Merging Equivalent Contexts for Scalable Heap-Cloning-Based Context-Sensitive Points-to Analysis*

Guoqing Xu
Computer Science and Engineering
Ohio State University
xug@cse.ohio-state.edu

Atanas Rountev
Computer Science and Engineering
Ohio State University
rountev@cse.ohio-state.edu

ABSTRACT

A context-sensitive points-to analysis maintains separate points-to relationships for each possible (abstract) calling context of a method. Previous work has shown that a large number of equivalence classes exists in the representation of calling contexts. Such equivalent contexts provide opportunities for context-sensitive analyses based on binary decision diagrams (BDDs), in which BDDs automatically merge equivalent points-to relationships. However, the use of a BDD "black box" introduces additional overhead for analysis running time. Furthermore, with heap cloning (i.e., using context-sensitive object allocation sites), BDDs are not as effective because the number of equivalence classes increases significantly. A further step must be taken to look inside the BDD black box to investigate where the equivalence comes from, and what tradeoffs can be employed to enable practical large-scale heap cloning.

This paper presents an analysis for Java that exploits equivalence classes in context representation. For a particular pointer variable or heap object, all abstract contexts within an equivalence class can be merged. This technique naturally results in a new non-BDD context-sensitive points-to analysis. Based on these equivalence classes, the analysis employs a last-k-substring merging approach to define scalability and precision tradeoffs. We show that small values for k can enable scalable heap cloning for large Java programs. The proposed analysis has been implemented and evaluated on a large set of Java programs. The experimental results show improvements over an existing 1-object-sensitive analysis with heap cloning, which is the most precise scalable analysis implemented in the state-of-the-art Paddle analysis framework. For computing a points-to solution for an entire program, our approach is an order of magnitude faster compared to this BDD-based analysis and to a related non-BDD refinement-based analysis.

Categories and Subject Descriptors

F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—*Program Analysis*

General Terms

Algorithms, measurement, experimentation

Keywords

Pointer analysis, points-to analysis, context sensitivity

*This material is based upon work supported by the National Science Foundation under CAREER grant CCF-0546040.

ISSTA'08, July 20–24, 2008, Seattle, Washington, USA.

1. INTRODUCTION

Context sensitivity in points-to analyses has been studied extensively; some of this work is summarized in [11, 8, 28]. Most such analyses compute a complete points-to solution for all variables in a program. Such algorithms usually have to sacrifice analysis precision for practical scalability. An alternative is a refinement-based approach that performs points-to analysis by answering pointer-related queries raised by a compiler or a client analysis on demand [32, 31]. The precision of this approach can be proportional to the resource constraints (such as time and memory) allowed to answer queries. The analysis can be more precise than those that compute a complete points-to solution, if its full precision is reached (i.e., the result is fully refined). However, this algorithm may not scale when computing solutions for a large number of variables [30], and therefore is not suitable for analyses that depend on points-to information for the entire program. In this paper we focus on analyses that compute complete points-to information, and propose an algorithm whose precision approaches that of the refinement-based algorithm, and that scales well to large Java programs.

In context-sensitive analyses a standard approach is to use a calling context abstraction that is a string of the call graph edges leading to the analyzed method. The length of each string is usually limited to a fixed number k. A different technique, used in [34, 36], does not limit the length of a context string, but excludes all contexts corresponding to call edges that are in a cycle in a precomputed call graph. Orthogonal to the context abstraction is the decision as to where to apply context sensitivity. All existing analyses use context-sensitive treatment of local variables, and some algorithms also do this for heap objects. Context-sensitive modeling of objects is sometimes also referred to as *heap cloning* [26]. Higher precision requires a longer context string, with enabled heap cloning. Of these two factors, heap cloning has been shown [20, 15] to be more important, because it allows analyses to distinguish different instances of a logical data structure. The importance of heap cloning can especially be seen in object-oriented programs [15].

To achieve scalability, binary decision diagrams (BDDs) have been employed [19, 34, 36, 3, 16] to avoid redundant representation of similar points-to relationships. A BDD provides an effective representation for context-sensitive analysis because many contexts are equivalent, in the sense that the points-to relationships computed under these contexts are the same [20]. A BDD automatically merges the representation of equivalent points-to relationships.

BDDs may add overhead and increase the running time of the analysis. In [10], the BDD implementation of a points-to analysis for C is on average two times slower than an implementation using a sparse bitmap. If we can identify the source of the equivalence, and then design a non-BDD analysis that explicitly merges points-to relationships that have equivalent abstract contexts, we would be able to reduce analysis running time while still keeping the size of the solution as small as that of a BDD-based implementation.

A context-sensitive analysis with heap cloning may not benefit as much from BDDs. Only short context strings can be used in such an analysis even if BDDs are employed [20], while an analysis without heap cloning [34, 36] can use context strings of arbitrary length. The effectiveness of BDDs decreases in the presence of heap cloning because there are many more equivalence classes of contexts and as a result fewer points-to relationships can be merged.

Our proposal. The focus in this paper is a cloning-based context-sensitive analysis with arbitrary callstring length as the representation of contexts. We first present a characterization of equivalence classes of contexts for pointer variables and pointer targets. This can be captured by an abstraction function that takes as input a full context string c_p for a pointer variable p and a full context c_o string for a pointer target o, and produces a pair of context substrings (sub_1, sub_2) that defines an equivalence class. All pairs (c_p, c_o) such that c_p maps to sub_1 and c_o maps to sub_2 belong to the corresponding equivalence class. The points-to relationships under the contexts within an equivalence class are guaranteed to be the same, and all these contexts can be merged without loss of precision.

Based on this characterization, we propose a new flow-insensitive context-sensitive points-to analysis for Java. First, an intraprocedural phase builds a symbolic points-to graph for each method, using symbolic objects as placeholders for the objects that are not visible in the method. Next, an interprocedural phase performs bottom-up traversal of the call graph, cloning pointer variables and objects from callees to callers, computing points-to relationships, and making the call graph context-sensitive. The analysis takes advantage of equivalence classes: cloning of variables and objects is performed only for chosen context substrings. Since an additional top-down phase is not needed, the symbolic points-to graphs of callees can immediately be discarded when cloning is completed, which reduces memory usage and improves scalability.

Tailored for this analysis, a tradeoff technique is defined: last-k-substring merging for callstrings. By selecting small values for k, it is possible to achieve scalable heap cloning. The analysis was evaluated using 19 programs. The experimental results show that even when $k = 1$, the results are more precise than an existing BDD-based 1-object-sensitive analysis with heap cloning [16], which currently is the most precise publicly available points-to analysis for Java that can compute an whole-program solution in a scalable manner. For computing a solution for an entire program, our approach is an order of magnitude faster compared to this analysis and to the refinement-based analysis discussed above.

2. EQUIVALENT CALLING CONTEXTS

Figure 1 shows an example adapted from [31], together with the corresponding call graph. The calling context abstraction is a string of call graph edge labels from *main* down to a certain method.

The term "equivalent context" was defined in [20] in terms of method-context pairs. Two such pairs (m_1, c_1) and (m_2, c_2) are equivalent if $m_1 = m_2$ and for every local pointer variable p in the method, the points-to set of p is the same under both contexts c_1 and c_2. In our running example, the method-context pairs (*insert*, *ac*) and (*insert*, *bfc*) are equivalent (the contexts use the edge labels from Figure 1) because the only *AddrBook* object on which *insert* is invoked is allocated in *addName*, and this object does not affect and is not affected by the callers of *addName*.

This definition is coarse-grained as it requires all pointer variables in a method to have the same equivalence classes. However, the equivalent contexts for different variables in a method can differ. Consider the constructor of *Vector* in the example. We denote a context-insensitive heap object abstraction by o_i, where i is the line number for the allocation site. Similarly, a context-insensitive pointer variable abstraction is denoted by p_i, where p is the name of the variable, and i is the line number where p is first used/defined. Using this notation, pointer variable t_3 can be decided to point to object o_3 within the constructor. Since no callers of the constructor can affect the points-to pair (t_3, o_3), all contexts for t_3 are equivalent (i.e., there is only one equivalence class of contexts for this variable). However, variable $this_4$ points to objects outside the constructor and its points-to pairs are affected by the callers of the constructor (e.g., the points-to sets for $this$ in contexts containing call graph edge i and ones containing edge j are different). In general, the set of equivalence classes for a method m as defined in [20] treats uniformly all variables in m, which may lead to unnecessary equivalent classes for some variables. Our goal is to provide a finer-grained definition of equivalent contexts in terms of pointer variables and heap objects, in order to exploit greater similarity (i.e., fewer equivalence classes for some variables).

