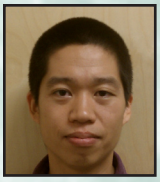
ISR Institute for Software Research
University of California, Irvine

Disjoint Reachability Analysis

James C. Jenista
University of California, Irvine
jjenista@uci.edu



Yong Hun Eom
University of California, Irvine
yeom@uci.edu

Brian Demsky
University of California, Irvine
bdemsky@uci.edu

June 2010

ISR Technical Report # UCI-ISR-10-4

Institute for Software Research
ICS2 221
University of California, Irvine
Irvine, CA 92697-3455
www.isr.uci.edu

Disjoint Reachability Analysis

UCI-ISR-10-4
June 2010

James Jenista, Yonghun Eom, and Brian Demsky

University of California, Irvine
Institute for Software Research

Abstract. We present a disjoint reachability analysis for Java. Our analysis computes extended points-to graphs annotated with reachability states. Each heap node is annotated with a set of reachability states that abstract the reachability of objects represented by the node. The analysis also includes a global pruning step which analyzes a reachability graph to prune imprecise reachability states that cannot be removed with local reasoning alone. We have implemented the analysis and evaluated it with several benchmarks. Our evaluation shows that the analysis reported the sharing for our benchmarks. We parallelized several benchmarks using the analysis results and obtained speedups of up to 61.6×.

1 Introduction

This paper introduces a static analysis that discovers disjoint reachability properties for Java. The analysis extends a standard pointer analysis with reachability states to maintain precise reachability properties. A reachability state for an object lists the allocation sites and the number of objects allocated at the site that may reach the given object. Reachability states enable the analysis to, for example, discover that objects allocated at a given site may be reached by an object allocated at site 1 or 2, but not both.

The analysis uses heap nodes in a points-to graph to abstract the objects allocated at a given allocation site. The analysis annotates the edges and nodes with sets of reachability states that abstract the heap reachability properties. These annotations enable our analysis to precisely reason about reachability in the presence of summarization and represent the key extension of our work beyond existing pointer analyses.

Our analysis is demand-driven — it takes as input a set of allocation sites that are of interest to the analysis client. The analysis then computes the reachability from the objects allocated at the selected allocate sites to all objects in the program.

To parallelize serial method calls, it is necessary to determine that they do not have conflicting data accesses. Our analysis enables new static-dynamic hybrid approaches to parallelizing code in which the combination of the static analysis results and some variant of a lock ensures the absence of conflicting accesses. This hybrid approach promises to allow a broader class of applications to be parallelized — in many cases, it can parallelize applications in which the absence of conflicting accesses cannot be statically determined and even applications that conditionally perform conflicting accesses.

The analysis results are also useful for verifying that sequential code was correctly parallelized. For example, the worker thread design pattern is commonly used to execute tasks in parallel. The worker thread pattern has significant advantages — it eliminates many deadlock concerns. However, this parallelization pattern typically relies on tasks

accessing disjoint parts of the heap. Disjoint reachability analysis can warn of possible sharing and the results often suggest locking strategies to avoid data races.

The paper makes the following contributions:

- **Disjoint Reachability Analysis:** It presents a new demand-driven analysis that discovers disjoint reachability properties. For example, it can determine that an object is reachable from at most one object abstracted by a summarized heap node or that an object is reachable from at most one of two different objects.
- **Reachability Abstraction:** It extends the points-to graph abstraction with reachability annotations to precisely reason about reachability properties.
- **Global Pruning:** It introduces a global pruning algorithm to improve the precision of reachability states.
- **Experimental Results:** It presents experimental results for several benchmarks. The results show that the analysis successfully discovers disjoint reachability properties and that it is suitable for parallelizing the benchmarks with significant speedups.

2 Example

Figure 1 presents an example that constructs several graphs. The `graphLoop` method populates an array with `Graph` objects. For this example, we assume that the analysis client needs to know the reachability of all objects in the program from the `Graph` objects allocated at line 4. Our analysis will show that each `Vertex` object is reachable from at most one `Graph` object. This information could be used to parallelize operations on different `Graph` objects. If a runtime check shows that method invocations operate on different `Graph` objects, then our static analysis results will imply that the method invocations operate on disjoint sets of `Vertex` objects.

```
1  public Graph[] graphLoop(int nGraphs) {
2   Graph[] a=new Graph[nGraphs];
3    for(int i=0; i<nGraphs; i++) {
4      Graph g=new Graph(); /* Analysis client flags this site */
5      Vertex v1=new Vertex();
6      g.vertex=v1;
7      Vertex v2=new Vertex();
8      v2.f=v1; v1.f=v2;
9      a[i]=g;
10   }
11   return a;
12 }
```

Fig. 1. Graph Example

3 Analysis Abstractions

This section presents the analysis abstractions. Abstractions are given for the input program, elements of the reachability graph, and reachability annotations that extend reachability graphs.

3.1 Program Representation

The analysis takes as input a standard control flow graph representation of each method. Program statements have been decomposed into statements relevant to the analysis: copy, load, store, object allocation, and call site statements. For a statement s in a method's control flow graph, we define the program point just before s as $\bullet s$ and the program point just after s as $s\bullet$.

3.2 Reachability Graph Elements

Our analysis computes a reachability graph for the exit of each program statement. Reachability graphs extend the standard points-to graph representation to maintain object reachability properties. Heap nodes represent objects in the heap. There are two heap nodes for each allocation site in the program — one heap node represents the single most recently allocated object at the allocation site, and the other is a summary node that represents all other objects allocated at the site[1].

In general, analysis clients only need to determine reachability from some subset of the objects in the program. The analysis takes as input a set of allocation sites for objects of interest — the analysis then computes for all objects in the program their reachability from objects allocated at those sites. We call the heap nodes for these allocation sites *flagged heap nodes* and shade them in all graphs in this paper.

The reachability graph G has the set of heap nodes $n \in N = \texttt{Allocation sites} \times \{0, \texttt{summary}\}$. The analysis client specifies a set of heap nodes $N_F = \texttt{Flagged allocation sites} \times \{0, \texttt{summary}\} \subseteq N$ that it is interested in determining reachability from.

Graph edges $e \in E$ abstract references $r \in R$ in the concrete heap and are of the form $\langle v, n\rangle$ or $\langle n, f, n'\rangle$. The heap node or variable that edge e originates from is given by $\mathrm{src}(e)$ and the heap node that edge e refers to is given by $\mathrm{dst}(e)$. Every reference edge between heap nodes has an associated field $f \in F = \texttt{Fields} \cup \texttt{element}$[2].

The equation $E \subseteq V \times N \cup N \times F \times N$ gives the set of reference edges E in a reachability graph. We define the convenience functions: $E_e(v) = \{\langle v, n\rangle \mid \langle v, n\rangle \in E\}$; $E_n(v) = \{n \mid \langle v, n\rangle \in E\}$; $E_e(n) = \{\langle n, f, n'\rangle \mid \langle n, f, n'\rangle \in E\}$; $E_n(n) = \{n' \mid \langle n, f, n'\rangle \in E\}$; $E_e(v, f) = \{\langle n, f, n'\rangle \mid \langle v, n\rangle, \langle n, f, n'\rangle \in E\}$; and $E_n(v, f) = \{n' \mid \langle v, n\rangle, \langle n, f, n'\rangle \in E\}$.

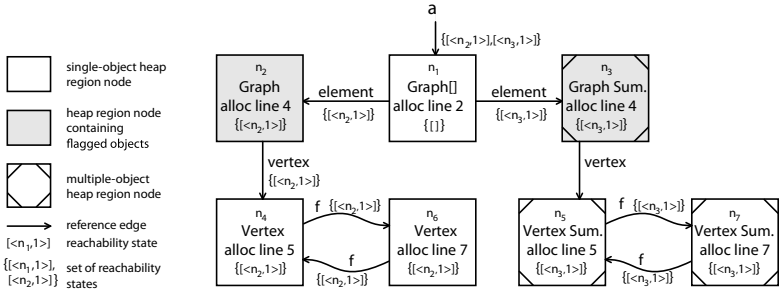


Fig. 2. Analysis result at line 11 of `graphLoop`.

Figure 2 presents the reachability graph at line 11 of the example program. Heap nodes are assigned unique identifiers of form $\texttt{n}_\texttt{i}$, where i is an unique integer. The node $\texttt{n}_1$ represents the `Graph` array object that is allocated at line 2, the nodes $\texttt{n}_2$ and $\texttt{n}_3$ represent the `Graph` objects that are allocated at line 4, the nodes $\texttt{n}_4$ and $\texttt{n}_5$ represent the `Vertex` objects that are allocated at line 5, and the nodes $\texttt{n}_6$ and $\texttt{n}_7$ represent the `Vertex` objects that are allocated at line 7. The heap nodes for the allocation site at

[1] Our implementation generalizes this to support abstracting the k most recently allocated objects from the allocation site with single-object heap nodes.

[2] The special field `element` represents all references from an array's elements.

line 4 are shaded to indicate that the analysis client is interested in reachability from the objects allocated at this site. We denote summary heap nodes as rectangles with chords across each corner.

