Saarland University
Faculty of Natural Sciences and Technology 1
Department of Computer Science

Bachelor thesis

Demand-driven Pointer Analysis on Explicit Dependence Graphs

submitted by

Klaas Boesche

on 13th October 2009

Supervisor

Jun.-Prof. Dr. Sebastian Hack

Advisor

Dipl.-Inform. Christoph Mallon

Reviewers

Jun.-Prof. Dr. Sebastian Hack,
Prof. Dr.-Ing. Andreas Zeller

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 13th October 2009

Acknowledgements

I thank my friends and family for supporting me throughout the years of my bachelor studies and especially Eva whose deep support is invaluable.

Thanks also go to Sebastian Hack for providing me with this interesting topic which gave me much insight into static analysis.

I furthermore thank Christoph Mallon for his work in extending the capabilities of jFirm and for fruitful discussions.

Thanks go to Nikolai Knopp who provided feedback on early versions of the implementation.

Finally, Markus Rabe has my deepest gratitude for proofreading large parts of this thesis and providing many quality improving comments.

Abstract

In this thesis I present a demand-driven pointer analysis for Java that aims at applications which require memory and time efficiency of the pointer analysis, such as tools for IDEs or JIT compilers. It furthermore allows clients to specify the precision needed and is context-sensitive and flow-sensitive.

The demand-driven aspect of my approach allows the analysis to ignore irrelevant parts of the program. It is also client-driven via an iterative refinement method that allows the client to terminate the analysis early in case sufficient precision is achieved.

The experimental evaluation of the approach however shows that the pointer analysis cannot be precise while terminating within a time budget. 66% of the analyzed queries exceeded even large budgets on average. I show that the flow-sensitive analysis of load statements requires too much computation without careful restriction of analyzed statements. Furthermore, I demonstrate that the refinement idea cannot be used easily in a flow-sensitive demand-driven pointer analysis.

Contents

Chapter 1. Introduction

One of the fundamental problems of software development is the ever increasing complexity. Controlling the complexity in software development is a major focus of research in computer science and software engineering. Within the many fields that aim at increasing developer and software efficiency *program analysis* has received much attention in recent years.

Researchers have developed many tools based on program analysis to aid the programmer in the various stages of development. These range from tools that help in code understanding such as call graph visualization[BD07] over automatic refactoring tools such as class hierarchy reengineering [ST00], to type state verification [FYD$^+$06] and defect detection tools [WZL07].

An important aspect in the implementation of such tools is the integration into the development process. This usually means making them part of an *integrated development environments* (IDE) and allowing the developer to get immediate feedback while working on the code that is analyzed. This however poses practical constraints on the execution time and memory usage of a tools implementation.

Similar requirements occur in the context of *just-in-time* (JIT) compilers. To enable such optimizations as method inlining, these compilers must make use of program analysis. Any time and memory needed for such optimizations however directly count towards the execution time and memory usage of the application.

A major obstacle in achieving efficiency and precision for program analysis is the pervasive use of pointers in modern object oriented languages. Pointers add one or multiple levels of indirection to the flow of data and control, making program analysis much harder. Resolving the targets of pointer variables is the goal of *pointer analysis*.

While much work has been done to create efficient pointer analyses in recent years [WL04, HL07, HL09], these still cannot be easily applied in the areas of IDE tools and optimizing JIT compilers due to the time and memory constraints.

1.1. Overview

In this thesis I extend the precision of the *demand-driven* [HT01] approach to pointer analysis, that tries to tackle the time and memory constraints. Instead of exhaustively computing the possible values of all pointer variables in the whole program, demand-driven pointer analyses answer *queries* for individual variables.

This allows other program analysis, also called *clients*, that make use of the pointer analysis to specify only relevant variables. The pointer analysis can then disregard unimportant program statements, effectively decreasing time and memory usage.

I base my work on the refinement approach described in 2005 by Sridharan *et al.* [SGSB05] and extended in 2006 by Sridharan and Bodík [SB06]. The refinement idea allows for early termination in cases where less precision is required. This is controlled by the client by accepting a coarse result from an intermediate stage of the analysis.

I increase precision of the demand-driven pointer analysis by adding information gained from analyzing the control flow. To this effect I adapt the approach of Sridharan and Bodík to an intermediate representation of Java programs called jFirm.

jFirm is an *explicit dependence graph* that includes both control flow and data dependencies between program statements. This allows the pointer analysis to be sensitive to the control flow, an aspect of precision still missing from previous demand-driven pointer analyses [HT01, SB06, SR05].

Evaluating the implementation with a client that tries to minimize the number of call targets proves the approach to lack practical usefulness. Although fast on average, two thirds of the queries can not be answered with necessary precision. The portion of the program that needs to be analyzed for loads is too large in these cases. The results also show that the refinement idea does not translate to a control flow sensitive analysis easily.

1.2. Thesis structure

The remainder of the thesis is organized as follows. Background in pointer analysis and jFirm is provided in Chapter 2. The full algorithm is described in Chapter 3. The implementation details, the method of evaluation and discussion of benchmark results can be found in Chapter 4. Conclusions and an outlook on extending this work are provided in Chapter 5.

Chapter 2. Background

In this chapter I describe the body of work that this thesis builds on. The first section gives background in pointer analysis. The second section explains the program representation that I use in this thesis.

2.1. Pointer Analysis

The aim of pointer analysis is to determine the values of pointer variables at compile time. Pointer variables reference other variables or objects, in the case of Java and other object-oriented languages. Thus the result of a pointer analysis for a given variable is a set of possible variables or objects, the *points-to set*, that is valid for any run of the analyzed program.

Computing points-to information in general has been shown to be undecidable [Lan92]. Any pointer analysis can thus only compute approximations. Although they are necessary, it is still desirable that the resulting points-to sets are *sound*, meaning that the points-to set for a variable is an over-approximation of the real set of pointer values that holds for all program runs.

In the following sections I describe the basics of pointer analysis and give an overview of several important approaches.

2.1.1. Abstract Objects

What exactly are the objects in a points-to set? We do not always know how many objects are allocated at runtime as it often depends on the input. We must thus distinguish between *abstract objects*, those that are used in the analysis, and the *concrete objects* that exist at runtime.

The common way to choose abstract objects for pointer analysis is by allocation site or statement. In this way, each allocation statement corresponds to an abstract object.

An abstract object however can represent multiple concrete objects. One example of this is an allocation site in a loop. If we do not (or cannot) analyze the loop conditions, we do not know how many objects are allocated at this allocation site. The same holds for allocation statements in methods: If multiple invocations are analyzed at the same time, there are multiple concrete objects, but we do not know which invocation corresponds to which object. See Section 2.1.5 for an approach that separates method invocations.

