Field Flow Sensitive Pointer and Escape Analysis for Java as Logic Programs using SSA

Prakash Prabhu
Indian Institute of Science

Priti Shankar
Indian Institute of Science

IISc-CSA-TR-2006-10
http://archive.csa.iisc.ernet.in/TR/2006/10/

Computer Science and Automation
Indian Institute of Science, India

September 2006

Field Flow Sensitive Pointer and Escape Analysis for Java as Logic Programs using SSA

Prakash Prabhu
Indian Institute of Science

Priti Shankar
Indian Institute of Science

September 19, 2006

Abstract

We present a flow and context sensitive combined pointer and escape analysis for Java. Our analysis provides both context and flow sensitivity for Java fields and it does so by extending the Heap Array SSA form. We first express a flow-sensitive pointer analysis as a set of deduction rules on database relations in Datalog, a logic programming language. We then get a context sensitive pointer analysis using techniques developed by Whaley and Lam and introducing new ϕ functions for Heap Array SSA. Adding a few more deduction rules gives an escape analysis. The advantages of this method are two fold: Firstly, a precise points-to analysis directly provides a precise escape analysis result with very little overhead and hence no separate heavy weight analysis to determine escape information needs to be carried out. Secondly, a precise points-to analysis can help in a variety of other compiler optimizations like code motion and dead store elimination. We have implemented this analysis in the $SOOT$ open source framework and used *bddbddb* (Binary Decision Diagram based Deductive Database) - a solver for Datalog rules, to obtain the analysis results. We have run our analysis on a set of 14 Java programs and compared our escape analysis results with a field flow insensitive (FFIS) analysis. Our experimental results show that the running times of our analysis are comparable to a FFIS analysis while doing much better in many cases. At the same time achieving improvements ranging from 21% to 79% in the number of one-level captured objects.

1 Introduction

The memory model of Java requires that all objects be allocated on the heap. These objects can be deallocated only by garbage collection. A similar situation exists for the Java synchronization model. Every object has a lock associated with it to ensure mutual exclusion when a synchronized method is invoked on it. Both heap allocation and synchronization on locks incur performance overhead.

Although, all Java objects are allocated on the heap, there may always be some objects which are allocated within a particular method and are never used once the method exits. If such objects are identified at compile time, they can be allocated on the stack frame of the method instead of the heap. This reduces the garbage collector time since these objects are freed with the unwinding of the method's stack frame. Similarly, if we are able to identify those objects that are accessed only by a single thread, then the unnecessary use of locks can be eliminated.

Escape Analysis [27] [6] is a static compiler analysis technique which identifies (a) objects that are local to a particular method and (b) those that are accessed only by a single thread. For such objects, the compiler can perform stack-allocation and lock elimination respectively. It thus helps to speed up programs by lessening the burden on the garbage collector and synchronization system.

Escape analysis works by determining whether an object (a) may escape a method (ie, is not local to the method that created it) and (b) may escape the thread that created the object (ie, other threads may access the object). If an object does not escape a method, it can be allocated on the method's stack frame. Similarly, if an object does not escape a thread, we can eliminate synchronization associated with the object.

Pointer analysis is a compiler analysis that attempts to statically determine whether two reference variables may point to the same storage location at runtime. There has been a lot of work done for pointer analysis in C in the last two decades. Java, when compared to C is a strongly typed language. Since the explicit manipulation of memory addresses by means of pointer arithmetic is disallowed by Java, there are only a few ways in which two variables can be aliased (two variables are aliases of each other if they may point to a common object), through a combination of reference variable assigns (*copy*), field stores (*putfield*) and field loads (*getfield*). Since Java is an object oriented language, there are several constructs of a Java program which are resolved only at runtime. For example, the presence of virtual method invocations inhibits the construction of a precise call graph at compile time. This affects any interprocedural pointer analysis carried out on a Java program. The pointer analysis for Java is thus quite different from that for C.

Many compiler optimizations require precise information about pointers in order to be effective. One of the most important applications of pointer analysis is to aid in code motion, particularly loop invariant code motion. Pointer analysis also has many other important applications like parallelization and dead store elimination. A precise pointer analysis can also obviate the need for a separate escape analysis and thus help in stack allocation and synchronization elimination. Of late, precision of pointer analysis is becoming more important in software verification and error diagnostics tools.

The rest of the article is organized as follows : Section 2 gives an overview of the issues involved in any pointer analysis. Section 3 describes our combined pointer and escape analysis in detail. Section 4 gives the implementation details and discusses the experimental results. Section 5 gives a discussion of related work and Section 6 presents our conclusions.

2 Pointer Analysis Background

There are a number of issues to be addressed while formulating any pointer analysis. The most important of them are [12] :

1. **Flow Sensitivity**: A flow sensitive analysis takes control flow into account while determining points-to relations at various program points. Since it can avoid generation of spurious points-to relations via non-existent control flow paths, it is important for precision of a pointer analysis.

2. **Context Sensitivity**: Context sensitivity, although in the worst case can lead to exponential analysis times, is important since it ensures that points-to relations from multiple calling contexts do not interfere with each other.

3. **Heap Modeling**: Any pointer analysis abstracts the many potential heap objects created at runtime by some finite number at compile time. The method usually adopted is to abstract the objects by their allocation sites, but shape analysis can provide a finer granularity of the object space. We chose the former since a non-trivial shape-analysis would further complicate the combined pointer and escape analysis.

4. **Aggregate modelling**: Whenever we have structured data types, distinguishing between their different components can lead to an accurate pointer analysis. In our analysis, we distinguish between the different fields of an object (and hence our analysis is field sensitive), while we do not distinguish between different elements of higher dimensional aggregates like arrays, though this analysis can be extended to accurately model arrays too, using the methods specified in [10].

2.1 Flow Sensitivity and SSA

The one main drawback using a flow sensitive analysis is that we need to maintain a points-to graph at every program point and iteratively propagate these graphs through the control flow graph until a fixed point is reached. This is costly both in terms of time and space. One way to reduce the cost is to use the Static Single Assignment (SSA) [7] intermediate representation form for pointer analysis.

An important advantage of SSA is that irrespective of whether an analysis performed on SSA program is flow sensitive or flow insensitive, we get similar results [11]. This is because, the SSA form encodes to a certain extent the control flow of the program in its subscripted variables. Hence it is enough to maintain a *single* points-to graph for the whole method. If the cost of translating a program into SSA form and then running a flow insensitive algorithm is less than a flow sensitive algorithm on the original program, then this approach could be used. However, translating a program into SSA itself may require pointer analysis, due to the presence of pointer variables in the original program. The general approach adopted is to :

1. Do a flow insensitive pointer analysis on the original program
2. Use the analysis results for a conservative translation of the program into SSA while taking taking special care of pointer update statements
3. Run a flow insensitive analysis on the translated SSA form and use it to refine Step 1

In [11], the authors give one such algorithm for C. Their algorithm can be tuned for precision by controlling the number of times the above steps are carried out.

Taking the case of Java, if a program uses no fields, the normal SSA form can be used. However the use of fields in Java give rise to the same issues as the use of pointer variables in C. Fields act as containers holding references to other objects. Hence, the normal SSA translation as applied to reference variables has to be extended to handle fields. This is done by the Heap Array SSA form [10]. In our work, we make use of this Heap Array SSA form and extend it interprocedurally over the super graph of a program by introduction of two more ϕ functions. We then explicitly represent the heap arrays along with the SSA numbers in the logic program that expresses our points-to analysis. Details of the Heap Array SSA form are mentioned in Section 2.5.

2.2 Context Sensitivity

To determine the points-to relations that may hold at any program point even in the presence of procedure calls, an interprocedural analysis has to be carried out. While an intraprocedural analysis is done by fixed point iteration over the control flow graph, an interprocedural analysis is done on the interprocedural control flow graph(ICFG) or the program supergraph [21].

The one important difference between data flow analysis on a supergraph and that carried out on a control flow graph is the presence of unrealizable paths in the supergraph, ie, those paths that begin at an invocation site a, flow through the method m that is invoked by the call, then return to a point after some other invocation site $b \neq a$. Data flow information if propagated along these paths can lead to a high degree of imprecision.

If an analysis distinguishes between different calling contexts and does not allow data flow information from multiple contexts to interfere with each other while avoiding unrealizable paths, it is context sensitive.