3.3 Reachability Annotations

This section overviews how the analysis extends a points-to graph with reachability annotations, Appendix B provides a more formal treatment. The analysis computes reachability states that abstract the reachability of all objects from objects of interest.

A reachability tuple $\langle n, \mu \rangle \in M$ is a heap node and arity pair where the arity value μ is taken from the set $\{0, 1, \texttt{MANY}\}$. The arity μ gives the number of objects from the heap node n that can reach the relevant object. The arity `0` means the object is not reachable from any objects in the given heap node, the arity `1` means the object is reachable from at most one object in the given heap node, and the arity `MANY` means the object is reachable from any number of objects in the given node. The arities have the following partial order $\texttt{0} \sqsubseteq \texttt{1} \sqsubseteq \texttt{MANY}$.

A reachability state $\phi \in \Phi$ contains exactly one reachability tuple for every distinct flagged heap node. For efficiency, our implementation elides arity-`0` reachability tuples. When we write reachability states, we use brackets to enclose the reachability tuples to make them visually more clear. For example, the reachability state $\phi_n = [\langle \texttt{n}_3, \texttt{1} \rangle] \in \Phi_{\texttt{n}_7}$ that appears on node $\texttt{n}_7$ in Figure 2 indicates that it is possible for at most one object in heap node $\texttt{n}_3$, and zero objects from any other flagged heap nodes (i.e. $\texttt{n}_2$) to reach an object from heap node $\texttt{n}_7$.

The function $\mathcal{A}^N : N \to \mathcal{P}(\mathcal{P}(M))$ maps a heap node n to a set of reachability states. The reachability of an object represented by the heap node n is abstracted by one of the reachability states given by the function $\mathcal{A}^N$. We represent $\mathcal{A}^N$ as a set of tuples of heap nodes and reachability states and define the helper function:

$$\mathcal{A}^N(n) = \{\phi \mid \langle n, \phi \rangle \in \mathcal{A}^N\}. \tag{3.1}$$

When a new reference is created, the analysis must propagate reachability information. Simply using the graph edges to do this propagation would yield imprecise results. To improve the precision of this propagation step, the analysis maintains for each edge the reachability states of all objects that can be reached from that edge. The function $\mathcal{A}^E : E \to \mathcal{P}(\mathcal{P}(M))$ maps a reference edge e to the set of reachability states of all the objects reachable from the references abstracted by e. For example, the set of reachability states $\{[\langle \texttt{n}_2, \texttt{1} \rangle], [\langle \texttt{n}_3, \texttt{1} \rangle]\}$ that appears on the variable `a`'s edge in Figure 2 indicates that all objects transitively reachable from the variable `a` have either the reachability state $[\langle \texttt{n}_2, \texttt{1} \rangle]$ or $[\langle \texttt{n}_3, \texttt{1} \rangle]$. We represent $\mathcal{A}^E$ as a set of tuples of edges and reachability states and define the helper functions:

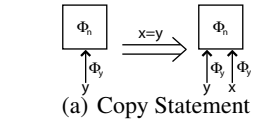
$$\mathcal{A}^E(v) = \{\phi \mid \langle \langle v, n \rangle, \phi \rangle \in \mathcal{A}^E\}, \tag{3.2}$$

$$\mathcal{A}^{E^\circ}(v) = \{\langle \langle v, n \rangle, \phi \rangle \mid \langle \langle v, n \rangle, \phi \rangle \in \mathcal{A}^E\}, \tag{3.3}$$

$$\mathcal{A}^E(e) = \{\phi \mid \langle e, \phi \rangle \in \mathcal{A}^E\}. \tag{3.4}$$

4 Intraprocedural Analysis

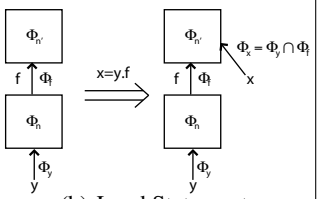
We begin by presenting the intraprocedural analysis. Section 5 will extend this analysis to support method calls. The analysis is structured as a fixed-point computation.



(a) Copy Statement

$$E' = (E - E_e(\mathtt{x})) \cup (\{\mathtt{x}\} \times E_n(\mathtt{y})) \quad (4.1)$$

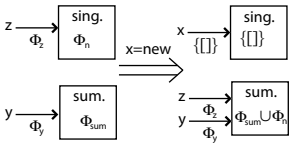
$$\mathcal{A}^{E'} = (\mathcal{A}^E - \mathcal{A}^{E^\circ}(\mathtt{x})) \cup (\{\mathtt{x}\} \times \mathcal{A}^E(\mathtt{y})) \quad (4.2)$$



(b) Load Statement

$$E' = (E - E_e(\mathtt{x})) \cup (\{\mathtt{x}\} \times E_n(\mathtt{y}, \mathtt{f})) \quad (4.3)$$

$$\mathcal{A}^{E'} = (\mathcal{A}^E - \mathcal{A}^{E^\circ}(\mathtt{x})) \cup \bigcup_{\langle n, \mathtt{f}, n' \rangle \in E_e(\mathtt{y},\mathtt{f})} \Big(\{\langle \mathtt{x}, n' \rangle\} \times \Big(\mathcal{A}^E(\langle \mathtt{y}, n \rangle) \cap \mathcal{A}^E(\langle n, \mathtt{f}, n' \rangle) \Big) \Big) \quad (4.4)$$



(c) Allocation Statement

1. Rewrite single-object node symbol into summary node symbol in all reachability states in the graph.
2. Merge single-object node into summary node.
3. Create new single-object node.
4. If the allocation site is flagged, generate appropriate reachability states from new node and edge. Otherwise, generate sets that include the empty reachability state.

Fig. 3. Transfer Functions for Copy, Load, and Allocation Statements

4.1 Method Entry

The method entry transform creates an initial reachability graph to model the part of the calling methods' heaps that are reachable from parameters. In the example, the initial reachability graph is empty because the method `graphLoop` does not take parameters. Method context generation is explained in detail in Section 5.1.

4.2 Copy Statement

A copy statement of the form `x=y` makes the variable `x` point to the object that `y` references. The analysis always performs strong updates for variables — it discards all the reference edges from variable `x` and then copies all the edges along with their reachability states from variable `y`. Equation 4.1 and Equation 4.2 give the transformations.

4.3 Load Statement

Load statements of the form `x=y.f` make the variable `x` point to the object that `y.f` references. Existing reference edges for the field are copied to `x` as given in Equation 4.3. Note that this statement does not change the reachability of any object. The reachability on new edges from `x`, as given in Equation 4.4, requires the intersection of $\mathcal{A}^E(\langle \mathtt{y}, n \rangle)$ and $\mathcal{A}^E(\langle n, \mathtt{f}, n' \rangle)$, because `x` can only reach objects that were reachable from both the variable `y` and a heap reference abstracted by the edge $\langle n, \mathtt{f}, n' \rangle$.

4.4 Object Allocation Statement

The analysis represents the most recently allocated object from an allocation site as a single-object heap node. A summary node for the allocation site represents any objects from the allocation site that are older than the most recent.

The object allocation transform merges the single-object node into the site's summary node. The single-object node is then the target of a variable assignment. As stated, the single-object node and its reachability information merge with the summary node. Note that if both the single-object node and the summary node appeared in the same

reachability state before this transform, afterward there will be two summary node tuples in the state. In this case the new arity for the summary heap node is given by $+_{\triangle}$, which is addition in the domain $\{0, 1, \geq 0\}$. Note that the reachability annotations enable the analysis to maintain precise reachability information over summarizations.

Finally, if the heap node is flagged, the analysis generates the set of reachability states $\{[\langle n_f, \mathtt{1}\rangle]\}$, where n_f is the given heap node, for the new object's node and edge. Otherwise, it generates the set $\{[]\}$ with the empty state for the node and edge.

4.5 Store Statement

Store statements of the form `x.f=y` point the `f` field of the object referenced by `x` to the object referenced by `y`. Equation 4.5 describes how a store changes the edge set.

$$E' = E \cup (E_n(\mathtt{x}) \times \{\mathtt{f}\} \times E_n(\mathtt{y})) \tag{4.5}$$

Let $o_{\mathtt{x}}$ be the object referenced by `x` in the concrete heap and $o_{\mathtt{y}}$ be the object referenced by `y`. The new edge from the object $o_{\mathtt{x}}$ to the object $o_{\mathtt{y}}$ can only add new paths from objects that could previously reach $o_{\mathtt{x}}$ to objects that were reachable from $o_{\mathtt{y}}$. In the reachability graph, the heap nodes $n_{\mathtt{x}} \in E_n(\mathtt{x})$ abstract $o_{\mathtt{x}}$ and the heap nodes $n_{\mathtt{y}} \in E_n(\mathtt{y})$ abstract $o_{\mathtt{y}}$.