2.1.2. Approaches to Pointer Analysis

The most common ways to do pointer analysis are to use *data flow analysis* or solve a *set constraint problem* generated from the analyzed program.

For data flow analysis each node in the control flow graph is associated with a transfer function that transforms incoming points-to information into outgoing information. This is usually a set containing pairs of pointer variables and abstract objects that they point to.

The transfer function of an assignment node `a = b` for example updates the points-to information for `a` by possibly discarding previous information for that variable (see Section 2.1.4 and adding points-to pairs *(a,o)* for each abstract object *o* pointed to by `b`.

Data flow analysis computes a fixed point solution where incoming and outgoing points-to sets have stabilized for all variables. For an in-depth treatment of data flow analysis see the book on data flow analysis by Khedker *et al.* from 2009 [KSK09].

Another traditional approach is to consider pointer analysis as a constraint problem in which each statement generates a *subset-constraint* on the points-to sets of the variables contained in the statement. For example an assignment statement `a = b` would generate the constraint $a \supseteq b$, meaning that `a` points to all that is pointed to by `b`.

The resulting set of constraints is then solved by computing the transitive closure of a graph with one node for each variable and edges generated from the constraints.

For a more detailed discussion of general constraint-based analysis refer to the book on program analysis of Nielson *et al.*[NNH99]. For recent work in constraint-based pointer analysis see the improvements made by Hardekopf and Lin in 2007 [HL07].

A third approach is the use of logic programs to implement pointer analysis. Languages such as Prolog or Datalog can directly express the facts and rules that govern how points-to information flows from one variable to another. Whaley and Lam in 2004 implemented a pointer analysis [WL04] on the basis of bddbddb [WACL05], a Datalog engine that uses Binary Decision

Diagrams (BDDs) to efficiently encode and compute the relations in a program analysis.

Most precise analyses that make use of these approaches are exhaustive. This means that they compute the points-to sets of all variables in the input program. Recent precise and exhaustive pointer analyses however often take several minutes and consume hundreds of megabytes of memory while analyzing medium sized programs [WL04, HL09] and are thus not applicable in the context of IDE tools and JIT compilers without further optimization.

2.1.3. Demand-driven Analysis

An alternative option is to work in a demand-driven fashion. This means that the pointer analysis answers queries from a *client* program. These have the form "What may variable x point to?". The analysis can then avoid unnecessary computation by considering only statements relevant to the given query. The analysis may also reduce memory usage, as less points-to sets need to be stored.

Demand-driven techniques were first used for pointer analysis by Heintze and Tardieu in 2001 [HT01]. They consider a subset of C and derive a control flow insensitive, demand-driven pointer analysis.

Listing 2.1 shows an example program in C. Given the query "What may p point to?" their algorithm in essence proceeds as follows: First we derive that p points to all variables whose address is directly assigned to p, in this case y via the assignment in line 3. For each simple assignment p = q, as in line 4, we raise the query "What may q point to?" and find that p points to all variables that q points to, the variable y in this case.

Load assignments like p = *r in line 7 are handled by finding the points-to set S of r and then raising a query for each variable v_i in S yielding points-to sets S_i. Variable p may then point to all variables in the union of all S_i. In this example this assignment adds the variable z to the points-to set of p.

To find out whether *u = &w might modify what variable p points to, we need to find the points-to set of variable u and check whether it contains the variable p. Alternatively we find the set of variables that may point to p and check whether it contains the variable u.

To be practical many demand-driven analyses allow for early termination and return less precise results when a set time limit has been reached. Another approach is to offer the client the option of specifying the degree of precision the analysis should try to achieve. Sridharan *et al.* [SGSB05] implemented this in a flow-insensitive, *client-driven* algorithm which iteratively refines the points-to set until it satisfies the needs of the client.

```
int w, x, y, z, *p, *q, **r, **u;
q = &x;
p = &y;
p = q;
r = &q;
*r = &z;
p = *r;
u = &q;
*u = &w;
```

Listing 2.1: Example for the approach of Heintze and Tardieu

The example in listing 2.2 illustrates their approach. Given the query “What does `x` point to?,” their algorithm first analyzes the statements in lines 1 and 6. The statement `Obj x = new Obj()` adds the abstract object *o1* to the points-to set of `x`.

For the load from the field `f` in the statement `x = v.f` the algorithm takes stores with the same field name and assumes in a first iteration that the variables which contain the fields *alias*. Two variables alias if they may point to the same object. If these variables alias, points-to information may flow through these fields. In the example the variable `y` is assigned to the field and points to the abstract object *o2*. The first iteration thus reports $\{o1, o2\}$ as the points-to set of `x` to the client.

If the client answers that the precision is not sufficient, the algorithm proceeds with another iteration. The next iteration checks whether variables in loads and stores that were assumed to alias in the previous iteration actually do alias. This is again done using the same approximation for newly encountered loads and stores. In this example `v` and `w` do not alias and thus the second and last iteration returns $\{o1\}$ as the points-to set of `x`.

```
1 Obj x = new Obj(); //Abstract object o1
2 Obj y = new Obj(); //Abstract object o2
3 Obj v = new Obj(); //Abstract object o3
4 Obj w = new Obj(); //Abstract object o4
5 w.f = y;
6 x = v.f;
```

Listing 2.2: Example for the approach of Sridharan et al.

In essence, the refinement works by beginning with a *field-based* pointer analysis, that is each field, independent of the actual instance that contains it, is considered as one global variable. Thus all points-to information from stores to this field merge and propagate to all loads.

In further refinement iterations each previously encountered field is handled *field-sensitive*, meaning that fields of different abstract objects are distinguished. Fields that are newly encountered in that iteration are handled field-based again. Thus in each iteration the points-to set might become smaller and thus more precise. The refinement stops when the client is satisfied with the result or no further refinement is possible as no new fields were encountered.

2.1.4. Flow-sensitivity

An important choice that affects precision is whether to observe control flow or disregard the order of statements. A flow-sensitive pointer analysis computes points-to information for every program point, while a flow-insensitive analysis would return points-to sets that hold during the entire program run.

Listing 2.3 shows a simple example. Three objects are assigned to variable a, in lines 1, 3 and 5. The points-to set computed by a flow-insensitive analysis would contain all three objects, while a flow-sensitive analysis would compute points-to sets for each program point where a is accessed. This would result in four points-to sets of variable a: one for each assignment to a, containing the assigned object, and one for the read in line 7, which would contain the abstract objects *o2* and *o3*.

```
1 Obj a = new Obj(); // Abstract object o1
2 if(...) {
3     a = new Obj(); // Abstract object o2
4 } else {
5     a = new Obj();  // Abstract object o3
6 }
7 ... = a;
```

Listing 2.3: Example for flow-sensitivity

This example shows an important property of flow-sensitive pointer analyses: *strong updates*. If a flow-sensitive pointer analysis encounters an assignment statement that overrides points-to information resulting from statements executed before the current one, it can discard that information and thus perform a strong update of the variables points-to set at this program point. A flow-insensitive analysis would have to include all previous information and would return larger points-to sets.