There are two main techniques for doing context sensitive pointer analysis:

- *Summary Based Analysis* : This technique operates in two phases. In the first phase a parameterized summary of every method is constructed in a bottom up order. In the next phase, the data flow facts are propagated in a top down fashion, during which the parameters for the summaries are plugged in as determined by the calling context. The summaries are computed only once for every method, but evaluated separately for every calling context. Examples of summary based analysis are: [27], [6] and [8].

- *Cloning Based Analysis* : This is a simple brute force technique where every path in the call graph is cloned, creating a clone of every method for every distinct context in which it can be called. A context insensitive algorithm run on these cloned paths of the expanded call graph gives a context sensitive analysis result. The problem with this approach is there could be an exponential number of cloned paths due to a large number of calling contexts in the program. However, usually a concise representation of these cloned paths is possible using a data structure like the Binary Decision Diagram [5]. Examples of this kind of analysis are: [26] and the call strings approach of [21].

We are using the analysis of [26] for achieving context sensitivity due to the following reasons:

- Our analysis aims at adding flow sensitivity to fields and locals of a method. A summary represents the points-to relations which hold at the exit point of the method while we require the points-to relations that hold even at intermediate points of the method.
- Summary based analysis represents the points-to relations in the form of variations of points-to graphs which can become quite big, while the cloning based analysis encodes the points-to relations in the boolean domain compactly in a reduced form.

2.3 Pointer Analysis as Logic Programs

Any program analysis which is expressed in the form of a set of constraint based rules can be expressed in a logic programming language [19]. Datalog is one such language. A Datalog program consists of a set of *deduction rules* on the tuples of a set of relations.

Given the nature of the points-to relations and constraints based on which they are built, points-to analysis can be naturally expressed as a logic program. The Java statements that affect the building of points-to relations (viz, *assign*, *load* and *store*, *new*) are given as input relations to logic program while the points-to relation is an output relation generated by the analysis. The transfer functions for these statements are then formulated as deduction rules in a Datalog program [26]. To arrive at the final set of points-to relations, the deduction rules are repeatedly applied on the initial and the newly generated relation tuples until a fixed point is reached (ie, there are no more new tuples generated in the output relations). Every application of a deduction rule is done by a sequence of relational algebra operations : *natural join, union, project, rename, difference and select* [25].

The analysis of [26] builds over Andersen's pointer analysis [2] by adding context numbers to the pointer relations. However in doing so, the size of these relations grow to a large extent due to the exponential number of contexts that are generated. To deal with this problem, these relations are interpreted as boolean expressions and represented efficiently by means of binary decision diagrams [5][3].

2.4 Binary Decision Diagrams (BDD)

BDDs are efficient representations of boolean functions. Any boolean function can be represented by a directed acyclic graph with a single root node and two terminal nodes, 0 and 1. The internal nodes are the variables of the boolean function under consideration. Each internal node has two children, labeled as the *low* child and the *high* child. The edge from a node to its low (high) child represents an assignment of 0 (1) to the variable corresponding to that (parent) node. A path from the root to the terminal labeled 1 represents a variable assignment for which the represented boolean function is true.

The most important aspect in points-to analysis where the power of BDDs comes to the fore is with respect to context sensitivity. Since BDDs efficiently represent large sets of redundant data, the points-to information which is common across different contexts (of which there are an exponential number) is shared at the data structure level. Also, the relational algebra operations for solving the deduction rules can be efficiently performed on BDDs since the cost of BDD operations (*and, or, difference, exist and replace*) depends on the size and shape of the BDD graphs and not on the number of tuples in the relation [25].

BDDs can be used to represent relations that are used in the points-to analysis. Every element d in an n-element domain is can be represented using $log_2(n)$ bits. A relation $R : D_1 \times ... \times D_n$ is represented as a boolean function $f : D_1 \times ... \times D_n \rightarrow \{0, 1\}$ such that $(d_1, ..., d_n) \in R$ iff $f(d_1, ..., d_n) = 1$ and $(d_1, ..., d_n) \notin R$ iff $f(d_1, ..., d_n) = 0$.

There is however one potential issue in using BDDs, that could theoretically be encountered. The size of a BDD depends on the variable ordering which is used while constructing the BDD. In general, the size could be as bad as the binary tree representation in case a bad ordering of the variables is chosen. Hence a proper heuristic ordering of the variables is necessary to prevent such an exponential representation.

2.5 Flow sensitivity for Fields : Heap Array SSA

The normal SSA form when directly applied to a Java program having fields does not precisely give a flow-sensitive analysis result since no distinction is made in the different instances of the object fields in the scalar form of SSA. There have been a number of extensions to scalar SSA to handle object fields. For our analysis, we chose the Heap Array SSA form [10].

Algorithm 1 Pointer Analysis using Object Numbering

1. counter=0;
2. For every *new* site in a method,
   Assign a unique number to the heap object h
   created at this site:
   $PointsTo(h) = \{counter++\}$
3. W $\leftarrow$ { st | st is a new statement of the form
   x = *new h()* }
4. while (W $\neq \Phi$) {

```
W = W - st;
switch (st)
{
   case new (x = new h()) :
         PointsTo(x) = PointsTo(x) ∪ PointsTo(h);
   case assign ( x = y ) :
         PointsTo(x) = PointsTo(x) ∪ PointsTo(y);
   case store ( H_i^f[x] = y ) :
         ∀n ∈ PointsTo(x),
            PointsTo(H_i^f[n]) = PointsTo(H_i^f[n]) ∪ PointsTo(y);
   case load ( x = H_i^f[y] ) :
         ∀n ∈ PointsTo(y),
            PointsTo(x) = PointsTo(x) ∪ PointsTo(H_i^f[n]);
   case dφ(H_i^f = dφ(H_j^f, H_k^f)) :
         X = Φ
         For every valid index n
         ( ∀ n such that PointsTo(H_j^f[n] ≠ Φ)),
            X = X ∪ { n }
            PointsTo(H_i^f[n]) = PointsTo(H_j^f[n])
         For every valid index m ∉ X of H_k^f,
            PointsTo(H_i^f[m]) = PointsTo(H_k^f[m])
   case cφ(H_i^f = cφ(H_{j1}^f, H_{j2}^f, ..., H_{jm}^f) :
         For every valid index n of H_{jk}^f (1 ≤ k ≤ m),
            PointsTo(H_i^f[n]) = PointsTo(H_{jk}^f[n])
   case mφ(a_i = cφ(a_{j1}, a_{j2}, ..., a_{jn})) :
         PointsTo(a_i) = PointsTo(a_i) ∪ PointsTo(a_{jk})
                         ∀ k such that (1 ≤ k ≤ n)
}
If the PointsTo set computed in the previous step
is different from the earlier PointsTo,
   W = W ∪ { s | s ∈ SSASuccessor(st) }
```

}

The heap array SSA form for fields is an extension of the Array SSA form introduced by Sarkar et al [13]. Array SSA form introduces a new ϕ function called the $define - \phi$ or $d\phi$ function which is inserted immediately after every definition of an array element(s). This function merges the values of the elements modified in the definition with the values that are not killed and available immediately before the definition. Apart from the the $d\phi$ function, the array SSA form also carries over the *control-*ϕ or $c\phi$ function from the scalar SSA which is inserted exactly at the same location (dominance frontier) as would have been inserted by the scalar SSA algorithm.

Heap Array SSA applies the array SSA form for Java objects. Accesses to an object field f are modeled by defining a one-dimensional heap array H^f. This heap array represents all instances of the field f that exists on the heap. Heap arrays are indexed by object references. A *load* of $p \cdot f$ is modeled as read of element $H^f[p]$ and the *store* of $q \cdot f$ is modeled as a write of element $H^f[q]$. Since there is no aliasing of fields in Java (accesses to two distinct fields refer to two distinct memory locations), heap arrays for two different fields can be regarded as truly distinct.

Converting a Java program into Heap Array SSA form and running a flow insensitive pointer analysis on it generates a flow sensitive pointer analysis result for both fields and local variables. The next section describes the object numbering algorithm we use in our analysis, for finding out points-to relations.

2.6 Object Numbering Algorithm

Algorithm 1 gives the object numbering algorithm used in our analysis. It is an adaptation of the global value numbering algorithm which was originally used to find equivalences between program variables [1]. Initially all the abstract heap objects defined by their allocation sites are assigned a unique object number. Then a worklist based algorithm propagates these object numbers to the reference variables and heap arrays through the edges of the value graph. The value graph would be constructed only for those statements that affect pointer analysis. The propagation is done until no more changes to the points-to relation occur.

