# STATIONARY FIELDS IN OBJECT-ORIENTED PROGRAMS

A DISSERTATION SUBMITTED TO
THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Christopher Unkel

October 2009

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Monica S. Lam)  Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Alex Aiken)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(David L. Dill)

Approved for the University Committee on Graduate Studies.

# Abstract

This dissertation introduces stationary fields, which are widely prevalent in Java programs, and which have a property useful for reasoning about programs.

Java programmers can document that the relationship between two objects is unchanging by declaring the field that encodes that relationship to be `final`. This information can be used in program understanding and detection of errors in new code additions. Unfortunately, few fields in programs are actually declared `final`. Programs often contain fields that could be `final`, but are not declared so. Moreover, the definition of `final` has restrictions on initialization that limit its applicability.

We introduce *stationary fields* as a generalization of `final`. A field in a program is stationary if, for every object that contains it, all writes to the field occur before all the reads. Unlike the definition of `final` fields, there can be multiple writes during initialization, and initialization can span multiple methods.

We have developed an efficient algorithm for inferring which fields are stationary in a program, based on the observation that many fields acquire their value very close to object creation. We presume that an object's initialization phase has concluded when its reference is saved in some heap object. We perform precise analysis only regarding recently created objects. Applying our algorithm to real-world Java programs demonstrates that stationary fields are very common. Furthermore, stationary fields are much more common than `final` fields: 44–59% vs. 11–17% respectively in our benchmarks.

Guided by the experimental results, we show two programming idioms that violate the stationary field property, how the definition may be expanded to cover these cases, and corresponding analyses to locate these fields.

We show four applications of stationary fields to program analysis and understanding, as first examples of how stationary fields are useful. Three of the applications are to concurrent programs, an increasingly important area. They take advantage of an important property of stationary fields: because they do not change after their initialization, inter-thread interference through them is impossible. This property makes reasoning about concurrent programs easier, and we expect applications of stationary fields to concurrent programs to be a fruitful avenue of further research.

# Acknowledgements

I offer my gratitude to those who have provided me advice, ideas, camaraderie, patience, and entertainment during my time at Stanford: first and foremost, my adviser Monica Lam; the members of my reading and orals committees; my colleagues at Stanford, Kealia, Sun, and Silicon Image; my friends; and my family. Without their support completing this research would have been impossible instead of challenging.

Finally, my thanks to the staff of MoonBean's Coffee for approximately $2 \times 10^3$ caffe lattes.

# Contents

# List of Figures

# Chapter 1

# Introduction

Tools for program analysis have provided large benefits in the form of increased programmer productivity. Tools have optimized programs, parallelized programs, found errors in programs, and allowed programmers to operate at higher levels of abstraction. Recently, much attention has been given to program analysis to aid in program correctness: tools that help programmers avoid or detect errors.

In order for program analysis tools to function, and to continue to provide benefits, they must be able to reason about the programs. Of course, programmers must also reason about the programs they write. For any important property of a program, the programmer should have a reason to believe the property holds. The key to allowing a tool to understand why the property holds is often recreating the programmer's reasoning. Understanding ways that programmers commonly reason about programs can therefore help us create better tools.

In programs written in an object-oriented programming language, it is often the case that one object has a fixed relationship to another object. For example, an object composed of smaller objects will have a fixed relationship to each of its components. Such relationships are frequently captured in some fields of an object soon after the object is created. The field may be written multiple times during the initialization phase, but it stabilizes before it is used and remains constant for the rest of its object's lifetime. We refer to such fields as *stationary*.

Knowing which fields are stationary provides an object-oriented basis for reasoning about aliases for objects across time. Programmers naturally use these invariants when reasoning about their programs. They don't worry that an object's parent has changed, because they know—or perhaps assume—that no code ever alters a child's parent. When programmers want to know the identity of the window's parent, they will start looking close to where the window is created to find out what the relationship is. Stationary fields offer a different approach to reasoning about objects. We shall show that it is relative easy to find stationary fields in a program. Once we know that a field is stationary, we can conclude that two reads of the same field of an object are the same without tracking all the pointers that may possibly point to the object of interest. This suggests that it is beneficial for compilers and program analysis tools to understand stationary fields, as programmers already do. This may lead to a more precise points-to analysis with less effort.

## 1.1 Final Fields in Java

The presence of unvarying relationships between objects is not a new observation, as this notion motivated the design of Java's final fields. A Java programmer can declare an *object instance field* (Java nomenclature for fields inside objects) to be `final`, which, informally, means that it does not change once initialized. *The Java Language Specification* observes: "Declaring a variable `final` can serve as useful documentation that its value will not change and can help avoid programming errors." [15] The Java compiler provides aid in avoiding mistakes by enforcing properties in code it compiles. These properties combine to ensure that `final` fields, once initialized by the constructor, are not modified by ordinary means. (Declaring a field `final` does not guarantee absolutely that it is constant: it may still be modified by extraordinary means such as reflection, native methods, and other implementation-dependent functionality.) Declaring a field `final` prevents programmers from making the mistake of modifying fields that should be constant, by directing the compiler to reject programs that erroneously change those fields.

The properties of `final` fields are defined so that verifying them fits within Java's model of compilation and dynamic class loading. Specifically, because Java classes may be compiled separately, and because they are then loaded separately, it must be possible to verify the property by examining a single class at a time. In fact, it is possible to verify the property by examining a single method at a time.

## 1.2 Automatic Inference of Final Fields

We speculate that there are many more fields that are being used like they are final without being declared as such. Automatic final field inference is useful because the information can serve as documentation and may be used for other analyses. It can also be used to find errors in subsequent code modifications in the software development process. Suppose a field is used originally as a final field; if subsequent code additions violate this pattern, then it may be worthwhile to flag the inconsistency as a possible error. Many recent projects have used the idea of intent to infer properties in the absence of accurate declarations or documentation [10, 18, 19, 21, 23, 44, 45]. Such properties have been used to find thousands of critical errors in programs.

We can infer if a field can be legally declared as final in a program, by examining the usage pattern of the field in the code. Note that in the presence of dynamic loading, the inferred property holds only for the code examined.

The Java final field modifier is defined to admit a relatively simple verification procedure. As a result, it is not as generally applicable as it could be. There are several important limitations:

- A final field must be assigned exactly once on each execution path through each constructor defined for the class containing it. This definition excludes the case where the field is first defined for the common case, and based on some exceptional conditions, the field can be updated before the field is ever accessed otherwise.

- A final field must not be assigned outside the constructors of the class declaring it. This is restrictive because the constructor might invoke utility procedures to perform initialization; or, an object factory may create and return the object, with the method using the factory responsible for the initialization.

- A final field is defined based on textual properties. A program may use a field as if it is a final even though it contains unexecuted code that violates such properties. This is especially prevalent if large, general Java libraries are used.
- Not all cases where a final field is read before it is assigned are identified and prohibited: those through aliases to `this`, or in methods invoked by the constructor, are not found. However, the fact that most cases are identified and prohibited suggests that it is undesirable to read such a field before it is written.

As we show in our empirical results, the definition of final fields greatly restricts the number of fields encoding unvarying relationships to which it can be applied.

## 1.3 Stationary Fields

In practice, the initialization of a stationary field may span different methods. The field may be written to multiple times during the initialization process. What is important is that the value of the field must stabilize before the field is used. That is, all reads of a stationary field of the same object are guaranteed to follow all the writes, and thus must yield the same value.

Thus, we say that a field $f$ in class $c$ is *stationary* in a program if all writes to field $f$ in every object $o$ of class $c$ occur before all reads of $f$ in $o$.

## 1.4 Inferring Stationary Fields

This thesis presents an algorithm for finding stationary fields in Java programs. The analysis is subject to the usual caveats regarding reflection and native methods, but is conservative otherwise. The results are thus useful for error detection and program understanding tools, but should not be used for program optimization in the general case.

Our algorithm makes the simplifying assumption that an object's stationary fields are initialized soon after the object is created, and that the initialization, which may involve multiple writes and in multiple procedures, is complete before it is released for use through other objects. Releasing an object for use usually means making some other object refer to it: placing it in a list, registering a handler that uses it, etc. In general, once an object is connected to other heap objects, it can be touched by many different pieces of code. Thus, the object's initialization phase usually ends when it is pointed to by some heap object. (In Java, all objects are allocated on the heap; only references to objects and primitive types such as `int` may be stored on the stack.)

We use a flow and context sensitive analysis to track the reads and writes of all pointer variables and heap object fields. We keep track of the identity of each newly created object accurately during its initialization phase. We accurately follow how its references are assigned to local variables and passed as input parameters or return values. However, once the object is stored into some heap reference, it is assumed to exit its initialization phase. We abstract its identity away and represent it with a special *lost* object. Any write to a field of this special object would render the field not stationary. Because we do not have to keep track of the pointees in the fields of heap objects, the algorithm converges quickly.

## 1.5 Experimental Results

We have implemented the algorithms presented in this report to automatically infer final fields and stationary fields in Java programs. We have applied the algorithms to 19 real-world Java programs and analyzed the results. To help validate our static analysis algorithm, we have also developed a dynamic analysis that records, for each field, if a write follows a read of the same location. Clearly, no such pairs should ever be found for any stationary fields identified by our algorithm. We have found this sanity check to be helpful in providing confidence that the algorithm and the implementation are correct. In addition, we also record the number of times stationary fields are read so as to assess if these fields are important in the execution of a program.

## 1.6 Contributions

The key contributions of this thesis are:

- The concept of *stationary fields*, a generalization of Java's `final` that captures a wider range of unchanging fields, primarily by relaxing the requirements on initialization.

- An efficient interprocedural algorithm for finding stationary fields in Java programs.

- Empirical results on the number of inferred final and stationary fields across 19 real-world Java programs. In total, our algorithm analyzed 88292 classes and 620884 methods. We found that stationary fields are widely present in real-world Java programs and more common than `final` fields: 44–59% vs. 11–17%. The experiments also address quantitatively how much each difference between the definitions of stationary and `final` contributes to the prevalence of stationary fields, and show that inferring `final` fields is valuable in its own right.

- Dynamic results showing that stationary fields are an important part of the runtime behavior of programs.

- Several example applications of stationary fields showing how stationary fields can be used for program analysis and understanding. First, we show three examples of stationary fields to concurrent programs. Because stationary fields do not change after their initialization, threads cannot interfere with each other by modifying them. We show how this property can be used to understand and optimize multithreaded programs. In our last application, we examine how stationary fields may be used to correctly maintain the invariants required when implementing an interface used in the Java libraries.

## 1.7 Paper Organization

The remainder of this dissertation is organized as follows. The next chapter explores stationary fields and their relationship to `final` in detail, culminating in our algorithm for inferring stationary fields. In Chapter 3 we present our experiences applying our algorithm to Java programs and our analysis of the results. Guided in part by these results, in Chapter 4 we show two extensions of the definition of stationary fields. In Chapter 5, we show several applications as examples of how stationary fields are useful. We briefly discuss related work in Chapter 6 and conclude in Chapter 7.

# Chapter 2

# Stationary Fields

In the first chapter, we introduced and defined stationary fields. This chapter explores stationary fields in more detail. We first give a brief example showing a stationary field and how the property it gives helps us reason about a program invariant.

Stationary fields are related to Java's `final` fields. For reference, we summarize the definition of `final` fields. The differences between `final` and stationary motivate the design of our algorithm for finding stationary fields.

## 2.1 Stationary and Nonstationary Fields

Consider the code fragment in Figure 2.1, which might appear in a GUI-based program. Assume that this is only a fragment of our button class, but that none of the other methods modify field `parent`. The button's `parent` field shows a common type of stationary field: it encodes the object's position in a hierarchy in which objects may be created or destroyed, but in which they never move. This example also shows the use of a factory method, a common motivation for initialization outside the constructor. The factory method itself calls the constructor for `Button`, so we cannot provide arguments. The example also shows why stationary fields are useful for reasoning about programs: we know that the button is always removed from the window to which it is added because the `parent` field is stationary. Another common case of stationary fields occurs when an object is created as a composite of smaller

```
class Button {
  private Window parent;

  void setParent(Window parent) {
    this.parent = parent;
  }

  void destroy() {
    parent.removeChild(this);
  }

  void onClick() {
    parent.closeWindow();
  }
}


Button b = ButtonFactory.newButton();
b.setParent(mainWindow);
mainWindow.addChild(b);
```

Figure 2.1: Code fragment from and using button object showing stationary fields.

objects.

Some examples of fields we would expect to be nonstationary include those encoding mutable current state of an object. For example, the current position of an iterator changes repeatedly, as does the current state of a pattern matcher such as a tokenizer. Another example is that of a *role* [22], where the state of an object is captured by the existence of a reference from some other object. Changing roles are encoded by nonstationary fields.

## 2.2 Final Fields

This section describes the algorithm used by the Java compiler to verify the legality of code that includes `final` instance fields. We then progress to how we might infer `final` fields.

### 2.2.1 Verifying Final Fields

The *Java Language Specification* provides an algorithm that is used by a Java compiler to verify that programs that declare `final` fields do not misuse them. Without loss of generality, let us just consider the more complex case of a *blank* field, where the final instance field is not initialized in the declaration in the following.

Intuitively, a final field is one that is assigned exactly once during any normal execution of the constructor of the class declaring it, and never assigned elsewhere. (There is no requirement that a final field be assigned should the constructor terminate abruptly, that is, by throwing an exception.) However, since it is undecidable to determine all the possible executed paths statically, the requirements for a final fields are specified by the verification procedure used, as outlined below.

Let `f` be a final field in class `c`.

1. There is an error if any of the methods that are not constructors of `c` contain an assignment to field `f`. Note that constructors of classes derived from `c` do not count as constructors of `c`.

2. Perform a data flow analysis to determine if the field `f` is *definitely assigned*, *definitely unassigned*, or neither, at each program point.

   - Field `f` is definitely unassigned at the entry of the constructor.
   - Executing an assignment "`this.f=`" or "`f=`", provided there is no local variable `f`, leaves field `f` definitely assigned.
   - At a control-flow join point, `f` is definitely assigned or unassigned, respectively, iff it is definitely assigned or unassigned, respectively, on both incoming branches. Otherwise, `f` is considered to be neither.
   - No other statements alter whether `f` is definitely assigned or definitely unassigned.

3. Report an error if:

   - `f` is not definitely assigned at the normal exit of the constructor.

- `f` is not definitely unassigned immediately prior to an assignment “`this.f=`” or “`f=`”.
- `f` is not definitely assigned immediately prior to a use of “`this.f`” or “`f`”.
- the constructor assigns to `f` through some variable other than `this`, e.g. with “`other.f=`”.

Notice that the legality of code that uses `final` can be verified by examining a single method at a time. This property of `final` allows it to fit within Java’s compilation and loading process.

### 2.2.2 Inferring Final

The algorithm for verifying `final` can also be used to infer if a program uses a field like it is final, assuming all the code to be executed is available. We simply assume that a field is final, and execute the procedure for verifying a final field. If no errors are produced, the field may safely be declared `final`, given all the code we have at hand. If we do this for all fields, the result is a superset of fields that are declared final, provided that the input code does not contain errors.

## 2.3 Inferring Stationary Fields

Our algorithm for inferring stationary reads accepts a Java program as input and outputs a set of stationary fields in that program. The algorithm is subject to the same caveats respecting reflection and native methods as final fields.

For stationary fields, programmers generally just read the initialization code and assume that the field is not changed in the rest of the program because they understand the meaning of the field. (Note that there may be errors in the program that violate this assumption.) Like the programmer, our algorithm tracks the reads and writes to each field carefully during initialization. But unlike the programmer, who instinctively knows which fields are stationary, our algorithm needs to analyze if a field is written into after initialization before it can declare the field to be stationary.

Our algorithm makes the simplifying assumption that an object's stationary fields are initialized before its reference is stored into any objects. We say that an object is *lost* once some other object points to it. Thus, a field $f$ is *stationary* if

1. all the reads of field $f$ in any object before it is lost occur after all the writes, and

2. there are no writes to field $f$ upon any lost object.

With this assumption, we can derive a relatively efficient stationary field analysis. Our algorithm uses a hybrid approach to modeling objects. The algorithm tracks aliases precisely for all objects until they are lost; it models all lost objects coarsely by representing them with a special object denoted $\perp$. Conversely, all field dereferences are modeled as yielding the $\perp$ object. Our algorithm is fast because tracking pointees in heap objects is what makes pointer alias analysis expensive.

(We note here that our lost objects are related to escaped objects, as defined by escape analysis. Escaped objects are those whose lifetime exceeds their static scope, or that may be accessed outside some given scope. Lost objects do not necessarily escape (for example if the object that contains the reference does not escape) and escaped objects are not necessarily lost (for example if the object escapes by being returned directly by the method.) Nonetheless, many of the ways objects may escape involve the creation of a reference in the heap. Our lost refers only to the existence of heap references.)

Our analysis is a flow-sensitive and context-sensitive interprocedural summary-based algorithm. It computes a fixed point that summarizes how each method may lose objects and render fields nonstationary in terms of its input parameters and return value. The result of the entire program is given by the summary computed for the entry method of the whole program. We track all the input and return parameters accurately across method invocations, and the assignments of parameters and local variables flow-sensitively.

Newly created objects are given local names preserving the context-sensitivity to the location of their creation. We rename newly created objects that have not been lost, and therefore have no aliases, as they are returned by methods. By giving such an object the name of the method that most recently returned it, we can distinguish between objects created at the same line in the program, if the call stack to that spot is different. This precision is important for analyzing factory methods. Using only local names also keeps the set of potential pointees small, speeding the computation.