To make a strong update however, the target of the assignment must refer to exactly one variable. If we do not know which variable the assignment may affect, we cannot safely discard previous information.

In the case of Listing 2.3 variable a is a local variable. In Java the assigned location is thus uniquely defined and a strong update is possible. Thus the assignments in lines 3 and 5 remove the abstract object *o1* from the points-to set of variable a in line 7.

Listing 2.4 shows the same example, except that a is a field of an object. Whether we can do a strong update now depends on what o points to. If it points to multiple abstract objects or one that corresponds to multiple concrete objects, we do not know which object's field really receives the assignment. We then cannot know which points-to information to keep and which we should discard. If it only points to a single concrete object, we can perform a strong update. Thus the abstract object *o1* must be kept in the points-to set of o.a in line 7.

```
1 o.a = new Obj(); // Abstract object o1
2 if(...) {
3     o.a = new Obj();
4 } else {
5     o.a = new Obj();
6 }
7 ... = o.a;
```

Listing 2.4: Example for strong updates on a field

The use of an intermediate program representation based on static single assignment (SSA) form has become standard in flow-sensitive pointer analysis. In SSA form each variable has only one static definition. A program is thus transformed to SSA form by introducing versions of the variables to make each definition unique.

For fields however this cannot be done without pointer analysis, as the definition and use of a field depends on the points-to set of the dereferenced variable. Most intermediate representations thus use a partial SSA form in which only local variables are transformed.

The advantage of using SSA form for flow-sensitive analysis is that for local variables a flow-insensitive analysis becomes automatically flow-sensitive due to the versioning of the variables.

Hardekopf and Lin [HL09] show how an exhaustive flow-sensitive pointer analysis can be scaled by improving a partial SSA form of the program with points-to information from a preliminary fast flow-insensitive points-to analysis and using Binary Decision Diagrams for points-to sets.

In a flow-sensitive demand-driven analysis queries of the form “What may variable `x` in line `i` point to?” can be answered. To my knowledge however no demand-driven pointer analysis so far has included flow-sensitivity to improve precision since Hind [Hin01] observed in 2001 that “all demand-driven analyses are flow-insensitive”.

He states that it “remains an open question as to whether the precise flow-sensitive analyses, such as those that use context-sensitivity or perform shape analysis, can be performed in a demand-driven manner.”

2.1.5. Context-sensitivity

A pointer analysis is called *context-sensitive* if it analyzes each method separately for each call. Consider Listing 2.5: If we ignore calling context, variables `a` and `b` would both point to the abstract object *o* from the allocation in line 7 and thus have the same points-to set. In a context-sensitive analysis however, we would need to distinguish abstract objects and calls by the context in which they occur. This is done by associating a *context*, a stack of call statements, with each abstract object and note the context when analyzing each call. Thus `a` and `b` point to different abstract objects, each a pair of a context and the allocating statement, as each is allocated during different calls of the `factory()` method.

```
1 method() {
2     Object a = factory();
3     Object b = factory();
4 }
5
6 Object factory() {
7     return new Object(); //Abstract object o
8 }
```

Listing 2.5: Example for context-sensitivity

A context-sensitive analysis thus matches calls and returns in a program, essentially inlining all method calls. In recursive programs unbounded contexts occur and thus termination is not guaranteed. Several approximations have been used: One approach is to limit the number of calls in a context to some number *k*, handling all calls after that in a context-insensitive way. A better approximation is to ignore contexts for calls after the first level of recursion [EGH94]. Another option is to treat calls inside strongly connected components of the call graph as belonging to the same context.

The last option is used by Whaley and Lam [WL04] and by Sridharan and Bodík [SB06]. Whaley and Lam are able to allow their exhaustive approach to scale by using BDDs to efficiently reduce the amount of memory that the large number of contexts need. However they do not use a context-sensitive construction of the call graph and context-sensitive abstract objects.

Queries for a demand-driven context-sensitive pointer analysis may be made more precise by providing the context in which to analyze the variable. Sridharan and Bodík extend the demand-driven algorithm of Sridharan *et al.*, explained in Section 2.1.3, to include context-sensitivity. They use context-sensitivity for calls and abstract objects, as well as on-the-fly context-sensitive call graph construction. They claim that their approach scales for programs with circa 8800 methods and 164000 relevant statements.

2.2. Explicit Dependence Graphs

An explicit dependency graph is an intermediate representation (IR) of a program in which definition and use dependencies as well as control flow and memory dependencies are made explicit by edges between statements.

In this thesis, I will use the jFirm intermediate representation which is based on Firm [LBBG05]. The goal of both, Firm and jFirm, is to provide a common IR for compiler, optimization, and program analysis research. Firm aims to offer a common IR to many languages, is implemented in C and currently offers mainly C and Java 1.4 frontends. In contrast jFirm is implemented in Java and provides a Java frontend that also works with current versions of Java. Additionally, jFirm provides a more direct mapping of Java bytecode operations to IR operations than the Firm IR.

Firm and jFirm make use of a partial SSA form (see Section 2.1.4). jFirm thus only resolves definitions and uses of local variables and abstracts them away in the final graph structure by directly connecting operations that define some value with operations that use the value.

Operations that modify or read memory such as PUTFIELD and GETFIELD are made memory-dependent. This means that each operation depends upon the memory generated by the operation before it and produces a new memory. The chain of memory dependencies also implies the order in which the statements must be executed and represents the order of statements in the original code as long as no optimization was made.

Figure 2.1 shows an example graph generated from the `remove` method in Listing 2.6. The basic structure is that of a control flow graph with *basic blocks* as nodes. Each basic block contains the nodes of a section of code that is not interrupted by jumps or jump targets. Basic blocks are connected by dashed control flow edges. Each control flow edge starts at the block that follows after executing the target of the edge, a jump or branching statement.

Other edges represent either data flow of any primitive or reference type, or memory dependencies between operations. Edges are directed in the way that they point from the node that uses the value to the defining node. The numbers at the edges correspond to the input of the source node. A PUTFIELD node for example has three inputs: a memory value on input 0, the reference (DATA) to the instance containing the field on input 1, and the reference or value to be stored on input 2. PROJ nodes extract produced values from nodes with multiple output values. Notice that the local variables `listNode`, `previous` and `next` are replaced by edges that directly link the defining and using operations.

```
void remove(Node listNode) {
    if(listNode == null) return;
    Node previous = listNode.previous;
    Node next = listNode.next;
    previous.next = next;
    next.previous = previous;
}
```

Listing 2.6: Example for jFirm

Table 2.1 shows an overview of the operations relevant to pointer analysis. For each operation their input and output values are listed.