Every reference variable is associated with a *PointsTo* set, which contains the list of object numbers pointed to by the variable. Similarly, every heap array element (heap array is indexed by an object number) has its own *PointsTo* set which has the object numbers pointed to by the corresponding object field as determined by object index and the field of the heap array. At the end of the analysis, the *PointsTo* set for the heap arrays and reference variables in SSA form give a flow-sensitive points-to relation.

3 Combined Pointer and Escape Analysis as Logic Programs

This section gives a detailed description of our combined pointer and escape analysis.

3.1 Motivation

Considering that there is much in common between escape analysis and pointer analysis, it is interesting to observe the subtleties involved in the relation between the two analyses. Consider the code shown in Listing 1. It shows a Java class *Complex* with four methods *multiply, square, shift and shiftData*. Consider the case when *multiply* method is called from *square* method. In this calling context, both the parameters passed to *multiply* are aliases of each other. However, in general, aliasing between parameters is dependent on the context is which a method is called.

```
public class Complex {

  private Integer real;
  private Integer imaginary;
  protected Complex origin;

  Complex(Integer r, Integer i)
  {
    real = r;
    imaginary = i;
  }

  public static Complex multiply(Complex a,Complex b)
  {
    Integer u = a.real;
    Integer v = b.real;

    Integer w = a.imaginary;
    Integer x = b.imaginary;

    int r = u.intValue() * v.intValue()
            - w.intValue() * x.intValue();

    int i = w.intValue() * v.intValue()
            + u.intValue() * x.intValue();

    u = new Integer(r);
    v = new Integer(i);
    Complex t = new Complex(u,v);
```

```
    return t;
  }

  public static Complex square(Complex a)
  {
    return multiply(a,a);
  }

  public void shift(Integer shiftVal)
  {
    Complex oldorigin = origin;
    this.origin = new Complex(shiftVal, shiftVal); // O1
    this.shiftData();
    this.origin = oldorigin;
  }

  public void shiftData()
  {
    int r = real.intValue();
    int i = imaginary.intValue();
    int scaler = origin.real.intValue();
    int scalei = origin.imaginary.intValue();
    r = r + scaler;
    i = i + scalei;
  }
}
```

Listing 1. Relation between Pointer and Escape Analysis

The points-to escape graph of [27] would represent the the two parameters nodes by two *different* outside nodes. Since a object is stack allocated only when it does not escape in all contexts in which the method is called, there is no need for distinction between the points-to relations for the different calling contexts. Thus, escape analysis cannot effectively answer questions like "When *multiply* is called from *square*, are the loads $b{\cdot}real$ and $b{\cdot}imaginary$ redundant ?".

Consider points-to analysis result of [26] for the method *shift*. Since the analysis is flow insensitive for fields, the updates done for fields are weak. At the end of the *shift*, the analysis would say that $this{\cdot}origin$ could point to the object *O1*. It would then conservatively infer that the object *O1* escapes the method although actually it doesn't due to the store statement at the end of the method.

So, although escape analysis is flow sensitive, it is not completely context sensitive and hence would not be precise enough for doing most of the compiler optimizations for which context sensitive pointer analysis is important. At the same time, although points-to analysis of [26] is context sensitive, it is not field flow sensitive and hence when this analysis is used for determining escape information, a less precise result is likely to be obtained.

If a pointer analysis is both flow and context sensitive, then not only can it generate a precise points-to information but also help in getting a precise escape analysis. In our work, we do one such analysis. The advantages of this method are: Firstly a precise points-to analysis directly provides a precise escape analysis result with very little overhead and hence no separate heavy weight escape analysis needs to carried out. Secondly, a precise points-to analysis can help in a variety of other compiler optimizations like code motion and dead store elimination.

3.2 Field Flow sensitive Pointer Analysis

This section expresses the field flow sensitive intraprocedural pointer analysis algorithm of Section 2.6 (Object Numbering algorithm) in Datalog. We use the same notation for the logic programs as [26].

DOMAINS

V	variable.map
H	heap.map
F	fields.map

RELATIONS

input	*new*	(v: V, h: H)
input	*assign*	(v_1: V, v_2: V)
input	*load*	(d: V, f: F, s: N, b: V)
input	*store*	(v: V, f: F, s: N, b: V)
input	*dphi*	(f: F, s_2: N, s_1: N, s_0: N)
input	*cphi*	(f: F, s_2: N, s_1: N)
input	*mphi*	(v_2: V, v_1: V)
output	*vPointsTo*	(v: V, h: H)
output	*hPointsTo*	(f: F, s: N, h_1: H, h_2: H)

RULES

$$\begin{aligned}
vPointsTo(v_1, h) \quad &:- \quad new(v_1, h) && (1)\\
vPointsTo(v_1, h) \quad &:- \quad assign(v_1, v_2),\\
& \qquad vPointsTo(v_2, h) && (2)\\
vPointsTo(v_1, h_2) \quad &:- \quad load(v_1, f, s_f, v_b),\\
& \qquad vPointsTo(v_b, h_1),\\
& \qquad hPointsTo(f, s_f, h_1, h_2) && (3)\\
hPointsTo(f, s_f, h_1, h_2) \quad &:- \quad store(v_1, f, s_f, v_b),\\
& \qquad varPointsTo(v_b, h_1),\\
& \qquad varPointsTo(v_1, h_2) && (4)\\
hPointsTo(f, s_2, h_1, h_2) \quad &:- \quad dphi(f, s_2, s_1, -),\\
& \qquad hPointsTo(f, s_1, h_1, h_2) && (5)\\
hPointsTo(f, s_2, h_1, h_2) \quad &:- \quad dphi(f, s_2, s_1, s_0),\\
& \qquad hPointsTo(f, s_0, h_1, h_2),\\
& \qquad hPointsTo(f, s_1, h_3, -),\\
& \qquad h_1 \neq h_3 && (6)\\
hPointsTo(f, s_2, h_1, h_2) \quad &:- \quad cphi(f, s_2, s_1),\\
& \qquad hPointsTo(f, s_1, h_1, h_2) && (7)\\
vPointsTo(v_1, h) \quad &:- \quad mphi(v_2, v_1),\\
& \qquad vPointsTo(v_1, h) && (8)
\end{aligned}$$

The semantics of the domains are the same as in [26]. The relations used in our analysis are also very similar, with the changes being additional attributes to accomodate the SSA numbers of the Heap Arrays and relations for the $d\phi$ and $c\phi$ statements:

1. *new* : It represents the initial points-to relation created at all the allocation sites. There is one tuple corresponding to each allocation site. For every new site $v_1 = new \ \ h()$, there is a tuple (v_1, h) in this relation.

2. *assign* : Represents the assignment statements. For every statement of the form $v_1 = v_2$, there is a tuple (v_1, v_2) in the *assign* relation.

3. *load* : Represents the load or the *getfield* statements. For every statement of the form $v_2 = H^f_{ssa_f}[v_1]$, there is a tuple (v_2, f, ssa_f, v_1) in this relation.

4. *store* : Represents the store or the *putfield* statements. For every statement of the form $H^f_{ssa_f}[v_1] = v_2$, there is a tuple (v_2, f, ssa_f, v_1) in this relation.

5. *dphi* : Represents the $d\phi$ statement. Since every $d\phi$ expression has only two arguments, a $d\phi$ statement is always of the form : $H^f_{s_2} = d\phi(H^f_{s_1}, H^f_{s_0})$.

For every such statement, there is a tuple of the form (f, s_2, s_1, s_0) in the *dphi* relation.

6. *cphi* : Represents the $c\phi$ statement. For every statement of the form $H^f_{s_0} = c\phi(H^f_{s_1}, H^f_{s_2}, ..., H^f_{s_n})$, there are tuples (f, s_0, s_i) $\forall i$ such that $1 \leq i \leq n$ in the *cphi* relation.
7. *mphi* : Represents the scalar $m\phi$ statement. For every statement of the form $v_0 = m\phi(v_1, v_2, ..., v_n)$, there are tuples (v_0, v_i) $\forall i$ such that $1 \leq i \leq n$ in the *mphi* relation.

The deduction rules capture the effect of the various statements that affect the building of the points-to relations. The semantics of the rules are as follows :

3.2.1 Rule (1) for new statement : $v_1 = new\ \ h()$

This rule creates the initial points-to relation for the reference variables.

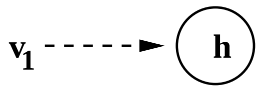

Figure 1: Effect of *new* statement

3.2.2 Rule (2) for assign statement $v_1 = v_2$

This rule updates the points-to set for v_1 based on the inclusion property that
$$points - to(v_2) \subseteq points - to(v_1)$$
The dotted edges are the ones that are newly created due to the particular statement under consideration.