### 2.3.1 Program Representation

While our algorithm is defined for the full Java programming language, for the sake of simplicity, we present the analysis over the following simplified language. There is a set of methods $m \in M$. Methods have a set of local variables $x, y \in V$. There is also a set $f \in F$ of object fields, which are accessed using syntax of the form $v.f$. The language has statements $s \in ST$ of the following forms:

- an assignment statement $x = y$
- an object creation statement $x = \texttt{new}()$
- a load statement $x = y.f$
- a store statement $x.f = f$
- a sequence of statements $s_1; s_2$
- an if statement $\texttt{if} \sim \texttt{then}\ s_1\ \texttt{else}\ s_2$
- a while loop $\texttt{while} \sim \texttt{do}\ s$
- a method declaration $m(x_1, \ldots, x_i)\{s; \texttt{return}\ y\}$
- a method invocation statement $x = m(y_1, \ldots, y_i)$

We use the notation $\sim$ as a placeholder for the branch and loop conditions, which are irrelevant to the analysis.

### 2.3.2 Effect of Each Method

The input to our algorithm is a Java program and the main function to be invoked, and the output of the algorithm is a set of stationary fields. The fixed point computation keeps track of all the nonstationary fields found so far in a set called $\bar{S}$; the complement of $\bar{S}$ at the end of the algorithm represents the result of the algorithm.

Our algorithm uses class hierarchy analysis (CHA) to compute the possible call targets at each invocation [8]. It assumes that the target of a call site may be any method consistent with the method being called and the type of the reference through which the method is invoked. We include the class initializers of all used classes as if they occurred at the start of the main function.

The analysis of each method $m$ operates on the following set of abstract objects $o \in O$:

- method input parameters $\mathbf{A} = \{\alpha_1, \alpha_2, \dots\}$. The summary is computed parametrically with respect to the input parameters.
- allocation-site objects. This is the set of objects allocated by $m$ and named by the allocation site. These objects are not aliased at the time of creation.
- call-site objects. This is the set of objects representing objects created by $m$'s callees and are named by the call site.
- the $\perp$ object, used as a placeholder for untracked objects.

A method $m$ and its callees may produce the following side effects that affect the computation of stationary fields:

- generate a set of nonstationary fields, which are added to the set of nonstationary fields found so far $\bar{S}$.
- lose a set of input objects, $L_m \subseteq \mathbf{A}$.
- write to a set of fields for each input parameter, $W_m$.
- read a set of fields for each input parameter, $E_m$.

- return a possible set of objects. It may
  - return some lost object or input parameters. These are represented by the set $R_m$.
  - return some object created by $m$ or its callees that has not been lost. In addition, it may have read a set of fields $D_m$ from the returned object, before it is returned. For the sake of simplicity, and without loss of precision, we assume that a method always returns a new object that has not yet been lost. That is, a fresh call-site object is always created to represent such an object where $m$ is invoked.
- generate a set of alias conditions $C_m$ of the form $\langle \alpha_i, \alpha_j, f \rangle$, indicating that $f$ is nonstationary if arguments $\alpha_i$ and $\alpha_j$ can be the same object.

### 2.3.3 Effect of Each Statement

To compute the effect of each method, the algorithm uses a flow-sensitive analysis statement by statement to track where objects become lost, and the pointer aliases and reads of objects before they are lost. Thus, besides computing the effect of each statement on the terms introduced above for each method summary, the analysis keeps track of the following terms flow-sensitively, before and after every statement:

- $P$: a set of points-to relations $\langle x, o \rangle$ indicating that local variable $x$ may point to object $o$.
- $U$: untracked objects, objects that are lost in this method or its callees.
- $Q$: a set of object fields $o.f$ indicating that field $f$ of object $o$ may have been read previously.

### 2.3.4 Inference Rules

The properties of the solution are given by inference rules in Figure 2.2. The notation $P[x \mapsto B]$ indicates $P$ with all points-to relations involving $x$ removed, and relations added for $x$ pointing to each element of $B$:

(NEW)

$$\frac{P' = P[x \mapsto \beta] \qquad \beta\,\mathit{fresh}}{m, P, U, Q \vdash x = \texttt{new}() \Rightarrow P', U, Q}$$

(ASSIGN)

$$\frac{P' = P\left[x \mapsto \{o|\,\langle y, o\rangle \in P\}\right]}{m, P, U, Q \vdash x = y \Rightarrow P', U, Q}$$

(LOAD)

$$\frac{P' = P[x \mapsto \{\perp\}] \qquad Q' = Q \cup \{o.f|\,\langle y, o\rangle \in P\}}{m, P, U, Q \vdash x = y.f \Rightarrow P', U, Q'}$$

(STORE)

$$\frac{\begin{array}{c} W_m \supseteq \{\alpha.f|\alpha \in \mathbf{A} \wedge \langle x, \alpha\rangle \in P\} \qquad U' = U \cup \{o|\,\langle y, o\rangle \in P\} \\ (\exists o)(\langle x, o\rangle \in P \wedge o \in U) \rightarrow \left(f \in \bar{S}\right) \qquad (\exists o)(o.f \in Q \wedge \langle y, o\rangle \in P) \rightarrow \left(f \in \bar{S}\right) \\ C_m \supseteq \{\langle \alpha_i, \alpha_j, f\rangle\,|i \neq j \wedge \alpha_i \in A \wedge \alpha_i.f \in Q \wedge \langle y, \alpha_j\rangle \in P\} \end{array}}{m, P, U, Q \vdash x.f = y \Rightarrow P, U', Q}$$

(SEQ)

$$\frac{m, P, U, Q \vdash s_1 \Rightarrow P', U', Q' \qquad m, P', U', Q' \vdash s_2 \Rightarrow P'', U'', Q''}{m, P, U, Q \vdash s_1; s_2 \Rightarrow P'', U'', Q''}$$

(IF)

$$\frac{\begin{array}{c} m, P, U, Q \vdash s_1 \Rightarrow P_1, U_1, Q_1 \\ m, P, U, Q \vdash s_2 \Rightarrow P_2, U_2, Q_2 \qquad U' = U_1 \cup U_2 \qquad P' = P_1 \cup P_2 \qquad Q' = Q_1 \cup Q_2 \end{array}}{m, P, U, Q \vdash \texttt{if} \;\sim\; \texttt{then}\; s_1 \;\texttt{else}\; s_2 \Rightarrow P', U', Q'}$$

(WHILE)

$$\frac{m, P', U', Q' \vdash s \Rightarrow P', U', Q' \qquad P' \supseteq P \qquad U' \supseteq U \qquad Q' \supseteq Q}{m, P, U, Q \vdash \texttt{while} \;\sim\; \texttt{do}\; s \Rightarrow P', U', Q'}$$

Figure 2.2: Inference rules for finding nonstationary fields $\bar{S}$.

(METHOD)

$$\frac{\begin{array}{c} R_m \supseteq \{o|\,\langle y,o\rangle \in P' \wedge o \in \mathbf{A}\} \\ (\exists o)(\langle y,o\rangle \in U') \rightarrow (\bot \in R_m) \qquad L_m = \{\alpha|\alpha \in U' \wedge \alpha \in \mathbf{A}\} \\ D_m = \{f|(\exists o)(o.f \in Q' \wedge \langle y,o\rangle \in P')\} \qquad E_m = \{\alpha.f|\alpha \in \mathbf{A} \wedge \alpha.f \in Q'\} \\ P = \{\langle x_1,\alpha_1\rangle,\ldots,\langle x_i,\alpha_i\rangle\} \qquad U = \{\bot\} \qquad Q = \emptyset \qquad m,P,U,Q \vdash s \Rightarrow P',U',Q' \end{array}}{\vdash m(x_1,\ldots,x_i)\{s;\ \texttt{return}\ y\}}$$

(INVOKE)

$$\frac{\begin{array}{c} a_i.f \in W_n \wedge (\exists o)(\langle y_i,o\rangle \in P \wedge o \in U')) \rightarrow \big(f \in \bar{S}\big) \\ W_m \supseteq \{a_i.f|a_i \in \mathbf{A} \wedge (\exists j)(a_j.f \in W_n \wedge \langle y_j,a_i\rangle \in P\} \\ P' = P[x \mapsto B] \qquad B \supseteq (\exists i)(\alpha_i \in R_n \wedge \langle y_i,o\rangle \in P)\} \qquad \beta \in B \\ \beta\ \mathit{fresh} \qquad \bot \in R_n \rightarrow \bot \in B \qquad U' = U \cup \{o|(\exists i)(\alpha_a \in L_n \wedge \langle y_i,o\rangle \in P)\} \\ Q' \supseteq Q \qquad Q' \supseteq \{\beta.f|f \in D_n\} \qquad Q' \supseteq \{o.f|(\exists i)(\alpha_i.f \in E_n \wedge \langle y_i,o\rangle \in P\} \\ (\exists i,j,o)(\langle \alpha_i,\alpha_j,f\rangle \in C_n \\ \wedge \langle y_i,o\rangle \in P \wedge \langle y_j,o\rangle \in P) \rightarrow \big(f \in \bar{S}\big) \\ C_m \supseteq \{\langle \alpha_i,\alpha_j,f\rangle|i \neq j \wedge \alpha_i.f \in Q \wedge \\ (\exists k)(\alpha_k.f \in W_n \wedge \langle y_k,\alpha_j\rangle \in P)\} \\ C_m \supseteq \{\langle \alpha_i,\alpha_j,f\rangle|i \neq j \\ \wedge(\exists k,l)(\langle \alpha_k,\alpha_l,f\rangle \in C_n \qquad \vdash n(z_1,z_2,\ldots)\{\ldots\} \\ \wedge \langle y_k,\alpha_i\rangle \in P \wedge \langle y_l,\alpha_j\rangle \in P)\} \end{array}}{m,P,U,Q \vdash x = n(y_1,y_2,\ldots) \Rightarrow P',U',Q'}$$

Figure 2.3: Inference rules for finding nonstationary fields $\bar{S}$.

$$P[x \mapsto B] \equiv \{\langle y, o\rangle \mid \langle y, o\rangle \in P \land x \neq y\} \cup \{\langle x, o\rangle \mid o \in B\}$$

This notation is used to discard the current contents of a variable and insert new ones, that is, to destructively update the points-to set of a variable.

The inference rules relate the points-to relations, lost objects, and reads prior to a statement to those after it. For example, rule LOAD, which reads:

$$\frac{P' = P[x \mapsto \{\bot\}] \qquad Q' = Q \cup \{o.f \mid \langle y, o\rangle \in P\}}{m, P, U, Q \vdash x = y.f \Rightarrow P', U, Q'}$$

says that $x = y.f$, when executed within method $m$ with prior points-to relations $P$, untracked objects $U$, and prior reads $Q$, results in points-to relations $P'$, unchanged untracked objects, and prior reads $Q'$; $P'$ is $P$ modified such that $x$ points only to $\bot$; and $Q'$ is the union of $Q'$ with $o.f$ for every object to which $y$ may point.

Analysis proceeds from a main function: our final conclusion is $\vdash \texttt{main}(x_1, \dots)\{\dots\}$. The inference rules provide the following:

- Following an allocation $x = \texttt{new}()$, $x$ points to a fresh object that represents all objects created by any execution of that allocation statement (NEW).

- Following an assignment $x = y$, $x$ points to all objects that $y$ pointed to; nothing is lost or read (ASSIGN).

- After a load $x = y.f$, $x$ points to the lost object $\bot$, which is used in place of all objects that are fetched from the heap. Field $f$ has been read on all objects that y may point to (LOAD).

- After a store $x.f = y$, objects pointed to by $y$ are lost. If $x$ pointed to any lost object, then $f$ is nonstationary. If $x$ points to an object passed as a parameter of $m$, then $m$ writes field $f$ of that parameter. If the store overwrites a previous read, the field is nonstationary. If the store is to a parameter, and the same field of some other parameter has previously been read, the store will overwrite the read if the two parameters alias; record this alias condition so that the caller may check it (STORE).

- In a sequence of statements, $s_1; s_2$, the points-to relations after $s_1$ are those before $s_2$; likewise for the lost objects and previous reads (SEQ).
- Anything that may be lost after either branch of an `if` ... `then` ... `else` ... statement is lost after the statement; any points-to relation possible after either branch is possible after the statement; likewise with prior reads (IF).
- The points-to set, lost objects, and prior reads at the exit of a loop must be a fixed point under the body of the loop, and the fixed point must include all points-to relations, lost objects, and prior reads at the entry of the loop (WHILE).
- All substatements of compound statements execute within the same method as the compound statement (SEQ, IF, WHILE).
- The statements inside a method definition are executed in the context of that method; when they begin only $\perp$ is lost, no fields have been read, and each formal parameter points to the corresponding placeholder object. Any parameter object that is lost at the end of the method is lost by the method. If a parameter object is pointed to by the return variable at the end of the method, the function returns that parameter. Any parameter field that has been read at the end of the method is lost by the method. Likewise, if a field of some object has been read, and that object is pointed to by the return variable, then that field has been read from the object returned by the method. If the return variable points to any lost object, the function also returns the lost object (METHOD).
- Rule INVOKE is the most complicated, as invoking a function has effects equivalent to some set of loads, stores, assignments, and allocations. Let $x = n(y_1, \ldots)$ be the invocation. If actual parameter $i$ may refer to an object that is lost after the invocation, and $n$ writes $f$ of its $i^{\text{th}}$ argument, then $f$ is assumed to be nonstationary, because the write in $n$ is to a lost object. This condition refers to the lost objects *after* the invocation because the summary for a method contains no information on the relative ordering of the writes and objects lost. The rule must conservatively assume that some objects are lost,

and then the writes occur. If the called function $n$ writes its $i^{\text{th}}$ argument, and the $i^{\text{th}}$ actual parameter may point to parameter $j$ of the *calling* method $m$, then the calling method writes field $f$ of argument $j$; writes are propagated outward. The properties of invocations just given are parallel to those for stores. Likewise, the fields written by the callee may overwrite previous reads, or imply that a field will be overwritten if two parameters of the caller alias, just as for a store.

After the invocation, $x$ points to anything $n$ may have returned. If $n$ may return parameter $i$, $x$ may point to anything parameter $i$ pointed to; $x$ may also point to $\perp$ if $n$ returns that.

It is also assumed that any function may return some newly created, not lost object, so $x$ points to a fresh placeholder object just as with an allocation. In this fashion, we keep the object names for allocation sites and call sites internal to a method. The effect is to substitute the caller's call-site name for the callee's internal name.

If $n$ loses parameter $i$, then anything pointed to by the argument $i$ is lost after the invocation of $n$.

Any fields of the returned object that were read by the callee have been read on the placeholder return object. If $n$ reads fields from a parameter, then those fields have been read on any object pointed to by the corresponding argument.

### 2.3.5 Solution Procedure

We seek the least solution consistent with the rules given above: we wish $\bar{S}$ as to be small as possible, and likewise for all the sets that comprise the method summaries, and $P$, $U$, and $Q$ for each statement.

The algorithm is efficient because the side effects of each method are represented in terms of its parameters, so no information needs to flow from the caller to the callee. Furthermore, each method can only refer to the placeholder objects representing the input parameters and the newly created objects returned by its callees. All indirect references are modeled simply as the lost object $\perp$.

```
**procedure** *findStationaryFields(main)*:
    *callgraph* = *computeCHACallgraph(main)*
    $\bar{S} := \emptyset$
    **for each** method $m \in$ *callgraph:*
        $W_m := \emptyset; L_m := \emptyset; R_m := \emptyset$
        $C_m := \emptyset; D_m := \emptyset; E_m := \emptyset$
    **for each** *scc* $\in$ *topologicalSort(findSCCs(callgraph))*
        **repeat**
            **for each** method $m \in$ *scc:*
                *summarizeMethod(m)*
        **until** $W_m$, $R_m$, $L_m$, $C_m$, $D_m$, $E_m$
            stabilize for all $m \in$ *scc*

**procedure** *summarizeMethod(m)*:
    **for each** $p \in$ *programPoints(m)*:
        $U_p := \emptyset; P_p := \emptyset; Q_p := \emptyset$
    *visitRule(methodRule(m))*
    **repeat**
        **for each** $s \in$ *statements(m)*:
            *visitRule(statementRule(s))*
        **until** $U_p$, $P_p$, $Q_p$ stabilize for all $p \in$ *programPoints(m)*
    *visitRule(methodRule(m))*

**procedure** *visitRule(r):*
    add minimum items to sets referenced by $r$ so that $r$ holds
```

Figure 2.4: Pseudocode for our algorithm for identifying stationary fields.

Figure 2.4 shows the pseudocode for our algorithm. The procedure is as follows:

1. Compute a call graph for the input program rooted at the entry function using CHA.

2. Begin by initializing sets we aim to compute to the empty set: we have found no fields that are written lost, and we have not discovered the side effects of any method.

3. Find strongly-connected components within the call graph and sort them in topological order.

4. Summarize all methods within each SCC, starting with leaf SCCs, and ending with the SCC containing `main`.

5. Nontrivial strongly connected components contain recursive cycles; the summaries methods within these SCCs depend on each other. Within such an SCC, iterate until a fixed point is reached for the summaries of all methods it contains.

The procedure to summarize a method $m$ is:

1. Maintain a value for $P$, $U$, and $Q$ at each program point; initialize these to the empty set.

2. Set $P$, $U$, and $Q$ prior to the first statement in the method in accordance with the METHOD rule.

3. Iterate over all statements in $m$ until a fixed point for $P$ and $U$ at all points is reached.

4. When visiting each statement, add the minimum items to $P$, $U$, and $Q$ following the statement to satisfy the appropriate rule. (Also add items to the summary for $m$ in $L_m$, or note the discovery of a lost write in $\bar{S}$, when appropriate.)

5. Update the summary for $m$ in accordance with the METHOD rule.

The above procedure constitutes a particular order for visiting the inference rules. We could of course iterate over them in any order until they converge. By using this order, we may recompute the $P$, $U$, and $Q$ for the program points inside a method each time we visit it. This reduces the space required for the algorithm, by eliminating the need to store this information for every program point simultaneously.