Another derivative IR of Firm has been used to construct a flow-sensitive pointer analysis [LL07], a flow-sensitive analysis using information from branch conditions [GLL07] and a context- and flow-sensitive analysis [LGL08]. They use an exhaustive approach that simulates the execution of the program.

Hardekopf and Lin [HL09] use the LLVM IR [Lat02] to design a flow-sensitive pointer analysis for C. LLVM uses a partial SSA form with annotated low level operations. Hardekopf and Lin improve their analysis by first using a flow-insensitive pointer analysis to compute def-use information for pointer variables. This allows their flow-sensitive analysis to scale up to 1.9 million lines of code and 1 million statements by their account.

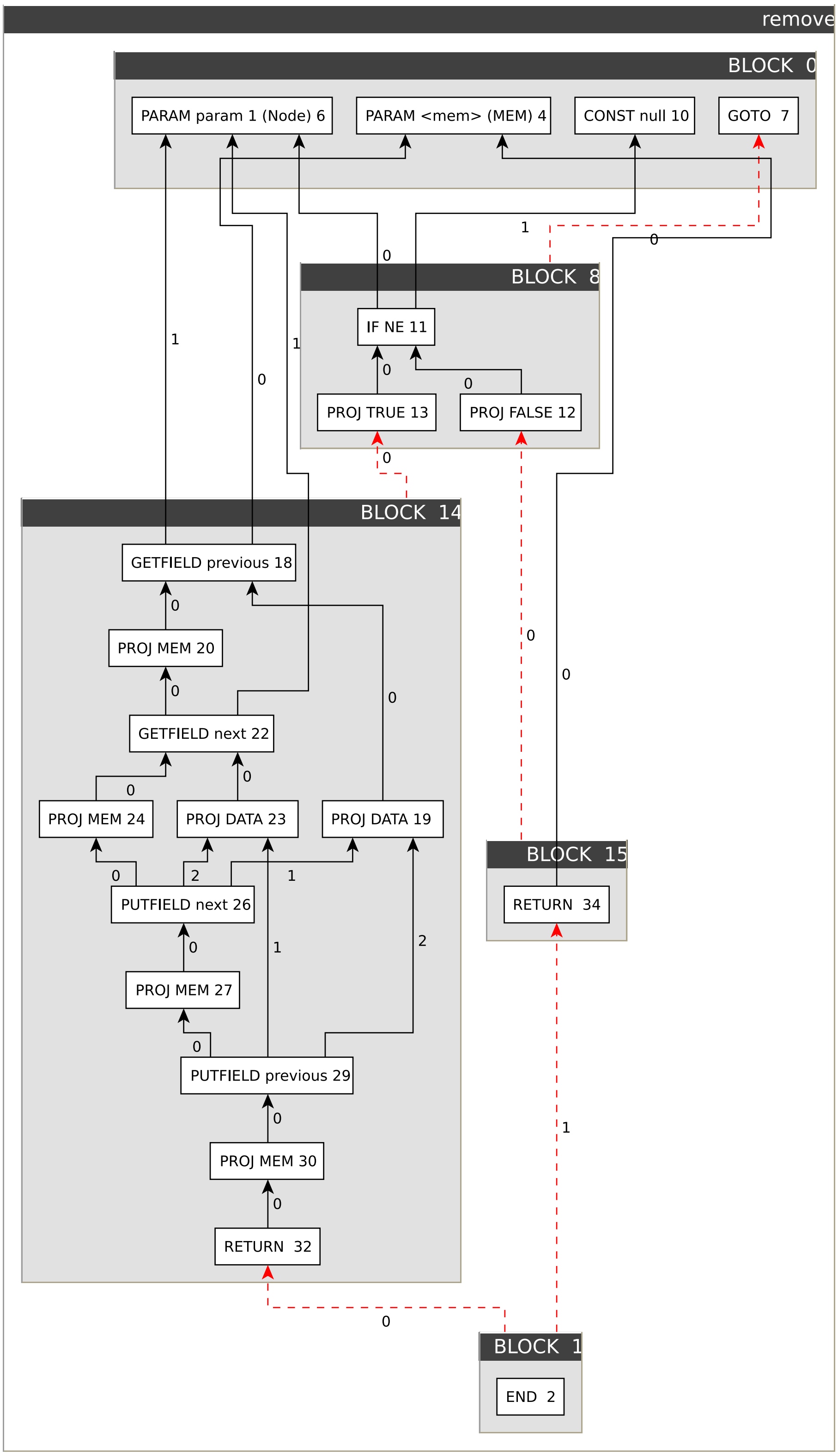

Figure 2.1.: Example method graph for jFirm

Opcodes	#In	Inputs in order	#Val	Produced Values
END	0	*	0	
BLOCK	0 to n	control-flow	0	
PHI	2 to n	either DATA or MEM	1	**
PROJ	1	DATA or MEM	1	same as input
PARAM	0		1	parameter/memory
CONST	0		1	constant value
ALOAD	3	MEM, array, index	2	MEM, loaded value
ASTORE	4	MEM, array, index, value	1	MEM
RETURN	1/2	MEM, value	1	control-flow
NEW	1	MEM	2	MEM, new value
NEWARRAY	2	MEM, size	2	MEM, new value
GETFIELD	2	MEM, instance	2	MEM, loaded value
PUTFIELD	3	MEM, instance, value	1	MEM
GETSTATIC	1	MEM	2	MEM, loaded value
PUTSTATIC	2	MEM, value	1	MEM
CALL	1 to n	MEM, this, parameters	1/2	MEM, return value
CAST	2	MEM, value	2	MEM, value
SWITCH	1	value	n	control-flow
IF	2	value, value	2	true, false
GOTO	0		1	control-flow

* The block containing END is control-flow successor to all RETURNs.

** Returns the input corresponding to the current control-flow to block.

Table 2.1.: jFirm Opcodes relevant to pointer analysis. Operations that use only primitive types have been left out.

Chapter 3. Algorithm

In my approach I combine several of the ideas outlined in the previous chapter to develop a flow- and context-sensitive, demand- and client-driven pointer analysis. I extend and modify the refinement algorithm of Sridharan *et al.* [SGSB05] and Sridharan and Bodík [SB06] to compute flow-sensitive results. To this effect I use the jFirm intermediate representation which has been described in Section 2.2.

In the following section I show the outer refinement loop. In Section 3.2 I describe and explain the algorithm for a single refinement iteration.

3.1. Refinement

Queries for the pointer analysis contain a context and a node that produces a reference value. The core idea is to begin searching at this node and follow all dependencies that determine the points-to set. This is a simple depth-first search as long as no values returned by loads from arrays, static fields or instance fields are encountered.