Equivalence classes for recursion-free programs. Consider a context-insensitive points-to relationship of the form (p, o) where p is a pointer variable and o is an object allocation site. A context-sensitive points-to relationship in a heap-cloning analysis is a 4-tuple (p, c_p, o, c_o), where c_p is a full context (callstring) that goes from *main* to the method m_p that defines p, c_o is a full context that goes from *main* to the method m_o that contains o, and p under context c_p may point to the objects created at o under context c_o.

To characterize the equivalence of calling contexts, we outline two hypothetical analyses. These analyses are conceptual constructs, not our actual algorithm. The first analysis performs bottom-up traversal and method inlining. Each inlining step creates a copy of the callee's body (this body itself has already been modified by earlier inlining steps) inside the caller, with the appropriate variable renaming, assignments for parameter passing, etc. The renaming attaches a context string to each pointer variable p and allocation site o. For example, if the callee method m contains an assignment $p := q$ there p and q are local variables, and the call graph contains an edge e from n to m, this statement would be inlined in n as $p^{(e)} := q^{(e)}$. If some method k calls n along a call graph edge f, then the inlining step for f would create a statement $p^{(f,e)} := q^{(f,e)}$ in k, where (f, e) is a partial callstring. Eventually all methods are inlined in *main*, all callstrings are complete, the original statement has been transformed to $p^{(e_0,\ldots,e)} := q^{(e_0,\ldots,e)}$ where the source of e_0 is $main$, and intraprocedural points-to analysis is applied to the body of *main*.

The second hypothetical analysis is a variation of the first one. After a method n is processed to inline all its call sites, intraprocedural points-to analysis is applied to the new body. This analysis of n produces tuples (p, c_p, o, c_o) where c_p and c_o start with call graph edges whose source is n. Suppose e is one of the edges from n to some callee m. It is easy to see that for any (p, c'_p, o, c'_o) that was computed in the earlier analysis of m (here c'_p and c'_o start with edges whose source is m), a corresponding (p, c_p, o, c_o) must exist in n's solution; here $c_p = (e) \circ c'_p$ and $c_o = (e) \circ c'_o$ where $\circ$ denotes list concatenation. This is because all statements in m's body whose cumulative effects allowed the creation of (p, c'_p, o, c'_o) are also present in the body of n, except that all pointer variables and allocation sites are tagged with updated contexts starting with e.

```
1  class Vector {
2     Object[] elems; int count;
3     Vector() { t = new Object[10];
4                this.elems = t; }
5     void add(Object p) {
6        t = this.elems;
7        t[count++] = p;//write t.arr_ele
8     }
9     Object get(int ind) {
10       t = this.elems;
11       p = t[ind]; return p;//read t.arr_ele
12    }
13 }
14 class AddrBook {
15    Vector names;
16    AddrBook() { t = new Vector();
17                 this.names = t; }
18    void insert(String n, ...) {
19       t = this.names; ...;
20       t.add(n);
21    }
22    void update() {
23       t = this.names;
24       for (int i = 0; i < t.size(); i++) {
25          Object name = t.get(i);
26          String nameStr = (String)name;
27          ...
28       }
29    }
30 }
```

```
31 class Client {
32   void addName() {
33      AddrBook book1 = new AddrBook();
34      book1.insert("Tom");
35      book1.update();
36   }
37   String lookup() {
38      int i = find(); // find is omitted
39      if (i == -1) addName();
40      else {
41         Vector v = new Vector();
42         Integer in = new Integer(i);
43         v.add(i);
44         in = (Integer)v.get(0);
45      }
46   }
47   static void main(String[] args) {
48      addName();
49      lookup(); }
50 }
```

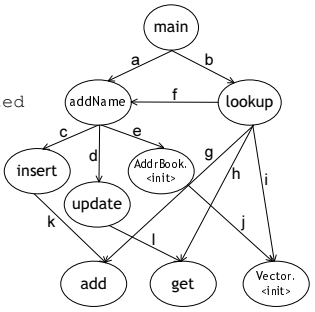

Figure 1: Running example: code and call graph.

However, not all tuples for n have this property: for example, if the call site in the original body of n is $p := m()$ and m creates and returns a new object o, the solution for n after inlining will contain $(p, \epsilon, o, (e))$, where ϵ is the empty context; for this tuple, there is no corresponding tuple in any callee of n. Furthermore, if originally n also contains a call $p.m2()$ for call graph edge $e2$, after inlining we will have $this^{(e2)} := p^{\epsilon}$ to represent parameter passing, and $(this, (e2), o, (e))$ will be computed for n. Again, for this tuple there is no corresponding tuple in any callee of n.

The lifetime of a tuple consists of a single creation event in some method, followed by a sequence of inlining steps that increase both calling contexts in synch, until full contexts starting at *main* are reached. For any (p, c_p, o, c_o) in the solution for *main* (i.e., in the final points-to solution), the method in which the initial creation event happened will be referred to as the *flowing point* (FP) for this tuple. This is the first method in which o can flow to p; here "flow" is defined by the underlying intraprocedural analysis. The FP is unique and belongs to a common prefix of c_p and c_o. If the common prefix ending at the FP were replaced with *any* chain from $main$ to the FP, the points-to relationship would still hold.

For example, in Figure 1 the FP for (t_3, bi, o_3, bi), where o_3 is the object created at line 3, is the constructor of *Vector*. This is also the FP for (t_3, aej, o_3, aej) and for $(t_3, bfej, o_3, bfej)$. In this case contexts bi, aej, and $bfej$ are equivalent. The FP for (p_{11}, adl, o_{34}, a), where o_{34} is the string constant "Tom" at line 34, is *addName*: if all callees of *addName* are transitively inlined into it along chain dl, o_{34} will flow to p_{11} through a sequence of assignments within *addName*. Furthermore, this flow is independent of the callers of *addName*.

A key observation is that once a tuple in created for the first time in its FP, it will exist in all transitive callers of the FP (where it will be associated with longer context strings). For any contexts c_p and c_o, the part that really contributes to (p, c_p, o, c_o) are the suffixes starting at the call edges that leave the FP. This naturally leads to the definition of equivalence classes for contexts, based on the following abstraction functions f_p and f_o:

$$f_p((p, (e_0, \ldots, e_{FP}, \ldots, e_p), o, c_o)) = (e_{FP}, \ldots, e_p)$$
$$f_o((p, c_p, o, (e_0, \ldots, e'_{FP}, \ldots, e_o)) = (e'_{FP}, \ldots, e_o)$$

where the source of e_{FP} and e'_{FP} is the FP and the source of e_0 is *main*. The suffix for p or o will be referred to as a *unique replacement context (URC)* for all contexts that map to it, with respect to the tuple. For example, the URCs for both t_3 and o_3 with respect to (t_3, c, o_3, c), where c is any chain from *main* to *Vector.*<*init*>, are an empty string ϵ. The URCs for p_{11} and o_{34} with respect to (p_{11}, adl, o_{34}, a) are dl and ϵ.

For each (p, c_p, o, c_o) in the final analysis solution, one can conceptually replace the contexts with their URCs. Each distinct URC pair $(f_p(c_p), f_o(c_o))$ defines a context equivalence class for both pointer variables and pointer targets, and characterizes equivalent contexts under which the points-to relationship holds. A context-sensitive analysis only needs to explicitly represent points-to relationships with URCs. To answer a query "which objects may p point to under context c", one can inspect the set of URCs computed by f_p, compute a subset U_p containing URCs that c maps to, and return the union of object-context pairs (o, c_o) such that $(p, c_p \in U_p, o, c_o)$ holds. The set of distinct URC pairs can be much smaller than the set of all pairs of full contexts.

This definition of equivalence classes provides some insights into why a BDD implementation may be less effective in scaling an analysis with heap cloning than a non-heap-cloning analysis. Without heap cloning, an element of the solution is a 3-tuple and an equivalence class is defined by a single string urc_p rather than by a pair (urc_p, urc_o). Thus, the number of equivalence classes is reduced significantly, and the number of times a pointer variable needs to be cloned is also greatly reduced. All these reductions lead to higher levels of similarity for a BDD to exploit.