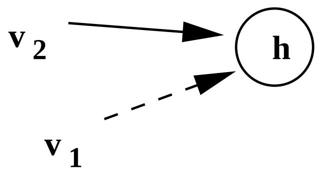

Figure 2: Effect of *assign* statement

3.2.3 Rule (3) for load statement $v_1 = H^f_{s_f}[v_b]$

This rule updates the points-to set for v_1 using the points-to set of $H^f_{s_f}[h1]$ where *h1* is the object pointed to by the index variable v_b of the Heap array instance $H^f_{s_f}$.

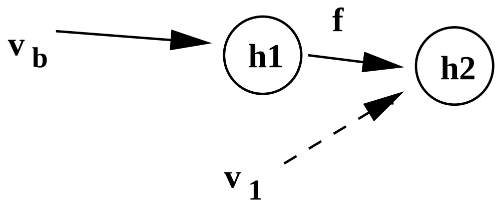

Figure 3: Effect of *load* statement

3.2.4 Rule (4) for store statement $H^f_{s_f}[v_b] = v_1$

This rule updates the points-to set for $H^f_{s_f}[h1]$ using the points-to set of v_1 where *h1* is the object pointed to by the index variable v_b of the Heap array instance $H^f_{s_f}$.

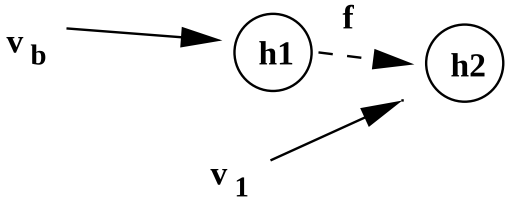

Figure 4: Effect of *store* statement

3.2.5 Rules (5) and (6) for dϕ statement : $H^f_{ssa2} = d\phi(H^f_{ssa1}, H^f_{ssa0})$

These rules are the ones which are crucial for adding flow sensitivity for fields. Whenever a dϕ statement is encountered, the points-to sets for lhs heap array instance H^f_{ssa2} is constructed from its arguments H^f_{ssa1} and H^f_{ssa0} as follows : All the points-to sets corresponding to the object indices of H^f_{ssa1} are carried over to H^f_{ssa2} while the points-to sets for the remaining object indices for H^f_{ssa2} come from H^f_{ssa0}. Whenever there is an object index common to both H^f_{ssa1} and H^f_{ssa0}, the points-to set from H^f_{ssa1} prevails. Note the ordering of the arguments for the dϕ argument is important and the heap array instance H^f_{ssa1} corresponds to the one that is defined in the *store* statement immediately before this dϕ statement.

3.2.6 Rule (7) for cϕ statement : $H^f_{ssa} = c\phi(H^f_{ssa_1}, H^f_{ssa_2}, ..., H^f_{ssa_n})$

The cϕ statement is represented as a set of tuples (f, ssa, ssa_i) $\forall i$ such that $1 \leq i \leq n$ in the *cphi* relation since all the arguments of cϕ are symmetric with respect to the lhs heap array instance. The effect of the *cphi* statement

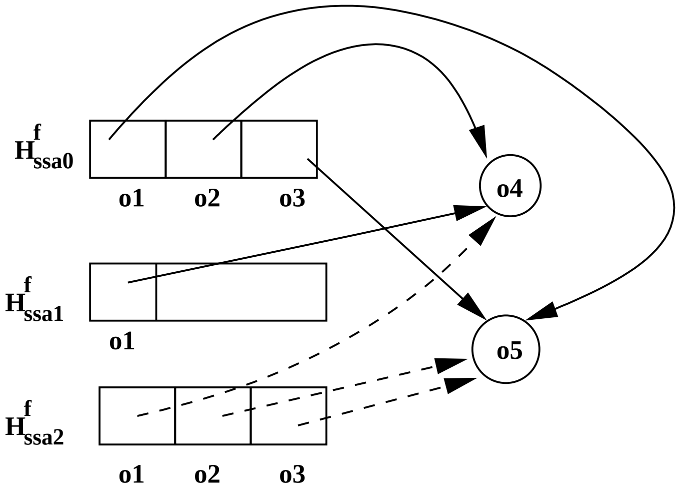

Figure 5: Example Effect of $d\phi$ statement $H^f_{ssa2} = d\phi(H^f_{ssa1}, H^f_{ssa0})$

is to merge the points-to sets of corresponding to all valid object indices of its arguments.

3.2.7 Rule (8) for mϕ statement : $v_0 = m\phi(v_1, v_2, ..., v_n)$

The effect of this statement is to merge the points-to sets corresponding to the arguments of the mϕ function into the points-to set of the lhs local variable.

3.3 Adding context sensitivity

We use the analysis of [26] for extending our analysis to accommodate context sensitivity. Since updates made to object fields are visible across methods, we have to handle :

- Effects of fields updates made by a called method which would be visible at the caller at the point of return
- Flow of correct object field values into a method depending on the invocation site from where it is called

We extend the Heap Array SSA form to the interprocedural case by introducing two new ϕ functions

- The invocation ϕ function : iϕ
- The return ϕ function : rϕ

The iϕ models the flow of values into a method corresponding to invocation site from where it is called. It selects exact points-to set that exists at the

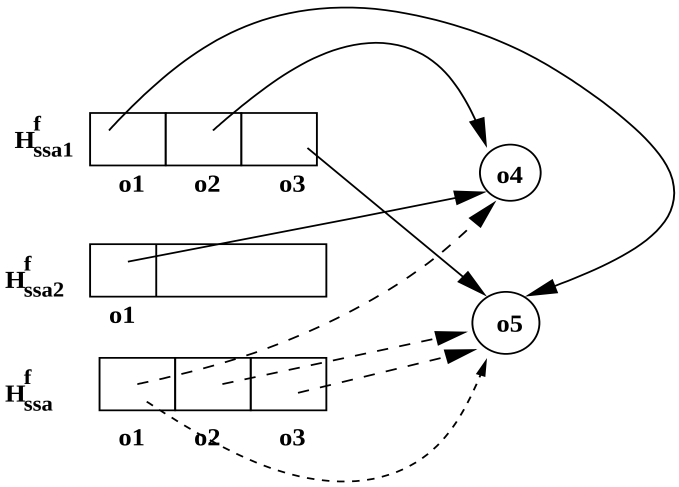

Figure 6: Example Effect of $c\phi$ statement $H^f_{ssa} = c\phi(H^f_{ssa1}, H^f_{ssa2})$

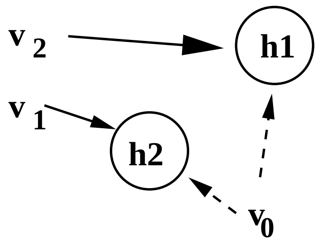

Figure 7: Example Effect of $m\phi$ statement $v_0 = m\phi(v_1, v_2)$

invocation site (at the point of call), on the basis of the invocation edge in the call graph. This is in contrast with the $c\phi$ function which merge the points-to sets flowing via different control flow paths at the point where it is placed. The iϕ function has the following form:

$$H^f_{ssa_i} = i\phi(H^f_{ssa_1}[e_1], H^f_{ssa_2}[e_2], ..., H^f_{ssa_n}[e_n])$$

where $H^f_{ssa_1}, H^f_{ssa_2}, ..., H^f_{ssa_n}$ are the heap array instances live at the point of call and $e_1, e_2, ..., e_n$ are the corresponding invocation edges in the call graph.

The rϕ models the merge of the points-to sets of the heap arrays that existed before the call was made and the points-to set of the heap arrays that were modified by the call that was made. The updated heap array after the call models the effect of all the methods transitively called in the call chain. It has the following form:

$$H^f_{ssa_r} = r\phi(H^f_{ssa_1}[e_1], H^f_{ssa_2}[e_2], ..., H^f_{ssa_n}[e_n], H^f_{s_{local}})$$

The presence of more than one edge in the rϕ function is due to virtual method invocations. $H^f_{ssa_i}$, $\forall i$ such that $1 \leq i \leq n$ is the heap array instance that is live at the end of the called method, with corresponding invocation edge e_i. Thus, for every possible concrete method that can be called at the call site, the rϕ collects the heap array instances for the field f, live at the end of the called method and merges the points-to sets into $H^f_{ssa_r}$. Those objects whose field f has not been modified get their points-to sets into $H^f_{ssa_r}$ from $H^f_{s_{local}}$, the heap array instance for f live in the calling method before the call was made.