The set of flagged heap nodes containing objects that could potentially reach $o_{\mathtt{x}}$ is given by the set of reachability states:

$$\Psi_{\mathtt{x}} = \mathcal{A}^N(n_{\mathtt{x}}) \cap \mathcal{A}^E(\langle \mathtt{x}, n_{\mathtt{x}}\rangle). \tag{4.6}$$

The reachability states of the objects reachable from $o_{\mathtt{y}}$ is

$$\Psi_{\mathtt{y}} = \mathcal{A}^E(\langle \mathtt{y}, n_{\mathtt{y}}\rangle). \tag{4.7}$$

We define $\cup_{\triangle}$ to compute the union of two reachability states. When two reachability states are combined, tuples with matching heap nodes merge arity values according to $+_{\triangle}$. We divide updating the reachability graph into the following steps:

1. **Construct the New Graph:** The analysis first constructs the new edge set as given by Equation 4.5.
2. **Update Reachability States of Downstream Heap Nodes:** The reachability of every object o' reachable from $o_{\mathtt{y}}$ is (i) abstracted by some $\psi_{\mathtt{y}} \in \Psi_{\mathtt{y}}$ and (ii) there exist a path of edges from the heap node that abstracts $o_{\mathtt{y}}$ to the heap node that abstracts o' in which each edge has $\psi_{\mathtt{y}}$ in its reachability state. The newly created edge can make the object o' reachable from the objects that can reach $o_{\mathtt{x}}$ — this set of objects is abstracted by some reachability state $\psi_{\mathtt{x}} \in \Psi_{\mathtt{x}}$. Therefore the new reachability state for o' should be $\psi_{\mathtt{y}} \cup_{\triangle} \psi_{\mathtt{x}}$. We capture this reachability change with the *change tuple set* $C_{n_{\mathtt{y}}} = \{\langle \psi_{\mathtt{y}}, \psi_{\mathtt{y}} \cup_{\triangle} \psi_{\mathtt{x}}\rangle \mid \psi_{\mathtt{y}} \in \Psi_{\mathtt{y}}, \psi_{\mathtt{x}} \in \Psi_{\mathtt{x}}\}$. Constraints 4.8 and 4.9 express the path constraint (ii). The analysis uses a fixed point to solve these constraints and then uses Equation 4.10 to update the reachability states of downstream nodes.

$$\Lambda^{\text{node}}(n_{\mathtt{y}}) \supseteq C_{n_{\mathtt{y}}} \tag{4.8}$$

$$\Lambda^{\text{node}}(n') \supseteq \{\langle \phi, \phi'\rangle \mid \langle \phi, \phi'\rangle \in \Lambda^{\text{node}}(n), \langle n, f, n'\rangle \in E, \phi \in \mathcal{A}^E(\langle n, f, n'\rangle)\} \tag{4.9}$$

$$\begin{aligned}\mathcal{A}^{N'}(n) = \{\phi' \mid \phi \in \mathcal{A}^N(n), \langle \phi, \phi'\rangle \in \Lambda^{\text{node}}(n)\} \cup \\ \{\phi \mid \phi \in \mathcal{A}^N(n), \nexists \phi'.\langle \phi, \phi'\rangle \in \Lambda^{\text{node}}(n)\}\end{aligned} \tag{4.10}$$

3. **Propagate Reachability from Downstream Nodes to Edges:** The analysis must propagate the reachability changes of objects back to any edge that models a reference that can reach the object. Constraint 4.11 ensures that edges contain reachability change tuples that capture the reachability changes in the incident objects. Constraint 4.12 ensures that the change set contains tuples to re-establish the transitive reachability state property.

$$\Lambda^{\text{edge}}(e) \supseteq \{\langle \phi, \phi' \rangle \mid \langle \phi, \phi' \rangle \in \Lambda^{\text{node}}(\text{dst}(e)), \phi \in \mathcal{A}^N(\text{dst}(e)), \phi \in \mathcal{A}^E(e)\} \quad (4.11)$$

$$\Lambda^{\text{edge}}(e) \supseteq \{\langle \phi, \phi' \rangle \mid \langle \phi, \phi' \rangle \in \Lambda^{\text{edge}}(e'), \phi \in \mathcal{A}^E(e), \text{dst}(e) = \text{src}(e')\} \quad (4.12)$$

4. **Propagate Reachability Changes Upstream of $o_{\mathtt{x}}$:** The reachability states of edges that model references that can reach $o_{\mathtt{x}}$ must be updated to reflect the objects they can now reach through the newly created edge. We define the change tuple set $C_{n_{\mathtt{x}}} = \{\langle \psi_{\mathtt{x}}, \psi_{\mathtt{y}} \cup_{\triangle} \psi_{\mathtt{x}} \rangle \mid \psi_{\mathtt{y}} \in \Psi_{\mathtt{y}}, \psi_{\mathtt{x}} \in \Psi_{\mathtt{x}}\}$ that updates the reachability states of edges that can reach o_x. Constraint 4.13 ensures that edges incident to the heap nodes that abstract $o_{\mathtt{x}}$ contain reachability change tuples that capture the reachability states of the newly reachable objects. Constraint 4.14 ensures that the change set contains tuples to re-establish the transitive reachability state property.

$$\Upsilon^{\text{edge}}(e) \supseteq \{\langle \phi, \phi' \rangle \mid \langle \phi, \phi' \rangle \in C_{n_{\mathtt{x}}}, \phi \in \mathcal{A}^E(e), \text{dst}(e) = n_{\mathtt{x}}\} \quad (4.13)$$

$$\Upsilon^{\text{edge}}(e) \supseteq \{\langle \phi, \phi' \rangle \mid \langle \phi, \phi' \rangle \in \Upsilon^{\text{edge}}(e'), \phi \in \mathcal{A}^E(e), \text{dst}(e) = \text{src}(e')\} \quad (4.14)$$

5. **Update Edge Reachability:** Finally, the analysis generates the reachability states for the edges in the new graph. Equation 4.15 computes the reachability states of all edges that existed before the store operation using the change tuple sets. Equation 4.16 computes the reachability for the newly created edges from the reachability of the edges for $\mathtt{y}$ with the constraint that every reachability state on the edge must be at least as large as the reachability state for the object $o_{\mathtt{x}}$. We define $\phi \subseteq_{\triangle} \phi'$ if $\forall \langle n, \mu \rangle \in \phi$ there exists a reachability tuple $\langle n, \mu' \rangle \in \phi'$ such that $\mu \sqsubseteq \mu'$.

$$\begin{aligned} \mathcal{A}^{E'}(e) = \mathcal{A}^E(e) \cup \{\phi' \mid \langle \phi, \phi' \rangle \in \Lambda^{\text{edge}}(e), \phi \in \mathcal{A}^E(e)\} \cup \\ \{\phi' \mid \langle \phi, \phi' \rangle \in \Upsilon^{\text{edge}}(e), \phi \in \mathcal{A}^E(e)\} \end{aligned} \quad (4.15)$$

$$\mathcal{A}^{E'}(\langle n_{\mathtt{x}}, \mathtt{f}, n_{\mathtt{y}} \rangle) \subseteq \{\phi \in \mathcal{A}^{E'}(\langle \mathtt{y}, n_{\mathtt{y}} \rangle) \mid \exists \phi' \in \mathcal{A}^{N'}(n_{\mathtt{x}}), \phi' \subseteq_{\triangle} \phi\} \quad (4.16)$$

Strong Updates While in general the analysis performs *weak updates* that simply add edges, under certain circumstances the analysis can perform *strong updates* that also remove edges to increase the precision of the results. Strong updates are possible under either of two conditions. First, when variable $\mathtt{x}$ is the only reference to a heap node $n_{\mathtt{x}}$. In this case we can destroy all reference edges from $n_{\mathtt{x}}$ with field $\mathtt{f}$ because no other variables can reach $n_{\mathtt{x}}$. Second, when the variable $\mathtt{x}$ references exactly one heap node $n_{\mathtt{x}}$ and $n_{\mathtt{x}}$ is a single-object heap node. When this is true $\mathtt{x}$ definitely refers to the object in $n_{\mathtt{x}}$ and the existing edges with field $\mathtt{f}$ from $n_{\mathtt{x}}$ can be removed.

For strong updates, the analysis first removes edges that the strong update eliminates. It then performs the normal transform as described in this section. Note that when strong updates remove edges, reachability of graph elements may change if the removed edges provided the reachability path. Therefore, reachability states may become imprecise. After a store transform with a strong update occurs, a global pruning step improves imprecise reachability states. Section 4.9 presents the global pruning step.

4.6 Element Load and Store Statements

Our analysis implements the standard pointer analysis treatment of arrays: Array elements are treated as a special field of array objects and always have weak store semantics. The analysis does not differentiate between different indices. This treatment can cause imprecision for operations such as vector removes that move a reference from one array element to another. Our implementation uses a special analysis to identify array store operations that acquire an object reference from an array and then create a reference from a different element of that array to the same object. Because the graph already accounts for this reachability, the effects of such stores can be safely omitted.

4.7 Return Statement

Return statements are of the form `return x` and return the object referenced by `x`. Each reachability graph has a special `Return` variable that is out of program scope. At a method return the transform assigns the `Return` variable to the references of variable `x`. We assume without loss of generality that the control flow graph has been modified to merge the control flow for all return statements.

