance of the TC implementation, and examining the proportional execution times of the componentisation, folding, and refinement stages in practice. Specifically, we:

- Compare the index-generation/query times required by TC against the alternatives.
- Evaluate the proportional execution time of the componentisation, folding, and refinement stages of TC.
- Quantify the solution space of the DaCapo/OpenJDK datasets.
- Compare the bridge-finder and refinement strategies of our algorithm against alternatives in the literature.

5.1.5 Distribution

All implementations are available online[7], including scripts to perform the experiments described in this section, and input-data for the DaCapo datasets.

The Datalog implementation relies on the proprietary LogicBlox engine, an academic or commercial license must be obtained. The logic program run by LogicBlox is provided in source-form in the appendix and the online distribution.

5.2 Datasets

We present a breakdown of the problem and solution sizes by benchmark in Table 1. Our input problems represent a large range in problem sizes, though the OpenJDK dataset is significantly larger than any of the DaCapo problems. It is interesting to observe that the edge sets are mostly equal in size to their respective vertex sets. Our benchmark suite is highly sparse, which points-to graphs are known to be in the literature.

The relationship between the problem size and the solution size is less clear. The $E_{pointsTo}$ set varies greatly in size when compared to V, which indicates variable degrees of connectedness in the input problems. Indeed, Table 2 demonstrates this. The average and maximal sizes of points-to sets in the DaCapo benchmarks varies significantly. We attribute the outlying OpenJDK to the relatively higher proportion of large points-to sets in strongly connected components (see Appendix A, Table 6).

Table 2 also shows the maximal bound for load/store edges, how often the most loaded/stored field is referenced. This number motivates reasoning about the work of the bridge-finder in practice. Since the load/store bound does not scale with program size, but rather problem-specific internals, we have evidence that the work of a bridge oracle does not scale quadratically with the size of the program.

5.3 Performance

The execution time and memory usage[8] for the various solvers are shown in Table 3. We clearly see the distinction between the more scalable implementations (LB and TC) and the non-scalable ones (WL and DP). In general, WL and DP exhibit the scalability that these techniques are known for, respectively cubic and quadratic time and space. The Datalog implementation performs well on the DaCapo benchmarks, particularly the larger `Tradebeans` and `Tradesoap`, however the time needed to process Open-

[7] `https://bitbucket.org/jensdietrich/gigascale-pointsto-oopsla2015`

[8] The memory consumption of LB is internally limited to 75% of system memory.

Table 3: Runtimes (s) and memory consumption (MB) for all benchmarks. The WL and TC implementations timed out after 2 hours on the OpenJDK problem.

Bench	WL_{time}	DP_{time}	LB_{time}	TC_{time}	WL_{mem}	DP_{mem}	LB_{mem}	TC_{mem}
sun	0.21	0.20	1.01	0.38	74	77	227	96
lus	0.19	0.18	1.01	0.42	75	78	224	96
lui	0.21	0.19	1.01	0.46	81	80	226	110
avr	0.51	0.32	1.02	0.60	89	92	225	127
ecl	0.48	0.40	1.04	0.74	122	145	232	170
h2	1.92	0.83	1.11	0.74	137	226	233	211
pmd	1.61	0.58	1.11	0.80	152	228	236	220
xal	1.72	0.71	1.09	0.84	161	235	239	223
bat	1.23	0.97	1.08	0.80	161	242	239	221
fop	2.70	2.05	1.12	1.08	242	244	245	262
tom	4.45	2.13	1.15	1.27	235	326	252	303
jyt	16.62	9.83	1.54	1.86	379	611	289	502
trb	122.90	52.36	1.95	4.34	796	1,022	378	910
trs	124.56	52.44	1.96	4.39	795	1,025	378	910
jdk	*	*	3,102.10	40.13	*	*	23,530	3,296

JDK is significantly worse than TC. The TC implementation is the most scalable option, as evidenced by the under-a-minute runtime and relatively small memory footprint, for the largest benchmark. Note that the computational improvement does not come at the cost of increased index-querying times, which are reported in Appendix B.