3.3.1 Placement of $i\phi$ and $r\phi$ functions

The $i\phi$ functions are placed at the entry point of a method. It is not required to place a $i\phi$ function for every heap array at the start of a method. Only those heap arrays that are *used* or $modified$ in the current method and all the methods it calls transitively would require a $i\phi$ function to be placed at the entry point. Similarly, a $r\phi$ function is required only for those heap arrays which are $modified$ transitively by a method call. The list of heap arrays for which $r\phi$ and $i\phi$ are required can be determined while performing a single traversal on the super graph of the program. This traversal is done in reverse topological order of the nodes of the reduced call graph (a call graph whose strongly connected components have been collapsed into single nodes).

3.3.2 Computing the Live Heap Array Instances

Although the previous step would have determined the placement of the $i\phi$ and $r\phi$ functions, still we have to determine their arguments and rename the heap array instances used in the arguments. For this we do a intraprocedural data flow analysis with a maximum of two iterations to compute the set of live heap array instances at every point in the method. Since in SSA, every heap array has a unique definition site, at most one SSA instance is live at any program point. At the end of the analysis, the heap array instances live at the *point of call* are used for plugging in the arguments of $i\phi$ while the heap array instances live at the *exit points* of a method are used to plugging in the arguments of $r\phi$.

3.3.3 Overall ϕ placement and Renaming

The placement of the *phi* functions ($m\phi$, $d\phi$, $c\phi$, $i\phi$ and $r\phi$) and their renaming are performed in the following steps:

1. Place the $d\phi$, $c\phi$ and the $m\phi$ functions using the dominance frontiers as done in [7]

2. Place the $r\phi$ and $i\phi$ functions using steps mentioned in Section 3.3.1

3. Apply Algorithm of [7] to rename the Heap Array Instances (results of $d\phi$, $c\phi$, $r\phi$, $i\phi$ and arguments of $d\phi$ and $c\phi$) and local variables (results and arguments of $m\phi$. Now only the arguments of $r\phi$ and $i\phi$ have to be decided).

4. Use the steps mentioned in Section 3.3.2 to get the live Heap Array Instances at the exits and at the call sites to plug in the arguments for $r\phi$ and $i\phi$ functions respectively

3.4 Context Sensitive Analysis as Logic Programs

We extend field flow sensitive analysis described in Section 3.2 to incorporate calling contexts by using the context sensitive analysis of [26]. The additional complications are due to the presence of $i\phi$ and $r\phi$ functions. We describe only those relations and rules that pertain to them. The rest of the relations and rules are context sensitive extensions of the relations and rules mentioned in Section 3.2 and those for parameter and return value bindings, invocation edge representations [26].

```
ADDITIONAL DOMAINS
 I                        invoke.map
 M                        method.map
 C
ADDITIONAL RELATIONS
 input   IE_c        (c1: C, i: I, c2: C, m: M)
 input   iphi        (f: F, s_i: Z, s_2: Z, i: I)
 input   rphi        (f: F, s_r: Z, s_l: Z, s_i: Z, i : I)
 output  vPointsTo   (c: C, v: V, h: H)
 output  hPointsTo   (c: C, f: F, s_f: N, v: V, h: H)
```

Two main relations that are new to this analysis as compared to [26] are the *iphi* and *rphi*. Other relations are context sensitive extensions of the *vPointsTo* and *hPointsTo* relations and one for invocation edge representations.

1. *iphi* : Represents the $i\phi$ statement. For every statement of the form $H^f_{s_0} = i\phi(H^f_{s_1}[e_1], H^f_{s_2}[e_2], ..., H^f_{s_n}[e_n])$, there are tuples (f, s_0, s_i, e_i) $\forall i$ such that $1 \leq i \leq n$ in the *iphi* relation.

2. *rphi* : Represents the $r\phi$ statement. For every statement of the form $H^f_{s_0} = i\phi(H^f_{s_1}[e_1], H^f_{s_2}[e_2], ..., H^f_{s_n}[e_n], H^f_{s_{local}})$, there are tuples $(f, s_0, s_{local}, s_i, e_i)$ $\forall i$ such that $1 \leq i \leq n$ in the *rphi* relation.

RULES

$$
\begin{aligned}
hPointsTo(c_2, f, s_i, h_1, h_2) \quad &:- \quad iphi(f, s_i, s_1, i), \\
& IE_c(c_1, i, c_2, -), \\
& hPointsTo(c_1, f, s_1, h_1, h_2). && (9) \\
hPointsTo(c_1, f, s_r, h_1, h_2) \quad &:- \quad rphi(f, s_r, -, s_i, i), \\
& IE_c(c_1, i, c_2, -), \\
& hPointsTo(c_2, f, s_i, h_1, h_2). && (10) \\
hPointsTo(c_1, f, s_r, h_1, h_2) \quad &:- \quad rphi(f, s_r, s_1, s_i, i), \\
& IE_c(c_1, i, c_2, -), \\
& hPointsTo(c_1, f, s_1, h_1, h_2), \\
& hPointsTo(c_2, f, s_i, h_3, -), \\
& h_1 != h_3. && (11)
\end{aligned}
$$

The semantics of these rules are as follows:

3.4.1 Rule (9) for $i\phi$ statement

This rule models the effect of the $i\phi$ statement. When there is a change in context from c_1 to c_2 due to an invocation site i, the heap array instance for a field f live in context c_2 (the lhs of the $i\phi$ statement with ssa number s_i) inherits its points-to set from the heap array instance live in c_1 before the call was made (argument s_1 corresponding to the invocation edge i in the $i\phi$ statement). The presence of IE_c makes sure that points-to set of multiple calling contexts don't interfere with each other.

3.4.2 Rules (10) and (11) for $r\phi$ statement

These rules are similar to Rules (5) and (6) that model the effect of $d\phi$ statement. Rule (10) makes sure that the points-to sets of the live heap array instance (s_i in context c_2) from a virtual method invocation (invocation edge i) are merged into that of lhs heap array instance (s_r) in context c_1. For those objects whose field f has not been changed due to the call, Rule (11) ensures that the lhs heap array instance gets the points-to sets from the local heap array instance live before the call (s_1 in context c_1).

3.5 Escape Analysis for Methods

The escape analysis for methods can be derived from the earlier rules by adding a few more relations and deduction rules. In addition, every class used in the program is uniquely numbered in the same domain as heap objects. We also require the heap array instances that are found to be live at the exits of methods to be encoded in a relation.

ADDITIONAL RELATIONS

input	*liveHArray*	(m: M, f: F, s: Z)
input	*threadParam*	(v: V)
input	*callEdge*	(m_1: M, m_2: M)
input	*allocated*	(h: H, m: M)
input	*classNode*	(c: V, h: H)
output	*escapes*	(h: H, m: M)
output	*aEscapes*	(h: H, m: M)
output	*captured*	(h: H, m: M)
output	*recaptured*	(h: H, m: M)

The semantics of these relations are as follows :

1. *liveHArray*: Represents the Heap Array SSA Instance(s) corresponding to field f that is live at the exit of method m.

2. *threadParam*: Represents the locals passed as a parameter to a thread object's method call.

3. *callEdge*: Represents the call edges in the call graph. Whenever there is a call from a method m_1 to a method m_2, there is a tuple (m_1, m_2) in the *callEdge* relation.

4. *allocated*: Represents the objects that are allocated within their corresponding methods

5. *classNode*: Represents a class used in the program. A class is assigned a number in both the variable and heap domains

6. *escapes*: Represents the set of objects escaping a method irrespective of whether the object was allocated in the method or not

7. *aEscapes*: Represents the set of objects escaping the methods in which they were *allocated*.

8. *captured*: Represents the objects allocated in a method and which do not escape from it.

9. *recaptured*: Represents the set of objects captured one-level higher in the call chain, ie, the objects which escape the method in which they are allocated but do not escape an immediate caller. Such objects can be stack allocated in the caller.