4.8 Control Flow Join Points

To analyze a statement, the analysis first computes the join of the incoming reachability graphs. The operation for merging reachability graphs r_0 and r_1 into r_{out} follows below:

1. The set of variables for r_{out} is the set of live variables at $\bullet s$.

2. The set of heap nodes for r_{out} is the union of the heap nodes in the input graphs. The union of the reachability states is taken, $\mathcal{A}^N_{\text{out}}(n) = \mathcal{A}^N_0(n) \cup \mathcal{A}^N_1(n)$.

3. The set of reference edges for r_{out} is the union of the reference edges of the input graphs. Recall that reference edges are unique in a reachability graph with respect to source, field, and destination. For a reference edge e, $\mathcal{A}^E_{\text{out}}(e) = \mathcal{A}^E_0(e) \cup \mathcal{A}^E_1(e)$.

4.9 Global Pruning

When strong updates remove edges, the reachability states may become imprecise. The call site transform given in Section 5 can also introduce imprecise reachability states. Our analysis includes a global pruning step that uses global reachability constraints to prune imprecise reachability states to improve the precision of the analysis results. The intuition behind global pruning is that multiple abstract states can correspond to the same set of concrete heaps, and the global pruning step generates an equivalent abstraction that locally has more precise reachability states.

Global Reachability Constraints Reachability information must satisfy two reachability constraints that follow from the discussion in Section 3.3.

- **Node reachability constraint:** For each node n, $\forall \phi \in \mathcal{A}^N(n)$, $\forall \langle n', \mu \rangle \in \phi$, if $\mu \in \{1, \texttt{MANY}\}$ then there must exist a set of edges $e_1, \ldots, e_m$ such that $\phi \in \mathcal{A}^E(e_i)$ for all $1 \leq i \leq m$ and the set of edges $e_1, \ldots, e_m$ form a path through the reachability graph from n' to n.
- **Edge reachability constraint:** For each edge e, $\forall \phi \in \mathcal{A}^E(e)$ there exists $n \in N$ and $e_1, \ldots, e_m \in E$ such that $\phi \in \mathcal{A}^N(n)$; $\phi \in \mathcal{A}^E(e_i)$ for all $1 \leq i \leq m$; and the set of edges $e_1, \ldots, e_m$ form a path through the reachability graph from e to n.

The first phase of the algorithm generates a reachability graph with the most precise set of reachability states for the nodes. The second phase of the algorithm generates the most precise set of reachability states for the edges.

1. Improve the precision of the node reachability states: The algorithm first uses the node reachability constraint to prune the reachability states of nodes. This phase uses the existing $\mathcal{A}^E$ to prune reachability tuples from imprecise reachability states to generate a more precise $\mathcal{A}^{N'}$ from the previous $\mathcal{A}^N$. The algorithm iterates through each flagged node n_f. The function $\mathcal{A}^{\mathcal{E}}_f(e)$ maps the edge $e \in E$ to the set of reachability states Φ for which each $\phi \in \Phi$ (1) includes a non-zero arity reachability tuple with the node n_f and (2) there exist a path from n_f to e for which every edge along the path contains ϕ in its set of reachability states. We compute $\mathcal{A}^{\mathcal{E}}_f$ using a fixed-point algorithm on the following two constraints:

$$\forall e \in E_e(n_f), \mathcal{A}^{\mathcal{E}}_f(e) \supseteq \mathcal{A}^E(e), \tag{4.17}$$

$$\forall e \in E, e' \in E_e(\mathrm{dst}(e)), \mathcal{A}^{\mathcal{E}}_f(e') \supseteq \mathcal{A}^E(e') \cap \mathcal{A}^{\mathcal{E}}_f(e). \tag{4.18}$$

For each node n and each reachability state $\phi \in \mathcal{A}^N(n)$ the analysis shortens ϕ to remove tuples n_f or n^*_f to generate a new reachability state ϕ' if ϕ does not appear in $\mathcal{A}^{\mathcal{E}}_f(e)$ of any edge e incident to n. This step does not prune n_f or n^*_f from the reachability states of flagged nodes n_f. The analysis then propagates these changes to $\mathcal{A}^E$ of the upstream edges using the same propagation procedure described by Equations 4.11 and 4.12 to generate $\mathcal{A}^E_r$.

2. Improve the precision of the edge reachability states: The algorithm next uses the pruned node reachability states in $\mathcal{A}^{N'}$ and $\mathcal{A}^E_r$ to generate a more precise $\mathcal{A}^{E'}$. The intuition is that an edge can only have a given reachability state if there exists a path from that edge to a node with that reachability state such that all edges along the path contain the reachability state. The analysis starts from every heap node n and propagates the reachability states of $\mathcal{A}^N(n)$ backwards over reference edges. The analysis initializes $\mathcal{A}^{E'} = \{\mathcal{A}^E_r(e) \cap \mathcal{A}^{N'}(n) \mid \forall e \in E, n = \mathrm{dst}(e)\}$. The analysis then propagates reachability information backwards to satisfy the constraint: $\mathcal{A}^{E'}(e) \supseteq \mathcal{A}^E_r(e) \cap \mathcal{A}^{E'}(e')$ for all $e' \in E_e(\mathrm{dst}(e))$. The propagation continues until a fixed-point is reached.

4.10 Static Fields

We have omitted analysis of static fields or globals. We assume that the preprocessing stage creates a special global object that contains all of the static fields and passes it to every call site. Through this semantics-preserving program transformation, static field store and load statements become normal store and load statements, respectively.

5 Interprocedural Analysis

The interprocedural analysis adds a call site transform to the intraprocedural analysis. It uses a standard fixed-point algorithm and begins by analyzing the main method. Our analysis processes each method using one context that summarizes the heaps for all call sites. A summary of the transform follows:

1. Compute the portion of the heap that is reachable from the callee.

2. Rewrite reachability states to abstract flagged heap nodes that are not in the callee heap with special out-of-context heap nodes.

3. Merge this portion of the heap with the callee's current initial graph. If the graph changes, schedule the callee for reanalysis.
4. Use the callee reachable portion of the heap to specialize the callee's current analysis result.
5. Remove the callee reachable portion of the heap and splice in the specialized callee results.
6. Merge nodes such that each allocation site has at most one summary heap node and one single object heap node.
7. Call the global pruning step introduced in Section 4.9 to improve the precision of the caller reachability graph.

5.1 Compute Callee Context Subgraph

For each call site, the analysis computes the subgraph $G_{\text{sub}} \subseteq G$ that is reachable from the call site's arguments. For each incoming edge $\langle n, f, n' \rangle \in E$ into G_{sub} where $n \notin G_{\text{sub}}$ and $n' \in G_{\text{sub}}$, the analysis generates a new *placeholder node* n_p and a new edge $e' = \langle n_p, \mathcal{R}(n') \rangle$ where $\mathcal{A}^E(e') = \mathcal{A}^E(e)$. The placeholder node n_p serves as a proxy flagged node for all reachability nodes in $\mathcal{A}^N(n)$ during global pruning. For each incoming edge $\langle v, n' \rangle \in E$ into G_{sub} where $n' \in G_{\text{sub}}$, the analysis generates a new placeholder variable v_p and *placeholder edge* $e_p = \langle v_p, n' \rangle$ where $\mathcal{A}^E(e_p) = \mathcal{A}^E(e)$.

5.2 Out-of-Context Reachability

Summarization presents a problem for out-of-context flagged heap nodes that appear in reachability states of in-context heap nodes. The interprocedural analysis uses placeholder flagged nodes to rewrite out-of-context flag heap nodes in reachability states. Each heap node n_f that appears in $\mathcal{A}^N$ of a placeholder node is (1) outside of the graph G_{sub} and (2) abstracts objects that can potentially reach objects abstracted by the subgraph G_{sub}. The analysis replaces all such nodes in all in-context reachability states with special out-of-context heap nodes for the allocation site. There can be up to two out-of-context heap nodes per an allocation site: one is a summary node and one abstracts the most recently allocated object from the allocation site. The purpose of these heap nodes is to allow the analysis of the callee context to age in-context, single-object heap nodes without affecting out-of-context flagged heap nodes that can reach objects in the callee's reachability graph.

The analysis maps (1) the newest single-object heap node for an allocation site that is out of the callee's context to the special single-object out-of-context heap node and (2) all other nodes for the allocation site that are out of the callee's context for the heap node to the special summary out-of-context heap node. The analysis stores this mapping for use in the splicing step. These special out-of-context nodes serve as placeholders to track changes to the reachability of out-of-context edges.

5.3 Merge Graphs

The analysis merges the subgraphs from all calling contexts using the join operation from Section 4.8.

5.4 Predicates

The interprocedural analysis extends all nodes, edges, and reachability states with a set of predicates. These predicates are included to prevent nodes and edges from leaking

from one call site to another and are used (1) to determine whether the given node, edge, or reachability state in the callee graph should be mapped to the caller and (2) to correctly propagate reachability states in the caller. Predicates are comprised of the following atomic predicates, which can be combined with logical disjunction (or):

- Edge e exists with reachability state ϕ in G_{sub} of the caller
- Node n exists with reachability state ϕ in G_{sub} of the caller
- Edge e exists in G_{sub} of caller
- Node n exists in G_{sub} of caller
- *true*

The caller analysis begins by initializing the predicates for all nodes, edges, and reachability states to tautologies. For example, the initial predicate for a node n is that the node n exists in the caller — this prevents node n from leaking from one call site to another. The initial predicate for a reachability state ϕ on node n is that node n exists in G_{sub} of the caller with reachability state ϕ.