Sridharan *et al.* [SGSB05] observe that a load from a field can only be affected by a matching store to that same field and that this can be used to implement a refinement strategy. In the first refinement iteration the algorithm assumes that all stores and loads on fields with the same name and containing class match. Thus the value stored into the field directly flows to the load. This enables the analysis to skip computing whether the base variables whose fields are accessed really alias but produces a possibly imprecise result.

In the algorithm an approximating *match* between a store and a load is a pair $((loadContext, loadNode), (storeContext, storeNode))$. The algorithm keeps track of the following sets:

- *removedMatches* The set of all matches that should be checked for aliases in further iterations.

- *usedMatches* The set of matches that have been used to approximate the result in the last iteration.

- *invalidMatches* The set of matches whose base references do not alias. This is used during the whole lifetime of the pointer analysis.

- *visitedNodes* The set of nodes that have been visited during the depth-first search for points-to information.

Figure 3.1 shows the refinement loop. In each iteration the *usedMatches* are added to the set of matches to be refined in later iterations. Two conditions can make the refinement stop: Either we have found no more matches during the last refinement iteration and thus maximum precision is achieved. Or the client decides that the precision is sufficient for its needs by returning true from the call to QueryAnswered.

```
PointsTo(context, node)
Clear(removedMatches)
repeat
    Clear(visitedNodes)
    Clear(usedMatches)
    pointsTo = FindPointsTo(context, node)
    AddAllTo(removedMatches, usedMatches)
until usedMatches == $\emptyset$
        $\lor$ QueryAnswered((context, node), pointsTo)
return $pointsTo$
```

Figure 3.1.: Refinement algorithm

3.2. Single iteration

During a single refinement iteration the analysis searches along the data dependencies in a depth-first manner for allocation sites. If the points-to set contains information from the return value of a load operation the method FindLoadPointsTo is called. The full pseudo code for FindPointsTo and FindLoadPointsTo can be found in the appendix in Figures A.2 and A.3.

To implement the refinement idea, FindLoadPointsTo uses another depth-first search along the chain of memory dependencies to find stores. If a call is encountered we inline each possible target method by entering the method at the return statement and exiting via the memory parameter. See Section 3.2.2 for context sensitivity and interprocedural details. The stores are handled by the methods PutStatic, ArrayStore and PutField. They each take the corresponding load and the store, the points-to set that will be returned from the load and the stack for the depth first search.

PutStatic in Figure 3.2 can easily check whether the PUTSTATIC and GETSTATIC operations really match by comparing equality between the accessed fields. Each static field is uniquely identified by the class that owns it and its name. If the fields are equal we can thus always perform a strong update. The strong update discards all points-to information generated by operations before the store. This means we simply stop the depth-first search at the store and do not add the node that produced the previous memory value to the *stack*. The relevant part of the memory value, that is the part that contains this field, was overwritten by the strong update. The use of refinement for static stores is thus not necessary.

```
PutStatic($((c, load), (context, store), pointsTo, stack)$)
if $store.field == load.field$
    AddAllTo($pointsTo$, FindPointsTo($context, store.value$))
    return
Push($stack$, $(context, store.memoryIn$)
return
```

Figure 3.2.: Method for handling PUTSTATIC nodes

Precise handling of arrays is very difficult as we would need to differentiate between the different indices and compute array boundaries. This requires analysis of primitive values and is often dependent upon input values and loop conditions. The most common approach is to merge all positions of the array into a single field and not analyze indices and boundaries at all. This precludes strong updates on all array stores as the field possibly corresponds to different array positions.

Figure 3.3 shows the ArrayStore method. Using the refinement idea for arrays assumes that arrays of compatible static types are the same in the first iteration (lines 5-11) and later refines this by checking whether the array reference nodes alias. If the match has been proven invalid we can skip analyzing this store further.

Lines 9-11 are used in the first iteration that encounters this store. If the match has not been used to approximate in previous iterations it is added to *usedMatches*. Then the points-to set of the stored value is found and added to the points-to set returned by FindPointsTo.

```
ArrayStore$((c, load), (context, store), pointsTo, stack)$
 1  $match = ((c, load), (context, store))$
 2  if $match \in invalidMatches$
 3      Push$(stack, (context, store.memoryIn))$
 4      return
 5  if IncompatibleTypes$(store.array, load.array)$
 6      Push$(stack, (context, store.memoryIn))$
 7      AddTo$(invalidMatches, match)$
 8      return
 9  if $match \notin removedMatches$
10      AddAllTo$(pointsTo,$ FindPointsTo$(context, store.value)$
11      AddTo$(usedMatches, match)$
12  else
13      $alias = ((c, load.array), (context, store.array))$
14      if $alias \notin checkingAlias$
15          AddTo$(checkingAlias, alias)$
16          $sPointsTo =$ FindPointsTo$(context, store.array)$
17          $lPointsTo =$ FindPointsTo$(c, load.array)$
18          RemoveFrom$(checkingAlias, alias)$
19          if $sPointsTo \cap lPointsTo == \emptyset$
20              AddTo$(invalidMatches, match)$
21          else
22              AddAllTo$(pointsTo,$ FindPointsTo$(context, store.value))$
23  Push$(stack, (context, store.memoryIn))$
```

Figure 3.3.: Method for handling ASTORE nodes

In later iterations we need to prove that the array references may refer to the same array. It may be possible that a cyclic dependency leads back to this combination of load and store. As observed by Sridharan and Bodík [SB06] if the existence of the alias between the two base reference variables depends only on itself, then the two references cannot refer to the same object. To ensure termination we must thus keep track of the aliases being checked in lines 13-18.

We check for the alias by computing the points-to set of both array references. If the intersection is the empty set this store can never affect this load and we add the match to *removedMatches*. In the other case we find the points-to set of the stored value and add it to the points-to set of the analyzed ALOAD.

Matches between PUTFIELD and GETFIELD nodes are handled in much the same way. Figure 3.4 shows only the differences. As fields are determined by the static type of the base reference and their name, we can improve our approximation by checking for equality and possibly add the match to the *invalidMatches*. Lines 2-5 replace lines 5-8 in ArrayStore. The main difference to handling matches between array load and stores is in lines 7-9 which replace line 22. If the points-to sets of the base references not only alias but also refer to a single concrete object, we can perform a strong update and do not visit the previous node on the memory chain.

```
PutField($(c, load), (context, store), pointsTo, stack$)
1  …
2  if $\neg(store.field == load.field)$
3        AddTo($invalidMatches, match$)
4        Push($stack, (context, store.memoryIn)$)
5        return
6  …
7              AddAllTo($pointsTo$, FindPointsTo($context, store.value$))
8              if IsSingleConcrete($sPointsTo$)
                  $\wedge$ IsSingleConcrete($lPointsTo$)
9                  return
```

Figure 3.4.: Method for handling PUTFIELD nodes

3.2.1. Single concrete objects

Determining whether an abstract object refers to a single concrete object is often difficult in pointer analysis. As said in Section 2.1.1 the allocation site might be inside a loop or the context of the method invocation might not be unique.