The disparity between scalable and poorly-scaling algorithms deserves further treatment. The OpenJDK problem is only 4x larger than the next largest benchmark (`Tradesoap`), and yet the WL and DP implementations timeout on this problem. Indeed, the LB implementation slows down by 1582x compared to its runtime on `Tradesoap`. The reason for this, and a great advantage of TC, is that the WL, DP, and LB executions are intrinsically tied to the size of the *output*, which is almost 2250x larger. Specifically, WL's worklist is populated by newly discovered points-to edges, which drives the search for more edges. Similar is true of the difference sets in DP and the *pointsTo* relation in LB. By contrast, TC is only affected by the output size when checking the alias relationship between load/store bases. Having a larger (more-populated compressed bit-vector) relation, does not demand significantly more computation to check aliases (non-empty intersection). Without compression, the memory consumption of WL, DP and LB will also be affected.

In light of the faster execution time for TC we are interested in how the computations break down. Referring to the stages of the algorithm discussed in Section 4, componentisation, folding and refinement, Figure 11 shows the proportion of execution time devoted to these stages. We observe that for the smaller DaCapo benchmarks, refinement makes up more than half the execution time, where larger benchmarks require proportionally less time. The ratio between componentisation and folding seems independent of problem size, though, at least for the largest benchmarks, folding accounts for the most time.

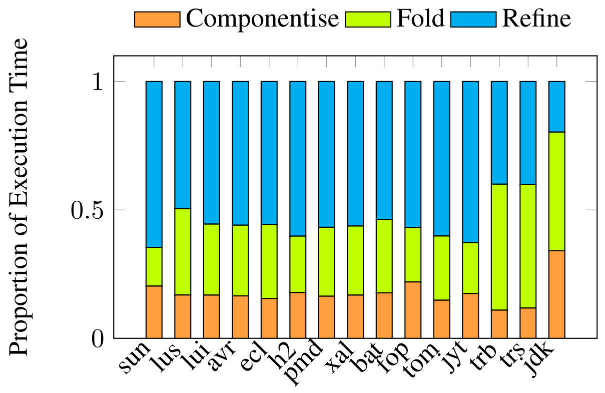

Figure 11: Execution time for TC devoted to componentisation, folding, and refinement, as a proportion of total time.

5.4 Bridge-Finders

Our implementation uses bi-directional Dyck-reachability [48] as its bridge-finder. This decision is motivated by a trade-off between having an accurate finder (which necessitates less work during refinement) and having a fast finder (less work during folding). Some alternative bridge-finders are shown in Table 4, displaying the number of potential bridges they find. Using the all-pairs bridge-finder produces prohibitively many pairs, between two and four orders of magnitude greater than the true amount. The same-field bridge-finder, which was used by Sridharan et. al. in [31], varies significantly in precision, most notably `pmd` produces 37x more potential bridges than Dyck.

Table 4: Comparison of different bridge-finders: every load with every store [All], loads and stores for the same field [Field], bi-directional Dyck-reachability used by TC [Dyck], and the true number of bridge edges [True].

Bench	All	Field	Dyck	True
sun	$8.61{\cdot}10^5$	3,934	886	747
lus	$1.46{\cdot}10^6$	5,037	793	770
lui	$2.87{\cdot}10^6$	8,722	1,228	1,212
avr	$3.58{\cdot}10^6$	4,630	756	722
ecl	$4.39{\cdot}10^6$	6,444	1,430	1,096
h2	$8.31{\cdot}10^6$	12,174	3,492	3,281
pmd	$2.20{\cdot}10^7$	65,849	1,908	1,752
xal	$2.45{\cdot}10^7$	31,041	2,233	2,104
bat	$1.20{\cdot}10^7$	19,405	2,397	1,948
fop	$1.43{\cdot}10^7$	12,551	1,962	1,744
tom	$4.93{\cdot}10^7$	30,863	8,052	7,700
jyt	$6.12{\cdot}10^7$	55,776	8,646	8,082
trb	$6.15{\cdot}10^8$	149,642	30,863	29,155
trs	$6.17{\cdot}10^8$	149,724	30,894	29,173
jdk	$7.44{\cdot}10^9$	538,274	84,415	64,716

Table 5: Comparison of refinement strategies. Init shows the initial precision (reported points-to relations vs actual) without refinement. TD shows the iterations count of the top-down refiner. Precision shows the final precision of top-down refinement. BU shows the bottom-up refiner's iteration count.