Our implementation incorporates an important optimization: once a field $f$ is assumed not to be stationary (known to be contained in $\bar{S}$), our implementation ignores side effects with respect to $f$ in $W$, $C$, $D$, and $E$ for every method. Information about such side effects has no further use; suppressing their storage keeps the method summaries small.

# Chapter 3

# Study of Programs

In this section, we present results of applying our algorithms to infer final and stationary fields in real-world Java programs.

## 3.1 Methodology

We applied our algorithms to a selection of real-world programs in order to gauge its effectiveness and discover the prevalence of stationary fields.

We implemented our analysis using the Joeq compiler infrastructure [42]. Experiments were conducted on an Opteron 150 (2.4 GHz) with Sun JDK 1.5.0_03 running on CentOS 4.

We selected a group of benchmarks from the most-downloaded Java programs on Sourceforge, selecting only those that would compile as standalone programs. (The version we use in each case is what was at the latest revision in the source code repository as of 15 November 2003.) These are all real-world programs with thousands of users. In addition, we include the SPEC JVM98 benchmarks because they come with input sets and thus make dynamic analysis possible. We exclude the JVM testsuite program check from the SPEC JVM98 benchmarks.

Figure 3.1 lists the programs we used as benchmarks, along with a description, statistics about their size, and the run time of our analysis. The figure gives the number of classes in the call graph for each program; the number of methods defined

| project | description | classes | methods | methods in callgraph | analysis time (m) |
|---|---|---|---|---|---|
| azureus | bittorrent client | 1710 | 16243 | 11576 | 5 |
| columba | graphical email client | 4447 | 37400 | 30494 | 25 |
| findbugs | find bugs in Java programs | 1929 | 16804 | 11511 | 6 |
| freetts | speech synthesis system | 3614 | 32291 | 25551 | 18 |
| gruntspud | graphical CVS client | 4479 | 37258 | 31245 | 28 |
| jbidwatcher | auction site tracking tool | 3816 | 33714 | 27375 | 24 |
| jboss | j2ee application server | 3687 | 33045 | 26052 | 20 |
| jedit | programmer's text editor | 4182 | 35856 | 30280 | 29 |
| jetty | HTTP server and servlet container | 1450 | 13001 | 9171 | 4 |
| jgraph | graph objects and algorithms | 3653 | 33090 | 26322 | 23 |
| joone | neural net framework | 3646 | 32937 | 25914 | 21 |
| jxplorer | LDAP browser | 4053 | 35459 | 29087 | 33 |
| l2j | game server | 1601 | 14170 | 10008 | 4 |
| megamek | networked Battletech game | 3983 | 36130 | 30105 | 29 |
| nfcchat | distributed chat client | 3610 | 32349 | 25565 | 19 |
| openwfe | workflow engine | 3669 | 33048 | 25937 | 19 |
| pmd | Java program analyzer | 1599 | 13680 | 10228 | 4 |
| spec/compress | modified Lev-Zimpel compression | 3613 | 32314 | 25608 | 19 |
| spec/db | in-memory database | 3605 | 32321 | 25613 | 19 |
| spec/jack | parser generator | 3654 | 32405 | 25865 | 19 |
| spec/javac | Java compiler | 3769 | 33256 | 26718 | 21 |
| spec/jess | expert shell system | 3749 | 32772 | 26170 | 19 |
| spec/mpegaudio | decompress MP3 audio | 3645 | 32508 | 25823 | 19 |
| spec/mtrt | ray tracer | 3627 | 32406 | 25734 | 19 |
| sshtools | ssh terminal | 3830 | 33827 | 26809 | 23 |
| umldot | make UML class diagrams from Java code | 3672 | 32542 | 26123 | 23 |

Figure 3.1: Benchmark programs used in our experiments.

| | **total** | **% stationary** | | | | | **% nonstationary** | | | | | **%** | **%** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **project** | **fields** | **final** | **uf** | **cgf** | **nf** | **total** | **final** | **uf** | **cgf** | **nf** | **total** | **final** | **sta.** |
| azureus | 1601 | 15 | 16 | 2 | 17 | 50 | 1 | 4 | 0 | 44 | 50 | 17 | 50 |
| columba | 4541 | 10 | 17 | 3 | 17 | 46 | 1 | 7 | 0 | 45 | 54 | 11 | 46 |
| findbugs | 1548 | 12 | 21 | 6 | 20 | 58 | 1 | 3 | 0 | 38 | 42 | 13 | 58 |
| freetts | 3289 | 12 | 17 | 5 | 17 | 51 | 1 | 3 | 0 | 44 | 49 | 13 | 51 |
| gruntspud | 4928 | 13 | 14 | 3 | 14 | 45 | 2 | 8 | 0 | 46 | 55 | 15 | 45 |
| jbidwatcher | 3601 | 11 | 17 | 3 | 16 | 48 | 1 | 4 | 0 | 47 | 52 | 12 | 48 |
| jboss | 3377 | 12 | 17 | 5 | 17 | 51 | 1 | 3 | 0 | 44 | 49 | 13 | 51 |
| jedit | 4511 | 13 | 16 | 3 | 14 | 46 | 1 | 7 | 0 | 45 | 54 | 15 | 46 |
| jetty | 1087 | 14 | 18 | 2 | 21 | 56 | 1 | 3 | 0 | 40 | 44 | 15 | 56 |
| jgraph | 3483 | 12 | 17 | 4 | 17 | 50 | 1 | 3 | 0 | 46 | 50 | 13 | 50 |
| joone | 3368 | 11 | 17 | 5 | 17 | 50 | 1 | 3 | 0 | 45 | 50 | 13 | 50 |
| jxplorer | 4334 | 13 | 14 | 3 | 15 | 45 | 1 | 7 | 0 | 46 | 55 | 14 | 45 |
| l2j | 1219 | 12 | 23 | 3 | 21 | 59 | 2 | 3 | 0 | 36 | 41 | 14 | 59 |
| megamek | 4679 | 9 | 15 | 3 | 16 | 44 | 1 | 11 | 0 | 44 | 56 | 11 | 44 |
| nfcchat | 3300 | 12 | 18 | 5 | 17 | 51 | 1 | 3 | 0 | 44 | 49 | 13 | 51 |
| openwfe | 3375 | 12 | 18 | 5 | 17 | 51 | 1 | 3 | 0 | 44 | 49 | 13 | 51 |
| pmd | 1160 | 14 | 19 | 2 | 22 | 57 | 1 | 3 | 0 | 39 | 43 | 14 | 57 |
| spec/compress | 3290 | 12 | 17 | 4 | 17 | 51 | 1 | 3 | 0 | 45 | 49 | 13 | 51 |
| spec/db | 3280 | 12 | 17 | 4 | 17 | 51 | 1 | 3 | 0 | 45 | 49 | 13 | 51 |
| spec/jack | 3336 | 12 | 17 | 4 | 17 | 51 | 1 | 3 | 0 | 44 | 49 | 13 | 51 |
| spec/javac | 3455 | 11 | 18 | 4 | 16 | 50 | 1 | 3 | 0 | 45 | 50 | 12 | 50 |
| spec/jess | 3347 | 12 | 18 | 4 | 17 | 51 | 1 | 3 | 0 | 44 | 49 | 13 | 51 |
| spec/mpegaudio | 3368 | 11 | 18 | 4 | 17 | 51 | 1 | 3 | 0 | 44 | 49 | 13 | 51 |
| spec/mtrt | 3314 | 12 | 18 | 4 | 17 | 51 | 1 | 3 | 0 | 45 | 49 | 13 | 51 |
| sshtools | 3616 | 11 | 17 | 4 | 20 | 53 | 1 | 3 | 0 | 43 | 47 | 12 | 53 |
| umldot | 3492 | 12 | 16 | 4 | 17 | 49 | 2 | 3 | 0 | 46 | 51 | 14 | 49 |

Figure 3.2: Percentages of reference-typed fields by stationary and final status, excluding packages `sun.*`. All percentages are of total fields. (**final**: declared final; **uf**: undeclared final; **cgf**: final in program's call graph; **nf**: cannot be inferred final; see Section 3.2 for definitions.)

| | total | % stationary | | | | | % nonstationary | | | | | % | % |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **project** | **fields** | **final** | **uf** | **cgf** | **nf** | **total** | **final** | **uf** | **cgf** | **nf** | **total** | **final** | **sta.** |
| azureus | 533 | 18 | 13 | 1 | 7 | 40 | 1 | 7 | 0 | 52 | 60 | 20 | 40 |
| columba | 1313 | 6 | 18 | 2 | 14 | 41 | 1 | 14 | 1 | 44 | 59 | 7 | 41 |
| findbugs | 487 | 7 | 29 | 14 | 18 | 67 | 1 | 3 | 0 | 29 | 33 | 8 | 67 |
| freetts | 186 | 4 | 22 | 6 | 15 | 47 | 1 | 1 | 0 | 51 | 53 | 5 | 47 |
| gruntspud | 1692 | 16 | 9 | 1 | 9 | 36 | 2 | 16 | 0 | 45 | 64 | 19 | 36 |
| jbidwatcher | 499 | 4 | 16 | 3 | 10 | 33 | 1 | 10 | 0 | 56 | 67 | 5 | 33 |
| jboss | 268 | 9 | 24 | 8 | 16 | 57 | 2 | 1 | 0 | 41 | 43 | 11 | 57 |
| jedit | 1401 | 16 | 15 | 1 | 7 | 39 | 1 | 15 | 0 | 44 | 61 | 18 | 39 |
| jetty | 43 | 7 | 37 | 0 | 21 | 65 | 0 | 5 | 0 | 30 | 35 | 7 | 65 |
| jgraph | 400 | 7 | 18 | 4 | 12 | 42 | 0 | 3 | 0 | 55 | 58 | 8 | 42 |
| joone | 289 | 3 | 17 | 11 | 13 | 45 | 1 | 3 | 0 | 51 | 55 | 4 | 45 |
| jxplorer | 1100 | 14 | 7 | 1 | 8 | 30 | 1 | 20 | 0 | 49 | 70 | 16 | 30 |
| l2j | 157 | 2 | 60 | 8 | 17 | 87 | 0 | 3 | 0 | 10 | 13 | 2 | 87 |
| megamek | 1586 | 4 | 13 | 1 | 14 | 31 | 1 | 26 | 0 | 42 | 69 | 5 | 31 |
| nfcchat | 221 | 4 | 28 | 7 | 13 | 52 | 1 | 1 | 0 | 46 | 48 | 5 | 52 |
| openwfe | 286 | 4 | 22 | 10 | 14 | 50 | 1 | 3 | 0 | 45 | 50 | 5 | 50 |
| pmd | 116 | 6 | 33 | 1 | 25 | 65 | 0 | 8 | 0 | 28 | 35 | 6 | 65 |
| spec/compress | 210 | 4 | 23 | 6 | 14 | 47 | 1 | 3 | 0 | 49 | 53 | 5 | 47 |
| spec/db | 200 | 4 | 24 | 6 | 14 | 48 | 1 | 2 | 0 | 50 | 52 | 5 | 48 |
| spec/jack | 256 | 3 | 23 | 5 | 18 | 50 | 1 | 4 | 0 | 46 | 50 | 4 | 50 |
| spec/javac | 375 | 2 | 23 | 3 | 10 | 39 | 1 | 5 | 2 | 53 | 61 | 3 | 39 |
| spec/jess | 266 | 3 | 24 | 5 | 16 | 47 | 1 | 6 | 0 | 45 | 53 | 4 | 47 |
| spec/mpegaudio | 288 | 3 | 32 | 5 | 17 | 56 | 1 | 2 | 0 | 41 | 44 | 3 | 56 |
| spec/mtrt | 234 | 3 | 24 | 5 | 15 | 48 | 1 | 3 | 0 | 48 | 52 | 4 | 48 |
| sshtools | 529 | 4 | 23 | 7 | 33 | 67 | 0 | 2 | 0 | 31 | 33 | 4 | 67 |
| umldot | 402 | 14 | 11 | 3 | 9 | 37 | 4 | 2 | 0 | 57 | 63 | 18 | 37 |

Figure 3.3: Percentages of reference-typed fields by stationary and final status, excluding packages `java.*`, `javax.*`, and `sun.*`.
All percentages are of total fields. (**final**: declared final; **uf**: undeclared final; **cgf**: final in program's call graph; **nf**: cannot be inferred final; see Section 3.2 for definitions.)

| | total | % stationary | | | | | % nonstationary | | | | | % | % |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **project** | **fields** | **final** | **uf** | **cgf** | **nf** | **total** | **final** | **uf** | **cgf** | **nf** | **total** | **final** | **sta.** |
| azureus | 1099 | 6 | 10 | 3 | 24 | 42 | 1 | 2 | 0 | 54 | 58 | 6 | 42 |
| columba | 2409 | 3 | 11 | 5 | 15 | 34 | 0 | 2 | 0 | 64 | 66 | 3 | 34 |
| findbugs | 803 | 6 | 11 | 8 | 19 | 45 | 1 | 2 | 0 | 52 | 55 | 7 | 45 |
| freetts | 2072 | 3 | 12 | 7 | 15 | 37 | 0 | 2 | 1 | 60 | 63 | 3 | 37 |
| gruntspud | 2437 | 3 | 11 | 4 | 17 | 35 | 0 | 2 | 1 | 62 | 65 | 3 | 35 |
| jbidwatcher | 2238 | 3 | 12 | 5 | 16 | 36 | 0 | 2 | 0 | 62 | 64 | 3 | 36 |
| jboss | 2095 | 3 | 12 | 7 | 16 | 37 | 0 | 2 | 1 | 60 | 63 | 3 | 37 |
| jedit | 2586 | 3 | 11 | 4 | 14 | 32 | 0 | 2 | 0 | 65 | 68 | 3 | 32 |
| jetty | 615 | 9 | 11 | 4 | 17 | 41 | 1 | 2 | 0 | 56 | 59 | 10 | 41 |
| jgraph | 2140 | 3 | 12 | 6 | 15 | 36 | 0 | 1 | 1 | 62 | 64 | 3 | 36 |
| joone | 2126 | 3 | 12 | 7 | 15 | 37 | 0 | 2 | 1 | 60 | 63 | 3 | 37 |
| jxplorer | 2251 | 3 | 11 | 4 | 16 | 34 | 0 | 2 | 0 | 64 | 66 | 3 | 34 |
| l2j | 891 | 6 | 20 | 11 | 17 | 55 | 1 | 1 | 0 | 43 | 45 | 7 | 55 |
| megamek | 2662 | 2 | 10 | 5 | 15 | 32 | 0 | 2 | 1 | 65 | 68 | 3 | 32 |
| nfcchat | 2072 | 3 | 12 | 7 | 15 | 37 | 0 | 2 | 1 | 60 | 63 | 3 | 37 |
| openwfe | 2098 | 3 | 12 | 7 | 15 | 37 | 0 | 2 | 1 | 60 | 63 | 3 | 37 |
| pmd | 665 | 8 | 11 | 4 | 18 | 42 | 1 | 2 | 0 | 55 | 58 | 9 | 42 |
| spec/compress | 2098 | 3 | 12 | 7 | 15 | 37 | 0 | 2 | 1 | 61 | 63 | 3 | 37 |
| spec/db | 2082 | 3 | 12 | 7 | 15 | 37 | 0 | 2 | 1 | 61 | 63 | 3 | 37 |
| spec/jack | 2130 | 3 | 12 | 7 | 15 | 37 | 0 | 2 | 1 | 61 | 63 | 3 | 37 |
| spec/javac | 2154 | 3 | 12 | 7 | 15 | 36 | 0 | 2 | 1 | 61 | 64 | 3 | 36 |
| spec/jess | 2191 | 3 | 14 | 6 | 15 | 39 | 0 | 2 | 1 | 59 | 61 | 3 | 39 |
| spec/mpegaudio | 2144 | 3 | 12 | 7 | 15 | 36 | 0 | 1 | 1 | 62 | 64 | 3 | 36 |
| spec/mtrt | 2116 | 3 | 12 | 7 | 15 | 37 | 0 | 2 | 1 | 61 | 63 | 3 | 37 |
| sshtools | 2173 | 3 | 12 | 6 | 16 | 37 | 0 | 1 | 1 | 61 | 63 | 3 | 37 |
| umldot | 2088 | 3 | 12 | 6 | 16 | 36 | 0 | 2 | 1 | 62 | 64 | 3 | 36 |

Figure 3.4: Percentages of primitive-typed fields by stationary and final status, excluding packages `sun.*`. All percentages are of total fields. (**final**: declared final; **uf**: undeclared final; **cgf**: final in program's call graph; **nf**: cannot be inferred final; see Section 3.2 for definitions.)

by those classes; and the number of methods actually included in the call graph for the program. These numbers include library methods and classes. Many of these are sizeable programs. Gruntspud, the largest, uses over four thousand classes, approximately half of which are in the application itself rather than the Java libraries.

The number of methods in the call graph is typically about two-thirds of those defined by the included classes. This indicates that many applications that use classes, especially from libraries, use only a portion of the functionality that those classes offer. The benchmarks' call graphs fall into two categories: less than 12,000 methods, or more than 25,000 methods. This is because the call graph for the Java libraries includes some very large strongly connected components. The smaller call graphs exclude one of these very large components.

The presence of this large component has a strong effect on the analysis time. Those applications that do not include it are completed in under five minutes; those that do include it generally take approximately twenty minutes to half an hour. The algorithm must iterate over all the methods in each SCC until a fixed point is reached for their summaries; in these cases one SCC includes nearly half the program. Even on the largest programs, analysis is complete in under two hours, an acceptable amount of time for yielding whole-program information. Our implementation is not highly optimized. It would be possible to apply techniques such as precomputation of results for the system library to minimize the run time.

## 3.2 Static Analysis of Stationary and Final Fields

Figures 3.2, 3.3, and 3.4 show the results of applying our algorithm to our benchmarks. Fields are classified either stationary or nonstationary. Only instance fields from which the methods in the call graph include at least one load are included. Figures 3.2 and 3.4 include all packages except `sun.*`, which contains primarily JVM internals, for fields of reference and primitive type respectively. The results for reference-typed fields in Figure 3.3 also exclude the core Java libraries in packages `java.*` and `javax.*` and should generally represent the application itself and its libraries.