Equivalence classes in the presence of recursion. In the presence of recursion we first consider a mapping of the infinite set of call chains to a finite set of repetition-free chains. TRIMCHAIN shown in Figure 2 provides an operational definition of this mapping. Given a call chain $(e_i, \ldots, e_j)$ in which a method could be repeated, the function scans the chain in reverse order, removes all substrings whose start point and end point are the same method, and returns a new string that is repetition-free. Note that this description is conceptual: the functionality defined by TRIMCHAIN could be implemented implicitly in an analysis algorithm.

TRIMCHAIN $((e_i, ..., e_j))$

```
1: Sequence newChain
2: for q from j downto i do
3:    if newChain does not contain e_q.src then
4:       newChain.addFirst(e_q)
5:    else
6:       find the edge e_k in newChain such that e_k.src = e_q.src
7:       remove all edges in newChain occurring before e_k
8:    end if
9: end for
10: Return newChain
```

Figure 2: TRIMCHAIN for handling of recursion.

The hypothetical inlining analyses described earlier can be modified as follows. First, whenever a statement is inlined from a callee to a caller, the function from above is applied to ensure repetition-free contexts. For example, if the callee has a statement $p^c := q^d$ where p and q are variables and c and d are contexts, inlining along a call graph edge e will generate in the caller's body a statement $p^{c'} := q^{d'}$ where $c' = TrimChain((e) \circ c)$ and d' is defined similarly. Second, the inlining process will iterate through a SCC a sufficient number of times to reach a fixed point for the set of unique statements in the body of each (inlining-modified) SCC method. There is a well-defined upper bound on the number of such iteration steps [29]. After everything is inlined in *main*, intraprocedural analysis computes a solution containing tuples (p, c_p, o, c_o) where c_p and c_o are repetition-free callstrings starting at *main*. For this solution we can define equivalence classes based on flowing points, similarly to how this was done for the recursion-free case. For a tuple (p, c_p, o, c_o) from the final solution, the FP is determined by the earliest time when an "ancestor" tuple was created. This FP defines the suffixes of c_p and c_o that are used as URCs.

3. POINTS-TO ANALYSIS

The approach from the previous section could potentially be applied to different existing or future points-to analyses. In this section we describe a new points-to analysis for Java that is based on this idea. The analysis is flow-insensitive and context-sensitive, with callstring contexts and heap cloning. Intraprocedurally the value flow propagation is subset-based, and interprocedural propagation uses a combination of subset-based and unification-based techniques. Our experiments indicate that this analysis is more precise and more efficient than a subset-based 1-object-sensitive analysis with heap cloning implemented with BDDs [16].

3.1 Intraprocedural Analysis

Flow graph. The intraprocedural phase first builds a flow graph representing pointer assignments, similarly to the pointer assignment graph (PAG) in Spark [18]. There are three types of nodes in a flow graph: allocation site nodes $\in \mathcal{O}$, variable nodes $\in \mathcal{V}$ (for local variables and formal parameters) and field dereference nodes $\in \mathcal{F}$ (nodes of the form $v.fld$, where $v \in \mathcal{V}$). Edges in the graph represent the flow of values: e.g., an assignment $p := q$ where $p, q \in \mathcal{V}$ is represented by a flow graph edge $q \rightarrow p$.

Symbolic points-to graph. Based on the flow graph of a method, we construct a symbolic points-to graph. A SPG for a method is an extension of a standard points-to graph, with three types of nodes: variable nodes $\mathcal{V}$, allocation nodes $\mathcal{O}$, and symbolic object nodes $\mathcal{S}$. A variable points-to edge is $(v \rightarrow o) \in (\mathcal{V} \rightarrow (\mathcal{O} \cup \mathcal{S}))$. A field reference edge is $(o_1 \xrightarrow{f} o_2) \in ((\mathcal{O} \cup \mathcal{S}) \xrightarrow{FIELDS} (\mathcal{O} \cup \mathcal{S}))$.

The SPG of a method is constructed from the flow graph. First, symbolic object nodes $so \in \mathcal{S}$ are created to represent objects that are not visible in the method. A symbolic object is created for each formal parameter and each call site that returns a value, and edges in the flow graph are added between the newly created objects and the corresponding variable nodes (i.e., the formal parameter node or the left-hand-side variable node of a call site). Second, closure over flow graph edges is computed so that any two nodes that are originally transitively connected are now directly connected.

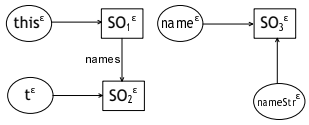

Figure 3: SPG for *update* from the running example.

Next, if there exits an edge between $o \in \mathcal{O} \cup \mathcal{S}$ and $n \in \mathcal{V}$ in the flow graph, a new edge $n \rightarrow o$ is added to the SPG. Both nodes in this edge are labeled with ϵ, signifying that the current callstring context associated with them is empty. Following this, it is necessary to propagate points-to relationships through object fields to account for the effects of assignments to/from field dereferences $v.f$. This propagation is a worklist-based fixed-point computation and the iteration stops when the SPG no longer changes. We use the approach from [18] for this computation.

After this propagation, for each node $v.f$ in the flow graph, all objects o in v's points-to set are considered and their f fields are examined. If the points-to sets of all $o.f$ are empty, a new symbolic node so is introduced and SPG edges $o \xrightarrow{f} so$ are added. Additional propagation is then performed to reach a final fixed point.

The SPG for method *update* from the example is shown in Figure 3. Symbolic object SO_1 is a placeholder for the unknown receiver object. Symbolic object SO_2 represents the object pointed-to by $this.names$. Finally, SO_3 is a placeholder for the object returned by the call to *get*. During the subsequent interprocedural analysis, the three symbolic objects will be resolved to real objects.

A SPG is similar to the points-to escape graph from [35]. However, in a points-to escape graph, an outside node (similar to our symbolic node) is created for each field dereference $v.f$ if v points to an escaped object, while we do not create a symbolic object node for $v.f$ if there already exists a symbolic or allocation node in the points-to set of some $o.f$ where v points to o.

Escape analysis. An object escapes a method if the object's lifetime exceeds the lifetime of the method [35, 6]. Since allocation nodes or symbolic nodes that do not escape a method are not pointed to by variables defined in the callers of the method, we use the results of the escape analysis to filter out such non-escaping allocation or symbolic nodes, as described in the next subsection.

A node n can directly escape a method if $n \in \mathcal{O} \cup \mathcal{S}$ and in the method's SPG there exists a points-to edge $p \rightarrow n$, where p represents a static field, a formal parameter of the method, or the return variable of the method. (Our implementation also takes into account escaping through exceptions.) A node $n \in \mathcal{O} \cup \mathcal{S}$ indirectly escapes if in the SPG it is reachable from a node that directly escapes the method. These definitions are extended from the object escapement definitions in [35, 6], with the inclusion of symbolic objects. Note that not all symbolic object nodes escape the method. For example, symbolic nodes that do not escape may correspond to objects created in the callees of the method, returned to the method, and then not propagated any further.

3.2 Interprocedural Analysis

After intraprocedural analysis is performed on each method, the interprocedural phase executes a bottom-up traversal of the SCC-DAG of the call graph to inline variables and objects. We use an initial imprecise call graph based on RTA [2]; during the analysis,

PROCESSSPG (Method m)

1: Set $clonedEdges$
2: COMPUTELOCALPOINTSTO(m, $clonedEdges$)
3: **for** each call graph edge e that goes to m **do**
4: **for** each SPG edge $ce \in clonedEdges$ **do**
5: CLONEEDGE(ce, e)
6: **end for**
7: MERGENODES(m, e)
8: **end for**
9: destroy the SPG of m

Figure 4: Processing during bottom-up traversal.

this graph may be refined as described later. The basic structure of the algorithm is the following: edges from the callee's SPG are inlined ("cloned") in the caller's SPG, with the corresponding update of context information (i.e., context length increases by 1). After this inlining, some symbolic nodes are merged with the corresponding real allocation nodes; this way, unresolved points-to relationship involving these symbolic nodes are finally resolved. As the bottom-up traversal progresses, the number of symbolic nodes continuously decreases until all of them are replaced by allocation nodes. Furthermore, whenever a FP is reached for some points-to relationship, this relationship is excluded from further inlining and is added to the final points-to solution for the program.