Bench	Init	TD	Precision	BU
sun	0.9857	2	1.0000	4
lus	0.9909	3	1.0000	4
lui	0.9978	2	0.9998	5
avr	0.9949	2	0.9953	6
ecl	0.9460	4	0.9606	5
h2	0.8844	3	1.0000	5
pmd	0.9736	3	1.0000	6
xal	0.9755	2	0.9998	6
bat	0.9857	3	1.0000	4
fop	0.9908	4	1.0000	8
tom	0.9830	4	1.0000	5
jyt	0.8155	3	0.9997	7
trb	0.9661	4	0.9988	6
trs	0.9662	4	0.9988	6
jdk	*	5	*	6

5.5 Refinement Strategies

The bottom-up refinement strategy used in our solver is more expensive than comparable top-down techniques. We justify its use by presenting precision data as compared with the top-down refinement strategy favoured by [31, 33]. Table 5 shows how precision varies due to refinement. Precision measures the proportion of correct points-to relations the top-down solver reports, i.e. its false-positive rate is $1 - $ precision. The initial precision (assuming all reported dyck-reachable bridges are correct) motivates the need for refinement, due to the (nearly 20%) false-positive rate of some benchmarks. Using top-down refinement (with usually fewer steps than bottom-up), we are able to bring precision close to 100%. Unfortunately, sets of bridge edges remain which mutually confirm the validity of each other, and so cannot be removed via top-down refinement[9]. In their own work, Sridharan et al., using different benchmarks, observe imprecision (11% pre-refinement, 7% post-refinement [33]) which is consistent with our results, allowing for the time-budget in their demand-driven analysis and the compounded imprecision of their same-field bridge-approximation.

6. Complexity Revisited

In this section we revisit the complexity analysis in order to explain the superior performance exhibited by our algorithm in experiments. We show how this can be attributed to the topology of the points-to graphs. We argue that the behaviour of all three steps of the algorithm is potentially better than quadratic in practice, an improvement over Sridharan and Fink's observation [32].

6.1 Componentisation

Componentisation can be performed in linear time using simple depth first search (DFS). Note that we can merge components using proxies. I.e., instead of replacing the component already assigned to vertices, we just treat this component as an alias for another component. In order to perform well during the folding stage, our implementation chooses a good component ordering here, which increases the componentisation time, especially for larger benchmarks (see Figure 11).

6.2 Folding

Zhang et al. have demonstrated that the complexity of the folding step is $\mathcal{O}(n + m \log m)$, where $n = |V|$ and $m = |E|$. Our implementation differs from their algorithm in that bridge vertices are used. The same-field oracle size reported in Table 4 is a conservative estimate for the steps that must be performed by our algorithm: all load/store pairs with identical fields must be processed. Checking that the store's source and load's sink are in the same Dyck-component can be done in constant time. Comparing the data in Tables 1 and 4 indicates that the number of these load/store pairs increases at most linear to the size of the graph, in fact, the largest values of ratios same-field oracle / vertex count can be found in smaller programs such as `luindex` and `xalan`.

The sparsity of load and stores edges can be attributed to the fact that Java encourages field encapsulation and

[9] The most imprecise benchmark shown is `eclipse`, at 96%, however experimentation on other benchmarks has revealed real-world adverse cases with as low as 40% precision for the `bloat` dataset that is included in DaCapo 2006.

field access through getters and setters through conventions, tool support and component models. Empirical studies have demonstrated that even when this convention is violated, direct access to exposed fields is rare [37].