RULES

$$
\begin{aligned}
escapes(h_2, m) \quad &:- \quad liveHArray(m, f, ssa), \\
&\qquad classNode(-, h_1), \\
&\qquad hPointsTo(-, f, ssa, h_1, h_2) && (12) \\
escapes(h, m) \quad &:- \quad vPointsTo(-, v, h), \\
&\qquad formal(m, -, v). && (13)
\end{aligned}
$$

$$
\begin{aligned}
escapes(h, m) \quad &:- \quad return(m, v), \\
&\qquad vPointsTo(-, v, h). && (14) \\
escapes(h, -) \quad &:- \quad threadParam(v), \\
&\qquad vPointsTo(-, v, h). && (15) \\
escapes(h_2, m) \quad &:- \quad escapes(h_1, m), \\
&\qquad hPointsTo(-, f, ssa, h_1, h_2), \\
&\qquad liveHArray(m, f, ssa). && (16) \\
aEscapes(h, m) \quad &:- \quad escapes(h, m), \\
&\qquad allocated(h, m). && (17) \\
captured(h, m) \quad &:- \quad !aEscapes(h, m), \\
&\qquad allocated(h, m). && (18) \\
recaptured(h, m_1) \quad &:- \quad !escapes(h, m_1), \\
&\qquad aEscapes(h, m_2), \\
&\qquad callEdge(m_1, m_2). && (19)
\end{aligned}
$$

The semantics of these rules are as follows :

3.5.1 Rules (12) to (15) : Direct Escape

These rules determine the set of objects that directly escape the method m under consideration. These are

1. objects whose reference has been stored in a static class variable (Rule 12)
2. objects representing the parameters (Rule 13)
3. objects which are returned from a method (Rule 14)
4. thread objects and objects passed to thread methods (Rule 15)

3.5.2 Rules (16) to (19) : Indirect Escape, Capture and Recapture

Rules (16)-(18) do a reachability analysis on the object connections determining all the indirectly escaping objects , ie, those reachable via a sequence of object references from a directly escaped object. The remaining objects are '*captured*', ie, those that are inaccessible outside the method under consideration. Finally, Rule (19) computes the one-level captured or recaptured objects, those that escape their method of allocation but are captured by a caller method.

4 Experimental Results

We implemented our analysis in the SOOT framework [24] and used *bddbddb* [25] for solving the Datalog rules to arrive at the final points-to sets and the list of escaping objects. We modified the *Shimple* intermediate representation of SOOT to support Heap Array SSA. We ran the analysis on some of the popular Java programs from SourceForge and programs from the SPEC JVM 98 Benchmarks set. All the programs were run in whole program mode in Soot with a precomputed call graph obtained using Class Hierarchy Analysis [15]. The programs with their vital statistics are listed in Table 1. The last six programs are from SPEC JVM 98 benchmark suite. The analyses were done on 4 CPU 3.20 GHz Intel Pentium IV PC with 1 GB of RAM running Ubuntu Linux.

We compare our analysis (field flow sensitive and context sensitive - FFS) with a context sensitive and field flow *insensitive* analysis (FFIS) to study the effect of field flow sensitivity on a pointer analysis in presence context sensitivity. A FFIS analysis is easily obtained from the FFS logic program by omitting the *ssa* number attribute for fields and the *phi* function relations and the corresponding rules. Figure 8 shows the comparative sizes of the *vPointsTo* relation for the two analyses. The sizes are almost the same for most of the programs. Since both the analysis are flow sensitive for locals (due to the use of scalar SSA), field flow sensitivity does not seem to affect the aliasing between the locals themselves much.

Figure 9 compares the sizes of the *hPointsTo* relation for the two analyses. As expected, FFS results in an enormous increase in the size of *hPointsTo* compared to FFIS. This is due to the distinction being made in the field instances at different program points using the heap array SSA. The *escapes* and *captured* relation sizes are shown in Figures 10 and 11 respectively. This *escapes* relation represents the global escape property, ie, not only does it compute the objects that escape their method of allocation, but also computes the escape state of the object with respect to the callers of the method of allocation, transitively. However, the *captured* relation computes only the captured property of an object with respect to to the method of allocation.

As seen from the graphs, the number of objects captured in their method of allocation is almost the same for both FFS and FFIS. This is suprising given the fact that FFS has an extra level of precision and hence should have discovered more number of captured objects. On the other hand, the size of the escapes relation is really big for FFIS compared to FFS, giving an indication that although both FFS might not help in identifying more captured objects in their method of allocation, it might discover more of these objects being captured higher up in the call chain.

Name	Description	Class File Size	Methods Analyzed	SSA Locals	Heap Arrays	New Sites
javacup	Parser Generator for Java	260K	538	8K	335	768
jlex	Lexical Analyzer Generator for Java	164K	496	9K	636	721
nfcchat-server	Distributed chat server	384K	1322	12K	1171	1227
jsch	A pure Java implementation of SSH2	476K	762	10K	515	998
saxon-xml	An open source XSLT and Xquery processor	420K	456	3K	197	481
jrat	Java Runtime analysis Kit	360K	1265	10K	992	1141
jxla	Web Log Parser	264K	794	10K	860	999
junit	Regression Testing Framework	548K	1699	16K	1593	1459
compress	Modified Lampel-Ziv method	48K	469	4K	353	320
raytrace	Ray tracer	120K	613	6K	409	442
db	Memory Resident Database	16K	489	4K	342	362
mpegaudio	MPEG Audio decoder	272K	763	10K	567	1442
mtrt	Ray-tracer where two threads work	120K	613	6K	409	442
check	Checker for JVM Features	92K	358	3K	289	332