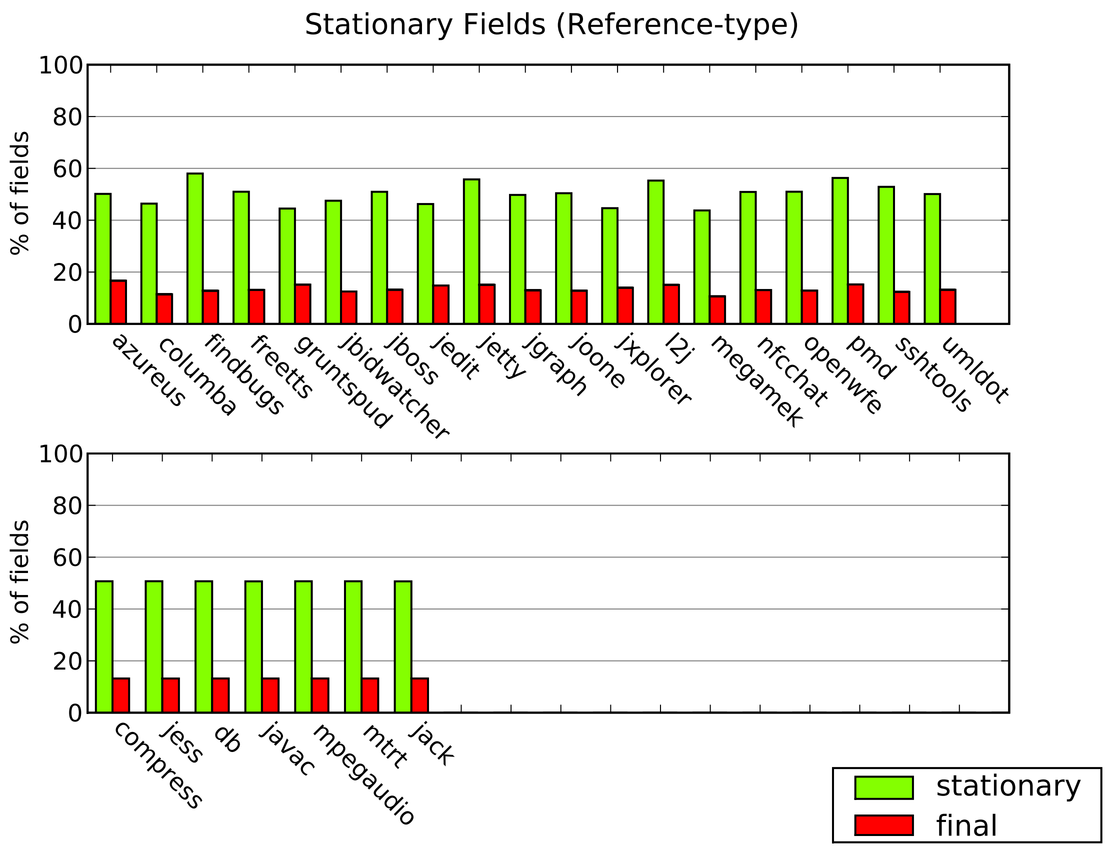

Figure 3.5: Stationary fields found by our inference algorithm: reference-typed fields, application only

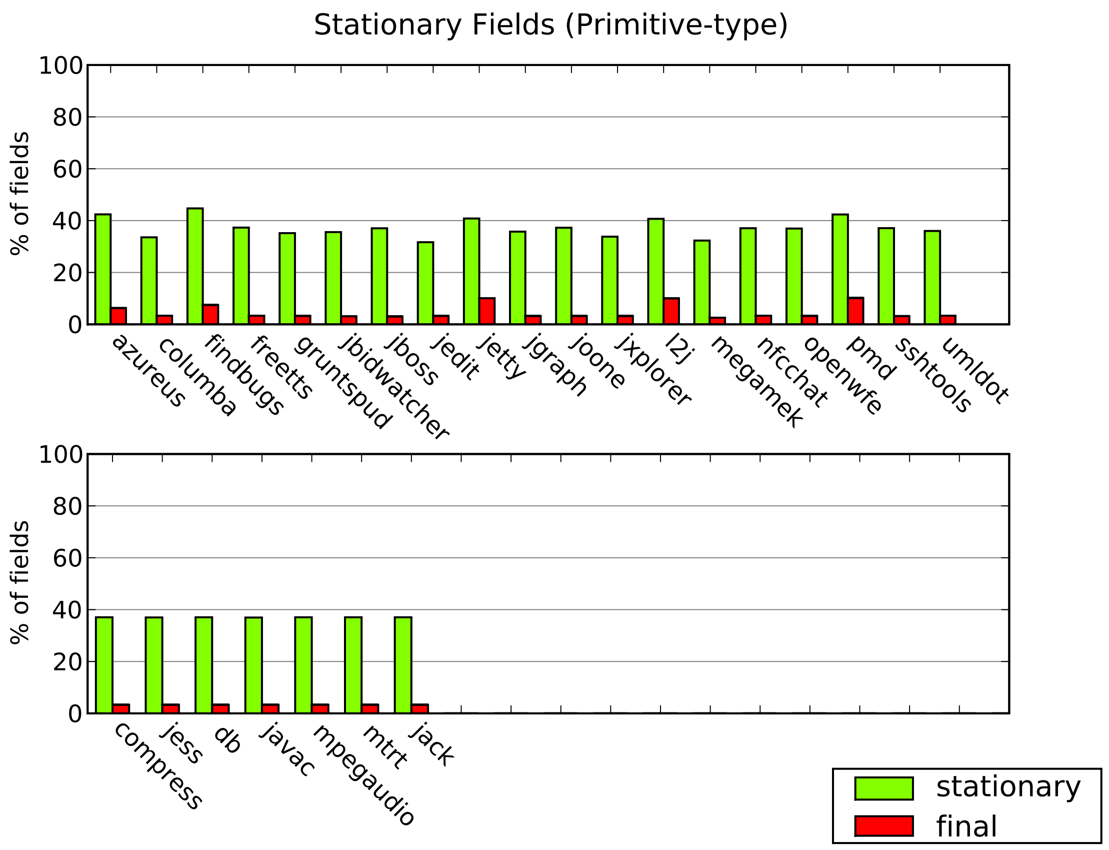

Figure 3.6: Stationary fields found by our inference algorithm: primitive-typed fields

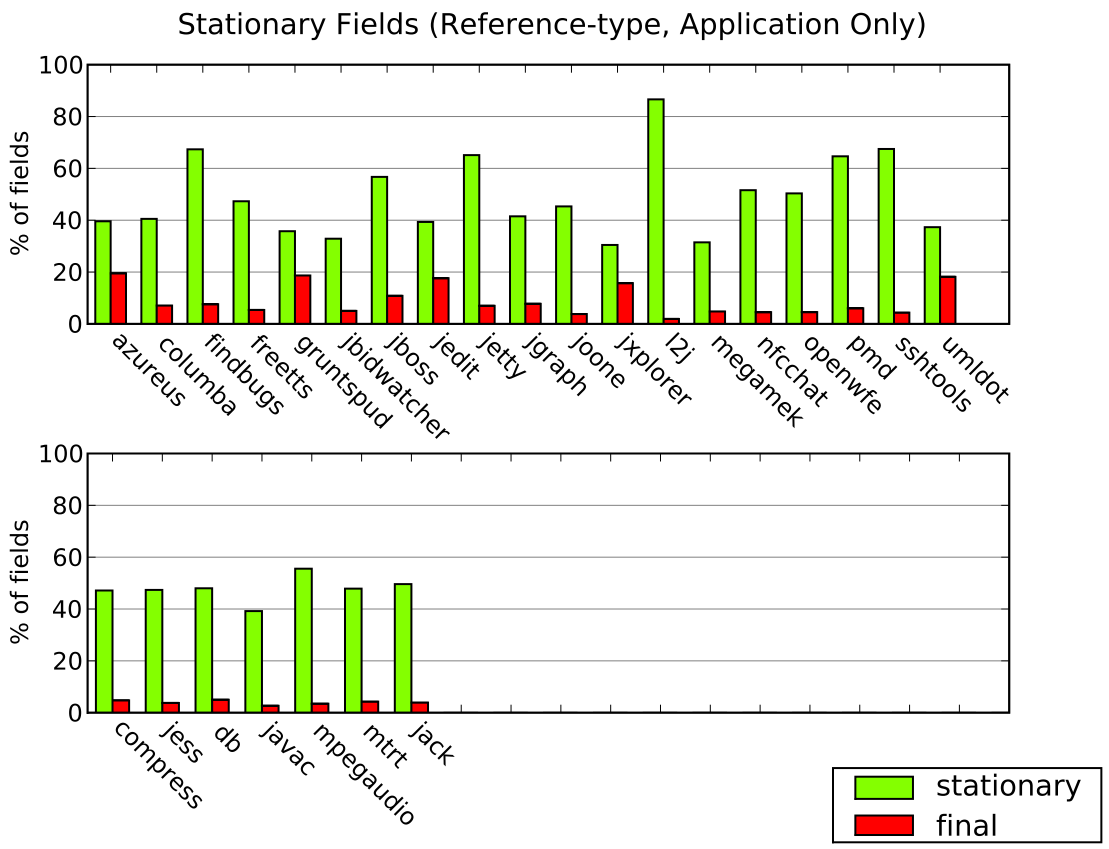

Figure 3.7: Stationary fields found by our inference algorithm: reference-typed fields, application only

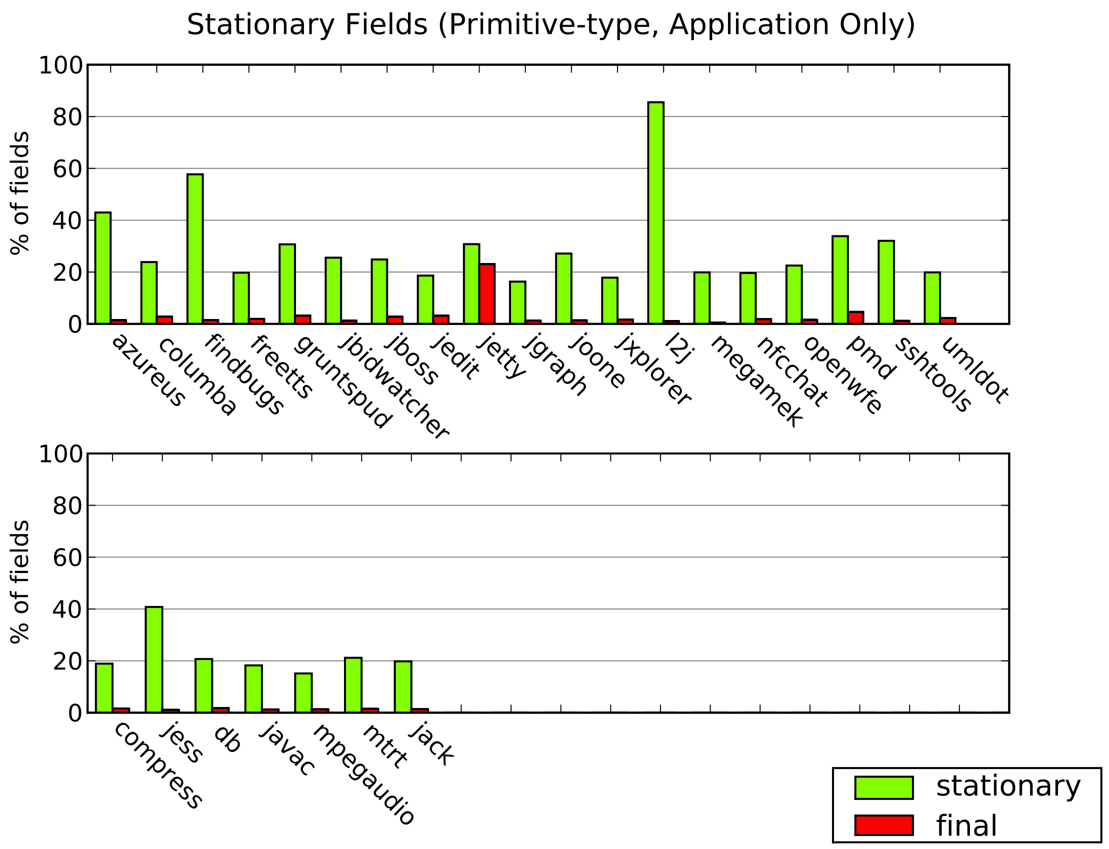

Figure 3.8: Stationary fields found by our inference algorithm: primitive-typed fields, application only

The results show that stationary fields are prevalent in Java programs: for reference-typed fields, the stationary percentage ranges from 44 to 59 in the programs, when the Java libraries are included. Even when the Java libraries are excluded, more than 30% of fields are stationary in every application. In application portion of some of the smaller programs, more than 60% of the fields are stationary. For fields that hold primitive types, stationary fields are modestly less common, but are still more than 30% in all the benchmarks.

It is surprising that about half of all the fields in each of these Java programs program are stationary. This suggests that significant portions of Java programs are "functional" in nature, where data are initialized and not changed later. Obviously, garbage collection plays an important role in encouraging this style of programming.

### Comparing Stationary with Final Fields

We also analyzed all the fields of the programs to determine if they were or could be declared final, using the algorithm given in Section 2.2.2. In the figures, fields are classified as one of:

- declared as **final**;
- undeclared final (**uf**), fields that are not declared final but for which such a declaration would be legal considering all the code contained in each program and its libraries;
- call-graph final (**cgf**), fields that are not final or undeclared final, but that could legally be declared final when considering only the code within the call graph we used for each program; or
- not final (**nf**).

There are many fewer declared `final` fields than stationary fields: less than 20% of the fields are declared final in both the full programs as well as just the application codes. The results suggest that automatic inference of final fields is useful, as many fields are used like they are final but are not declared as such.

Nonetheless, almost 20% of the fields in full programs are found to be stationary yet cannot be inferred to be final. This means that the relaxed definition of initialization of stationary fields is significant.

### 3.2.1 Stationary Nonfinal Fields

The table in Figure 3.9 quantifies the reasons why stationary is more applicable than final. The three main reasons are fields that are:

- **uninitialized**, potentially uninitialized at the end of a constructor;
- **multiply initialized**, potentially multiply initialized within the constructor; or
- **outside constructor**, assigned outside the constructor of the class defining that field.

Notice that fields can belong to more than one category, so the columns sum to more than 100%. The results show that potential lack of initialization is the most common reason that fields cannot be declared final, appearing for a majority of fields. Assignment outside the constructor is next most common, occurring for around half of fields. Multiple assignment in the constructor is not common, but does occur in a handful of places in all programs.

In every program, the number of uninitialized fields is strictly greater than the number of fields assigned outside the constructor. This might be surprising; after all, if we don't assign the field inside the constructor, we would certainly expect an assignment somewhere else. Bear in mind, however, that this result only reports that a field is *potentially* unassigned. Constructors may leave fields unassigned on some path, allowing them to retain their default null value. Also, it is possible for some constructors of a class to initialize a field while others do not.

### 3.2.2 Inferred Final Fields

Our results show that many programs seem to be missing opportunities to declare fields to be final—even including the Java libraries, fewer than half the fields that

| **project** | **uninit.** | **init. outside ctor.** | **mult. init.** |
|---|---|---|---|
| azureus | 88 | 42 | 3 |
| columba | 90 | 32 | 5 |
| findbugs | 87 | 46 | 3 |
| freetts | 90 | 33 | 5 |
| gruntspud | 92 | 36 | 5 |
| jbidwatcher | 90 | 33 | 6 |
| jboss | 90 | 33 | 6 |
| jedit | 92 | 31 | 5 |
| jetty | 87 | 42 | 3 |
| jgraph | 91 | 32 | 6 |
| joone | 91 | 33 | 5 |
| jxplorer | 90 | 33 | 6 |
| l2j | 88 | 43 | 3 |
| megamek | 89 | 34 | 6 |
| nfcchat | 90 | 32 | 5 |
| openwfe | 90 | 32 | 5 |
| pmd | 89 | 43 | 2 |
| spec/compress | 90 | 33 | 5 |
| spec/db | 90 | 33 | 5 |
| spec/jack | 90 | 34 | 5 |
| spec/javac | 90 | 33 | 5 |
| spec/jess | 90 | 33 | 5 |
| spec/mpegaudio | 90 | 32 | 5 |
| spec/mtrt | 90 | 33 | 5 |
| sshtools | 76 | 43 | 5 |
| umldot | 90 | 32 | 6 |

Figure 3.9: Reasons why stationary fields cannot be declared `final`. Percentages of stationary, non-final, reference-typed fields.

could be marked final are.

There are several reasons why there are so many undeclared final fields. First, our analysis only considers classes that are used by the program; there may be other classes inside these libraries that mutate these fields. Second, there may be public fields for which legitimate client code could modify them. Third, the declaration of final could simply be missing.

When we look at final in the application code itself, the percentage of declared final fields is even lower. Indeed, some applications, such as l2j, do not declare *any* fields to be `final`! (The 3 final fields reported for l2j are not within the application itself, but rather within the `com.sun.management` package.) This is probably because there is no feedback from the compiler when a `final` declaration is missing, as it is compiling only a single class at a time. However, many of these fields are private, so the fields could not be modified outside the translation units. Java compilers might consider emitting a warning when a private field could be declared final but is not, in the same vein that some emit warnings for private fields that are never read within the body of the defining class.

The number of callgraph final fields suggests that many library methods that mutate some fields are unused by applications. That is, even though a library defines an object as mutable, an application treats it as fixed. It could also represent object configuration parameters that applications tend to leave in the default settings. The presence of call-graph-final fields outside of the Java libraries indicates some dead code within applications, but also that applications are using only portions of the (non-Java core) libraries that they are packaged with.