Store operations can change the reachability states of both edges and heap nodes. When the propagation of a change set creates a new reachability state on a node or an edge, the new state inherits the predicate from the previous state on the node or edge, respectively. Object allocation operations can merge single-object heap nodes into the corresponding summary node. In this case, predicates for the nodes are or'ed together. Likewise, if the operation causes two edges to be merged, their predicates are also or'ed together. Duplicated reachability states may also be merged and their predicates are or'ed together. Predicates can be ignore for the load statement and the computation of change sets as the predicates produced by those operations do not appear on caller visible graph elements.

Newly created nodes or edges are assigned the *true* predicate.

5.5 Specializing the Graph

The algorithm uses G_{sub} to specialize the callee heap reachability graph G_{callee}. The analysis makes a copy of the heap reachability graph G_{callee}. It then prunes all elements of the graph whose predicates are not satisfied by the caller subgraph G_{sub}. The callee predicates of each heap element in G_{callee} are replaced with the caller predicate for the heap element in G_{sub} that satisfied the callee predicate.

If a reachability state contains out-of-context heap nodes, then the analysis uses the stored mapping to translate the out-of-context heap nodes to caller heap nodes. The stored mapping may map multiple heap nodes to the same out-of-context summary heap node. If the arity of the reachability tuple for an out-of-context heap node was `1`, then the analysis generates all permutations of the reachability state using the stored mapping from Section 5.2. If the arity was `MANY`, the analysis replaces the reachability tuple with a set of reachability tuples that contains one tuple for each heap node that mapped to the out-of-context summary node and that tuple has arity `MANY`.

5.6 Splice in Subgraph

This step splices the physical graphs together. The placeholder nodes are used to splice references from the caller graph to the callee graph. The placeholder edges are used to splice caller edges into the callee graph.

Finally, the reachability changes are propagated back into the out-of-context heap nodes of the caller reachable portion of the reachability graph. The analysis uses predicates to match the reachability states on the original edges from the out-of-context portion of the caller graph into G_{sub}. The analysis generates a change set for each edge that tracks the out-of-context reachability changes made by the callee. It then solves constraints of the same form as Constraints 4.11 and 4.12 to propagate these changes to upstream portions of the caller graph.

5.7 Merging Heap Nodes

At this point, the graph may have more than one single object heap node or summary heap node for a given allocation site. The algorithm next merges all but the newest single object heap node into the summary heap node. It rewrites all tokens in all reachability states to reflect this merge, and then updates the arities.

5.8 Global Pruning

Finally, the analysis calls the global pruning algorithm to remove imprecision potentially caused by our treatment of reachability from out-of-context heap nodes.

6 Evaluation

We have implemented the analysis in our compiler and analyzed both out-of-order Java applications [1] and Bamboo applications [2]. In out-of-order Java the developer annotates code blocks as reorderable to decouple these blocks from the parent thread of execution. OoOJava uses disjointness analysis combined with other analyses to generate a set of runtime dependence checks that guarantee that the parallel execution preserves the behavior of the original sequential code — therefore the annotations do not affect the correctness of the program.

Bamboo extends Java with task extensions designed for parallel programming and uses static analysis to prevent data races between tasks. Collectively, Bamboo can be viewed as a generalization of the worker thread pattern. The Bamboo task scheduler uses the disjointness analysis results to generate a locking strategy that ensures that the application does not simultaneously execute tasks that may update the same object.

Bamboo programs are a natural choice for benchmarks as the tasks' parameter objects are typically intended to be disjoint and therefore provide a source of programs with flagged allocation sites. Analyzing tasks only means that task parameter objects may be live from references in the scheduler, and therefore the first strong update condition (see Section 4.5) does not apply to those heap nodes. The analysis and benchmarks are available at `http://demsky.eecs.uci.edu/compiler.php`.

6.1 Benchmarks

We analyzed and parallelized the following three out-of-order Java benchmarks: KMeans, a data clustering benchmark from STAMP [3]; RayTracer, a ray tracer from Java Grande [4]; and Power, a power pricing benchmark from JOlden [5].

We analyzed and parallelized the following six Bamboo benchmarks ported from the Java Grande benchmark suite[4]: MonteCarlo, a Monte Carlo simulation; Series, a Fourier series computation; and Fractal, which computes a Mandelbrot set. We also analyzed and parallelized: KMeans; Tracking, a vision benchmark from the San Diego Vision Benchmark Suite [6]; and FilterBank, a multi-channel filter from StreamIt [7].

Benchmark	Sharing	Time (s)	Lines	Speedup
RayTracer	0	54.3	3,258	7.8×
KMeans-OoOJava	0	26.2	3,541	5.8×
Power	0	11.2	2,275	6.0×

Fig. 4. Out-of-order Java Results (8 cores)

Benchmark	Sharing	Time (s)	Lines	Speedup Bamboo	Speedup C
Tracking	0	19.3	5,218	26.2×	26.1×
KMeans-Bamboo	2	3.7	2,893	38.9×	35.1×
MonteCarlo	0	2.1	3,638	36.2×	34.2×
FilterBank	0	0.1	1,555	37.5×	37.5×
Series	0	0.2	1,639	61.2×	57.6×
Fractal	1	0.1	1,568	61.6×	58.0×

Fig. 5. Bamboo Results (62 cores)

Benchmark	Sharing	Time (s)	Lines
Bank	0	4.3	2,059
Chat	3	3.7	1,744
jHTTPp2	0	4.2	2,679
MultiGame	10	180.9	3,099
Spider	0	10.1	1,827
TicTacToe	0	1.6	1,766
WebCommerce	0	11.5	2,090
WebPortal	0	2.7	2,213

Fig. 6. Analysis Only Bamboo Results

We also analyzed several additional Bamboo applications. jHTTPp2 is an open source proxy server available from `http://jhttp2.sourceforge.net/`. Bank implements a simple banking server. WebPortal assembles information from online data sources into a web page. Spider crawls the web. TicTacToe is a tic-tac-toe game server. WebCommerce is a web application server. MultiGame is a multiplayer game.

6.2 Disjoint Reachability Analysis Results

We ran 17 benchmarks through the analysis on a 2.27 GHz Xeon. Figures 4, 5, and 6 present the results and the time columns show the analysis time. Most benchmarks took only a few seconds to analyze. The benchmarks ranged from 5,218 to 1,555 lines of code, with an average of 67.7 methods per benchmark.

Disjoint reachability analysis identified a total of 16 possible sharing classes between flagged heap nodes over 17 benchmarks. A sharing class is a set of flagged heap nodes that the analysis identified as not definitely disjoint. Therefore, runtime objects that map to these nodes may have sharing during execution. A developer might examine the results for particular program points to learn about possible sharing.

The other benchmarks were reported to have disjoint heaps reachable from flagged nodes. We checked that the analysis reported all sharing by manual inspection. The amount of sharing is relatively small as we only checked the sharing properties required to verify that the intended parallelization was safe. The analysis detected real sharing in MultiGame between player objects that prevents any substantial parallelization.

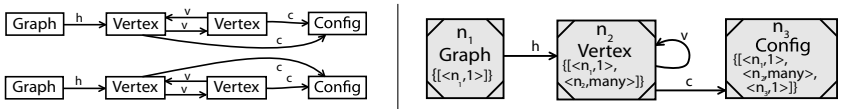
We used disjoint analysis to explore whether main feeders in Power were disjoint. To explore this question, we manually flagged the allocation site. Disjoint analysis confirmed that the power distribution networks reachable from main feeders were disjoint.

6.3 Parallelization Speedups

Three of the benchmarks were automatically parallelized using out-of-order Java for a 2.27GHz 8-core Intel Xeon. The analysis results for RayTracer, KMeans, and Power were used to parallelize the benchmarks. Figure 4 presents the speedups for these benchmarks on 8 cores. The speedups are relative to the single-threaded Java versions compiled to C using the same compiler. The OoOJava version of KMeans parses an input file while the Bamboo version hard codes the input, causing the difference in scalability.

Six of the benchmarks were automatically parallelized using Bamboo for Tilera's TILEPro64 processor. Without disjointness analysis, the benchmarks must be executed sequentially. The analysis results for MonteCarlo, KMeans, FilterBank, Fractal, Series and Tracking were used to automatically generate parallel implementations. Figure 5 presents the speedup for a 62-core[3] Bamboo binary parallelized from analysis results

[3] Two cores in the parallelized binary are reserved for PCI device support.



(a) Concrete heap (b) Reachability graph

Fig. 7. An example concrete heap with a reachability graph

relative to a 1-core Bamboo binary and a 1-core C binary. We omit results for the server benchmarks due to both the lack of network support in our runtime (the current runtime executes directly on the hardware with no underlying O/S) and the difficult of accurately benchmarking server applications.

The speedups are significant. While disjoint reachability analysis is critical for generating correct parallel implementations, the benchmarks contain a large degree of parallelism and other applications may not see the same magnitude of speedup.