We first discuss the analysis of non-SCC methods, each of which is processed by function PROCESSSPG shown in Figure 4. At the time a method m is processed, all of m's callees have already been processed by this same function, and their SPGs have been inlined in m's SPG.

Resolving local points-to relationships. For a method m, COMPUTELOCALPOINTSTO resolves local points-to and field reference relationships and adds them to the final points-to solution produced by the analysis. It also computes a set $clonedEdges$ of SPG edges that need to be cloned to (i.e., inlined in) m's callers. The rules used in this computation are shown in Figure 5(1).

The function resolves not only points-to relationships in m, but also relationships that were unresolved in m's callees but now are resolvable in m. It searches for edges in m's SPG, including edges cloned from its callees, that can be resolved. Whenever such an edge is found, the corresponding 4-tuple is added to the final points-to solution for the program. Variables can point only to assignable objects, and based on this we use types to filter out infeasible tuples. A SPG edge $v^c \rightarrow o^d$ where $o \in \mathcal{O}$ indicates that m is a FP for the pair (v, o) and an equivalence class defined by (c, d) is found. Thus, this edge does not need to be cloned to m's callers.

A points-to edge starting at v^c needs to be cloned only if the target node is a symbolic object node so^d and this symbolic node escapes. If an (allocation or symbolic) object node escapes m, all its outgoing edges are added to set $clonedEdges$. Note that even in the case of an edge between two ordinary (i.e., non-symbolic) objects $o_1, o_2 \in \mathcal{O}$, the edge still needs to be cloned if o_1 and o_2 escape. The reason is that there could be symbolic object nodes reachable from o_2, and these nodes could correspond to different real objects in different callers (i.e., contexts).

Cloning of SPG edges into callers. After the set of SPG edges that need to be cloned is computed, the edges are cloned for each of m's callers (lines 4-6 in Figure 4), and the newly cloned edges are merged with corresponding SPG edges in the caller (line 7 in Figure 4). Function CLONEEDGE is straightforward and its code is not shown. Consider an edge $ce = (p^c, q^d)$ in m's SPG that should be cloned due to some call graph edge e, where e's source is n and e's target is m. A cloned node $p^{c'}$ is created in n's SPG, where the new context c' is the concatenation of (e) and c. If many

(1) COMPUTELOCALPOINTSTO

$$\frac{v^c \longrightarrow o^d \in SPG, v \in \mathcal{V}, o \in \mathcal{O}}{(v, c, o, d) \in \text{ final points-to solution}}$$

$$\frac{o_1^c \xrightarrow{f} o_2^d \in SPG, o_1, o_2 \in \mathcal{O}}{(o_1, c, f, o_2, d) \in \text{ final points-to solution}}$$

$$\frac{v^c \longrightarrow so^d \in SPG, so^d \text{ escapes}, v \in \mathcal{V}, so \in \mathcal{S}}{v^c \longrightarrow so^d \in clonedEdges}$$

$$\frac{o_1^c \xrightarrow{f} o_2^d \in SPG, o_1^c \text{ and } o_2^d \text{ escape}, o_1, o_2 \in \mathcal{O} \cup \mathcal{S}}{o_1^c \xrightarrow{f} o_2^d \in clonedEdges}$$

(2) MERGENODES

$$\frac{\text{actual } v_i, \text{formal } f_i, v_i^\epsilon \longrightarrow o^c \in SPG_n \quad f_i^\epsilon \longrightarrow so^\epsilon \in SPG_m, so \in \mathcal{S}, clone(so^\epsilon, e) = so^{(e)}}{(o^c, so^{(e)}) \in match}$$

$$\frac{\text{left-hand-side } v, \text{return var } ret, v^\epsilon \longrightarrow o_1^c \in SPG_n \quad ret^\epsilon \longrightarrow o_2^d \in SPG_m, o_1, o_2 \in \mathcal{O} \cup \mathcal{S}, clone(o_2^d, e) = o_2^{d'}}{(o_1^c, o_2^{d'}) \in match}$$

$$\frac{(o_1^c, o_2^d) \in match, o_1^c, o_2^d \in \mathcal{O} \cup \mathcal{S} \quad \exists f : o_1^c \xrightarrow{f} o_3^p, o_2^d \xrightarrow{f} o_4^q \in SPG_n}{(o_3^p, o_4^q) \in match}$$

Figure 5: Rules for interprocedural phase.

cloned edges ce for e include p^c, only one $p^{c'}$ is created. Similarly, a cloned node $q^{d'}$ is created in n's SPG. These nodes are connected by new edges in n's SPG.

Merging caller nodes and cloned callee nodes. Interprocedural propagation is achieved by merging the cloned versions of the object subgraphs rooted at formal parameters/return variables in the callee with their corresponding object subgraphs rooted at actual parameters/left-hand-side variables in the caller.

For each caller of m, function MERGENODES, outlined in Figure 5(2), merges the newly cloned nodes and the corresponding nodes in the caller's SPG. Consider call site $v = v_0.m(v_1, v_2, \ldots)$ corresponding to a call graph edge e from method n to method m. The initialization step of MERGENODES computes a set $match$ of ordered pairs of nodes in n's SPG. These pairs are the staring points of the object subgraph merging described below. Consider an actual parameter v_i (including v_0) and its corresponding formal parameter f_i in m. In m's SPG, there is an edge $f_i^\epsilon \longrightarrow so^\epsilon$ where so is a symbolic object. This edge is cloned in n's SPG as $f_i^{(e)} \longrightarrow so^{(e)}$. For any edge $v_i^\epsilon \longrightarrow o^c$ in n's SPG, the pair $(o^c, so^{(e)})$ is added to $match$.[1] Also, consider the return variable ret in m and the left-hand-side variable v at the call site. For any edge $v^\epsilon \longrightarrow o_1^c$ in n's SPG and $ret^\epsilon \longrightarrow o_2^d$ in m's SPG, a pair with o_1^c and the clone of o_2^d is added to $match$. For each of these initial pairs, the analysis simultaneously traverses the two sub-graphs of n's SPG that are reachable from the objects in the pair (last rule in Figure 5) and includes all traversed pairs in $match$.

Given the constructed set $match$, a helper function MERGE is applied to each pair in the set. This function is illustrated in Figure 6. In (a), if both two nodes are symbolic object nodes, node so_2^d in the callee is *replaced* by node so_1^c in the caller, in the sense that an edge is added between the source node of each of so_2^d's incoming edges and so_1^c, and an edge between so_1^c and the target

[1] At this moment, the only context that could be associated with v_i is ϵ; thus, the only possible edges for v_i have as source v_i^ϵ.

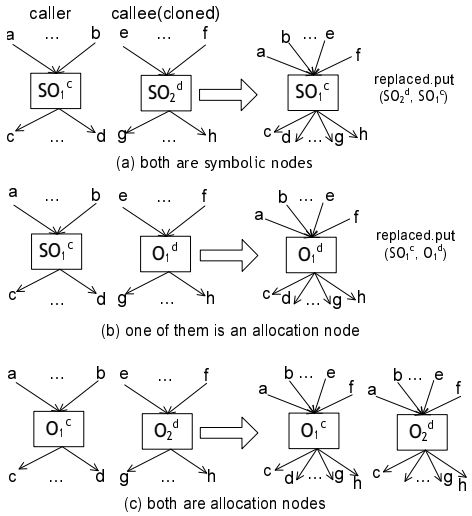

Figure 6: Illustration of function MERGE**.**

node of each of so_2^d's outgoing edges. In (b), if one of the nodes is an allocation node, the symbolic node is replaced by the allocation node. In (c), if both o_1^c and o_2^d are allocation nodes, they must be preserved — for any node that is directly connected to one of these two nodes, the algorithm creates an edge to connect it with the other. A map *replaced* is maintained to map a node p to the node q by which p is replaced, so that when p is processed again, q can be used instead. Merging of two nodes occurs only when the type of one node is assignable to the type of the other; otherwise the node cloned from the callee is discarded and nodes reachable from it are subsequently ignored. The nodes that have been replaced are eventually discarded after MERGENODES returns. In the presence of recursive data structures, of course, we need to check if an edge has been visited before its target node is merged.

After function MERGENODES finishes, a symbolic node cloned from the callee m is instantiated by either a symbolic node or an actual object node in caller n, and a symbolic node in n is either instantiated by an actual object node from m or is not affected by the merging. At this time, the SPG for m can be safely destroyed, because all points-to relationships that can be resolved in m (and its callees) have already been resolved and added to the final points-to solution for the program, and all relationships that are still unresolved have been cloned into m's callers.