### 3.2.3 Nonstationary Final Fields

In general, we would expect most `final` fields to be stationary as well. However, there are a number of nonstationary fields reported in the final, undeclared final, and call-graph final categories, primarily undeclared final. First, there is a category of fields that are final but that cannot be shown stationary by our analysis. It is possible for a constructor to create a reference to `this` in another object, or to pass `this` as an argument to a function that does so. Any field initialized after that point in the constructor is nonstationary. In our experience, a common reason is the creation of a object of non-static inner class during construction. The inner class contains a reference to its outer class, and so the outer class is necessarily lost. For example, classes in the Java SWING library often create an inner class to use as an event handler. This implies that no field in a class derived from these classes is found to be stationary, because base class constructors execute before derived class constructors. The programs with significant numbers of nonstationary, undeclared final fields are all GUI programs.

Second, because our algorithm is conservative, it is also possible for an imprecision within it to report a stationary field as nonstationary, even if no lost writes or overwritten reads are in fact possible. The primary cause of this is the lack of relative ordering information between what is written and what is lost in the summaries of methods.

### 3.2.4 Semi-Stationary Fields

Stationary fields are defined such that they are read only after all the writes have been performed. Sometimes, a field may need to be read as part of the initialization. For example, a program may choose to write to a field only if it has not previously been assigned. This would require that the field be read first to check for nullness before the write operation. Such an action would render the field nonstationary by our definition.

Even though reads during initialization may access fields before they stabilize, our algorithm can identify the reaching definitions for such reads accurately as long as the objects accessed have not been lost. Thus, we refer to a field as *semi-stationary* if all the reads of a field of an object either occur before the object is lost, or after all the writes to the field have taken place. Semi-stationary fields are interesting because precise alias information is available for all their writes, and definitions reaching the read accesses are well identified.

Figure 3.10 compares the fraction of fields that are stationary and semi-stationary.

| | **all fields** | | **application only** | |
|---|---|---|---|---|
| **project** | **sta.** | **semi-sta.** | **sta.** | **semi-sta.** |
| azureus | 50 | 52 | 40 | 40 |
| columba | 46 | 48 | 41 | 42 |
| findbugs | 58 | 61 | 67 | 69 |
| freetts | 51 | 53 | 47 | 49 |
| gruntspud | 45 | 46 | 36 | 37 |
| jbidwatcher | 48 | 49 | 33 | 36 |
| jboss | 51 | 53 | 57 | 59 |
| jedit | 46 | 48 | 39 | 40 |
| jetty | 56 | 59 | 65 | 72 |
| jgraph | 50 | 51 | 42 | 42 |
| joone | 50 | 52 | 45 | 48 |
| jxplorer | 45 | 47 | 30 | 33 |
| l2j | 59 | 62 | 87 | 89 |
| megamek | 44 | 45 | 31 | 32 |
| nfcchat | 51 | 53 | 52 | 53 |
| openwfe | 51 | 53 | 50 | 52 |
| pmd | 57 | 60 | 65 | 71 |
| spec/compress | 51 | 53 | 47 | 50 |
| spec/db | 51 | 53 | 48 | 52 |
| spec/jack | 51 | 53 | 50 | 55 |
| spec/javac | 50 | 52 | 39 | 44 |
| spec/jess | 51 | 52 | 47 | 50 |
| spec/mpegaudio | 51 | 53 | 56 | 57 |
| spec/mtrt | 51 | 53 | 48 | 52 |
| sshtools | 53 | 55 | 67 | 68 |
| umldot | 49 | 50 | 37 | 38 |

Figure 3.10: Percentage of reference-typed fields that are semi-stationary. **All fields** excludes `sun.*`; **application only** also excludes `java.*` and `javax.*`.

The difference between the two is relatively small, never comprising more than 7% of total fields. This suggests that the simpler and stronger definition of stationary fields provides most of the benefits of our approach to track objects carefully before they are lost.

We note that if all objects of a certain class never escape into the heap, but are used and modified locally, then all the fields in the class are considered semi-stationary. For example, in the SPEC program db, the top-level database object has this property. Its fields are repeatedly changed, but it is only stored in local variables.

## 3.3 Implementation Validation with Dynamic Analysis

As a way of validating our implementation, we compared the dynamic behavior of the SPEC JVM98 programs with the results of our inference algorithm. We used bytecode rewriting to instrument the programs, recording all instance field reads and writes, and when heap references to objects were created. For ease of implementation, we only instrumented the application code, and not the Java libraries (specifically, we excluded classes in the `java` and `sun` packages) This allowed the instrumentation to use the Java libraries without interfering with the bootstrapping process of the JVM. Behavior of native methods and reflection was not captured by bytecode instrumentation.

In all cases where a field was overwritten at runtime, or where it was written while after a heap reference existed, the static analysis correctly identified this possibility. While not exhaustive, the dynamic analysis serves as a substantial independent test suite, which our implementation passes.

## 3.4 Dynamic Analysis of Stationary and Final Fields

To determine if stationary and final fields are important to a program's execution, we instrumented the SPEC JVM98 benchmarks programs to count the number of times each instance field was read during execution. Figures 3.11 and 3.12 show the results,

| | total | % stationary | | | | | % nonstationary | | | | | % | % |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **project** | **reads** | **final** | **uf** | **cgf** | **nf** | **total** | **final** | **uf** | **cgf** | **nf** | **total** | **final** | **sta.** |
| spec/compress | 1144M | 0 | 33 | 0 | 13 | 46 | 0 | 38 | 0 | 16 | 54 | 0 | 46 |
| spec/db | 231M | 14 | 0 | 0 | 0 | 14 | 0 | 22 | 0 | 64 | 86 | 14 | 14 |
| spec/jack | 47M | 2 | 18 | 0 | 11 | 32 | 0 | 2 | 0 | 67 | 68 | 2 | 32 |
| spec/javac | 111M | 1 | 6 | 0 | 17 | 23 | 0 | 1 | 0 | 75 | 77 | 1 | 23 |
| spec/jess | 104M | 0 | 2 | 0 | 2 | 4 | 0 | 3 | 0 | 94 | 96 | 0 | 4 |
| spec/mpegaudio | 492M | 0 | 79 | 0 | 6 | 85 | 0 | 0 | 0 | 15 | 15 | 0 | 85 |
| spec/mtrt | 129M | 0 | 14 | 0 | 36 | 50 | 0 | 28 | 0 | 22 | 50 | 0 | 50 |

Figure 3.11: Percentages of dynamic reads of reference-typed fields, excluding packages `sun.*`. All percentages are of total fields. (**final**: declared final; **uf**: undeclared final; **cgf**: final in program's call graph; **nf**: cannot be inferred final; see Section 3.2 for definitions.)

| | total | % stationary | | | | | % nonstationary | | | | | % | % |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **project** | **reads** | **final** | **uf** | **cgf** | **nf** | **total** | **final** | **uf** | **cgf** | **nf** | **total** | **final** | **sta.** |
| spec/compress | 795M | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 99 | 99 | 0 | 1 |
| spec/db | 92M | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 100 | 100 | 0 | 0 |
| spec/jack | 63M | 0 | 1 | 0 | 11 | 12 | 0 | 0 | 0 | 88 | 88 | 0 | 12 |
| spec/javac | 132M | 0 | 5 | 0 | 0 | 5 | 0 | 0 | 0 | 95 | 95 | 0 | 5 |
| spec/jess | 139M | 0 | 19 | 0 | 0 | 19 | 3 | 0 | 0 | 78 | 81 | 3 | 19 |
| spec/mpegaudio | 304M | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 0 | 98 | 98 | 0 | 2 |
| spec/mtrt | 167M | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 100 | 100 | 0 | 0 |

Figure 3.12: Percentages of dynamic reads of primitive-typed fields, excluding packages `sun.*`.
All percentages are of total fields. (**final**: declared final; **uf**: undeclared final; **cgf**: final in program's call graph; **nf**: cannot be inferred final; see Section 3.2 for definitions.)

for reference- and primitive-typed fields respectively. The dynamic numbers exhibit more variation than the static numbers, ranging from 4 to 78% for reference fields. These numbers suggest that stationary fields are accessed in real programs. Reads of final fields are rare; reads of undeclared final fields on the other hand are common, demonstrating the value of inference of final fields. For fields of primitive type, most reads are directed at nonstationary fields; this suggests that fields of reference and primitive types are used differently in programs. For example, it is common to create a data structure out of stationary reference fields, and then the primitive fields within that structure are accessed and mutated.

We also analyzed some of the fields most frequently used during execution to gain a better understanding of how stationary and final fields are used in practice.

Among frequently accessed stationary fields are such things as: the input stream of a scanner, the table for a hash table, as well as the hash code and key of a table entry, virtual "this" pointers (e.g. "`this$0`") used by inner classes, and input buffers.

79% of the reference field accesses in mpegaudio are directed at stationary fields that can be declared final, but are not declared as such in the program. The application mpegaudio is obfuscated so it is hard to tell what these fields are exactly. However, they are of type array of array of float. From this, and given that the program is an MPEG audio decompressor, we may guess that these are statically allocated arrays for signal processing.

36% of reference fields in mtrt are to stationary but not final fields. The main reason is that the initialization of an important field is performed outside the constructor, in a method called `Initialize`. This again shows that it is important to relax the initialization constraints.

Among nonstationary fields, indexes are the most common, such as into buffers and other arrays and vectors. Also appearing as nonstationary fields are reallocated structures, such as an input or vector buffer that may need to be resized, or a lazily generated list of database entries in sort order. In mtrt, 28% of the field reads are directed at nonstationary fields but can be declared as final fields. In fact, these fields are also stationary. Our algorithm cannot identify these fields as such because of imprecision discussed in Section 3.2.3. For primitive-typed fields, the only substantial reads of stationary, non-final fields occur in jack; the fields carry information about whether a buffer is read-only or writeable.

## 3.5 Bounds Derived from Dynamic Analysis

As described in Section 3.3, we used bytecode instrumentation to classify fields as either dynamically stationary or nonstationary. By combining these results with the results of our static analysis, we may classify fields as:

- **Statically stationary.** Applying our analysis shows that these fields are stationary in all executions of the program.
- **Dynamically stationary.** These fields were not modified after being read during the benchmark execution.
- **Nonstationary.** These fields were modified after being read during the benchmark execution.

Some fields are dynamically stationary but not statically stationary: our analysis was not able to show that these fields were stationary, but they were during this particular execution of the program.

Figure 3.13 shows the results of classifying the fields accessed during the execution of the SPEC benchmark programs in this way, displaying the fields found to be in the statically stationary and dynamically stationary categories as a percentage of the fields used during execution.

Note that fields represented here are only a sample of the overall fields: those that were accessed during benchmark execution. The numbers for statically stationary fields therefore differ from the results presented earlier, which were for all fields used in the static call graph of the program. Also, as mentioned earlier, this dynamic analysis excludes the Java standard libraries for ease of implementation. The results therefore represent only the application code, where we may expect relatively good coverage given that these programs are constructed as benchmarks.

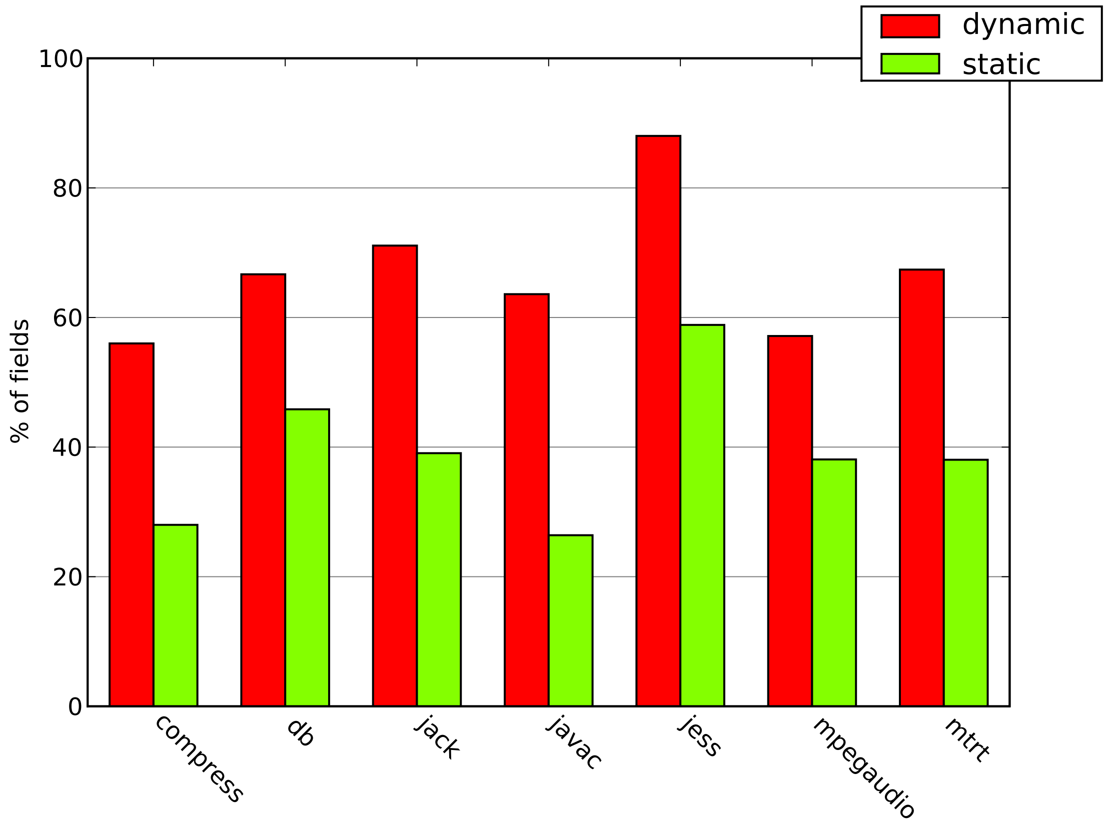

Figure 3.13: Statically and dynamically stationary fields in the SPECJVM benchmarks

These dynamic results are subject to the usual limitations of any dynamic analysis: they are limited by the coverage of the program executions monitored. They therefore can only give an upper bound on the possible stationary fields, and it is not a tight upper bound: some of the fields found to be dynamically stationary may be nonstationary in another execution, for example given input data other than that provided for the benchmarks.

Nonetheless, the dynamic results show that our static analysis has captured a good percentage of the fields that may be stationary, finding at least half of the dynamically stationary fields to be statically stationary across the benchmarks.

## 3.6 Distribution of Stationary and Nonstationary Fields Within Objects

In previous sections we have measured and discussed the overall prevalence of stationary fields. However, instance fields are not created, stored, and manipulated independently from each other. They are grouped together into objects of difference classes.

One thing we might measure is how stationary and nonstationary fields are grouped together into classes: Is the stationary or nonstationary status of objects within on object related? Are there objects composed entirely of stationary fields? Nonstationary fields?

The charts in Figures 3.15–3.17 show how stationary fields are grouped into classes. Figure 3.14 shows a larger version of this chart, for the Azureus application.

On each of these charts, the $x$-axis represents the number of stationary fields within a class, and the $y$-axis the number of nonstationary fields. At each location, the area of the circle represents the number of classes in the program with the corresponding number of stationary and nonstationary fields as determined by our static analysis. Any class with more than 20 stationary or nonstationary fields is represented at the upper or right edge of the chart. Lines of constant $x + y$ correspond to objects of constant total size (number of fields.) Along the bottom ($y = 0$) of each

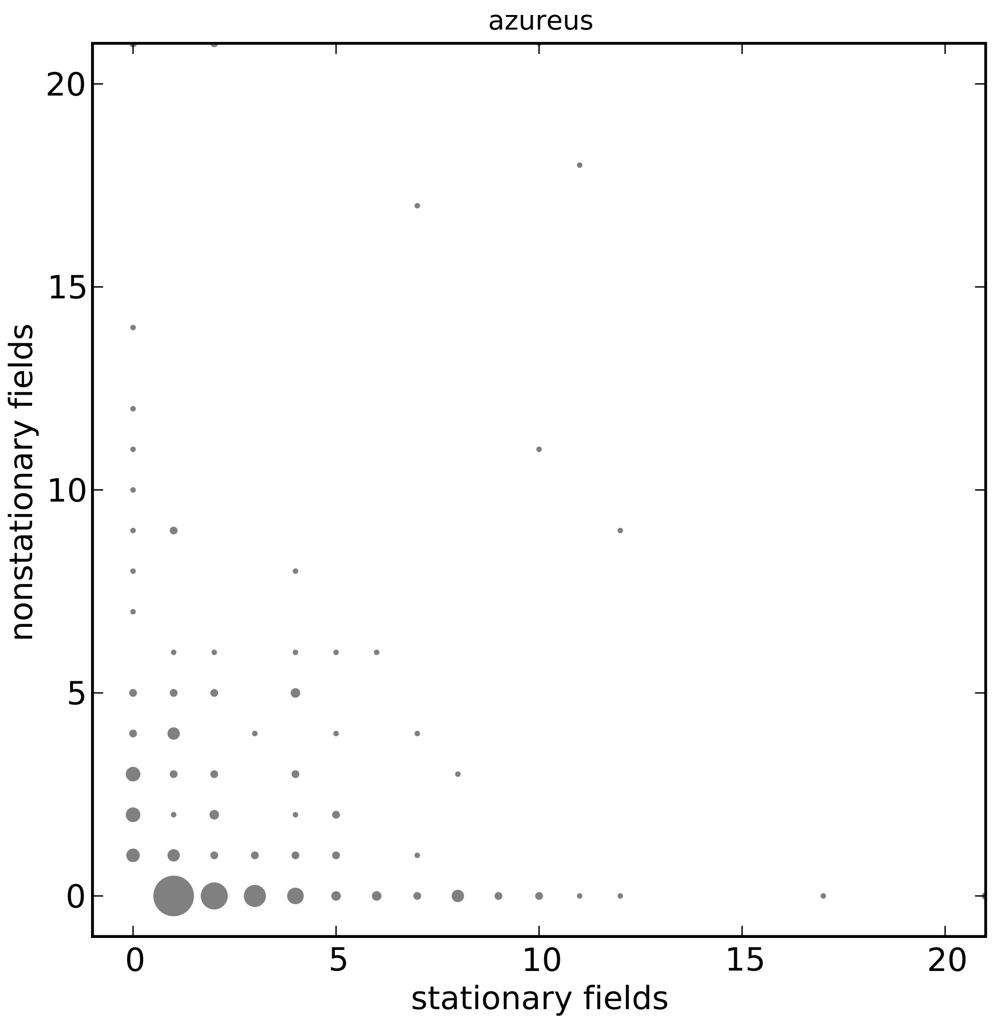

Figure 3.14: Distribution of stationary fields within classes in the Azureus benchmark (application classes only)