7 Related Work

We discuss related work in shape analysis, alias analysis, pointer analysis, logics, static analysis, and type systems.

7.1 Shape Analysis

Disjoint reachability analysis discovers properties that are related to but different from those discovered by shape analysis [8–12]. Shape analysis, in general, discovers local properties of heap references and from these properties infers a rich set of derived properties including reachability and disjointness. Where shape analysis can find properties that arise from local invariants, disjoint reachability analysis can find the relative disjointness and reachability properties for any pair of objects. Disjoint reachability analysis complements shape analysis by discovering disjoint reachability properties for arbitrarily complex structures. Calcagno et al. present a shape analysis that focuses on discovering different heap properties [13].

We motivate our discussion of shape analysis with a concrete heap example. Figure 7(a) illustrates a simple concrete heap where a `Graph` can reach several `Vertex` objects that all point to a graph-local `Config` object. We expect that many real programs construct data structures with similar sharing patterns to this example. A possible reachability graph in Figure 7(b) contains enough information to show that `Config` and `Vertex` objects are reachable from at most one `Graph` object. Some shape analyses [8, 9] focus on local shape properties (does a tree stay a tree?) and understandably lose precision with the above example or the singleton design pattern. Singleton design patterns include references to globally shared objects. Some parallelizable phases may not even access the shared object, but the presence of a shared object will cause problems for many shape analysis. Our analysis can infer that operations on different graphs that access both `Vertex` and `Config` objects may execute in parallel. Note that this result is independent of the relative shape of Vertex objects in the heap.

Marron et al. extend the shape approach for more general heaps with edge-sharing analysis [12, 14]. Their analysis can discover that the `Vertex` objects from different `Graph` objects are disjoint. However, their edge-sharing abstraction is localized and therefore cannot discover that `Config` objects are not shared between graphs.

TVLA [10] is a framework for developing shape analysis. Disjointness properties can be written as instrumentation predicates in TVLA, but the system will evaluate them

using the default update rule, providing acceptable results only for trivial examples. To maintain precision, update rules for the disjointness predicates must be supplied, a task that we expect is equivalent in difficulty to disjoint reachability analysis. While TVLA contains reachability predicate update rules, these cannot capture that an object is reachable from exactly one member of a summarized region. Furthermore, it appears that TVLA does not scale to the size of our benchmarks.

Separation logic can express that formulas hold on disjoint heap regions [15]. Distefano et al. propose a shape analysis for linked lists based on separation logic [16]. Raza et al. extend separation logic with labels that relate assertions at one program point to another in an effort to identify statements that can be parallelized [17]. These shape analysis based on separation logic are at an early stage and cannot extract disjoint reachability properties for our examples.

7.2 Alias and Pointer Analysis

Alias analysis [18, 19] and pointer analysis [20, 21], like disjoint reachability analysis, analyzes source code to discover heap referencing properties. Conditional must not aliasing analysis by Naik and Aiken [22] is similar, but their type system names objects by allocation site and loop iteration. Unlike our analysis, their approach cannot maintain disjointness properties for mutation outside of the allocating loop. Chatterjee et al. describe a modular points-to analysis that does not extract disjoint reachability properties, but introduces an alternative approach to abstracting caller contexts [23].

7.3 Other Analysis and Type Systems

Sharing analysis [24] computes sharing between variables. Sharing analysis could not determine disjoint reachability properties for the example in Figure 1 of our paper as it would lose information about the relative disjointness of graphs in the array.

Connection analysis discovers which heap-directed pointers may reach a common data structure [25]. There are a finite number of pointers in a program which implies that connection analysis can only maintain a finite number of disjoint relations. For example, connection analysis cannot determine that all of the `Graph` objects in our paper's example reference mutually disjoint sets of `Vertex` objects.

Ownership type systems have been developed to restrict aliasing of heap data structures [26, 27]. While disjoint reachability analysis can infer similar properties, it does not require user annotations.

Program verification [28] can also discover reachability properties. Their work focuses on reachability from named nodes rather than arbitrary heap objects.

8 Conclusion

If a compiler can determine that code blocks perform memory accesses that do not conflict, it can safely parallelize them. Traditional pointer analyses have difficulty reasoning about reachability from objects that are represented by the same node. We present disjoint reachability analysis, a new analysis for extracting reachability properties from code. The analysis uses a reachability abstraction to maintain precise reachability information even for multiple objects from the same allocation site. We have implemented the analysis and analyzed 17 benchmark programs. The analysis results enabled parallelization of several benchmarks that achieved significant performance improvements.

References

1. Jenista, J.C., Eom, Y., Demsky, B.: OoOJava: An out-of-order approach to parallel programming. In: Second USENIX Workshop on Hot Topics in Parallelism. (2010)
2. Zhou, J., Demsky, B.: Bamboo: A data-centric, object-oriented approach to multi-core software. In: Proceedings of the 2010 Conference on Programming Language Design and Implementation. (2010)
3. C. Minh et al: STAMP: Stanford transactional applications for multi-processing. In: Proceedings of the 2008 IEEE International Symposium on Workload Characterization
4. Smith, L.A., Bull, J.M., Obdrzalek, J.: A parallel Java Grande benchmark suite. In: SC2001
5. Cahoon, B., McKinley, K.S.: Data flow analysis for software prefetching linked data structures in Java. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques. (2001)
6. Venkata, S.K., Ahn, I., Jeon, D., Gupta, A., Louie, C., Garcia, S., Belongie, S., Taylor, M.B.: SD-VBS: The San Diego vision benchmark suite. In: Proceedings of the 2009 IEEE International Symposium on Workload Characterization. (2009)
7. M. Gordon et al: A stream compiler for communication-exposed architectures. In: Proceedings of the 2002 International Conference on Architectural Support for Programming Languages and Operating Systems
8. Chase, D.R., Wegman, M., Zadeck, F.K.: Analysis of pointers and structures. In: Proceedings of the 1990 Conference on Programming Language Design and Implementation
9. Ghiya, R., Hendren, L.J.: Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In: Proceedings of the 1996 Symposium on Principles of Programming Languages
10. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems (2002)
11. McPeak, S., Necula, G.C.: Data structure specifications via local equality axioms. In: Proceedings of the 2005 International Conference on Computer Aided Verification
12. M. Marron et al: A static heap analysis for shape and connectivity: Unified memory analysis: The base framework. In: Proceedings of the 2006 Workshop on Languages and Compilers for Parallel Computing
13. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: Proceedings of the 2009 Symposium on Principles of Programming Languages
14. Marron, M., Méndez-Lojo, M., Hermenegildo, M., Stefanovic, D., Kapur, D.: Sharing analysis of arrays, collections, and recursive structures. In: Proceedings of the 2008 Workshop on Program Analysis for Software Tools and Engineering
15. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of the 2002 IEEE Symposium on Logic in Computer Science
16. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. LNCS **3920** (2006) 287–302
17. Raza, M., Calcagno, C., Gardner, P.: Automatic parallelization with separation logic. LNCS **5502** (2009) 348–362
18. Diwan, A., McKinley, K.S., Moss, J.E.B.: Type-based alias analysis. In: Proceedings of the 1998 Conference on Programming Language Design and Implementation
19. Ruf, E.: Partitioning dataflow analyses using types. In: Proceedings of the 1997 Symposium on Principles of Programming Languages
20. Shapiro, M., Horwitz, S.: Fast and accurate flow-insensitive points-to analysis. In: Proceedings of the 1997 Symposium on Principles of Programming Languages

21. Landi, W., Ryder, B.G., Zhang, S.: Interprocedural modification side effect analysis with pointer aliasing. In: Proceedings of the 1993 Conference on Programming Language Design and Implementation
22. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: Proceedings of the 2007 Symposium on Principles of Programming Languages
23. Chatterjee, R., Ryder, B.G., Landi, W.A.: Relevant context inference. In: Proceedings of the 1999 Symposium on Principles of Programming Languages
24. Méndez-Lojo, M., Hermenegildo, M.V.: Precise set sharing analysis for Java-style programs. In: Proceedings of the 2008 International Conference on Verification, Model Checking, and Abstract Interpretation
25. Ghiya, R., Hendren, L.J.: Connection analysis: A practical interprocedural heap analysis for C. International Journal of Parallel Programming (1996)
26. Clarke, D.G., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: Proceedings of the 2002 International Conference on Object-Oriented Programming, Systems, Languages and Applications
27. Heine, D.L., Lam, M.S.: A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In: Proceedings of the 2003 Conference on Programming Language Design and Implementation
28. Yorsh, G., Rabinovich, A., Sagiv, M., Meyer, A., Bouajjani, A.: A logic of reachable patterns in linked data-structures. The Journal of Logic and Algebraic Programming (2007)

A Design Decisions

Many heap analyses that attempt to extract more precise properties than pointer analysis attempt to extract shape properties. In general, extracting shape properties has proved difficult. Our analysis is designed to carefully avoid the difficult problem of reasoning about data structure shapes and to instead extract disjoint reachability properties.

We note that pointer analysis in some circumstances can extract reachability information for a set of statically named data structures. Disjoint reachability analysis was designed to maintain reachability annotations for heap nodes and therefore can reason about the mutual disjoint reachability of an unbounded number of data structures.