6.3 Refinement

Refinement is performed in k steps corresponding to the level of recursion. We have observed that k is a small constant (BU column of Table 5), and that the size of the bridge oracle set l is roughly linear in the size of the graph (Table 4). Finally, we use a variant of the simple cubic transitive-closure algorithm from [40]. The cubic complexity stems from the fact that m edges must be visited, n^2 in the worst case. As our graphs are very sparse ($m \approx n$), the algorithm runs in quadratic time. Moreover, in [40], the authors assume that the complexity of a single successor set merge operation is $\mathcal{O}(n)$. This is only the case if a graph is rather dense (which is not the case here), and successor sets consist mainly of literals – sections that cannot be compressed. Our implementation avoids this by encoding vertices during depth-first traversal, ensuring that successor sets often consist of vertices with adjacent indices, hence more readily compressed. The sets are made even sparser by not recording all successors, but only the points-to sets (i.e. the partial successor sets only consisting of heap object vertices). For all programs we analysed, including the OpenJDK data set, we found that the size of these sets was 10 or less for over 92.8% of the variable vertices. The lowest value (by far) was observed for the OpenJDK dataset. However, when we consider the sizes of *unique* points-to sets, then the share of small points-to sets for the OpenJDK increases to 98.03%. This indicates that the larger points-to sets tend to be the points-to sets of variable vertices in strongly-connected components, and they can therefore be shared. Also note that the presence of a very few very large points-to sets that cause the average size of the points-to sets to increase with the size of the program (Table 2) does not necessarily make the algorithm slow as long as a dense representation can be found[10]. Then set merging becomes a defacto constant time operation. The same applies to the intersection operation needed during certification. Therefore, the transitive closure can also be computed in near-linear time. This is consistent with the results in Figure 11. It is particularly interesting to see that large programs need relatively less time for refinement, even though this part of the algorithm has the highest computational complexity.

7. Related Work

7.1 Transitive Closure

Naive algorithms to compute the transitive closure of directed graphs have cubic time and square space complexity. The computation of transitive closure can be mapped to binary matrix multiplication, and existing sub-cubic matrix multiplication algorithms can be used [3, 10, 11, 35]. More recent approaches to compute transitive closure precompute an index using effective data-structures that support fast queries. In many cases, single source – single sink queries can be answered in constant time. Index construction must strike the right balance between construction time, memory space required and query time. Often, index generation algorithms can take advantage of certain characteristics of the graph.

In chain / path compression [13], a chain cover is built and reachability information can be effectively compressed using these chains. In tree cover [1], a spanning tree is constructed, and reachability information is represented by labels that describe descendants within the tree. Both approaches work well if long chains and spanning trees exist and can be discovered, but have problems dealing with cross-chain and non-tree edges, respectively. Several improvements have been suggested, including combinations of tree and chain cover [14], and probabilistic approaches [47]. An alternative is to build an index around hubs within the graph, i.e., vertices or edges with high betweenness centrality. 2-HOP [8] and 3-HOP [15] are based on this idea, 3-HOP uses “highways” to compress reachability information, while 2-HOP uses single hub vertices. Based on our experimental results, we found that points-to graphs do not have any of the topology features, which the aforementioned approaches require. Points-to graphs tend to be sparse, have no hubs, and there are no long chains or trees that dominate these graphs.

A simple yet very effective approach to compute the transitive closure has been proposed by Nuutila [21]. The algorithm is based on Tarjan’s algorithm to detect strongly connected components [36]. While the graph is traversed using simple depth first search, successor sets are built and propagated down. In particular, the successor sets of child vertices are merged into the successor set of a parent vertex, and successor sets are shared amongst the vertices within a strongly connected component. Much of the computational complexity is shifted to merging successor sets. This can be done effectively by choosing suitable data-structures. Nuutila suggested the use of interval lists, while van Schaik and de Moor proposed compressed bit vectors [40]. Compressed bit vectors have the advantage that blocks of 32 or 64 vertices can be processed without computational overhead, a technique similar to the Four Russians trick [3]. Suitable compression schemes include WAH [42], PWAH [40] and CONCISE [9].