For method invocations the solution is to use context sensitivity and associate the calling context with each abstract object. In recursive invocations however abstract objects may still correspond to multiple concrete ones. See the next section for context sensitivity and recursion.

In loops the usual approximation is to just assume that all allocation sites do not correspond to single concrete objects. If we know however that a abstract object in a loop must be the *most recently* allocated object in that loop we can improve upon that approximation. This allows us to use that abstract object as a single concrete object inside the loop.

In our analysis there are only three cases in which a points-to set of only one abstract object may not refer to the most recently allocated object at that site. The trivial case is that the points-to set contains a special abstract object that corresponds to the null reference. A single abstract object might also flow through an array. In that case we do not know whether it is the most recent object, as the indices allow access to objects that might be allocated in previous loop iterations. If an array load thus affects the computed points-to set, we set a flag on it which is propagated to other points-to sets that include it. The third case occurs if the context of that abstract object is not unique and contains multiple invocations of a method. We check this by marking contexts that represent more than one method invocation. See the next section for details.

In all other situations the points-to set either includes only one most recently allocated object or it must include multiple abstract objects. This results from the way in which the analysis searches for abstract objects and the properties of the program representation.

Assume that we have a query for the points-to set of a local variable `x` after a loop which is defined inside the loop by an allocation `x = new ...`. As it is accessible after the loop it must be declared before the loop. Java however does not compile this if `x` is not also defined before the loop. In static single assignment form the variable must then have a PHI definition inside the basic block. The PHI either corresponds to the definition before or inside the loop, depending on which control flow path has been taken by the program. The analysis however is not path-sensitive and must merge the incoming points-to information and thus includes at least two abstract objects.

For loads from fields the same argument can be applied, except that we search for definitions along the memory dependencies. There must be a PHI node for the memory value, as both control flow predecessors produce a different memory value. The analysis also explores both memory chains and merges incoming points-to information. If one store and allocation site is found inside the loop we must still analyze the other memory dependency. Either we find another abstract object or no definition is found and we must either assume that it may return all possible objects or that it may be null. We can only assume the latter if we trace the memory dependencies back to the program entry.

3.2.2. Context sensitivity

For a context sensitive pointer analysis we must keep a stack of calls or a *context* that led to the currently analyzed method. The analysis proceeds from the use of a variable to the definition and thus in reverse direction of control and data flow. Accordingly it enters a method via all return statements and exits it via the parameters. On entering a method the call statement is pushed on the call stack and all nodes within that method are associated with that context.

This however is unbounded in the case of recursion. To solve this we test if a new call pushed on the stack is already present and would thus be a recursive call. All methods called in that recursive cycle are part of a strongly connected component in the call graph. We collapse this strongly connected component by associating further calls within that cycle with the same context. This context is also associated with all nodes and abstract objects within any contained method. These abstract objects may refer to multiple concrete objects and not only to the most recently allocated one as described in the previous section.

Chapter 4. Evaluation

4.1. Implementation

The analysis is implemented on the jFirm framework which is still in development. It currently does not support exception handling, reflection and native methods. It also does not handle MULTIANEWARRAY bytecode instructions. My implementation currently handles methods for which no jFirm graph can be constructed unsoundly by ignoring them as targets of calls.

4.1.1. Call graph

To determine call targets I implemented a coarse call graph. Possible targets are determined by considering the possible types of the receiving objects only based on the static type of the receiving variable. This is called a call graph based on class hierarchy analysis [DGC95].

This call graph is computed on initialization and is used to build a context sensitive call graph during analysis. In a context sensitive call graph the possible targets for each call statement are analyzed for each context in which the call statement can occur.

On encountering a call we compute the points-to set of the receiving object in the current context. We then filter the possible targets from the coarse call graph with the types of the objects in the points-to set.

4.1.2. Optimization

As there are repeated calls to PointsTo and FindPointsTo with the same paramaters, we cache the results returned by these methods. The cache for FindPointsTo is cleared for each refinement pass of the query. Otherwise, matches that should be refined are not be visited again and the possibly less precise cached points-to sets are used.

Another optimization skips call targets entirely when searching for stores on static or instance fields. On initialization we compute the sets of static and instance fields modified by a method and all called methods. We first compute the fields modified within each method. We then compute a reverse topological order of the strongly connected components (SCCs) of the coarse call graph. In a last step we propagate the sets of written fields from called SCCs to their callers. Every method within a SCC has the same sets of written static and instance fields.

When we would normally descend to the called methods of a CALL node during FindLoadPointsTo we filter the target methods. A method whose respective set of written fields does not include the field of the analyzed load operation is not analyzed.

4.2. Setup

I evaluated the pointer analysis with a client that asks for the points-to sets of the receiving objects of virtual calls. I exclude calls that already have a single target in the type based coarse call graph. The analysis begins at the main method and recursively analyzes all found target methods belonging to the benchmark program.

The client stops the refinement when there is only one type of objects in the points-to set. This represents the goal of finding the lowest number of possible targets while stopping the pointer analysis as early as possible.

Information about the possible targets is important for method inlining in compilers as well as for many IDE tools such as call graph visualization in program understanding. The points-to sets of receiving objects are also of interest to typestate verification which usually tracks state as a sequence of calls on an object.

I configured the analysis to stop and return a coarse result when more than a budget of relevant nodes is visited during the analysis. Either the points-to set computed during the last refinement iteration or the set that contains all possible abstract objects is returned. This simplifies the implementation and use of the pointer analysis in contrast to using actual time constraints.

4.2.1. Benchmarks

To evaluate the performance of the analysis I used programs from the DaCapo [BGH$^+$06] benchmark suite version 2006-10-MR2. The same programs, although from a slightly older version of the suite, were used by Sridharan and Bodík [SB06] and the same version was used by Gutzmann *et al.* [GLL07].

I use the distribution provided on the DaCapo website that separates the program and its dependencies into two archives.

The analysis was run on these benchmarks using Java 1.6.0_10 and a Intel Core Duo processor with 1.8 GHz and 2 GB of memory running Kubuntu 8.10. The Java virtual machine was configured to use a maximum of 1.5 GB of memory.

The tables 4.1, 4.2 and 4.3 show various statistics about the programs that were used.

In table 4.1 only those classes were considered that belong to the program itself. I restrict the client to only query variables within these classes for their points-to set. The points-to analysis however needs to consider methods inside the dependencies. Table 4.2 lists statistics that include all classes and methods that the pointer analysis may ever encounter.

Note that in both tables the *Nodes* column also includes nodes not relevant for pointer analysis such as arithmetic operations.