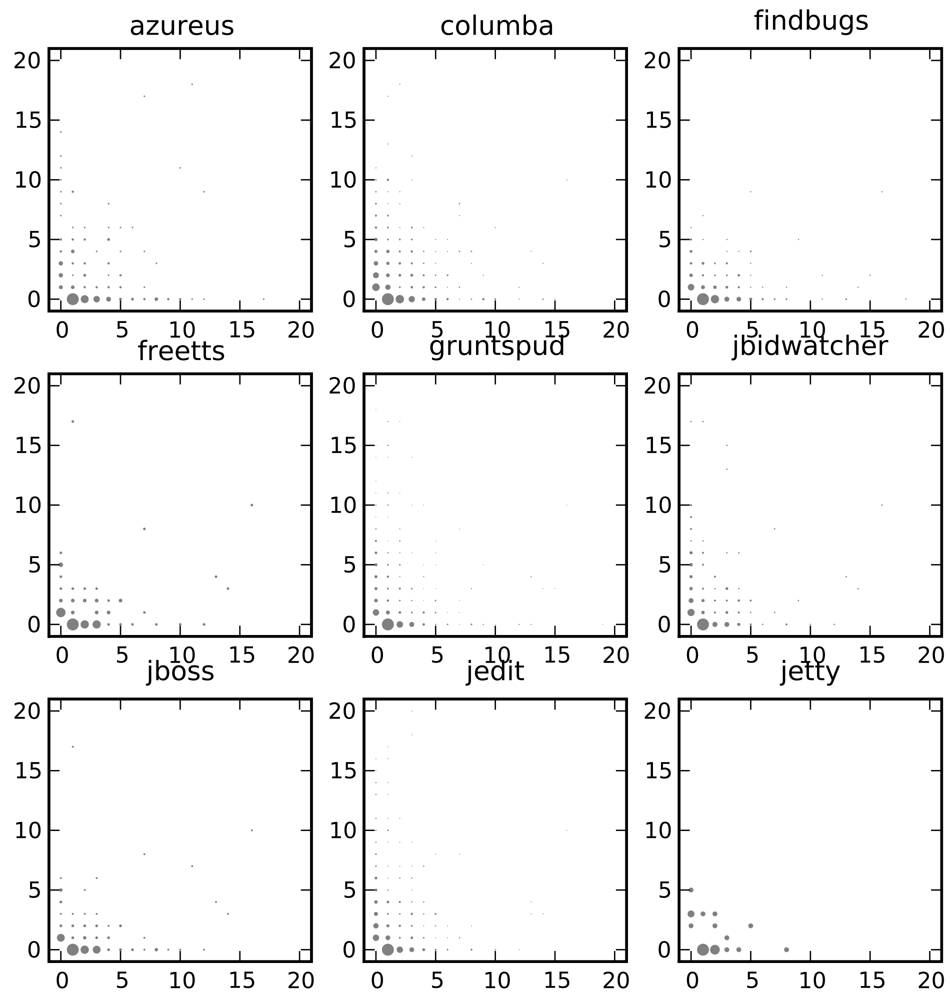

Figure 3.15: Distribution of stationary fields within classes (application classes only), benchmarks azureus–jetty

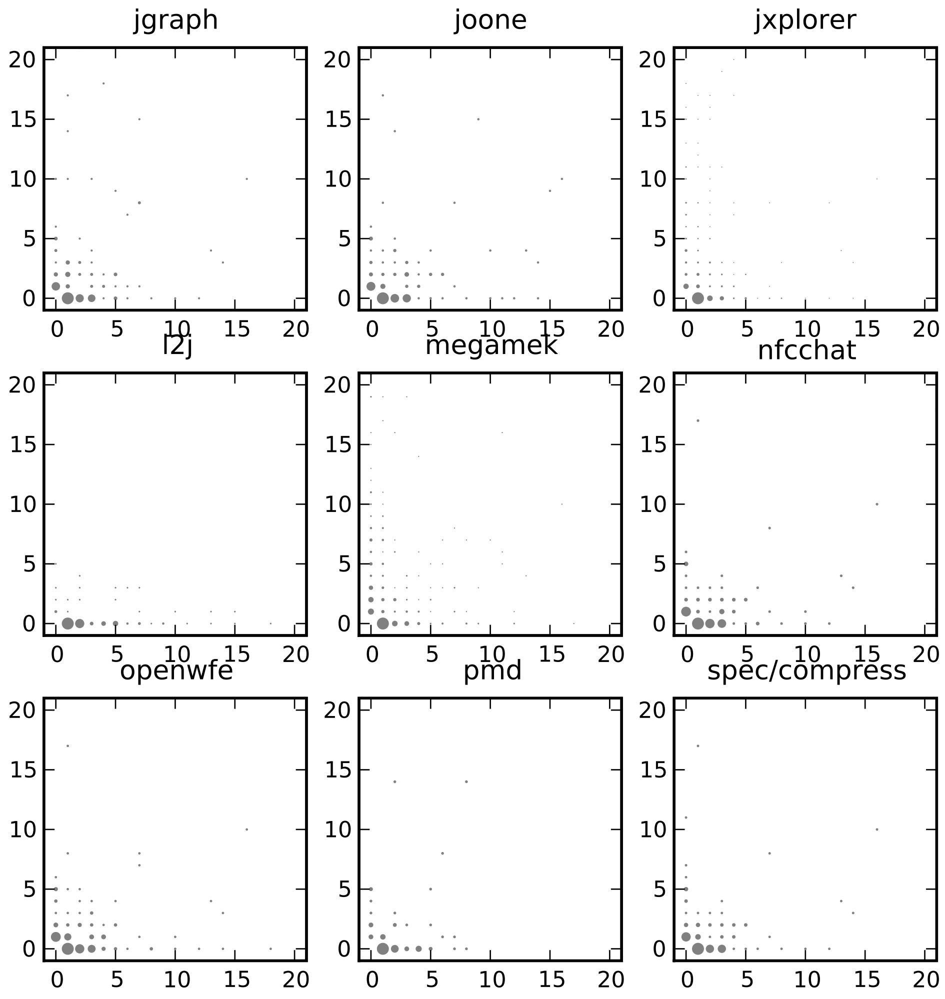

Figure 3.16: Distribution of stationary fields within classes (application classes only), benchmarks jgraph–spec/compress

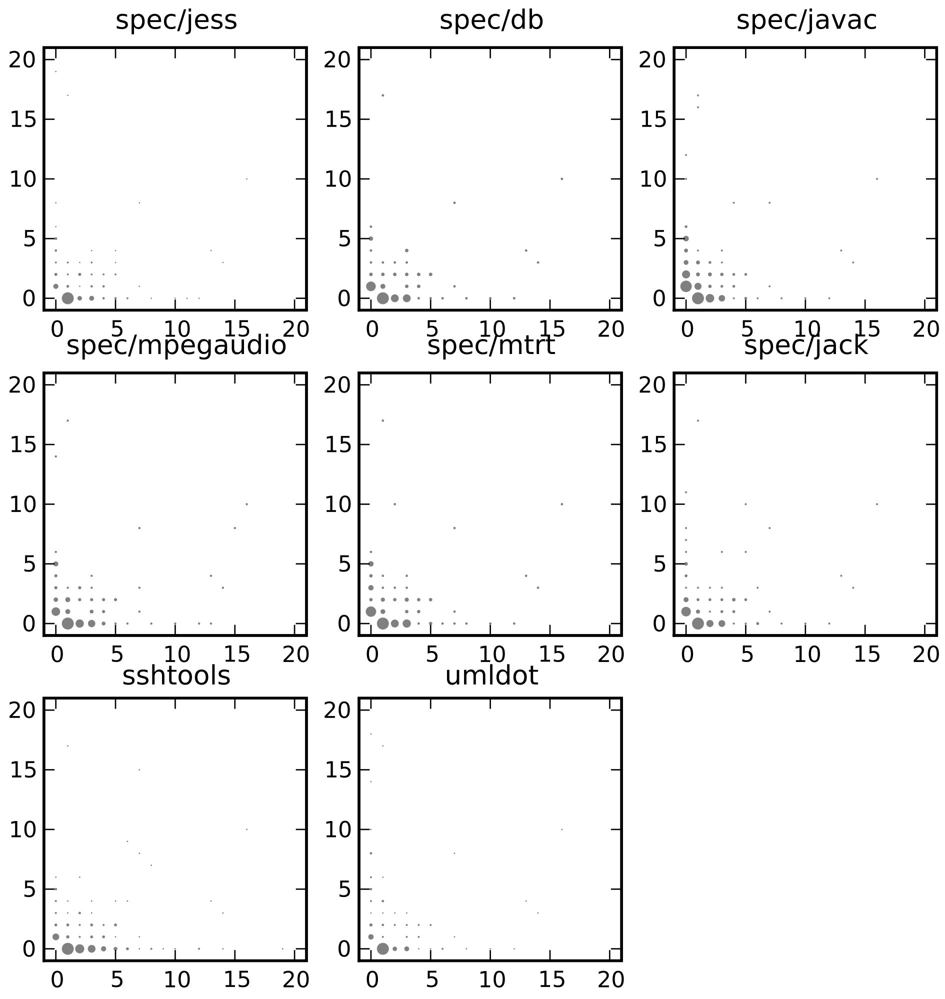

Figure 3.17: Distribution of stationary fields within classes (application classes only), benchmarks spec/jess–umldot

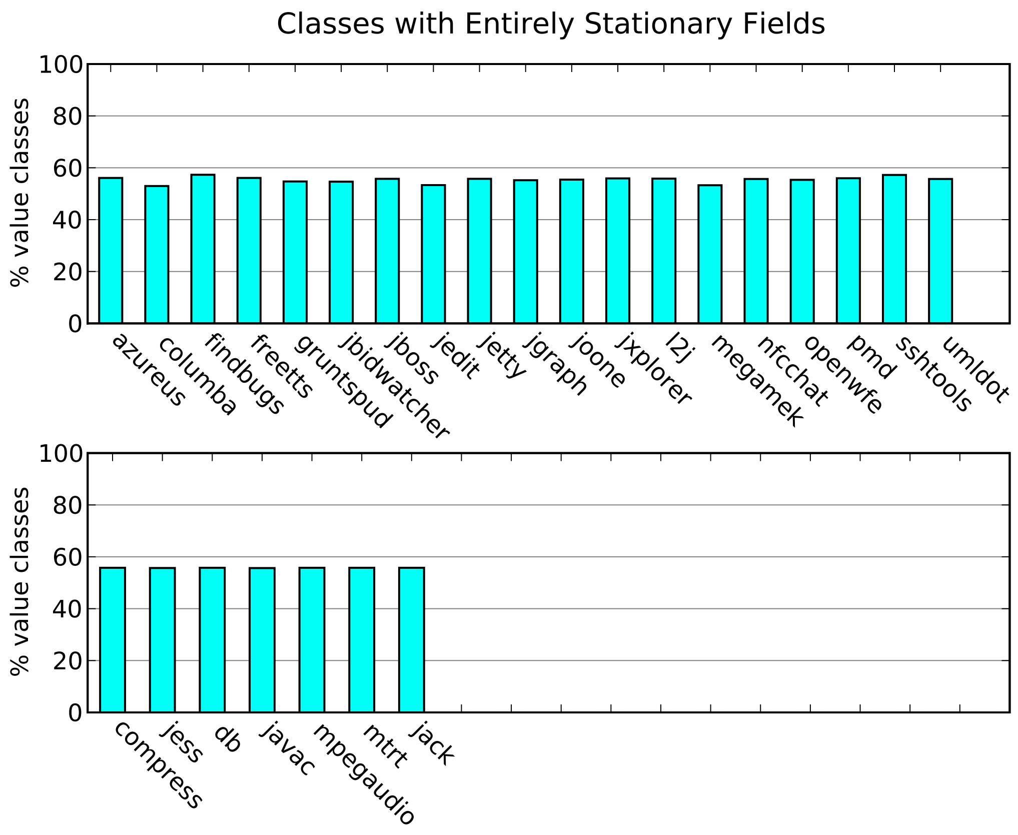

Figure 3.18: Percentage of classes composed entirely of stationary fields

chart are classes with entirely stationary fields, and along the left edge classes with entirely nonstationary fields.

The charts show that stationary fields are correlated within classes. If classes were created with a random mixture of stationary and nonstationary fields, lines of constant size would form a binomial distribution. Instead, there is more weight towards the edges of the graph.

Classes with entirely nonstationary fields may represent analysis imprecision. If the object that contains them is lost very early in initialization, before any fields are initialized, the analysis will be unable to find any stationary fields.

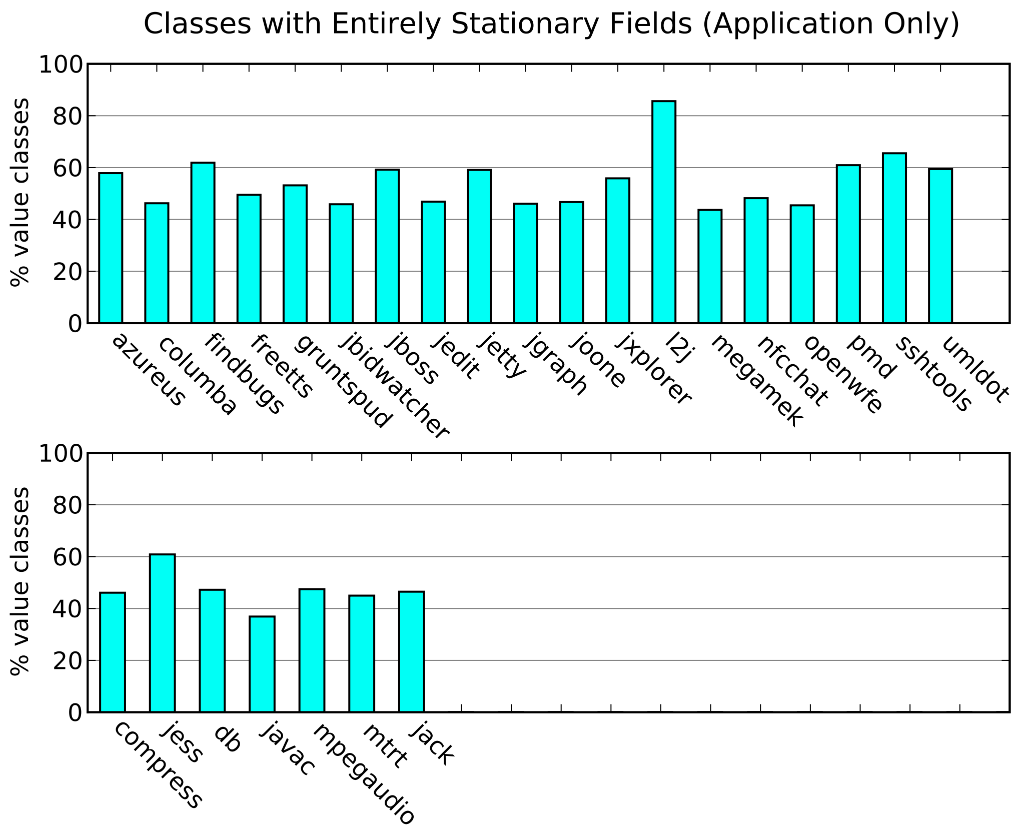

Figure 3.19: Percentage of classes composed only of stationary fields (application only)

### 3.6.1 Value Classes

Classes that contain entirely stationary fields are of special interest. They are sometimes known as *value classes*, and have some valuable properties. Because they have no state that changes, they may be shared freely, without worrying that other references may interfere by modifying the object. For example, Java provides immutable boxed numeric types (e.g. `java.lang.Integer`) so that numeric types may be stored in generic data structures. These types consist of a single immutable field of corresponding primitive type. Because all data contained in these classes is immutable, code need not worry about sharing instances of them.

Such classes also may be shared freely between threads without locking (provided that their initialization is occurs before they are shared, as is the case for fields our analysis determines to be stationary.) Goetz [14] defines an annotation `@Immutable` for a similar property: those classes whose instance fields are all `final`.

Figures 3.18 and 3.19 shows the fraction of classes in our benchmarks that contain only stationary fields. All of our benchmark programs contain many classes composed only of stationary fields: over 40% even when considering only the application's code excluding the Java libraries, with the single exception of javac. This percentage is in fact comparable to the overall prevalence of stationary fields. If stationary and nonstationary fields were mixed randomly together into classes, we would expect the percentage of classes with entirely stationary fields to be less than the the percentage of stationary fields. One reason there are so many such classes is that there are many classes that contain a single field. Another is the presence of classes with many fields, all of which are found to be nonstationary (perhaps due to analysis imprecision.) For example, the azureus benchmark contains a class with 302 fields, all of which our analysis believes are nonstationary. (The class represents the main window of the application; the fields are the many dialog boxes, menus, menu items, buttons, subpanels, etc., that the main window contains or invokes.) With so many nonstationary fields consumed by a small number of classes, a larger fraction of the remaining fields are stationary. Of course, another possible reason that there are more classes with no nonstationary fields than predicted by chance is that programmers are intentionally creating such classes due to the benefits such as those given above.