7.2 CFLR and Points-To Analysis

Yannakakis [46] introduced the foundations of CFLR, showing the relationship between recursively defined relations and Context-free reachability. There are several known algorithms for computing both general CFLR, and variants with restricted graph and grammar classes (such as Dyck grammars on bi-directed graphs). A cubic algorithm for gen-

[10] Compressed sparse bit vectors are symmetric in the sense that they can handle very dense and very sparse areas of the graph equally well.

eral context-free languages that is based on dynamic programming, was proposed by Melski and Reps [19]. The worst-case runtime complexity of the cubic algorithm was improved by Chaudhuri, using the Four Russians trick [7], i.e. dense set operations, reducing the runtime to $\mathcal{O}(\frac{n^3}{\log n})$. Reps [23] applied CFLR to phrase an Andersen-style analysis [2] for C programs. This was later adapted to a field-sensitive Andersen-style analysis for Java in the work of Sridharan et al. [31, 33].

Though our work presents the same points-to formulation as Sridharan et al., the ideas are very different. Most notably, refinement in their work is not equivalent to our "certification"-style techniques. Broadly, their match edges are refined when the client analysis is unsatisfied with the current precision of the result, meaning the points-to set is recomputed in their absence *via conventional worklist techniques*. Notably, "valid" matches are never remembered between queries, and the removal of matches increases the work done by future iterations, doubly so in [31], where the match elided both context and field information. Since we treat confirmed bridges as assignments, the regular-language reachability analysis truly is solved by transitive closure, which the context-insensitive variant [33] cannot leverage, despite having a regular approximation.

Zhang et al. [48] have proposed a fast algorithm to solve the CFLR problem for Dyck grammars on bi-directional graphs. Their algorithm requires the graph have new edges that simultaneously reverse both the direction and the labels, then it may recursively merges vertices on this bi-directed graph whenever matching labels are encountered, producing growing equivalence classes of vertices. Reachability between vertices is then defined with respect to common membership in one of these classes in time nearly linear to the edge-set ($O(|V| + |E|\log(|E|))$). They propose a memory-alias grammar as an application of their algorithm, which performs a Steensgard's style analysis [34] with field-sensitivity. Whilst useful for fast pre-analyses in some points-to analyses (e.g. [44] and our own work), it lacks precision on its own. For our algorithm, the fast Dyck algorithm was fundamental to find potential bridges efficiently with an acceptable false positive rate.

Points-to analysis for C was solved using CFLR by Zhang et al. in [49]. Their Andersen-style inclusion-based points-to is simpler than our Java analysis (which has field sensitivity), but the scalability challenges faced by those authors are relevant. Firstly, they have a more traditional CFLR formulation, which is modified to selectively propagate information along *meaningful* summary edges, unlike WL, which propagates over all edges. The authors also adapt Chaudhiri's [7] sub-cubic improvement for faster runtimes. Though they have a similar motivation, their approach significantly differs from ours, as we abandon the CFLR machinery entirely in favour of transitive-closure.

Sridharan and Fink have investigated the scalability of Andersen style analysis and found that it becomes quadratic if the graph is sparse enough [32]. Their algorithm is based on Pierce's difference propagation algorithm [22]. This has similarities with our algorithm where we propagate points-to sets. In [44], Xu et al. introduce a points-to analysis that performs a pre-analysis that determines whether two program variables may be aliases. The pre-analysis is used for filtering out infeasible CFL-paths using an augmented CFLR algorithm that takes the pre-analysis into account. In contrast, our approach uses a pre-analysis to find potentially matching load/store pairs (not to exclude pairs). Instead of a standard CFLR algorithm, we propose a transitive closure data-structure as a computational vehicle for solving CFLR problems to obtain high-performance. In [43], a points-to analysis uses equivalence classes to merge abstract contexts and employs a last-k-substring merging for trading-off scalability and precision. Our approach is context-insensitive but can be extended to context-sensitivity using cloning ideas.