Benchmark	Classes	Methods	Calls	Nodes	Loads	Stores
antlr	228	2483	23355	195010	8941	3544
bloat	360	3836	25127	224322	10559	4701
chart	515	6090	26227	276483	11770	5287
jython	919	8547	31584	374341	17320	8805

Table 4.1.: Benchmark statistics including only classes, methods and nodes within the analyzed benchmark programs.

Benchmark	Classes	Methods	Calls	Nodes	Loads	Stores
antlr	5482	46356	181175	3218024	104063	56571
bloat	2519	22072	88426	1454481	42522	23321
chart	6181	54077	203206	3696396	118588	73058
jython	6436	54106	196131	3535677	114730	63082

Table 4.2.: Benchmark statistics including classes, methods and nodes in dependencies of the analyzed benchmark.

Table 4.3 shows the memory usage of all jFirm method representations in the first column. The second column shows the amount of memory needed to keep the coarse call graph and the last column shows how much memory was occupied by the optimization described in 4.1.2.

An average total of 598 MB is likely too much to use this analysis for our target applications. For JIT compilers this effectively reduces the amount of memory available to the executed program by that amount. Assuming that 2 GB of main memory is a common amount available to developers this also subtracts a significant part of the memory available to the IDE, the client and other programs. jFirm however can also be used by the IDE and the client programs as it includes a full intermediate representation.

The current implementations of jFirm and the pointer analysis however are not optimized much with respect to memory usage. Set and map implementations are mostly taken from the Java standard library and only replaced by more space efficient versions in critical cases.

One of these is the optimization for side effect field stores made by method calls. As table 4.3 shows this still requires 151 MB of memory on average. Without the optimization however evaluation was all but impossible as field loads would require analyzing calls to large irrelevant parts of the code.

Benchmark	jFirm IR	CHA call graph	field optimization
antlr	377	87	165
bloat	172	45	41
chart	438	105	187
jython	421	93	182
average	363	84	151

Table 4.3.: Memory usage in MB for unchanging parts of the analysis.

4.3. Results

The tables 4.4 and 4.5 show the results of running the call target client with a pointer analysis that is limited to 1000 and 5000 nodes. The second column gives the total number of queries and the third column the number of those that were not cached. Caching reduces the number of analyzed queries by 30.3% on average.

The “Budget” column shows the number of queries that resulted in the analysis visiting too many nodes and returning a coarse result. This number does not change if we increase the limit to 5000 or 10000 nodes.

The reason for this can be found in table 4.6. It displays the number of load nodes analyzed per query on average. It also shows the difference between a query that did not complete within the limit of 1000 nodes and queries that did. We can see that the number of encountered loads is a major factor in whether a query exceeds the budget.

The time taken on average for a non-cached query is quite fast with an average of 6.3 milliseconds with a budget of 1000 nodes. For 5000 nodes however this increases to 334.7 ms. Tests with a budget of 10000 nodes show that the time needed explodes quickly to over one second without any precision gain.

The precision is given as the percentage of queries which resulted in a single target method for the virtual call or a single concrete object for the receiver points to set. In the second case the pointer analysis cannot return a more precise result. The precision comes mostly from the non-cached queries that did not run out of budget, since nearly all budget-exceeding queries returned the set of all possible abstract objects. These cannot prove that a call has a single target because the call has multiple targets in the coarse call graph.

The fact that the regularly terminating queries only analyzed 0.6 loads on average means that the algorithm cannot analyze most field loads. The implementation needs the optimization from Section 4.1.2 to reduce the number of calls analyzed during loads and thus reduce the number of nodes. However, it still does not succeed as the underlying algorithm needs to look at too many nodes while searching for stores.

The maximal amount of memory required during a query is shown in the last column of tables 4.4 and 4.5. This adds to the memory requirements displayed in table 4.3 and results in an average total of 264 MB for the analysis with a limit of 1000 nodes. Using a budget of five times as many nodes increases the memory usage by a factor of 14.5.

I also analyzed the average number of refinement passes per query. Budget-exceeding queries always terminated within the first refinement pass while for other queries the average was 1.004 passes. This reflects that the former are more complex as more loads need to be analyzed.

It also shows that the refinement idea is not directly applicable to a flow-sensitive demand-driven analysis. As no strong updates are performed on the first encounter of a store to the field, the analysis must search for other stores and needs to visit many more nodes. However, checking whether it is possible to perform a strong update on the first encounter would make further refinement passes superfluous.

	Queries	Not Cached	Budget	Time	Precision	Memory
antlr	4899	1237	614	4.6 ms	10.4 %	10 MB
bloat	2669	1851	1201	3.8 ms	21.4 %	13 MB
chart	4096	2300	1448	5.6 ms	17.2 %	7 MB
jython	3014	2742	2150	11.3 ms	4.5 %	12 MB
average	2916	2033	1353	6.3 ms	13.4 %	11 MB

Table 4.4.: Results of the call targets client with a limit of 1000 nodes.

	Time	Precision	Memory
antlr	271.6 ms	10.4 %	139 MB
bloat	188.0 ms	21.4 %	166 MB
chart	381.9 ms	17.2 %	166 MB
jython	497.2 ms	4.5 %	167 MB
average	334.7 ms	6.3 %	159.5 MB

Table 4.5.: Results of the call targets client with a limit of 5000 nodes.

		Loads
antlr	budget	12.0
	regular	0.4
bloat	budget	6.8
	regular	0.5
chart	budget	9.8
	regular	0.5
jython	budget	16
	regular	0.8
average	budget	11.2
	regular	0.6

Table 4.6.: Average number of load nodes analyzed during queries.

Chapter 5. Conclusions

In this thesis I have described a new approach to demand-driven pointer analysis on Java. The approach extends a context-sensitive demand-driven pointer analysis [SB06] with flow-sensitivity. To this effect it uses the intermediate representation jFirm (see Section 2.2) that explicitly includes data and control flow as well as memory dependencies.

The general idea is to make use of refinement and iteratively increase the precision. In each iteration we use a depth-first search for allocation sites that make up the points-to set. Load statements are analyzed by performing a depth-first search for stores along the memory dependencies. The core idea of refinement is to first assume newly encountered load and store pairs to access the same location. In the next iteration we then check whether this really holds. This also yields the possibility to perfom strong updates which enable the analysis to stop the search for more stores.

I implemented this approach and optimized the implementation by skipping calls during the search for store statements that do not write to the field. I evaluated the implementation using a client of the pointer analysis that tries to minimize the number of targets for a context-insensitive call.

The evaluation shows a flaw in the approach itself. The analysis fails when searching for the points-to set from load statements as the number of nodes that needs to be visited is too large. Only queries that access very few or no fields at all are completed successfully.