Example. Figure 7 illustrates the bottom-up phase of the analysis for *addName* and its callees. The SPG for each method is shown; the shaded nodes need to be cloned to the callers of the method. We add a globally unique number after "SO" to name a symbolic node that has not been cloned, so that we can distinguish between nodes cloned from callees and nodes introduced in the SPG during the intraprocedural phase. For each SPG, the set of escaped nodes *esc* and the map for node replacement *replaced* are also shown. The map contains a pair (o_1, o_2) if o_1 is replaced by o_2 during merging. When *addName* is processed, points-to pairs for all variables in it and its callees (except for its own *this*) under all contexts become fully resolved. All resolved points-to edges are immediately excluded from the propagation. For example, after COMPUTELOCALPOINTSTO completes for *addName*, only two nodes need to be cloned and merged into the callers of *addName*.

Unification due to merging. The merging process, although conservative, does not strictly correspond to the semantics of a subset-based analysis. In some cases unification may occur due to bidirectional value flow, as illustrated by the following example:

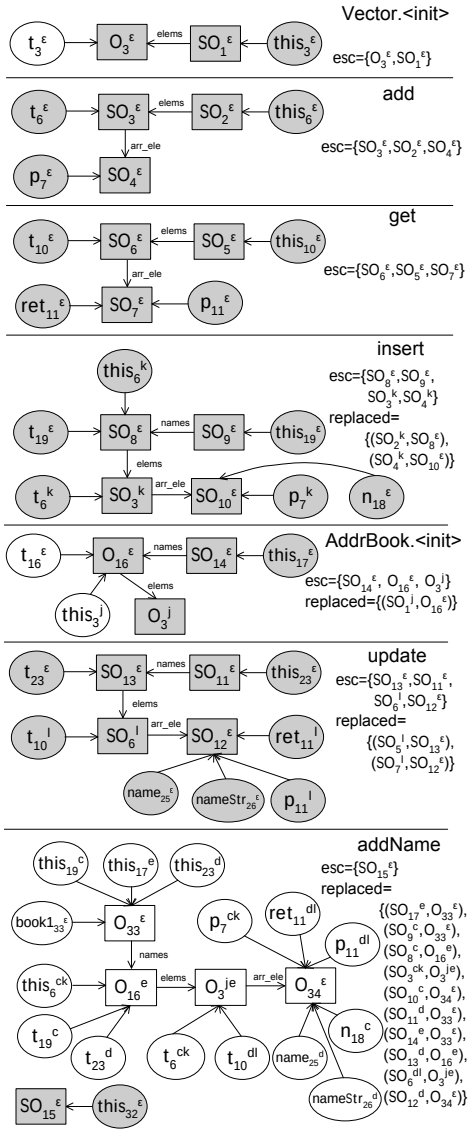

Figure 7: Illustration of the bottom-up phase.

```
n(C c){
 F f = new F(); //O1
 c.fld1 = f;
 m(c);
 O i = new O(); //O2
 f.fld2 = i;
}
```

```
m(C d){
  F g = new F();  //O3
  d.fld1 = g;
  O h = g.fld2;
}
```

The original SPGs of methods n and m are shown in Figure 8(a) and (b), respectively. SO_1 and SO_2 represent symbolic object nodes created for the formal parameters of n and m. SO_3 is the symbolic object created for the field deference node $g.fld2$ in m. O_1, O_2, and O_3 are used to represent the allocation nodes in the two methods. We use c to denote the call graph edge from n to m.

The SPG for n after cloning and merging is shown in Figure 8(c). Unification could come from the redirection of points-to edges. For example, since O_1 and O_3 are allocation nodes, during merging both of them need to be preserved and variables pointing to one object are forced to point to the other. However, edges $f^\epsilon \rightarrow O_3^c$ and $g^c \rightarrow O_1^\epsilon$ are spurious (shown in dotted lines), as O_3^c cannot flow to f and O_1^ϵ cannot flow to g. Unification could also come

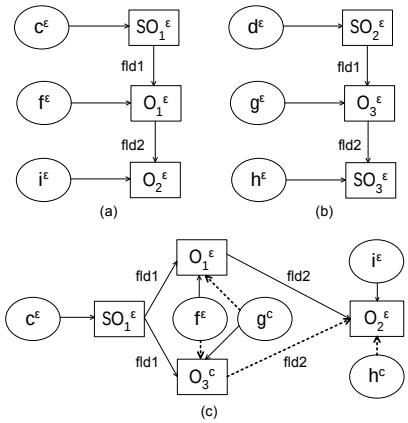

Figure 8: Illustration of unification.

from the redirection of field reference edges. In this example edge $O_3^c \rightarrow O_2^\epsilon$ is infeasible because SO_3, which is created for $g.fld2$, comes directly from field *fld2* of O_3, and is not related to the value of *fld2* for $d.fld1$. Subsequently, $h^c \rightarrow O_2^\epsilon$ is not feasible as well.

The approximations outlined above are due to the use of SPGs. While one could consider replacing SPGs with other more precise representations (e.g., flow graphs [18, 25, 32, 31]), it is not clear whether inlining along long call chains can be made scalable with such alternatives. Section 4 shows that our analysis has higher precision and lower cost than a fully subset-based 1-object-sensitive analysis. This indicates that the loss of precision due to unification may have been offset by the precision gains due to stronger context sensitivity. Furthermore, we have investigated some SPG enhancements to avoid such approximations, and found them to have no useful effect on analysis precision.

Dynamic dispatch. In the presence of dynamic dispatch, propagation occurs on all possible call graph edges. Since an imprecise call graph based on RTA is used, a call graph edge could be decided to be spurious in a certain context after it is processed. All nodes that are propagated up through that edge in that specific context must be removed. Removing these nodes is difficult, because they could be scattered in many SPGs. We use the observer pattern to solve this problem. For a call site that could invoke more than one target method, suppose the variable pointing to the receiver is r. Each pair (r, t_i) is treated as a *subject*, where t_i is a subtype of r's declared type. For each (r, t_i), all nodes that have been cloned into the current method from the callee corresponding to receiver type t_i are treated as *observers* of this subject. If r is cloned into a method m with a cloned node r^c, all observers that are also cloned in m need to replace r with the cloned node r^c in their subject. Once a 4-tuple (r, c, o, d) is added to the final points-to solution, the observers of subject (r^c, $o.type$) are notified to release this subject, because the corresponding call graph edge is not spurious in the current context. If r^c does not need to be cloned any more, a "remove" message is sent to all remaining observers of r^c. An observer removes itself from the SPG it belongs to if it receives a "remove" message from any subject it observes. Once the interprocedural phase finishes, the entire call graph is scanned and call graph edges that are infeasible under all calling contexts are removed. Even though this approach is effective in avoiding spurious points-to relationships, it cannot completely eliminate them. For example, a symbolic object can still be merged with an object of an assignable type after being cloned along a spurious call chain.

Static fields. Many cloned nodes represent static fields. Since object nodes that can be pointed to by static fields always escape a method, they could be cloned a large number of times. A majority of these cloned (allocation or symbolic) nodes can only be pointed to by static fields, and cloning them does not contribute much to the precision of the analysis. Thus, static fields are treated context-insensitively: we only keep global copies of these field nodes, and do not create clones during propagation.

3.3 Last-k-Substring Merging

Although our analysis is efficient, it does not scale to large Java programs without applying any tradeoffs. The problem lies in the cloning of allocation and symbolic nodes — they have to be cloned upwards as long as they escape a method, since they could potentially be pointed to by variables. However, if a cloned object node could never be pointed to a variable, it becomes redundant. If a node is deep inside the merged subgraphs defined by set $merge$ (i.e., a long reference distance away from the entry points of these subgraphs), it is less likely to be pointed directly by local variables in methods upward along the call chain.

A common tradeoff in a context-sensitive analysis is k-callstring-length, which limits the context to be the last k call graph edges. However, applying this approach may result in a large number of spurious points-to relationships that are computed along *unrealizable paths*, because nodes are not distinguished if the last k call graph edges of their contexts are the same. In fact, merging of nodes in unrealizable paths is one of the key factors that causes imprecision [20]. This tradeoff is also not suitable for our analysis. Although the number of nodes that need to be cloned can become smaller, the number of edges coming in/leaving each node could be significantly larger. This creates a large amount of work for function MERGENODES. In fact, we implemented the 3-callstring-limit technique in our analysis, and the analysis was very slow — it ran for more than an hour on the smallest benchmark — and less precise than the 1-object-sensitive analysis (discussed later).