8. Conclusion

We presented a new approach for solving the field-sensitive points-to problem for Java. The proposed algorithm is based on (1) an incremental reflexive transitive-closure problem, and (2) a pre-computed set of potentially matching load/store pairs to speed up the fix-point calculation. For the pre-computation we use the latest work in Dyck-Reachability to obtain matching load/store pairs efficiently and effectively. The algorithm computes a least-fix-point solution, for which we give a parameterised worst-case runtime complexity. We conducted experiments for the DaCapo benchmark, comparing our approach against commercial and academic alternatives. Our experiments demonstrate significant improvements, especially for large datasets. For the OpenJDK library, our approach computes a points-to index, with over 1.5 billion tuples, in under a minute.

While our algorithm provides a precise solution for the problem addressed – field-sensitive points-to analysis for Java – we acknowledge that this problem formulation itself lacks precision [24]. We see our work as a stepping stone, and more work is needed to investigate whether similar algorithms can be devised that support context-sensitivity [20, 26, 27, 29], flow-sensitivity [6, 41] and call graph construction on the fly [12, 20, 38]. This increases the complexity of the problem. On the other hand, the computed points-to relation becomes sparser, and this could at least partially offset the performance penalty for algorithms [17]. In particular, we can expect this to happen for algorithms that take full advantage of sparse data-structures, similar to the work presented here.

Acknowledgments

This project was supported by Oracle Labs, Australia, and ARC grant DP130101970. We would like to thank Cristina

Cifuentes, Herbert Jordan, Padmanabhan Krishnan, Yi Lu, K. R. Raghavendra, Andrew Santosa, Pavle Subotic, Johannes Tandler, Daniel Wainwright, and Chenyi Zhang for their valuable feedback, as well as Yannis Smaragdakis for help with LogicBlox.

References

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings SIGMOD'89*. ACM, 1989.

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

[3] V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. On economic construction of the transitive closure of a directed graph. *Soviet Math. Dokl.*, 11:1209–1210, 1970.

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, 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 *Proceedings OOPSLA'06*. ACM, 2006.

[5] E. Bodden, A. Sewe, J. Sinschek, M. Mezini, and H. Oueslati. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceeding ICSE '11*. ACM, 2011.

[6] R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant context inference. In *Proceedings POPL'99*. ACM, 1999.

[7] S. Chaudhuri. Subcubic algorithms for recursive state machines. In *Proceedings POPL'08*. ACM, 2008.

[8] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[9] A. Colantonio and R. Di Pietro. Concise: Compressed ncomposable integer set. *Information Processing Letters*, 110(16): 644–650, 2010.

[10] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9 (3):251–280, 1990.

[11] F. L. Gall. Powers of tensors and fast matrix multiplication. *arXiv preprint arXiv:1401.7714*, 2014.

[12] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):685–746, 2001.

[13] H. Jagadish. A compression technique to materialize transitive closure. *ACM Transactions on Database Systems (TODS)*, 15 (4):558–598, 1990.

[14] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *Proceedings SIGMOD'2008*. ACM, 2008.

[15] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *Proceedings SIGMOD'2009*. ACM, 2009.

[16] O. Lhoták. Spark: A flexible points-to analysis framework for java, 2002.

[17] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(1):3, 2008.

[18] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings USENIX'05*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[19] D. Melski and T. Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248(1):29–98, 2000.

[20] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 1–11. ACM, 2002.

[21] E. Nuutila. *Efficient transitive closure computation in large digraphs*. PhD thesis, PhD thesis, Helsinki University of Technology, 1995. Acta Polytechnica Scandinavica, Mathematics and Computing in Engineering Series, 1995.

[22] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Online cycle detection and difference propagation: Applications to pointer analysis. volume 12, pages 311–337. Kluwer, Dec. 2004. .

[23] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11):701–726, 1998.

[24] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proceedings CC'03*. Springer, 2003.