# Chapter 4

# Extensions to the Definition of Stationary Fields

The study of programs presented in the previous chapter showed that stationary fields account for approximately half of the fields in real Java programs. The more stationary fields we find, the more valuable stationary fields can be to program analysis and understanding. However, there may be fields that behave similarly to stationary fields, and have similar properties, but do not meet the definition of stationary fields. Identifying these fields can yield similar benefits to finding more stationary fields.

In this chapter, we identify two examples of programming idioms that may cause fields that seem to be constant to be nonstationary. We also give algorithms for locating these fields, and examining how common these idioms are in our benchmark programs.

## 4.1 Lazy Initialization

One reason fields may be nonstationary according to our definition, even if they seem to be unchanging, is if the programmer has attempted to defer their initialization until well after the object's creation. Such fields may be initialized lazily, with their initialization not attempted until the program needs to access the field. This is often an attempt to optimize the program. By deferring an expensive initialization, an

```
class Foo {
  private Object f;

  public Object getf(void) {
    if (this.f == null) {
      this.f = new X();
    }
    return this.f;
  }
}
```

Figure 4.1: Code showing lazy initializer.

object may be used more quickly following its creation. Furthermore, if the program never attempts to access the field, the cost of the initialization is saved entirely.

Figure 4.1 shows code exemplifying a common pattern for lazy initialization. Upon object creation, `f` retains the default null value. Access to field `f` is provided by an accessor function `getx`. If `f` has not be previously initialized, the function does so, setting it to a value guaranteed to be nonnull (in this case a newly created object). It then returns the value of `f`.

Field `f` will be nonstationary in any program that calls `getf`. Method `getf` must read `f` before it writes `f`, in order to test whether `f` has been previously initialized. The write initializing `f` occurs after this read, rendering `f` nonstationary. Although it would be possible to use a separate boolean flag indicating when the field has been initialized, this introduces a space overhead, so it is common to use the null value (or occasionally other sentinel value) in the field instead, as our example shows.

However, when invoked on the same object, `getf` will always return the same value. We therefore say that `getf` is a *stationary method*, by reference to the property of a stationary field that every time it is read on a particular object, the same value results. Other examples of stationary methods would include simple accessors of stationary fields, for example a method whose entire body is "`return this.f;`" where `f` is a stationary field.

A field that is properly lazily initialized appears to have very similar properties to a stationary field. Although the field is nonstationary, and the initial null value is read from the field before its terminal value is assigned, the propagation of the null value is limited. It is used only to guard the initial assignment. The nonstationary property of the field is apparent only to a very small section of code.

Given this, we may wish to search for fields that are lazily initialized, as they appear to most of the program to be constant, and so locating them will yield the same benefits as finding stationary fields.

### 4.1.1 Algorithm

We propose an algorithm for finding lazily initialized fields. The design of our algorithm is based on the idea that lazy initializers are frequently quite stylized, closely resembling the example in our figure. Specifically, we look for fields where:

- all access to the field is confined to a single method in the program;
- the field is reference-typed;
- the null value is used as the sentinel indicating that the field is uninitialized; and
- the single method that may read or write the field serves as an accessor, always returning the terminal value of the field.

The first step in our analysis is to locate all reference-typed fields in a program that are read or written only in a single method. For each such field $l$ and method $m$, we perform an analysis to determine whether the corresponding method $m$ is a lazy initializer of $l$. It must meet the following criteria:

- $m$ assigns to $l$ only through the `this` pointer.
- $m$ assigns to `this`.$l$ iff `this`.$l$ was null at entry to $m$.
- The return value of $m$ is always the same as the value of `this`.$l$ at the end of the $m$.

To determine this, we perform a flow-sensitive, intra-procedural analysis. The analysis accepts a candidate lazy field $l$ and method $m$. We present the analysis over the reduced language given in Section 2.3.1, with the following additions:

- There is a distinguished local variable `this`.
- Branch conditions of the form `if` $x =$ `null` `then` $s_1$ `else` $s_2$ are used.

At each point in the method, the analysis determines:

- The initial null state $N_l$, a set of boolean values indicating the possible nullness of $l$ at method entry. If $\mathbf{T} \in N_l$, then $l$ may have been null at method entry; if $\mathbf{F} \in N_l$, then $l$ may have been nonnull at method entry.
- Boolean $S_l$ indicating that $l$ has been properly initialized: that it has been assigned in this method if $l$ was null upon entry to the method. $S_l$ must be true at the end of the method for it to be a lazy initializer of $l$. (For the case where $l$ was nonnull upon entry, it is not allowed to be overwritten; this is checked at each assignment.)
- Initial variables $I_l$: a set of local variables that are known to hold the value of that $l$ held at the entry to the current method.
- Current variables $C_l$: a set of local variables that are known to hold the value of $l$ at the program point in question.

The results of the analysis are described by the inference rules given in Figure 4.2. The inference relate the values of $N_l$, $S_l$, $I_l$, and $C_l$ before a statement to those after it. For example, rule LazyAssign, which reads

$$\text{(LazyAssign)}\quad \frac{y \in I_l \rightarrow x \in I'_l \qquad I'_l \supseteq I_l - \{x\} \qquad y \in C_l \rightarrow x \in C'_l \qquad C'_l \supseteq C_l - \{x\}}{l, N_l, S_l, I_l, C_l \vdash x = y \Rightarrow N_l, S_l, I'_l, C'_l}$$

states that after an assignment $x = y$, all variables held the initial value of `this`.$l$ prior to the statement still do, except for $x$, which holds the initial value after the assignment if $y$ held it prior to the assignment; and likewise for variables that hold the current value of `this`.$l$. The analysis proceeds from the definition of the candidate

(LazyAssign)
$$\frac{y \in I_l \rightarrow x \in I'_l \qquad I'_l \supseteq I_l - \{x\} \qquad y \in C_l \rightarrow x \in C'_l \qquad C'_l \supseteq C_l - \{x\}}{l, N_l, S_l, I_l, C_l \vdash x = y \Rightarrow N_l, S_l, I'_l, C'_l}$$

(LazyLoad)
$$\frac{\begin{array}{c}((y = \texttt{this}) \wedge f = l \wedge \neg S_l) \rightarrow x \in I'_l \\ I'_l \supseteq I_l - \{x\} \qquad (y = \texttt{this}) \wedge f = l \rightarrow x \in C'_l \qquad C'_l \supseteq C_l - \{x\}\end{array}}{l, N_l, S_l, I_l, C_l \vdash x = y.f \Rightarrow N_l, S_l, I'_l, C'_l}$$

(LazyStoreThis)
$$\frac{S'_l = \mathbf{T} \qquad \mathbf{F} \notin N_l \qquad C'_l = y}{l, N_l, S_l, I_l, C_l \vdash \texttt{this}.l = y \Rightarrow N_l, S'_l, I_l, C'_l}$$

(LazyStore)
$$\frac{(x = \texttt{this} \vee f \neq l)}{l, N_l, S_l, I_l, C_l \vdash \texttt{x}.f = y \Rightarrow N_l, S_l, I_l, C_l}$$

(LazySeq)
$$\frac{l, N_l, S_l, I_l, C_l \vdash s_1 \Rightarrow N'_l, S'_l, I'_l, C'_l \qquad l, N'_l, S'_l, I'_l, C'_l \vdash s_2 \Rightarrow N''_l, S''_l, I''_l, C''_l}{l, N_l, S_l, I_l, C_l \vdash s_1; s_2 \Rightarrow N''_l, S''_l, I''_I, C''_l}$$

(LazyIf)
$$\frac{\begin{array}{c}N'_l = N'_{l1} \cup N'_{l2} \qquad S'_l = S'_{l1} \wedge S'_{l2} \qquad I'_l = I'_{l1} \cap I'_{l2} \qquad C'_l = C'_{l1} \cap C'_{l2} \\ l, N_l, S_l, I_l, C_l \vdash s_1 \Rightarrow N'_{l1}, S'_{l1}, I'_{l1}, C'_{l1} \qquad l, N_l, S_l, I_l, C_l \vdash s_2 \Rightarrow N'_{l2}, S'_{l2}, I'_{l2}, C'_{l2}\end{array}}{l, N_l, S_l, I_l, C_l \vdash \texttt{if} \ \sim \ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Rightarrow N'_l, S'_l, I'_l, C'_l}$$

(LazyIfNull)
$$\frac{\begin{array}{c}N_{l1} = b(N_l, \mathbf{T}, x \in I_l) \qquad N_{l2} = b(N_l, \mathbf{F}, x \in I_l) \qquad S_{l2} = S_l \vee x \in I_l \\ N'_l = N'_{l1} \cup N'_{l2} \qquad S'_l = S'_{l1} \wedge S'_{l2} \qquad I'_l = I'_{l1} \cap I'_{l2} \qquad C'_l = C'_{l1} \cap C'_{l2} \\ l, N_{l1}, S_l, I_l, C_l \vdash s_1 \Rightarrow N'_{l1}, S'_{l1}, I'_{l1}, C'_{l1} \qquad l, N_{l2}, S_{l2}, I_l, C_l \vdash s_2 \Rightarrow N'_{l2}, S'_{l2}, I'_{l2}, C'_{l2}\end{array}}{l, N_l, S_l, I_l, C_l \vdash \texttt{if}\ x = \texttt{null}\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Rightarrow N'_l, S'_l, I'_l, C'_l}$$

$$b(N_l, t, p) \equiv \begin{cases} N_l & \text{if } p \text{ is false;} \\ N_l \cap \{t\} & \text{if } p \text{ is true.} \end{cases}$$

Figure 4.2: Inference rules for finding lazy initializers.

(LazyWhile)
$$\frac{N'_l \supseteq N_l \qquad S'_l \rightarrow S_l \qquad I'_l \subseteq I_l \qquad C'_l \subseteq C_l \qquad N'_l, S'_l, I'_l, C'_l \vdash s \Rightarrow N'_l, S'_l, I'_l, C'_l}{l, N_l, S_l, I_l, C_l \vdash \texttt{while} \sim \texttt{do}\ s \Rightarrow N'_l, S'_l, I'_l, C'_l}$$

(LazyNew)
$$\frac{I'_l = I_l - \{x\} \qquad C'_l = C_l - \{x\}}{l, N_l, S_l, I_l, C_l \vdash x = \texttt{new}() \Rightarrow N_l, S_l, I'_l, C'_l}$$

(LazyInvoke)
$$\frac{I'_l = I_l - \{x\} \qquad C'_l = C_l - \{x\}}{l, N_l, S_l, I_l, C_l \vdash x = m(y_1, y_2, \dots) \Rightarrow N_l, S_l, I'_l, C'_l}$$

(LazyMethod)
$$\frac{\begin{array}{ccccc} & & & y \in C'_l & S'_l \\ N_l = \{\mathbf{T}, \mathbf{F}\} & S_l = \mathbf{F} & I_l = \emptyset & C_l = \emptyset & l, N_l, S_l, I_l, C_l \vdash s; \Rightarrow N'_l, S'_l, I'_l, C'_l \end{array}}{\vdash m(\texttt{this}, x_1, \dots, x_i)\{s; \texttt{return}\ y\} \text{ initializes } l}$$

Figure 4.3: Inference rules for finding lazy initializers.

lazy initializer method: the final judgment is $\vdash m(\texttt{this}, x_1, \dots)\{\dots\}$ initializes $l$. If, from the rules, we can infer this judgment, $m$ is a lazy initializer of $l$.

The remaining rules provide the following:

- After a load $x = y.f$, $x$ holds the initial value of $l$ if $y$ is `this`, $f$ is $l$, and $l$ has not yet been set. Otherwise it does not hold the initial value of $l$. Iff $y$ is `this` and $f$ is $l$, $x$ holds the current value of $l$. (LazyLoad)

- After a store $\texttt{this}.l = y$, $l$ has been initialized (S is true) if it is known to have been initially null; if it is not known to be initially null, $m$ cannot be a lazy initializer of $l$. Source variable $y$ now holds the current value of $l$. (LazyStoreThis)

- If $m$ contains a store to $l$ other than through `this`, it is not considered a lazy initializer of $l$. We require initialization to occur through the `this` pointer. Any writes through other variables could overwrite a previously initialized $l$. (LazyStore)

- In a sequence $s_1; s_2$, the conditions prior to $s_1$ are those at entry to the sequence;

those after $s_1$ are those before $s_2$; and the conditions at the conclusion of the sequence are those after $s_2$. (LazySeq)

- In an if statement, the conditions prior to each branch are the same as on entry to the if statement. After the if statement, the nullness of $l$ is that at the conclusion of either branches; variables hold the current or initial value of $l$ if they held it at the end of both branches; and $l$ has been initialized if it was on both branches. (LazyIf)

- If the branch condition of an if statement compares a variable that holds the initial value of $l$ to null, then $l$ is known to be null on the then taken branch and nonnull on the else branch. $S$ is properly initialized on the else branch ($S$ holds the condition "if $l$ was null at entry, it has been set"; since $l$ was not null at entry, the condition is trivially true.) (LazyIfNull)

- The nullness of $l$, current and initial variables, and whether $l$ has been initialized at the exit of a loop must be a fixed point under the body of the loop. (LazyWhile)

- Object creations and method invocations have no effect except to kill any assigned variable from those holding the current or initial values of $l$. (LazyNew, LazyInvoke)

- At the start of a candidate lazy initializer, $l$ may be either null or nonnull. $l$ has not been initialized. No variables hold its current or initial value. If $l$ has not been initialized at the conclusion of $m$, $m$ is not a lazy initializer of $l$. If $m$ does not return the final value of $l$, it is not a lazy initializer of $l$. (LazyMethod)

### 4.1.2 Program Study

We applied this algorithm for finding lazy initializers to our benchmark programs. The results are shown in Figure 4.4. We find twenty-three fields with lazy initializers in most of our benchmarks; these fields are entirely within the Java standard libraries. The lazy initializer pattern is not a dominant one in the benchmarks. However, as it

| **program** | **fields** | **% fields** |
|---|---|---|
| azureus | 8 | 0.4 |
| columba | 26 | 0.4 |
| findbugs | 9 | 0.4 |
| freetts | 23 | 0.5 |
| gruntspud | 28 | 0.4 |
| jbidwatcher | 23 | 0.5 |
| jboss | 22 | 0.5 |
| jedit | 25 | 0.4 |
| jetty | 7 | 0.4 |
| jgraph | 23 | 0.5 |
| joone | 26 | 0.5 |
| jxplorer | 22 | 0.4 |
| l2j | 7 | 0.4 |
| megamek | 28 | 0.5 |
| nfcchat | 23 | 0.5 |
| openwfe | 23 | 0.5 |
| pmd | 7 | 0.4 |
| spec/compress | 23 | 0.5 |
| spec/jess | 23 | 0.5 |
| spec/db | 23 | 0.5 |
| spec/javac | 23 | 0.5 |
| spec/mpegaudio | 23 | 0.5 |
| spec/mtrt | 23 | 0.5 |
| spec/jack | 24 | 0.5 |
| sshtools | 22 | 0.4 |
| umldot | 23 | 0.5 |

Figure 4.4: Fields with lazy initializers.
Shown as number of fields found, and as a percentage of all reference-typed fields in each benchmark.

does occur in the standard libraries, it covers about half a percent of the fields in all the benchmarks. Recognizing it would allow a tool to treat these fields similarly to stationary fields.

## 4.2 Object Disposal

### 4.2.1 Motivation

So far, we have focused primarily on objects' initialization and the beginning of their lifetime. In the same way that the early life of an object may be special in that writes are allowed to fields that are otherwise constant to initialize them, some otherwise constant fields may be written near the death of their containing object.

One pattern that occurs is for fields to be set to null just before the object is dead. The programmer may think that this will aid the garbage collector in reclaiming objects.

Otherwise constant fields with such writes are not stationary. We can look for cases where writes at object disposal time are preventing fields from being stationary by looking for fields that are stationary except for one or more writes with a null value. Such fields have the property that their observed value may contain a single non-null to null transition, essentially parallel to final fields which may observe a single null to non-null transition.

Although these fields do not have the full properties of stationary fields, they may still be useful for program analysis. Although one does not necessarily have the same value every time they are read, reading it cannot yield two distinct objects. Unless the program specifically checks for null values in the field, the only practical difference between these fields and stationary fields is that a program that uses the value after it is set to null may generate a null pointer exception. Any property that is based on the field not holding different objects should still hold.

### 4.2.2 Algorithm

To locate these fields, we make a simple change to our stationary fields inference algorithm. We change it to ignore all fields writes where the right hand side is the null constant. (We look only for the null constant itself and perform no analysis of whether variables must be null.) The change is confined to the STORE rule, which is modified to treat such writes as if they did not exist. The input language is extended to include store statements with a null right-hand side ($x.f = \mathtt{n}ull$); and such statements have no effect, as specified by the following additional inference rule:

$$\text{(STORENULL)}\quad \frac{}{m, P, U, Q \vdash x.f = \mathtt{null} \Rightarrow P, U, Q}$$

### 4.2.3 Program Study

We applied this modified version of our stationary fields inference algorithm to the same benchmarks we used for in Chapter 3. The results are shown in Figure 4.5, which shows the additional number of fields that are accepted by the modified algorithm in each benchmark. In each benchmark, about 1-2% of reference-typed fields are nonstationary only because of one or more writes with the value of null. The small number of fields found relative to the number of stationary fields indicates that programmers are not making many otherwise stationary fields nonstationary for the reasons given above. Nonetheless, because a weakened version of the stationary property holds for these fields, this additional small fraction of fields may be useful for program analysis.