Based on the observation that most redundant nodes are allocation and symbolic nodes, we propose a precision-scalability tradeoff referred to as *last-k-substring merging*. When an allocation or symbolic node n^{c_1} in a method m needs to be cloned to a caller of m, if there already exists a node n^{c_2} in the caller such that c_1 and c_2 share the last k call graph edges, we use n^{c_2} as the cloned node for n^{c_1}, instead of creating a new node. Applying this tradeoff has two advantages. First, last-k-substring merging does not limit the length of a context string. Second, node merging is caller-based, and cannot happen indiscriminately throughout the program. This significantly reduces the chances of nodes along unrealizable paths being merged. This approach has another desirable property: for a node, the longer its context string is, the more likely it will be merged with another node. The reason is that a node with a long context is cloned and propagated from a method "at the bottom" of the call graph, and there could be many call chains along which the (original) node is cloned. In the SPG of a method far from the node's introducing method, such a node is usually deep and far from the local variables of the method, and is thus less likely to be directly pointed to by variables in this method or in its callers (i.e., it is more likely to be redundant).

Example. We use the following example to illustrate the difference between last-k-substring-merging and the commonly used k-callstring-length approach. Suppose there are two partial call chains where symbols represent methods: $e \rightarrow p \rightarrow a \rightarrow b \rightarrow c$ and $f \rightarrow p \rightarrow d \rightarrow b \rightarrow c$. To simplify the example, assume that $k = 1$. Consider a symbolic or object node n originally introduced in method c, and assume that n needs to be cloned all the way up along the two call chains. The 1-callstring-length approach does not distinguish these two contexts, because they share the last

call graph edge $b \rightarrow c$. In the setting of the bottom-up propagation described in this paper, after method b is processed in the inter-procedural phase, such an approximation would merge nodes n^{abc} and n^{dbc}. The proposed k-substring-merging approach, however, will still create two distinct nodes in this case, because when n^{bc} is cloned, there does not exist a node n^{*bc} in either method a or d. Our approach merges nodes only at the caller method where these two call chains intersect (e.g., at method p in this example, where n^{pabc} and n^{pdbc} share a common suffix).

The two properties of last-k-substring-merging discussed above can be seen in this example. First, it does not limit the length of a context. Although node n^{pabc} and node n^{pdbc} are merged at p, different clones of this node still need to be created for e and f. However, by applying 1-callstring-length, all nodes of the form n^{*bc} are merged. Second, in an "merge" method such as p, nodes that have longer contexts (i.e., propagated from farther callees) are more likely to be merged than nodes that have shorter contexts (i.e., propagated from nearer callees). In our example, in method p, instances of n (from c) are merged, while instances of nodes introduced in b are not. The reason why this is desirable is that in method p, clones of n are more likely to be redundant (i.e., never pointed-to by a variable node) than clones of nodes introduced in b.

3.4 Handling of Recursion

As described earlier, in the presence of recursion the infinite set of call chains is mapped to a finite set of repetition-free call chains in order to ensure termination. Of course, when processing a SCC the analysis propagates along all chains, not just along repetition-free ones. The definition of function TRIMCHAIN from Section 2 essentially provides an algorithm to handle an SCC — a cloned node n^{c_1} does not need to be created in a caller if there already exists a node n^{c_2} in this caller such that c_2 is a suffix of c_1. Any SPG edge that would have been created for n^{c_1} during cloning is redirected to n^{c_2}. When a SCC is encountered in the bottom-up phase, all its methods are processed until the fixed point is reached (i.e., no nodes need to be cloned between any two methods in the SCC). If desired, the last-k-substring merging approach can also be applied to speed up this computation.

4. EMPIRICAL EVALUATION

We implemented the analysis using the Soot 2.2.3 [33] framework. The analyses included the Sun JDK 1.3.1_20 libraries, in order to to enable comparisons with the analyses implemented in Paddle [16]. All experiments were performed on a machine with an Intel Xeon 2.8GHz CPU, and run with 3G heap size (option Xmx3072m). The set of benchmarks, shown in Figure 1, includes the SPECjvm98 suite, programs from the Ashes suite [1], programs from the DaCapo suite [7], and several other programs. The programs from DaCapo were from version beta050224; this version is obsolete, but we used it to allow comparison with previous work [20, 31]. We did not include some programs from this version because we did not manage to deploy all library classes these programs depend on.

Table 1 shows the number of methods reachable from *main* in the call graph computed by Spark's default context-insensitive points-to analysis [18]. The table also shows the number of statements in Soot's Jimple representation of these methods, and the size of the largest SCC in the call graph. Library methods are included in these measurements, since the evaluated whole-program analyses have to analyze them together with the client code. For each benchmark, a SCC can contain more than 1000 methods. For some programs such as `compress` and `db`, the SCC contains almost half of all reachable methods. This shows the importance of context-sensitive modeling of recursion, which is impossible to achieve without careful merging of equivalent contexts.

Benchmark	#Methods	#Statements	Max SCC	Suite
compress	2344	47510	1107	SPECjvm98
db	2352	47717	1119	SPECjvm98
jack	2606	56909	1123	SPECjvm98
javac	3520	71132	1191	SPECjvm98
jess	2772	54897	1126	SPECjvm98
mpegaudio	2528	58738	1121	SPECjvm98
mtrt	2485	50541	1118	SPECjvm98
soot-c	4583	78761	1207	Ashes
sablecc-j	8789	146271	1872	Ashes
jflex	4008	80615	1185	-
muffin	4326	82991	1245	-
jb	2393	48277	1106	-
jlex	2423	52622	1097	-
java_cup	2605	54306	1125	-
polyglot	2322	46217	1097	-
antlr	2998	61382	1148	DaCapo
bloat	4994	98750	2055	DaCapo
jython	4136	85607	1710	DaCapo
ps	5278	95790	1711	DaCapo

Table 1: Java benchmarks.

We managed to run our analysis for some (relatively small) programs without applying any tradeoffs. However, the approach did not scale to larger programs such as `soot-c` and `sablecc-j`. By employing the last-k-sub-string merging with $k \leq 2$, the analysis was able to scale to all programs. We present experimental results for the following analyses: 1-object-sensitive analysis [23] with a (1-limited) context-sensitive heap abstraction, implemented with BDDs in Paddle [16] and referred to as *1H* in their work; *Refine*, the refinement-based analysis described in [31] (run without any budget); *1EPA*, our *e*quivalence-based *p*oints-to *a*nalysis with last-1-substring merging; *2EPA*, our analysis with last-2-substring merging; and *Full*, our analysis with last-2-substring merging only within SCCs.

The *1H* algorithm was chosen because [20] showed it to be the most precise of a set of analyses that included the algorithms from [36, 34] and k-callstring analyses. An analysis that could be even more precise than *1H* is the 3-object-sensitive analysis implemented with BDDs in the data race detection tool from [24]. However, at present this tool is not publicly available, and the same analysis implemented in Paddle does not scale to any of our benchmarks.

Although it has been shown in [20] that the k-callstring analyses are less precise than 1H, we implemented a 3-callstring analysis. As mentioned earlier, the analysis was very slow, and the results were imprecise compared to *1H*, which agreed with the conclusions from [20]. The *Refine* algorithm was chosen because it has been shown to be more precise than *1H*, although it may not perform well to compute a complete solution for the entire program [30].