[25] L. Shang, X. Xie, and J. Xue. On-demand dynamic summary-based points-to analysis. In *Proceedings CGO'12*. ACM, 2012.

[26] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*. New York University, 1978.

[27] O. Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University Pittsburgh, 1991.

[28] Y. Smaragdakis and M. Bravenboer. Using datalog for fast and easy program analysis. In O. de Moor, G. Gottlob, T. Furche, and A. Sellers, editors, *Datalog Reloaded*, volume 6702 of *LNCS*, pages 245–251. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-24205-2. .

[29] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. *ACM SIGPLAN Notices*, 46(1):17–30, 2011.

[30] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras. Introspective analysis: context-sensitivity, across the board. In *Proceedings PLDI'2014*. ACM, 2014.

[31] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for java. In *Proceedings PLDI'06*. ACM, 2006.

[32] M. Sridharan and S. J. Fink. The complexity of Andersens analysis in practice. In *Static Analysis*, pages 205–221. Springer, 2009.

[33] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for java. In *Proceedings OOPSLA'05*. ACM, 2005.

[34] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings POPL '96*. ACM, 1996.

[35] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[36] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[37] E. Tempero. How fields are used in Java: An empirical study. In *Proceedings ASWEC'09*. IEEE, 2009.

[38] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings OOPSLA'00*. ACM, 2000.

[39] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings CASCON '99*. IBM Press, 1999.

[40] S. J. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In *Proceedings SIGMOD'11*. ACM, 2011.

[41] J. Whaley and M. S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Static Analysis*, pages 180–195. Springer, 2002.

[42] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems (TODS)*, 31(1):1–38, 2006.

[43] G. Xu and A. Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *Proceedings ISSTA '08*. ACM, 2008.

[44] G. Xu, A. Rountev, and M. Sridharan. Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings ECOOP'09*. Springer, 2009.

[45] D. Yan, G. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for java. In *Proceedings ISSTA'11*. ACM, 2011.

[46] M. Yannakakis. Graph-theoretic methods in database theory. In *Proceedings PODS'90*. ACM, 1990.

[47] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. *Proceedings of the VLDB Endowment*, 3(1-2):276–284, 2010.

[48] Q. Zhang, M. R. Lyu, H. Yuan, and Z. Su. Fast algorithms for dyck-cfl-reachability with applications to alias analysis. In *Proceedings PLDI'13*. ACM, 2013.

[49] Q. Zhang, X. Xiao, C. Zhang, H. Yuan, and Z. Su. Efficient subcubic alias analysis for c. In *Proceedings OOPSLA'14*, OOPSLA '14, New York, NY, USA, 2014. ACM.

A. Benchmark Statistics

This appendix contains some details about the size of points-to sets (Table 6), and statistics of edge labels in the input problem (Table 7).

Table 6: Percentage of Small Points-to sets. Small is defined as having a size of 10 or less. The proportion is shown where points-to sets of strongly connected components are counted repeatedly (shared) and only once (unique).

Bench	Shared small sets	Unique small sets
sun	97.96	97.62
lus	99.49	99.22
lui	99.56	99.45
avr	98.81	97.95
ecl	99.66	99.16
h2	94.89	97.15
pmd	97.46	98.46
xal	98.66	97.68
bat	98.96	98.26
fop	98.25	98.86
tom	98.79	98.57
jyt	99.03	98.06
trb	98.05	98.27
trs	98.05	98.27
jdk	92.83	98.03

Table 7: Breakdown of edge labels in the input problem.