Table 1: Benchmark Programs used in Analysis

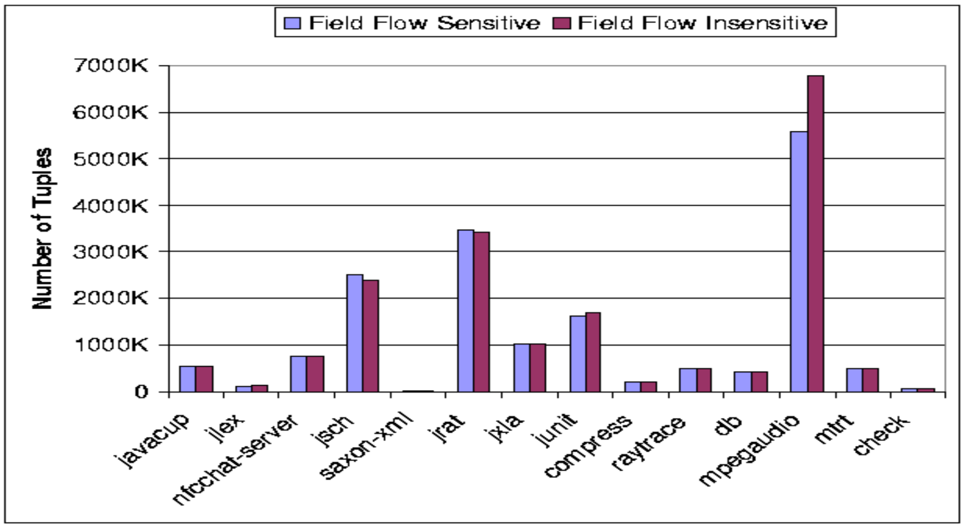

Figure 8: Size of the *vPointsTo* relation

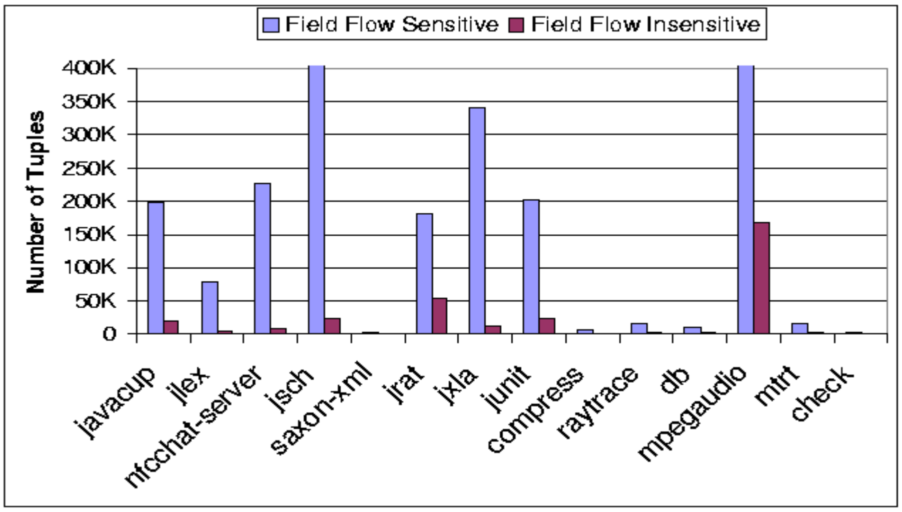

Figure 9: Size of the *hPointsTo* relation

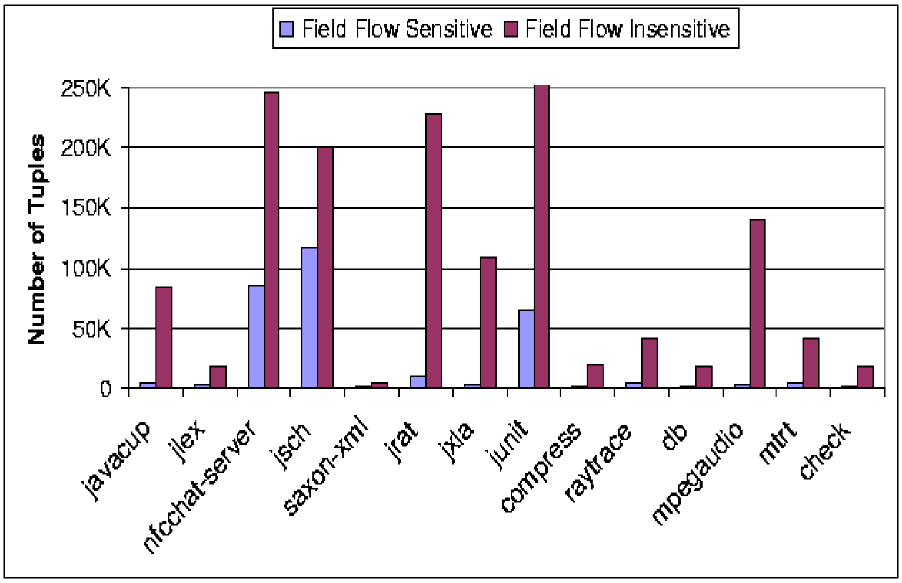

Figure 10: Size of the *escapes* relation

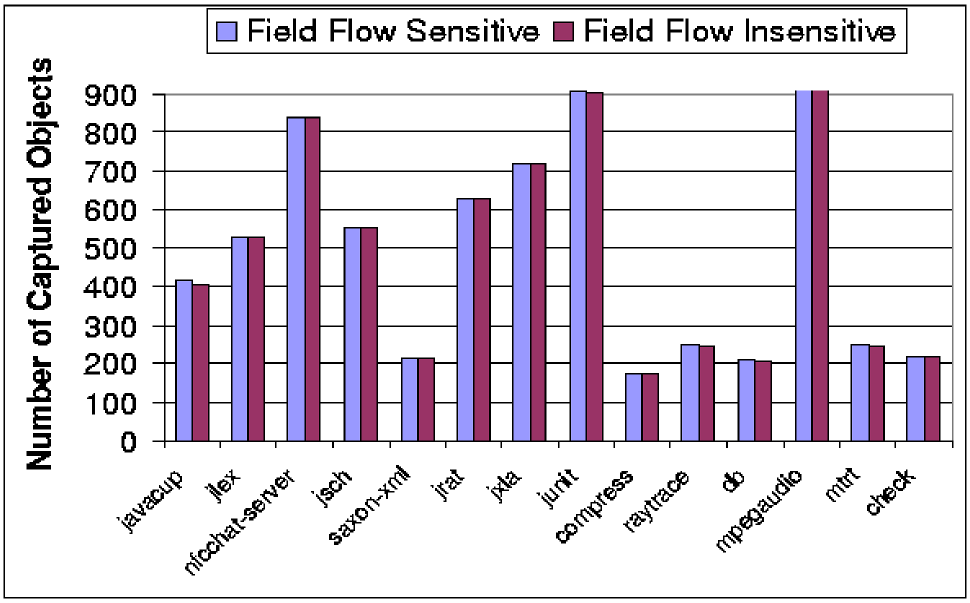

Figure 11: Number of captured objects

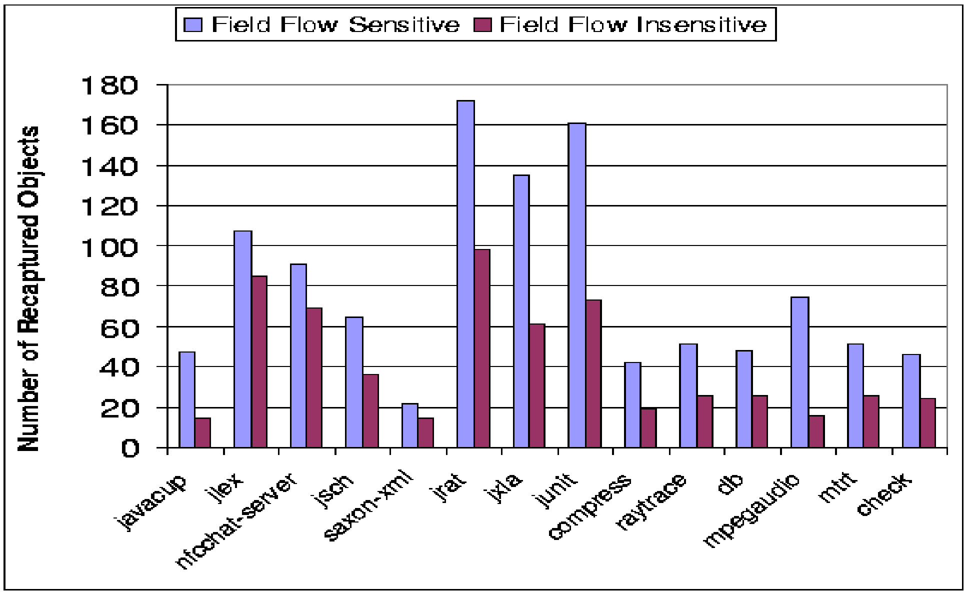

Figure 12: Number of recaptured objects

Figure 12 compares the number of *recaptured* objects (ie,objects that escape their method of allocation, but are caught in their immediate caller) discovered by FFS and FFIS. FFS discovers between 21% (jlex) to 79% (mpegaudio) more recaptured objects than FFIS. The average improvement in number of recaptured objects is 48%. Such recaptured objects can be stack allocated in the calling method's stack. This is especially beneficial when coupled with partial specialization of Java methods [20]. This behavior in the variation in number of captured and recaptured objects may be due to the observed fact that most of methods in object-oriented programs like Java are small and frequently used.

FFS results would be different from FFIS for the *captured* objects only when there are at least two field assigns in a method to the same field (so that strong updates are possible). We measured the percentage of the methods, in the 14 programs that were tested, where (a) an object allocation takes place and (b) there is not more than one field assign (putfield) to any one field. (a) is necessary because these are the methods that are relevant for escape analysis. This is shown in Table 2. For all programs, this percentage is consistently above 86, thus giving some indication as to why FFS and FFIS don't make much difference to the *captured* objects. However, the computation of *recaptured* objects takes into account the flow sensitivity of field assigns across two methods and hence FFS gives a better *recaptured* set than *captured*.

Many of these programs have factory methods which allocate and immediately return an object for which both FFS and FFIS give identical results. One alternate way to work around this is to inline most of the factory methods and compute the *captured* set. We ran our analysis with inlining disabled since it can result in large increase in the number of abstract heap objects and instead computed the *recaptured* relation.

The analysis times reported by *bddbddb* for FFS and FFIS is shown in Table 3. They include the time taken for analyzing the specified program bytecode and any library calls made in the program. For most of the programs with class sizes less than 250K or so, FFS takes more time but comparable to that taken by FFIS. For the other programs, FFS takes much less time than FFIS. Table 4 shows the percentage of the time that *bddbddb* spends in solving the rules that compute the $hPointsTo$ relation. For most of the larger programs, more than 50% percent of time is spent in solving these rules for FFIS than FFS.

Some observations can be made about the behaviour of *bddbddb* by juxtaposing the analysis times with the sizes of the *hPointsTo* relation for the following programs : *nfcchat-server, jsch, jrat, junit and mpegaudio*. All these programs have their bytecode sizes greater than 250K. Although the sizes of the *hPointsTo* relation is very large for a FFS analysis, the analysis times of the FFIS analysis for these programs is way above that of the corresponding FFS analysis. This behaviour is consistent with the fact that operations of the underlying BDDs take time proportional to the size and shape of the BDDs and are not dependent on the number of tuples in the parent database relation. Table 4 further strengthens this notion by indicating that although for FFIS, the size of *hPointsTo* relation is small, most of the analysis time is spent in solving rules pertaining to this relation.