4.1 Analysis Precision

Downcasts safety. The first set of experiments considered the number of downcasts for referenced-type variables proven to be safe. Table 2 shows the total number of casts (#Casts) and how many can be proven safe by the five analyses. A "-" is used if *Full* runs out of memory. For most programs, *Refine* is the most precise algorithm. Our *1EPA* and *2EPA* analyses are much more precise than *1H*, but are not as precise as *Refine*, because many nodes that represent different contexts are merged too early. Our analyses are more precise than *1H* because they do not limit the lengths of context strings, which enables larger-scale heap cloning. As pointed out in [31], *1H* cannot prove a cast as simple as `Iterator i = x.iterator(); o = (T)i.next();` because the elements contained in `x` are more than one (object) level far from `i`. Cloning nodes for only one level does not suffice to precisely treat such

Benchmark	#Casts	1H	Refine	1EPA	2EPA	Full
compress	6	0	**2**	**2**	**2**	**2**
db	24	6	**19**	13	17	17
jack	148	42	**70**	27	49	-
javac	317	40	**63**	49	55	-
jess	66	8	**42**	31	38	**42**
mpegaudio	13	**4**	3	3	**4**	**4**
mtrt	10	4	**5**	4	4	**5**
soot-c	1067	89	**253**	121	142	-
sablecc-j	458	30	**67**	43	62	-
jflex	580	2	**328**	17	43	-
muffin	148	21	**89**	44	69	-
jb	38	2	**34**	13	24	-
jlex	47	3	**27**	7	14	20
java_cup	460	0	**394**	286	372	380
polyglot	9	1	**6**	4	4	5
antlr	81	3	**39**	17	28	-
bloat	1298	79	**153**	125	148	-
jython	458	30	**197**	130	167	-
ps	676	**189**	41	49	49	-

Table 2: Number of downcasts determined to be safe.

Benchmark	#VCS	1H	Refine	1EPA	2EPA	Full
compress	197	194	**195**	**195**	**195**	**195**
db	336	332	322	332	332	332
jack	1190	**1168**	**1168**	1164	**1168**	-
javac	4609	3980	**4284**	4053	4054	-
jess	1604	1559	**1576**	1563	1563	1563
mpegaudio	481	442	442	442	442	442
mtrt	1099	1088	1088	1088	1088	1088
soot-c	5830	4917	**5653**	5526	5599	-
sablecc-j	6050	5648	**5850**	5759	5766	-
jflex	2399	1998	**2393**	2389	2389	-
muffin	2664	2508	**2527**	2510	2510	-
jb	599	460	**579**	538	538	-
jlex	772	771	771	771	771	771
java_cup	2200	2160	**2189**	2183	2184	2184
polyglot	177	163	163	163	163	163
antlr	4451	3611	**4258**	4144	4180	-
bloat	11855	10986	**11624**	11367	11454	-
jython	6050	5685	5686	**5875**	**5875**	-
ps	2361	2054	**2146**	2077	2077	-

Table 3: Number of virtual call sites resolved to a unique target.

widely used data structures. In addition, the last-k-substring merging approach to a large degree prevents nodes in unrealizable paths that share the same sub-context-strings from merging, by allowing nodes to be merged only in the same caller.

Note that *1H* produces much more precise result for `ps`. In this program, a large number of objects are read from a stack object that is pointed to by many variables. Our analysis with $k = 2$ cannot distinguish the elements in the stack, because many symbolic object nodes representing these elements are merged before the objects that the stack variables point to are merged. This is a typical problem of callstring-based analyses when they handle containers — in order to distinguish elements for different containers, nodes representing container elements must not be merged before the nodes representing containers are merged, and this is usually too expensive for analyzing large programs. Our result on this program also agreed with the explanation from [31], because *Refine* also fails to produce precise result for this particular program.

Virtual call resolution. The second set of experiments considered the numbers of virtual call sites that can be statically resolved to a unique target method. Table 3 shows the total number of virtual call sites (#VCS) and the number of resolved sites. *Refine* is still the most precise analysis. Note that typically even *1EPA* can resolve more (or the same number of) virtual call sites than *1H*.

4.2 Analysis Cost

Time and memory. Table 4 shows running time and peak memory consumption. We exclude the measurements for *Full* because

Benchmark	Time (sec)				Memory (Mb)			
	1H	Ref	1EPA	2EPA	1H	Ref	1EPA	2EPA
compress	1570	1249	2+31	53	385	928	320	342
db	1564	1404	3+29	59	402	840	344	350
jack	1771	1869	3+36	741	512	599	455	490
javac	3789	2648	5+312	1480	528	764	1241	1410
jess	2130	2058	3+74	333	318	760	316	361
mpegaudio	1687	1303	2+41	120	389	858	371	500
mtrt	1626	1494	4+51	81	405	282	314	389
soot-c	3720	6256	14+405	2078	588	1456	1278	1976
sablecc-j	4859	10073	12+786	1470	885	881	1548	2100
jflex	4676	4157	4+535	634	667	1498	656	978
muffin	4980	5067	4+125	137	630	1292	912	986
jb	1599	1269	4+139	146	361	1543	324	445
jlex	1565	1234	3+47	87	384	1106	338	412
java_cup	1709	1812	3+110	135	361	606	377	445
polyglot	1545	1125	4+28	58	324	267	302	334
antlr	2280	1696	3+123	399	419	428	579	1213
bloat	6054	4792	8+360	1380	833	793	1498	2411
jython	7115	2423	6+205	377	826	491	1096	1101
ps	3340	3401	5+800	1324	529	1228	1599	2287

Table 4: Analysis time and peak memory consumption.

it did not scale. *1H* was run using javabdd [12] as the backend to manipulate BDDs. According to [17] and also our own experience with small programs, running Paddle with BuDDy [4] as the backend can make the analysis run about 2 times faster. However, for most large programs in our benchmark set, BuDDy crashed the JVM for some unknown reasons. The column for *Refine* shows time and memory that the analysis used to answer queries for all variables in the programs.

Column Time-1EPA shows two numbers, representing the running times of the two analysis phases. Column Time-2EPA column shows the total running time, since the first phases of the two analyses take the same time. The results indicate that both *1EPA* and *2EPA* are much faster than both *1H* and *Refine*. In fact, this would still be the case even if *1H* were run with BuDDy. The time reduction is achieved partially by the non-BDD-based implementation, and partially by the nature of a summary-based analysis — a non-SCC method is processed only once in the interprocedural phase.

In general, *1H* used less memory than *1EPA* and *2EPA*, especially for large programs such as `soot-c` and `sablecc-j`. The major reason is that our analysis, even with $k = 1$, cloned many more nodes than *1H* did. The majority of cloned nodes that reside in memory for a long time are allocation nodes and symbolic object nodes, as they have to be cloned and propagated up if they escape a method. Our analysis only merges equivalent contexts of a pair of variable and object, while a BDD may also be able to exploit the similarity of points-to sets of different variables (even for context-insensitive analysis [3]). This may also explain why the BDD analysis achieves more space efficiency than our analysis. Memory consumed by *Refine* is primarily for caching matched object pairs. Even though our experiments with *Refine* did not record points-to sets of variables (i.e. a points-to set for a variable was discarded immediately after being computed), *Refine* used more memory than our analysis for many programs.

Context string length. Table 5 shows the average context string length for variable nodes (VLen) and allocation nodes (OLen) in the final points-to sets. Thus, each context string associated with a variable or an object represents a URC for that variable or object. The average lengths of the URCs are quite small — for many programs, both lengths are smaller than 1. This supports the observation that a large number of contexts share similarities, which serves as the motivation of using BDDs to represent analysis information. Note that for every program in the table, the lengths of URCs for objects are larger than those for variables, which indicates that on average more clones are needed for objects than for variables.

Benchmark	1EPA		2EPA	
	VLen	OLen	VLen	OLen
compress	0.87	0.93	1.06	1.10
db	0.81	0.91	1.08	1.10
jack	1.53	1.75	1.78	3.86
javac	0.67	0.43	1.56	3.99
jess	0.98	0.99	1.62	1.72
mpegaudio	0.96	1.70	1.10	2.27
mtrt	1.07	1.30	1.11	1.35
soot-c	0.45	0.93	0.89	1.58
sablecc-j	0.39	0.77	0.39	0.87
jflex	1.04	1.16	1.27	1.71
muffin	0.56	0.82	0.67	1.24
jb	0.53	0.83	0.56	1.00
jlex	1.10	1.27	1.42	1.69
java_cup	0.86	0.90	1.04	1.52
polyglot	0.85	1.03	1.00	1.05
antlr	1.15	1.68	2.23	3.45
bloat	0.37	0.80	1.23	1.56
jython	0.74	0.94	0.87	1.69
ps	0.73	1.31	1.00	2.42

Table 5: Average context string length.

5. RELATED WORK

There is a very large body of work on points-to analysis; useful summaries are available in [11, 8, 28]. The discussion in this section is restricted to context-sensitive algorithms that are most closely related to our technique.