Bench	Alloc	Assign	Load	Store
sun	3,306	9,972	2,305	374
lus	2,633	9,266	2,515	580
lui	3,273	9,903	3,340	859
avr	4,526	16,009	3,684	977
ecl	7,129	27,535	4,575	961
h2	7,339	41,392	6,709	1,243
pmd	7,552	38,676	11,109	1,992
xal	8,760	40,102	11,813	2,083
bat	10,322	43,905	7,176	1,686
fop	20,462	53,350	7,212	1,992
tom	22,962	69,473	15,198	3,251
jyt	30,830	210,346	14,685	4,173
trb	69,597	335,195	49,794	12,383
trs	69,718	336,279	49,858	12,408
jdk	347,515	1,411,505	157,804	47,322

B. Query Performance

We show the single-source single-sink, and single-source all-sinks query times for the benchmarks. The TC implementation uses compressed bit-vectors to store points-to information, which may affect the query performance. The query times for the WL and TC implementations are shown in Table 8. Query times do not scale with program size, but rather internal forces like cache locality and problem density. Furthermore, decompressing the compressed bit-vector has some overhead, but this at worst doubles the query time.

Table 8: Query times for all benchmarks, shown as ms per 1000 queries, Single-sink and All-sinks. The oracles necessary to determine single-source single-sink query times for OpenJDK are too large for experimentation.

Bench	WL_{single}	TC_{single}	WL_{all}	TC_{all}
sun	2.2	3.7	0.3	0.8
lus	2.9	5.3	0.5	1.5
lui	3.0	5.5	0.4	0.8
avr	2.0	3.4	0.3	0.3
ecl	2.3	4.0	0.3	0.3
h2	0.6	0.9	0.3	0.5
pmd	1.2	1.5	0.3	0.3
xal	1.4	1.9	0.3	0.3
bat	1.5	2.2	0.3	0.3
fop	0.8	1.5	0.3	0.5
tom	0.9	1.9	0.3	0.3
jyt	0.3	0.4	0.3	0.4
trb	0.4	0.5	0.2	0.4
trs	0.4	0.5	0.2	0.4
jdk	*	*	*	98.4

C. Datalog Source-Code

This appendix contains the source code of the Datalog program used in the experiments with LB. Note that the type-cardinality directive `lang:physical:capacity` and import scripts must be generated as-needed.

```
/*
 * Types
 */
FieldSignatureRef(?x),
FieldSignatureRef:Value(?x:?s) ->
    string(?s).
HeapAllocationRef(?x),
HeapAllocationRef:Value(?x:?s) ->
    string(?s).
VarRef(?x),
VarRef:Value(?x:?s) ->
    string(?s).

/*
 * EDB
 */
// var = new Obj();
Alloc(?var, ?obj) ->
    VarRef(?var),
    HeapAllocationRef(?obj).

// dst = src;
Assign(?src, ?dst) ->
    VarRef(?src),
    VarRef(?dst).

// dst = (type)src;
Cast(?src, ?dst) ->
    VarRef(?src),
    VarRef(?dst).

// dst = base.field;
Load(?base, ?dst, ?field) ->
    VarRef(?base),
    VarRef(?dst),
    FieldSignatureRef(?field).

// base.field = src;
Store(?src, ?base, ?field) ->
    VarRef(?src),
    VarRef(?base),
    FieldSignatureRef(?field).

/*
 * IDB
 */
Bridge(?dst, ?src) <-
    FieldIsStored(?src, ?field, ?obj),
    FieldIsLoaded(?dst, ?field, ?obj).

FieldIsLoaded(?target, ?field, ?obj) <-
    NewLoad(?target, ?field, ?base),
    VarPointsTo(?obj, ?base).

FieldIsStored(?src, ?field, ?obj) <-
    NewStore(?src, ?field, ?base),
    VarPointsTo(?obj, ?base).

NewAssign(?to, ?from) <-
    Assign(?from, ?to).
NewAssign(?to, ?from) <-
    Cast(?from, ?to).

NewLoad(?target, ?field, ?base) <-
    Load(?base, ?target, ?field).

NewStore(?src, ?field, ?base) <-
    Store(?src, ?base, ?field).

VarPointsTo(?obj, ?var) <-
    Alloc(?var, ?obj).
VarPointsTo(?obj, ?to) <-
    NewAssign(?to, ?from),
    VarPointsTo(?obj, ?from).
VarPointsTo(?obj, ?dst) <-
    Bridge(?dst, ?src),
    VarPointsTo(?obj, ?src).
```