We included an interprocedural analysis because reachability is a transitive property. Therefore, skipping method calls would likely introduce imprecision that would propagate throughout the graph.

B Semantics for Intraprocedural Analysis

Define the concrete heap $H = \langle O, R \rangle$ as a set of objects $o \in O$ and a set of references $r \in R \subseteq O \times \{\texttt{Fields}\} \times O$. We assume a straightforward collecting semantics for the statements in the control flow graph that are relevant to our analysis. The collecting semantics would record the set of concrete heaps that a given statement operates on.

The concrete domain for the abstraction function is a set of concrete heaps $h \in \mathcal{P}(H)$. The abstract domain is defined in Section 3.2. The abstract state is given by the tuple $\langle E, \mathcal{A}^N, \mathcal{A}^E \rangle$, where E is the set of edges, $\mathcal{A}^N$ is the mapping from nodes to their sets of reachability states, and $\mathcal{A}^E$ is the mapping from edges to their sets of reachability states. We next define the lattice for the abstract domain. The bottom element has the empty set of edges E and empty reachability information for both the nodes $\mathcal{A}^N$ and the edges $\mathcal{A}^E$. The top element for the lattice has (1) all the edges in E that are allowed by type constraints between all reachability nodes, (2) each heap node n has tuples in

$\mathcal{A}^N$ for the powerset of all heap nodes that are allowed by types to reach n, and (3) each edge $\langle n, f, n' \rangle \in E$ has the powerset of the maximal set of tuples in $\mathcal{A}^E$ that are allowed by type constraints.

We next define the partial order for the reachability graph lattice. Equation B.1 defines the partial order. The definition for the $\subseteq_\triangle$ relation between reachability states is given in the Update Edge Reachability step of Section 4.5.

$$\langle E, \mathcal{A}^N, \mathcal{A}^E \rangle \sqsubseteq_A \langle E', \mathcal{A}^{N'}, \mathcal{A}^{E'} \rangle \text{ iff } E \subseteq E' \wedge \langle \mathcal{A}^N, \mathcal{A}^E \rangle \sqsubseteq \langle \mathcal{A}^{N'}, \mathcal{A}^{E'} \rangle \tag{B.1}$$

$$\begin{aligned} \langle \mathcal{A}^N, \mathcal{A}^E \rangle \sqsubseteq \langle \mathcal{A}^{N'}, \mathcal{A}^{E'} \rangle \text{ iff } & \forall n \in N, \forall \phi \in \mathcal{A}^N(n), \exists \phi' \in \mathcal{A}^{N'}(n), \\ & \phi \subseteq_\triangle \phi' \wedge \big(\forall \langle n_1, f_1, n_2 \rangle, ..., \langle n_k, f_k, n \rangle \in E, \\ & \phi \in \mathcal{A}^E(\langle n_1, f_1, n_2 \rangle) \cap ... \cap \mathcal{A}^E(\langle n_k, f_k, n \rangle) \Rightarrow \\ & \phi' \in \mathcal{A}^{E'}(\langle n_1, f_2, n_2 \rangle) \cap ... \cap \mathcal{A}^{E'}(\langle n_k, f_k, n \rangle) \big) \end{aligned} \tag{B.2}$$

The join operation ($\langle E_1, \mathcal{A}^N{}_1, \mathcal{A}^E{}_1 \rangle \sqcup \langle E_2, \mathcal{A}^N{}_2, \mathcal{A}^E{}_2 \rangle$) on the heap reachability graph lattice simply takes the set unions of the individual components: $\langle E_1 \cup E_2, \mathcal{A}^N{}_1 \cup \mathcal{A}^N{}_2, \mathcal{A}^E{}_1 \cup \mathcal{A}^E{}_2 \rangle$.

We next define several helper functions. Equation B.3 defines the meaning of the statement that object o is reachable from the object o' in the concrete heap R. We define the object abstraction function $\texttt{rgn}(o)$ to return the single object heap node for o's allocation site if o is the most recently allocated object and the allocation site's summary node otherwise. Equation B.4 returns the number of objects abstracted by heap node n_f that can reach the object o. Equation B.5 abstracts the natural numbers into one of three arities. Equation B.6 computes the abstract reachability state for object o in the concrete heap $\langle O, R \rangle$.

$$\begin{aligned} \texttt{rch}(o', o, R) = \exists f, o_1, f_1, ..., o_l, f_l . & \langle o', f, o_1 \rangle, ..., \langle o_i, f_i, o_{i+1} \rangle, ..., \\ & \langle o_l, f_l, o \rangle \in R \end{aligned} \tag{B.3}$$

$$\texttt{count}(o, O, R, n_f) = \mid \{ o' \mid \forall o' \in O . \texttt{rgn}(o') = n_f, \texttt{rch}(o', o, R) \} \mid \tag{B.4}$$

$$\texttt{abst}(n) = \begin{cases} \texttt{0} & n = 0 \\ \texttt{1} & n = 1 \\ \texttt{MANY} & \text{otherwise} \end{cases} \tag{B.5}$$

$$\phi(o, O, R) = \{ \langle n_f, \texttt{abst}(\texttt{count}(o, O, R, n_f)) \rangle \mid n_f \in N_F \} \tag{B.6}$$

We next define abstraction functions that return the most precise reachability graph for the set of concrete heaps $h \subseteq \mathcal{P}(H)$. We use the standard subset partial ordering relation for our concrete domain of sets of concrete heaps. Equation B.7 generates the edge abstraction, Equation B.8 generate the reachability state abstraction for each node, and Equation B.9 generates the reachability state abstraction for each edge. Note that from the form of the definition of the abstraction function, we can see that it is monotonic. We mechanically synthesize a concretization function $\gamma(\langle E, \mathcal{A}^N, \mathcal{A}^E \rangle) = \sqcup \{ h \mid \alpha(h) \sqsubset \langle E, \mathcal{A}^N, \mathcal{A}^E \rangle \}$ to create a Galois connection. The pair α and γ do not form a Galois insertion as two abstract reachability graphs can have the exact same set of concretizations. The global pruning algorithm addresses the practical effects on analysis

precision of this issue by converting abstract reachability graphs into equivalent graphs that contain locally more precise reachability states.

$$\alpha_E(h) = \{\langle \texttt{rgn}(o), f, \texttt{rgn}(o')\rangle \mid \forall\langle o, f, o'\rangle \in R, \forall\langle O, R\rangle \in h\} \quad \text{(B.7)}$$

$$\alpha_{\mathcal{A}^N}(h) = \{\langle \texttt{rgn}(o), \phi(o, O, R)\rangle \mid \forall o \in O, \forall\langle O, R\rangle \in h\} \quad \text{(B.8)}$$

$$\alpha_{\mathcal{A}^E}(h) = \{\langle\langle \texttt{rgn}(o'), f, \texttt{rgn}(o'')\rangle, \phi(o, O, R)\rangle \mid \forall o \in O, \forall\langle o', f, o''\rangle \in R, \forall\langle O, R\rangle \in h.\texttt{rch}(o'', o, R)\} \quad \text{(B.9)}$$

C Termination

Termination of the analysis is straightforward. Reachability graphs form a lattice, and for a given set of allocation sites the lattice is of finite height. All transfer functions in the analysis are monotonic except stores with strong updates and method calls. With a simple modification to enforce monotonicity the analysis will terminate.

Our approach to enforcing monotonicity is to store the latest reachability graph result for every back edge and program point after a method call. The fixed point interprocedural algorithm takes the join of its normal result with these graphs to ensure the local result becomes no smaller.

D Soundness of the Core Intraprocedural Analysis

In this section, we outline the soundness of the core intraprocedural analysis. For all soundness lemmas, we argue $(\alpha \circ f)(h) \sqsubseteq_A (f^\# \circ \alpha)(h)$, where f represents the concrete operation and $f^\#$ is the corresponding transfer function on the abstract domain, to show soundness.

Lemma 1 (Soundness of Copy Statement Transfer Function). *The transfer function for the copy statement* `x=y` *is sound with respect to the concrete copy operation.*

Proof Sketch: The soundness of the transfer function for the copy statement `x=y` is straightforward. After the execution of the copy statement on the concrete heap, the variable `x` references the object that `y` referenced before the statement. We note that applying the abstraction function after the concrete copy statement yields the exact same abstract reachability graph as applying the abstraction function followed by the transfer function for the copy statement, therefore the copy transfer function is sound.

Lemma 2 (Soundness of Load Statement Transfer Function). *The transfer function for the load statement* `x=y.f` *is sound with respect to the concrete load operation.*

Proof Sketch: The soundness of the transfer function for the load statement `x=y.f` is also relatively straightforward. After the execution of the load statement on the concrete heap, the variable `x` references the object referenced by the `f` field of the object referenced by `y`. After abstraction, the edge for `x` would reference the same objects as the `f` field of the objects referenced by `y` and have the same reachability set.