The implementation does not profit from refinement, and the results from Section 4.3 show that a flow-sensitive analysis cannot use this idea easily. One reason is that refinement does not enable strong updates in early passes and thus leads to possibly more nodes being visited than necessary.

5.1. Future work

It remains an open problem how flow-sensitivity can be used in a context-sensitive demand-driven pointer analysis. A fundamental issue is the reduction of the search space for the points-to information flowing through load statements. Achieving this without increasing memory usage too much must thus be an integral part of future attempts in flow-sensitive demand-driven analysis.

Client-driven techniques such as refinement add a useful option for early termination to demand-driven analysis. It is however not clear how this can be incorporated into a flow-sensitive pointer analysis. The potential benefits however make this an interesting goal for future research.

Appendix A. Algorithm

The following pages provide the method listing missing from Chapter 3.

FindPointsTo in Figure A.2 shows the basic depth-first search algorithm used to search for the points-to set during a single refinement iteration. FindParamPointsTo and FindCallPointsTo are omitted. These methods query the call graph for the callers and targets respectively and return the union over the points-to sets returned by calls to FindPointsTo with the appropriate nodes in the calling or target methods.

Figure A.3 shows the FindLoadPointsTo method. It describes the depth-first search over the memory edges for stores that affect the analyzed loads.

```
Call$((context, call), workList)$
    $targets = \text{GetCallTargets}(callGraph, (context, call))$
    **for** each $(method, callContext) \in targets$
        $\text{AddToWorkList}(workList, (callContext, method.end.memoryIn))$
```

Figure A.1.: Algorithm for handling a call during the analysis of a load

```
FindPointsTo($c$, node)
if $(c, node) \in visitedNodes$
    return $\emptyset$
AddTo($visited, (c, node)$)
$pointsTo = \emptyset$
$opCode = node.opCode$
if opCode == PROJ
    $pOpCode = node.producer.opCode$
    if $pOpCode \in \{$GETFIELD, GETSTATIC, ALOAD$\}$
        $pointsTo =$ FindLoadPointsTo($c, node.producer$)
        if pOpCode == ALOAD
            $pointsTo.fromArray =$ true
    elseif $pOpCode \in \{$NEW, NEWARRAY$\}$
        $pointsTo = \{(c, node.producer)\}$
    elseif pOpCode == CAST
        $pointsTo =$ FindPointsTo($c, node.producer.reference$)
    elseif pOpCode == CALL
        $pointsTo =$ FindCallPointsTo($c, node.producer$)
elseif opCode == PARAM
    $pointsTo =$ FindParamPointsTo($c, node.producer$)
elseif opCode == PHI
    for $i = 0$ to $n.inCount$
        AddAllTo(pointsTo, FindPointsTo($c, node.in[i]$))
elseif opCode == CONST
    $pointsTo = \{(c, node)\}$
return $pointsTo$
```

Figure A.2.: Algorithm for a single refinement iteration

```
FindLoadPointsTo($c$, load)
pointsTo = $\emptyset$
opCode = load.opCode
workList = []
AddTo(workList, ($c$, load))
workVisited = $\emptyset$
while $\neg$ IsEmpty(workList)
    (context, node) = RemoveFirst(workList)
    if (context, node) $\in$ workVisited
        continue
    AddTo(workVisited, (context, node))
    workOpCode = node.opCode
    if workOpCode == PHI
        for $i$ = 0 to node.inCount
            AddToWorkList(workList, (context, node.in[$i$]))
    elseif workOpCode == CALL
        Call((context, node), workList)
        AddToWorkList(workList, (context, node.memoryIn))
    elseif workOpCode == PUTFIELD $\wedge$ opCode == GETFIELD
          $\vee$ PutField(($c$, load), (context, node), pointsTo, workList)
    elseif workOpCode == ASTORE $\wedge$ opCode == ALOAD
          $\vee$ ArrayStore(($c$, load), (context, node), pointsTo, workList)
        AddToWorkList(workList, (context, node.memoryIn))
    elseif workOpCode == PUTSTATIC $\wedge$ opCode == GETSTATIC
          $\vee$ PutStatic(($c$, load), ((context, node), pointsTo, workList)
    elseif workOpCode == PARAM
        callers = GetCallers(callGraph, (context, param))
        for each (callContext, callNode) $\in$ callers
            AddToWorkList(workList, callContext, callNode.memoryIn))
    else AddToWorkList(workList, (context, node.memoryIn))
return $pointsTo$
```

Figure A.3.: Algorithm for finding the points-to set of the reference value returned by a field load

Bibliography

[BD07] Johannes Bohnet and Jürgen Döllner. Cga call graph analyzer - locating and understanding functionality within the gnu compiler collection's million lines of code. In *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 161–162. IEEE Computer Society Press, 0 2007.

[BGH$^+$06] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101, London, UK, 1995. Springer-Verlag.

[EGH94] 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.

[FYD$^+$06] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 133–144, New York, NY, USA, 2006. ACM.

[GLL07] Tobias Gutzmann, Jonas Lundberg, and Welf Löwe. Towards path-sensitive points-to analysis. In *Seventh IEEE International Working Conference on Source Code Analysis And Manipulation (SCAM)*, pages 59–68, September 2007.

[Hin01] 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.

[HL07] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. *SIGPLAN Notices*, 42(6):290–299, 2007.

[HL09] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 226–238, New York, NY, USA, 2009. ACM.

[HT01] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–34, 2001.

[KSK09] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 2009.

[Lan92] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.

[Lat02] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[LBBG05] Götz Lindenmaier, Michael Beck, Boris Boesler, and Rubino Geiß. Firm, an intermediate language for compiler research. Technical Report 2005-8, March 2005.

[LGL08] Jonas Lundberg, Tobias Gutzmann, and Welf Löwe. Fast and precise points-to analysis. In *Eigth IEEE International Working Conference on Source Code Analysis And Manipulation (SCAM)*, pages 133–142, September 2008.

[LL07] Jonas Lundberg and Welf Löwe. A scalable flow-sensitive points-to analysis. In *Compiler Construction - Advances and Applications, Festschrift on the occasion of the retirement of Prof. Dr. Dr. h.c. Gerhard Goos*. Springer Verlag, 2007. accepted.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[SB06] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. *SIGPLAN Notices*, 41(6):387–400, 2006.

[SGSB05] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. *SIGPLAN Notices*, 40(10):59–76, 2005.

[SR05] Diptikalyan Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 117–128, New York, NY, USA, 2005. ACM.

[ST00] Gregor Snelting and Frank Tip. Understanding class hierarchies using concept analysis. *ACM Transactions on Programming Languages and Systems*, 22(3):540–582, May 2000.

[WACL05] 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.

[WL04] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM Press, June 2004.

[WZL07] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *Proceedings of the 11th European Software Engineering Conference held jointly with 15th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 35–44, September 2007.