5 Related Work

Our work is inspired by Whaley and Lam's work on context sensitive analysis [26] and Fink and Sarkar's work [10] on Heap Array SSA. Our work can be considered as extending the flow sensitive analysis of the latter to the interprocedural case and incorporating it into the framework of the former to obtain a field flow and context sensitive pointer analysis. Whaley and Lam also specify a thread escape analysis while we have used the pointer analysis results to infer a *method* escape analysis.

One of the earliest context-insensitive and flow-insensitive pointer analysis was due to Anderson [2] which is inclusion based, solved using subset constraints. Steensgaard [23] gives flow insensitive analysis based on a type system with bidirectional assignments which runs in linear time.

The analysis of [9], formulated for C, is both context sensitive and flow sensitive in presence of pointers involving both scalar and aggregate data types and also in the presence of function pointers. It computes both *may* and *must* pointer relations. Context sensitivity is handled by regarding every path in the call graph as a separate context, with recursive methods approximated by two nodes. Our analysis is a *may* pointer analysis and handles multiples levels of indirection by implicitly using the canonical representation in the compiler in-

Program Name	Percentage
javacup	91
jlex	94
nfcchat-server	97
jsch	86
saxon-xml	91
jrat	91
jxla	95
junit	96
compress	91
raytrace	89
db	89
mpegaudio	92
mtrt	89
check	94

Table 2: Percentage of Relevant Methods having at most one *putfield* to the same field

Program Name	FFS	FFIS
javacup	330	300
jlex	113	78
nfcchat-server	664	1572
jsch	798	1065
saxon-xml	10	6
jrat	475	12000
jxla	395	569
junit	556	4145
compress	248	145
raytrace	89	80
db	40	29
mpegaudio	1594	4135
mtrt	90	81
check	8	9

Table 3: Analysis times in seconds reported by *bddbddb*

termediate form where multiple levels of field accesses are split into a sequence of one-level field accesses and assignments to reference variables. Context sensitivity is handled in the same way (cloning), the only difference being the cloned paths are represented by BDDs using bddbddb.

Landi and Ryder [14] formulate a conditional may alias analysis for C, handling multiple levels of indirection, for both scalars and aggregate data types. Given a set of alias pairs that true at the entry of a function, they find the set of the alias pairs that hold at the end of the function (ie, *conditioned* on the alias pairs at the entry). This is similar to having one level of context sensitivity. Also, their analysis makes use of predicate inference rules to compute the may alias relations which can easily be formulated as a Datalog logic program.

Reps describes techniques for performing interprocedural analysis using logic databases [16] and gives the relation between context-free reachability, logic programs and constraint based analyses [18][19]. All program analysis problems whose logic programs are chain programs are equivalent to a context-free reachability problem. Sridharan and Bodik express a context sensitive, flow insensitive pointer analysis for Java as a context-free reachability problem [22].

Initially when we tried to use context free reachability for escape analysis, we faced the same problem that they faced, that is, the how to deal with the fact that although both (a) the field accesses (putfield and getfield) and (b) method invocation and returns can individually be modelled as context free languages, together the balanced parentheses of field accesses and method calls are not nested. They handle this by approximating the CFL of method calls to a regular language (recursive calls are treated as jumps and hence they get a finite language), then using its finite state machine in lock step with the CFL-reachability of field accesses. The approach taken by Whaley is to clone the methods for context sensitivity and hence not using CFL-reachability for method calls. For field accesses, we could not use CFL due to the presence of logic program rules which were not chain rules, for eg, that for *dphi* function rule that adds the flow sensitivity.

In [17], the authors propose *object sensitivity* as a substitute for context sensitivity using call strings. Object names are used, instead of context numbers, to distinguish the pointer analysis results of a method which can be invoked on them. Object names represent an abstraction of a sequence of object allocation sites on which methods can be invoked. They give examples where object sensitivity gives a more precise result than context sensitivity and also examples where context sensitivity gives a more precise result than context sensitivity. Their analysis is flow insensitive. The field flow sensitive portion of our analysis could be used with object sensitivity instead of context sensitivity. They have applied their points-to relations to get a side-effect analysis and a def-use analysis while we have applied our analysis to compute escaping objects.

Choi et al's escape analysis [6], similar to [27], maintains a connection graph at every point in the program and hence is field flow sensitive. Hasti and Horwitz [11] describe a general iterative approach by which any flow-insensitive points-to analysis could be used along with SSA to obtain higher precision. Blanchet's escape analysis [4] which uses scalar SSA, is context sensitive and flow sensitive for locals but not for fields. In our analysis we observed that a field flow *sensitive* and context sensitive analysis helps identify more number of recaptured objects than a corresponding field flow *insensitive* and context sensitive analysis.

6 Conclusions

In this paper, we presented a combined pointer and escape analysis which is both flow sensitive and context sensitive for fields. We have extended the Heap Array SSA form interprocedurally to obtain field flow sensitivity in the presence of context sensitivity. This analysis is expressed as a set of deduction rules in Datalog, a logic programming language. We have implemented our analysis in a bytecode analysis framework and used *bddbddb* - a solver for Datalog and tested it on a set of 14 Java programs. Our results indicate that field flow sensitivity helps to identify more number of captured objects at higher levels of the call chain. Also, the running times of our analysis are comparable to a corresponding field flow insensitive analysis, while in many cases running much faster than the latter.

Acknowledgments

We thank John Whaley for the *bddbddb* package and clarifying some of our doubts regarding his paper. We thank the people at McGill for the SOOT framework and answering our queries posted in their mailing list.

References

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, New York, NY, USA, 1988. ACM Press.

[2] Lars O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Unversity of Copenhagen, May 1994.

[3] Marc Berndl, Ondrej Lhotak, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 103–114, New York, NY, USA, 2003. ACM Press.

[4] Bruno Blanchet. Escape analysis for javatm: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, 2003.

Program Name	FFS	FFIS
javacup	51	56
jlex	71	64
nfcchat-server	24	67
jsch	40	58
saxon-xml	65	43
jrat	55	98
jxla	62	70
junit	45	93
compress	55	30
raytrace	47	44
db	43	22
mpegaudio	44	78
mtrt	47	44
check	41	44

Table 4: Percentage of Analysis Time spent in solving rules for *hPointsTo* relation

[5] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[6] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19, New York, NY, USA, 1999. ACM Press.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[8] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256, New York, NY, USA, 1994. ACM Press.

[9] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256, New York, NY, USA, 1994. ACM Press.

[10] Stephen J. Fink, Kathleen Knobe, and Vivek Sarkar. Unified analysis of array and object references in strongly typed languages. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, pages 155–174, London, UK, 2000. Springer-Verlag.

[11] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 97–105, New York, NY, USA, 1998. ACM Press.

[12] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61, New York, NY, USA, 2001. ACM Press.

[13] Kathleen Knobe and Vivek Sarkar. Array ssa form and its use in parallelization. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 107–120, New York, NY, USA, 1998. ACM Press.

[14] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 235–248, New York, NY, USA, 1992. ACM Press.

[15] Ondrej Lhotak and Laurie Hendren. Scaling java points-to analysis using spark. In *12th International Conference on Compiler Construction (CC), volume 2622 of LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[16] David Melski and Thomas Reps. Interconvertbility of set constraints and context-free language reachability. In *PEPM '97: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 74–89, New York, NY, USA, 1997. ACM Press.

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

[18] Thomas Reps. Program analysis via graph reachability. In *Information and Software Technology 40*, pages 701–706, November 1998.

[19] Thomas W. Reps. Demand interprocedural program analysis using logic databases. In *Workshop on Programming with Logic Databases (Book), ILPS*, pages 163–196, 1993.

[20] Ulrik P. Schultz, Julia L. Lawall, and Charles Consel. Automatic program specialization for java. *ACM Trans. Program. Lang. Syst.*, 25(4):452–499, 2003.

[21] M. Sharir and A. Pnueli. Two approaches for interprocedural data flow analysis. *Program Flow Analysis : Theory and Applications*, June 1981.

[22] Manu Sridharan and Rastislav Bodik. Refinement-based context-sensitive points-to analysis for java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 387–400, New York, NY, USA, 2006. ACM Press.

[23] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM Press.

[24] Raja Vallee-Rai. Soot - a java optimization framework. In *IBM Center for Advanced Studies Conference(CASCON)*, todo, January 1999. todo.

[25] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog and binary decision diagrams for program analysis. In Kwangkeun Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, volume 3780. Springer-Verlag, November 2005.

[26] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM Press.

[27] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 187–206, New York, NY, USA, 1999. ACM Press.