The soundness of the edge set transform follows from the definition of α_E — all objects that `y.f` could possibly reference are included in the set $E_n(\texttt{y}, \texttt{f})$. Therefore, applying the abstraction function followed by removing the previous edges for `x` and

adding the set of edges $\{\mathtt{x}\} \times E_n(y)$ gives an E set that contains all of the edges generated by applying the transfer function and then abstraction function.

From the definition of $\alpha_{\mathcal{A}^E}$ we can determine that for each n that could abstract the object referenced by y and each corresponding n' that could abstract the object referenced by y.f, that the reference y.f could only reach objects with reachability states included in the set $\mathcal{A}^E(\langle \mathtt{y}, n\rangle) \cap \mathcal{A}^E(\langle n, \mathtt{f}, n'\rangle)$. Note the subtle point that the correctness of the intersection operation follows from the edge reachability aspect of the abstraction function definition (and not from the lattice ordering) — there must exist a path through the y reference and y.f to any objects that can be reached by the new x and by the abstraction function both y and y.f will include the reachability states of those objects. Therefore, the application of the abstraction function followed by the transfer function generates a set of reachability states for edges of y that include all of the reachability states generated by applying the concrete load statement followed by the abstraction function.

Lemma 3 (Soundness of Allocation Statement Transfer Function). *The transfer function for the allocation statement* x=new *is sound with respect to the concrete allocation operation.*

Proof Sketch: The transfer function for the allocation statement is similarly straightforward. The execution of the allocation statement on the concrete heap followed by the abstraction function yields an abstract reachability graph in which the previous newest allocated object at the site is now mapped to the summary node. The allocation statement transfer function applied to the abstraction function yields the exact same reachability graph and therefore the transfer function is sound.

If the allocation site is flagged, the new heap node has a single reachability state that contains a single reachability token with its own heap node and the arity 1. The variable edge contains the same set of reachability states. If the allocation site is not flagged, the sets of reachability states contains only the empty reachability state.

Lemma 4 (Soundness of Store Statement Transfer Function). *The transfer function for the allocation statement* x.f=y *is sound with respect to the concrete store operation.*

Proof Sketch: We define $o_{\mathtt{x}}$ to be the concrete object referenced by x and $o_{\mathtt{y}}$ to be the concrete object referenced by y. The store operation can only add new paths in the concrete heap that include the newly created reference $\langle o_{\mathtt{x}}, f, o_{\mathtt{y}}\rangle$. In the abstraction, $E_n(\mathtt{x})$ gives the heap nodes that abstract the objects that x may reference and $E_n(\mathtt{y})$ gives the heap nodes that abstract the objects that y may reference. The concrete operation x.f=y creates a reference from the f field of the object that x references to the object that y references. Applying the abstraction function, the creation of this new reference in all concrete heaps represented by the abstract heap adds a set of edges $E_{\text{new}} \subseteq E_n(\mathtt{x}) \times \{\mathtt{f}\} \times E_n(\mathtt{y})$ to the abstract heap. Since the application of the transfer function to the initial abstraction adds a larger set of edges, it generates an abstract edge set that is higher in the partial order and therefore our treatment of edges in the store statement is sound.

We next discuss the soundness of the transfer function with respect to the reachability states for nodes. We note that the addition of the concrete reference can only (1) introduce new reachability from objects that could reach $o_{\mathtt{x}}$ to objects that $o_{\mathtt{y}}$ can reach and (2) allow edges that could reach $o_{\mathtt{x}}$ to reach objects that $o_{\mathtt{y}}$ can reach. The set $\Psi_{\mathtt{x}}$ defined in Equation 4.6 abstracts the reachability states for the objects that can reach $o_{\mathtt{x}}$ by the abstraction function. Similarly, $\Psi_{\mathtt{y}}$ from Equation 4.7 abstracts the allocation sites for the objects that can reach the objects downstream of $o_{\mathtt{y}}$.

By the abstraction function and the partial order, if an object represented by a heap node $n_{\mathtt{y}} \in E_n(\mathtt{y})$ can reach an object represented by the heap node n' with the abstract reachability state ϕ, then there must exist a path of edges from $n_{\mathtt{y}}$ to $n' \in N$ in the abstract reachability graph in which every edge along the path has ϕ in its set of reachability states and n' has ϕ in its set of reachability states and $\phi \in \Psi_{\mathtt{y}}$. By the abstraction function, the set of reachability states $\psi_{\mathtt{x}} \in \Psi_{\mathtt{x}}$ for $n_{\mathtt{x}}$ abstract $o_{\mathtt{x}}$'s reachability from all objects from flagged nodes. Therefore, the constraints given by Equations 4.8 and 4.9 will propagate the correct reachability change set to n' and Equation 4.10 applies these reachability changes to n'. This implies that the set of reachability states for the nodes is higher or equal in the partial order of reachability graphs to the graph generated by applying a concrete operation followed by abstraction and therefore the node reachability states are sound.

We next discuss soundness with respect to edges that are upstream of the objects downstream of $o_{\mathtt{y}}$ in the pre-transformed concrete heap. Consider an object o abstracted by the heap region n that the store operation changed its reachability state from ϕ to ϕ'. By the abstraction function and partial order function, for any reference in the conrete heap, which we abstract by e, that can reach an object represented by the heap node n, there must exist a path of edges from e to n in the pre-transformed heap in which ϕ is in the reachability state of each edge along the path. Therefore, Constraints 4.11 and 4.12 propagate the reachability change tuple $\langle \phi, \phi' \rangle$ to e which Equation 4.15 will then apply to e and all edges along the path from e to e'.

Finally, we discuss soundness with respect to edges upstream of $o_{\mathtt{x}}$ that the newly created edge allows to reach objects downstream of $o_{\mathtt{y}}$. Consider any upstream reference in the concrete heap, which we abstract by the edge e that can reach an object abstracted by the heap node $n_{\mathtt{x}} \in E_n(\mathtt{x})$ — any reachability state it has for the source object of the store must be abstracted by $\phi \in \Psi_{\mathtt{x}}$ in pre-transformed abstract reachability graph and there must exist a path of edges from e to $n_{\mathtt{x}}$ such that ϕ is in the reachability state of every edge along the path. Therefore, Constraints 4.13 and 4.14 propagate the new reachability change tuples $\{\langle \phi, \phi \cup \psi_{\mathtt{y}} \rangle \mid \psi_{\mathtt{y}} \in \Psi_{\mathtt{y}}\}$ to e and Equation 4.15 will then apply the change tuple to e.

At this point, only the new edge remains. Constraint 4.16 simply copies the reachability states from the edge for $\mathtt{y}$ whose reachability must be the same. It eliminates reachability states that are smaller in the partial order than any state in the source node as they must be redundant with some larger state. The previous three paragraphs imply that the set of reachability states for edges are higher or equal in the partial order of reachability graphs to the graph generated by applying a concrete operation followed by abstraction and therefore the edge reachability states are sound.

Lemma 5 (Soundness of Global Pruning Transformation). *The global pruning transformation is sound (it generates an abstraction that abstracts the same concrete heaps).*

Proof Sketch: We begin by overviewing the soundness of the first phase of the global pruning algorithm. Consider a flagged heap node n_f and a node n that contains n_f in its reachability state ϕ with non-0 arity. From the abstraction function and partial order, if there is no path from n_f to n with ϕ in each edge's set of reachability states, then objects in the reachability state ϕ cannot be reachable from objects abstracted by n_f. Therefore, removing n_f from the reachability set ϕ on n and adding this new reachability set to all edges that (1) have ϕ in their reachability state and (2) have a path to n in which all edges along the path have ϕ in their reachability state generates an abstract reachability graph that abstracts the same concrete heaps and therefore the first phase is sound.

We next discuss the soundness of the second phase. Consider an edge e with a reachability state ϕ. If there is no path from edge e to some node n with all edges along the path containing ϕ in their sets of reachability states and node n including ϕ in its set of reachability states, then dropping ϕ from edge e's set of reachability states yields an abstract state that abstracts the same concrete heaps because if a reference abstracted by e could actually reach an object with the reachability state ϕ then the path would exist. Therefore, the second phase is sound.

E Interprocedural Analysis

We next outline the soundness of the interprocedural analysis. There is a small issue in the interprocedural analysis with the abstraction function for single-object heap nodes. It is possible to have a callee method that only conditionally allocates an object at an allocation site that the caller has a single-object heap node for. The mapping procedure will then merge the caller's single-object heap node into the summary node even though it may represent the most recently allocated object from the site. One can see that this does not pose a correctness issue through a simple transform of the program that adds a special instruction at each method return that allocates an unreachable object at the given allocation site if the callee did not. It is straightforward to see that such a transform preserves the semantics of the program because it does not change the reachable runtime object graph and after this transform the abstract semantics exactly match the concrete program.

We outline the soundness of the interprocedural analysis by analogy to the intraprocedural analysis with inlining. We note that the callee operates on a graph that is a superset of the callee reachable part of the heap. If we consider only those elements that are in the callee reachable part of the heap, the analysis (1) generates a reachability subgraph that is greater in the partial order than the reachability graph that the inlined version would have and (2) all of those elements get mapped to the caller's heap. We note that reachability state changes on the placeholder edges and edges from placeholder nodes summarize the reachability changes of upstream edges and are sound for the same reasons as the store transfer function.