Most early context-sensitive pointer analyses were designed for C programs. A recent analysis for C that is close to ours is the algorithm from [25]. This approach is based on the observation that context insensitivity does not lose precision if the program is free of procedural side effects. A bottom-up phase propagates procedure summaries from callees to callers to make the program free of procedural side effects, which is similar to cloning symbolic points-to graph nodes from callees to callers in our analysis. The top-down phase then context-insensitively computes the points-to pairs. The analysis can scale to large programs without applying any tradeoffs, which is impressive for a subset-based context-sensitive points-to analysis with full heap cloning.

One difference with this earlier work is that we propagate symbolic points-to graphs instead of flow graphs in the bottom-up phase, and the points-to relationships are computed during the propagation. As mentioned in Section 3.1, symbolic points-to graphs for callees are immediately removed after they are processed and nodes are cloned, whereas their analysis requires flow graphs of all reachable methods to reside in memory during the bottom-up propagation, which may cause the analysis to run out of memory. From our experience, it is impossible for a Java analysis to keep everything in memory until the bottom-up phase finishes. SPGs also allow us to introduce some unification during propagation, which also contributes to the good scalability of this analysis.

Liang and Harrold [21] propose a unification-based analysis for C that uses parameterized pointer information to compute summary information (similar to our symbolic points-to graph) for a procedure; this information can be instantiated by a client at a specific call site by binding the symbolic names. Lattner et al. [15] develop a context-sensitive, unification-based, heap-cloning-based analysis, and successfully applied it on the entire Linux kernel. Their experimental results suggest that heap-cloning greatly compensated the precision loss caused by unification. Kahlon [13] proposes a staged analysis algorithm that partitions the whole program into small subsets and uses a divide-and-conquer approach that concurrently finds solutions in these subsets. A summarization-based approach is used for scalable context-sensitive alias analysis.

Chatterjee et al. [5] develop a flow-sensitive context-sensitive relevant context inference technique for a Java-like subset of C++. This approach constructs procedural summary functions in a bottom-up phase, and instantiates these functions in a top-down phase. Ruf [27] uses a flow-insensitive context-sensitive unification-based alias analysis to remove redundant Java synchronizations. This analysis uses method summaries to model context sensitivity, and also has a bottom-up phase followed by a top-down phase. Grove and Chambers [8] define a family of analyses that could provide points-to information for Java. In this general framework context sensitivity could be achieved either with functional or with callstring context abstractions [29].

Object-sensitive analysis [22, 23] uses the static abstraction of the receiver object as the abstract context, exploiting common patterns in object-oriented code. Studies have indicated [20, 24] that k-object-sensitive points-to analysis, even for small k, can produce more precise results than callstring-based approaches. We compare experimentally our callstring-based analysis with the BDD-based 1-object-sensitive analysis with heap cloning from [20]. The results indicate that our analysis achieves both better running times and higher precision.

Binary decision diagrams (BDDs) have been used recently to implement points-to analyses [3, 34, 36, 20, 16, 24, 10]. Representative context-sensitive analyses implemented with BDDs are [34, 36], which for the first time allow the use of arbitrary-length callstring contexts. However, these analyses use a context-insensitive heap abstraction, which leads to precision loss [20, 16]. Naik et al. present a race detection tool based on a 3-object-sensitive analysis implemented with BDDs [24]. Its precision relative to *1H* and *Refine* remains to be studied. Lhoták and Hendren [20] identify and measure equivalence classes of abstract contexts, and suggest that the use of BDDs could be avoided if context equivalence classes can be found manually. Hardekopf and Lin [10] develop an optimized subset-based analysis and implement it using both BDDs and bitmaps. Their result suggest that the BDD implementation is two times slower than the bitmap implementation, but is much more space efficient. These two recent reports were the direct inspiration for this paper, leading to our techniques for explicit merging of calling contexts in order to achieve both time and space efficiency.

Refinement-based analyses have been proposed to generate relatively imprecise initial results and then refine parts of the solution based on needs of clients. Guyer and Lin [9] present a points-to analysis for C that takes a client-driven point of view and identifies statements that cause imprecision for this client. Additional analysis is then performed to achieve better sensitivity for those problematic statements. Sridharan and Bodik [31] present a combined demand-driven and refinement-based context-sensitive analysis. Their results suggest that the analysis has lower memory requirements and is more precise than the 1-object-sensitive analysis, when used to answer individual pointer-related queries. Although refinement-based analyses could potentially produce more precise results than analyses that compute a complete points-to solution for the entire program, they may have limited generality. As pointed out in [34], to answer queries as simple as "which variables point to a certain object" would require a complete solution to be computed, and the algorithms used by refinement-based analyses may not scale under these circumstances [30].

Recent work [14] proposes a variant of the classic callstring model of context sensitivity [29]; it would be interesting to see whether our analysis can be restated using this theoretical formulation.

6. CONCLUSIONS

This paper presents a technique that characterizes equivalence classes of calling contexts. Based on this technique, we design a new points-to analysis for Java that merges equivalent contexts

without relying on BDDs. We also propose a last-k-substring-merging tradeoff that allows tuning of analysis cost and precision. An experimental study show that with small values of k the analysis is able to run faster and achieve better precision than an 1-object-sensitive analysis. One direction of future work is to exploit the same ideas for efficient representation of context information for other interprocedural analyses (e.g., context-sensitive side-effect analysis and def-use analysis, similar to the approach used in [23]).

Acknowledgments. We would like to thank the ISSTA reviewers for their valuable and very thorough comments and suggestions.

7. REFERENCES

[1] Ashes Suite Collection, www.sable.mcgill.ca/software.

[2] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, 1996.

[3] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Conf. Programming Language Design and Implementation*, pages 103–114, 2003.

[4] BuDDy Library, freshmeat.net/projects/bdd-buddy.

[5] R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *Symp. Principles of Programming Languages*, pages 133–146, 1999.

[6] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 1–19, 1999.

[7] DaCapo Benchmarks, www.dacapo-bench.org.

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

[9] S. Guyer and C. Lin. Client-driven pointer analysis. In *Static Analysis Symp.*, pages 214–236, 2003.

[10] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Conf. Programming Language Design and Implementation*, pages 290–299, 2007.

[11] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.

[12] Javabdd Library, javabdd.sourceforge.net.

[13] V. Kahlon. Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis. In *Conf. Programming Language Design and Implementation*, 2008.

[14] U. Khedker and B. Karkare. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: Resurrecting the classical call strings method. In *Int. Conf. Compiler Construction*, pages 213–228, 2008.

[15] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Conf. Programming Language Design and Implementation*, pages 278–289, 2007.

[16] O. Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, 2006.

[17] O. Lhoták. Personal communication, Oct. 2007.

[18] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *Int. Conf. Compiler Construction*, pages 153–169, 2003.

[19] O. Lhoták and L. Hendren. Jedd: A BDD-based relational extension of Java. In *Conf. Programming Language Design and Implementation*, pages 158–169, 2004.

[20] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: Is it worth it? In *Int. Conf. Compiler Construction*, pages 47–64, 2006.

[21] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *Static Analysis Symp.*, pages 279–298, 2001.

[22] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Int. Symp. Software Testing and Analysis*, pages 1–11, 2002.

[23] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Software Engineering and Methodology*, 14(1):1–41, 2005.

[24] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Conf. Programming Language Design and Implementation*, pages 308–319, 2006.

[25] E. Nystrom, H. Kim, and W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Static Analysis Symp.*, pages 265–280, 2004.

[26] E. Nystrom, H. Kim, and W. Hwu. Importance of heap specialization in pointer analysis. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 43–48, 2004.

[27] E. Ruf. Effective synchronization removal for Java. In *Conf. Programming Language Design and Implementation*, pages 208–218, 2000.

[28] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Int. Conf. Compiler Construction*, pages 126–137, 2003.

[29] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[30] M. Sridharan. www.sable.mcgill.ca/pipermail/soot-list/2006-January/000477.html.

[31] M. Sridharan and R. Bodik. Refinement-based context-sensitive points-to analysis for Java. In *Conf. Programming Language Design and Implementation*, pages 387–400, 2006.

[32] M. Sridharan, D. Gopan, L. Shan, and R. Bodik. Demand-driven points-to analysis for Java. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 59–76, 2005.

[33] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Int. Conf. Compiler Construction*, pages 18–34, 2000.

[34] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Conf. Programming Language Design and Implementation*, pages 131–144, 2004.

[35] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 187–206, 1999.

[36] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *Conf. Programming Language Design and Implementation*, pages 145–157, 2004.