### 4.2.4 Resource Management

The most common use of writing null values was for resource management: an object would nullify its reference to another object so that the referred object would be freed immediately by the garbage collector even in the referring object could not be. The majority of these examples occurred in a single library, the Eclipse SWT, which is a windowing library used by several of the GUI applications.

| **program** | **fields** | **% fields** |
|---|---|---|
| azureus | 52 | 2.4 |
| columba | 76 | 1.3 |
| findbugs | 41 | 1.9 |
| freetts | 69 | 1.4 |
| gruntspud | 80 | 1.2 |
| jbidwatcher | 73 | 1.4 |
| jboss | 72 | 1.5 |
| jedit | 73 | 1.2 |
| jetty | 34 | 2.1 |
| jgraph | 72 | 1.5 |
| joone | 68 | 1.4 |
| jxplorer | 69 | 1.2 |
| l2j | 35 | 2.0 |
| megamek | 66 | 1.1 |
| nfcchat | 69 | 1.4 |
| openwfe | 68 | 1.4 |
| pmd | 35 | 2.0 |
| spec/compress | 68 | 1.4 |
| spec/jess | 68 | 1.4 |
| spec/db | 68 | 1.4 |
| spec/javac | 72 | 1.5 |
| spec/mpegaudio | 69 | 1.4 |
| spec/mtrt | 68 | 1.4 |
| spec/jack | 68 | 1.4 |
| sshtools | 77 | 1.5 |
| umldot | 68 | 1.4 |

Figure 4.5: Fields that are stationary except for being set to null.
Shown as number of fields found, and as a percentage of all reference-typed fields in each benchmark.

### 4.2.5 Other Uses of Null

This analysis also identified some nearly constant fields that did not fit within the definition of a stationary field. Several fields were set repeatedly to null, and this was the only value they were ever assigned. Because some of the writes occurred after the objects were heap-referenced, our analysis could not identify them as stationary. Indeed, because the fields may have been read before they were set again to null, they do not meet the definition of stationary. These fields point to another possible expansion of the definition of stationary: fields for which all of the *non-silent* writes precede all of the reads.

# Chapter 5

# Applications of Stationary Fields

In this chapter we show four applications of stationary fields, as examples of how stationary fields may be useful for program analysis and understanding. The first three example applications are to concurrent programs. Concurrent programs are currently very common, and are expected to become more so given the recent shift in commodity hardware from uniprocessors to multicore processors. These first three applications take advantage of the property that threads cannot interfere with each other by modifying stationary fields, as stationary fields do not change.

## 5.1 Lock Elision

Our first application shows how stationary fields may be applied to locks, which are a fundamental tool used by concurrent programs. We use stationary fields to show that some locks in programs are unnecessary. These results can allow programmers to understand one instance where their applications are needlessly using locks, and allow them to optimize their applications by removing extraneous synchronization.

Multithreaded programs require a way to prevent threads from interfering with each other as they operate on shared data structures. One concurrency error that can occur is a race: a condition in which two threads may access the same memory location, no ordering between the two accesses is enforced, and at least one of the accesses involved is a write.

Java provides locks as the primary synchronization primitive: only one thread can hold a lock at any time, and a thread that attempts to acquire a lock held by another thread will block. Programmers prevent races by ensuring that whenever two threads may attempt to access the memory location simultaneously, the use of some common lock will delay one of the threads and prevent a race.

### 5.1.1 Stationary Fields and Races

A field found to be stationary by our algorithm cannot be involved in a race. In general, a stationary field can only be involved in a race during its initialization: after initialization is complete, there are by definition no more writes. Because at least one of the accesses in a race must be a write, a race cannot occur at this point.

This leaves the possibility of a race during initialization. However, a field found to be stationary by our inference algorithm cannot have a race during initialization either. Our algorithm's simplifying assumption that initialization occurs before an object is referenced by another object implies that initialization is performed by a single thread: an object can only become visible to a thread other than its creator by being referenced by some object on the heap. That is, tracking whether objects are lost also serves as a rudimentary escape analysis. Local variables are not shared between threads in Java. Because a race involved accesses by two threads and the object is only visible to one, a race cannot occur.

### 5.1.2 Unnecessary Synchronizations

The use of a lock is indicated in a Java program through the use of the `synchronized` keyword. Usually it is applied to a method, indicating that the invoking thread must hold the lock associated with the `this` object during the execution of the method; such methods are *synchronized methods*. The keyword may also be applied to a block of code, with a specific lock designated.

The primary use of locks is to prevent data races, and so we would expect that every synchronized method contains some computation that could involve a race.

However, some synchronizations may not be needed: if a synchronized method accesses only stationary fields, then the synchronization does not protect against any races.

### 5.1.3 Algorithm

An algorithm for finding synchronizations that prevent no races is straightforward, given a program, a list of stationary fields provided by our inference algorithm, and the call graph used by it. Given a synchronized method $m$, we:

- Calculate all transitive callees of $m$, by traversing the call graph to find every method is reachable from $m$.
- Obtain the set of all fields referenced by the callees by a simple scan of each method.
- If all the fields are stationary, the synchronization for $m$ prevents no races.

### 5.1.4 Experimental Results

We applied this algorithm to our benchmark programs. Figure 5.1 shows the fraction of synchronized methods that use only stationary fields. Surprisingly, it is approximately 7% across nearly all the benchmarks, rather than just the handful of methods that one might expect. In part this may be defensive. The primary drawbacks to extraneous synchronization are reduced performance and the possibility of deadlocks. Programmers are generally not as worried about these as they are about creating races, which can be the implication of a missing lock. Many of the synchronized methods are in the Java libraries and may be needed in circumstances outside these programs to guard against races in other code. Looking at only the application code, most programs contain very few synchronized methods, and few of those use only stationary fields. For example, although 18% of the synchronized methods in the application code of joone use only stationary fields, that is only 8 methods.

| | **all methods** | | **app. only** | |
|---|---|---|---|---|
| **program** | **methods** | **% methods** | **methods** | **% methods** |
| azureus | 20 | 8 | | |
| columba | 57 | 8 | 2 | 4 |
| findbugs | 11 | 5 | 0 | 0 |
| freetts | 48 | 7 | 0 | 0 |
| gruntspud | 54 | 8 | 2 | 5 |
| jbidwatcher | 51 | 8 | 2 | 11 |
| jboss | 49 | 7 | 0 | 0 |
| jedit | 49 | 7 | 0 | 0 |
| jetty | 19 | 9 | | |
| jgraph | 51 | 8 | 0 | 0 |
| joone | 56 | 8 | 8 | 18 |
| jxplorer | 54 | 8 | 1 | 5 |
| l2j | 20 | 9 | 0 | 0 |
| megamek | 51 | 6 | 0 | 0 |
| nfcchat | 48 | 7 | 0 | 0 |
| openwfe | 48 | 7 | 0 | 0 |
| pmd | 18 | 8 | | |
| spec/compress | 48 | 7 | 0 | 0 |
| spec/jess | 50 | 7 | 0 | 0 |
| spec/db | 48 | 7 | 0 | 0 |
| spec/javac | 48 | 7 | 0 | 0 |
| spec/mpegaudio | 48 | 7 | 0 | 0 |
| spec/mtrt | 48 | 7 | 0 | 0 |
| spec/jack | 48 | 7 | 0 | 0 |
| sshtools | 56 | 8 | 7 | 13 |
| umldot | 48 | 7 | 0 | 0 |

Figure 5.1: Synchronized methods that use only stationary fields.
Shown as the number and percentage of synchronized methods, considering all synchronized methods in each benchmark's call graph, and for the application's methods only. The applications corresponding to the blank entries contain no synchronized methods outside the Java libraries.

## 5.2 Optimizing a Software Transactional Memory

In the previous section we examined applying stationary fields to locks, which are the currently dominant method of concurrency control in Java programs. Current research is exploring other method: software transactional memories (STMs). A STM provides a way to declare that the effects of a section of code should not be interleaved with the effects of other threads. The STM enforces atomicity by adding barriers to memory access, so that it may prevent transactions from interfering with each other.

A transactional memory may provide either strong or weak isolation semantics. In a weakly isolated STM, atomic blocks are only guaranteed to appear atomic to other atomic blocks; non-transactional code may observe intermediate results of an atomic block. Conversely, in a strongly isolated STM, atomic blocks appear atomic to all other code. Strong isolation provides a more intuitive guarantee to the programmer. However, it it more expensive to provide the stronger semantics. For example, the STM must ensure that every read, whether inside an atomic block or not, does not observe an intermediate result of an atomic block; this may require a memory barrier on every read.

### 5.2.1 Experimental Results

We investigated the benefits of using stationary fields to optimize AJ, a strongly isolated Java STM. [6] We ran our stationary fields inference algorithm on the transactional program, and provided the results to AJ, which then eliminated the memory barriers on the stationary fields. We examined the performance of the benchmark program with and without this information. AJ includes a local stationary fields analysis, which finds stationary fields that can be detected while examining a single class at a time (i.e. only private fields.) We also examined disabling this local analysis, so that the total effects of using stationary fields could be discerned, as well as the incremental value of using our more powerful, whole-program, stationary fields analysis.

We performed these experiments on a single benchmark used in the evaluation of

| isolation | analysis | num. swaps | num. fields swapped | time per field (msec) | total swap swap time (s) |
|---|---|---|---|---|---|
| strong | none | $69 \pm 1.0$ | $522 \pm 2.9$ | $28.14 \pm 1.25$ | 14.69 |
| strong | local | $65 \pm 2.3$ | $447 \pm 1.9$ | $29.63 \pm 1.85$ | 13.24 |
| strong | static | $57 \pm 5.8$ | $435 \pm 1.3$ | $33.85 \pm 1.30$ | 14.72 |
| strong | local + static | $56 \pm 2.0$ | $377 \pm 1.2$ | $33.74 \pm 3.58$ | 12.73 |
| weak | none | $1 \pm 0.0$ | $9 \pm 0.0$ | $13.50 \pm 0.17$ | 0.12 |
| weak | static | $1 \pm 0.0$ | $9 \pm 0.0$ | $13.44 \pm 0.08$ | 0.12 |

Figure 5.2: Barrier swap performance of software transactional memory on SPECjbb benchmark using stationary fields analysis.

AJ: a version of the SpecJBB2005 benchmark modified to use atomic regions. The experiments were performed in the same environment as the evaluation of AJ. We ran each configuration five times and we report mean results with corresponding standard deviations.

We ran the STM with and without our stationary fields analysis (indicated as "static" in the results.) We also ran it with and without AJ's stationary fields analysis (indicated as "local.")

Figure 5.2 shows the barrier swap costs incurred by each configuration. It shows the number of times the hot swap was used, the number of fields for which barriers were swapped, the time in swaps per field, and the total time spent swapping barriers.

As expected, providing static information about stationary fields does reduce the number of swapped fields. The reduction in fields swapped indicates that stationary fields are used in the benchmark program. Because of the cost of inserting an optimized barrier (and the very large potential cost of undoing the placement of a barrier based on an incorrect hypothesis) the STM only inserts optimized barriers for frequently used fields. For example, our stationary fields analysis found 58 (435 − 377) fields over the AJ's local analysis that the STM considered important enough to optimize.

AJ includes many optimizations, some of which diminish the potential benefits of stationary fields. For example, because the STM batches the barrier swaps, the number of times the hot swap mechanism is invoked is not greatly reduced. The hot swap mechanism has a large fixed overhead each time it is invoked. Because of this, there is little improvement in total barrier swap time despite the reduced number of fields for which barriers must be inserted.

AJ also includes optimizations to improve the steady-state performance. One of the main features of AJ is that it uses optimized barriers based on the observed access patterns of fields. By observing the dynamic behavior of each field, it produces a hypothesis about the access pattern of the field. Based on this hypothesis it may choose to insert a cheaper barrier (at the expense of a very expensive operation if the hypothesis is incorrect.) These dynamic optimizations can capture the properties of stationary fields, and so they may blunt the benefits of providing static information about stationary fields.

Figures 5.3 and 5.4 show the throughput of the SpecJBB benchmark in each configuration. Throughput is given as microseconds per SpecJBB business operation (smaller numbers are better.) Figure 5.3 shows the throughput in each configuration as a function of the number of threads. Figure 5.4 shows the peak throughput of each configuration: the performance with the number of threads that produces the best throughput. The results show that using stationary fields to optimize the STM does produce an improvement in application throughput (with at least 95% confidence for strong/none vs. strong/local+static, and for weak/none vs. weak/static.) The results also show a small speedup by adding the static analysis to the local one for the strong isolation case; however, the result is not statistically significant at the 95% level given the small number of runs.

## 5.3 Optimizing Interthread Communication Monitoring

### 5.3.1 Motivation

Many Java programs, especially those that run in server environment and GUI applications, use multithreading extensively. Server-side programs often use threads in order to handle multiple clients. Client-side programs often use threads to remain

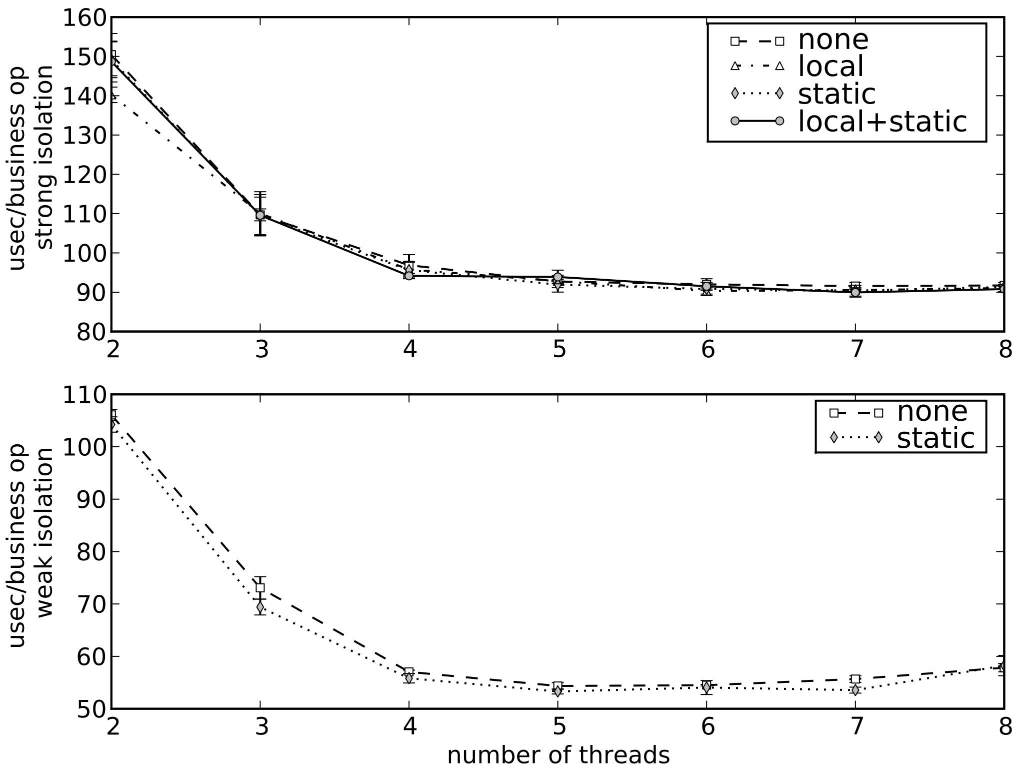

Figure 5.3: Performance of a software transactional memory on SPECjbb benchmark using stationary fields analysis.

| **isolation** | **analysis** | **$\mu$s/business op.** | **threads** |
|---|---|---|---|
| strong | none | $91.57 \pm 0.92$ | 7 |
| strong | local | $90.31 \pm 0.78$ | 6 |
| strong | static | $90.40 \pm 1.39$ | 7 |
| strong | local + static | $89.94 \pm 1.19$ | 7 |
| weak | none | $54.35 \pm 0.63$ | 5 |
| weak | static | $53.30 \pm 0.48$ | 5 |

Figure 5.4: Performance of software transactional memory on SPECjbb benchmark using stationary fields analysis (optimal number of threads).

responsive to the user even while long-running computation or slow network accesses are in progress. The Java standard libraries and standard Java programming idioms encourage, or in some cases even require, the use of multiple threads. For example, multithreading is the primary way to support concurrent I/O.

Support for multithreading is built into Java. Synchronization primitives are included in the language, facilities for starting threads are defined as part of the standard library, and even some concurrent data structures are provided.

Threads communicate through data structures that they share. The patterns of exchange between threads can therefore be arbitrarily complex, as they are limited only by the data structures that may be created.

In large programs, which may have many different types of threads running for many different purposes, the communication between threads may grow so complex that programmers do not understand them. They could use a tool that reports communication between threads to understand the structure of their programs.

### 5.3.2 Dynamically Monitoring Thread Communication with Tags

One way to obtain information about the interthread communication is to monitor the runtime behavior of a program. Flows2 [27] is one such tool. It uses bytecode instrumentation to modify an application such that direct communication between threads can be recorded.

The goal of flows2 is to capture the *thread communication graph* of the execution of a program. In the thread communication graph, each thread in the program is a node, and there is an edge from thread $t_1$ to $t_2$ iff $t_2$ reads a value from a variable and that value was written by $t_1$: that is, if there is direct information flow from $t_1$ to $t_2$. The graph may also be labeled to indicate some property of the flow; for example it may be labeled with a field $f$ to indicate that the communication occurred through that field.

Flows2 observes the communication by tagging each field with the identity of the last thread that wrote it. (Only fields need be instrumented because local variables cannot be shared between threads in Java.) Each write of any field is instrumented to also update the relevant tag with the identity. Each read of a field is instrumented to record a flow from the last thread to write that field, as recorded in the tag, to the reading thread. When the program terminates, the resulting graph is reported to the programmer. Similar tagging techniques could also be used to enforce restrictions on communication between threads, instead of simply recording them. Optimizations based on stationary fields could be applied in the same ways, which we shall now describe.

### 5.3.3 Reducing the Space Overhead with Stationary Fields

This tagging method imposes a substantial overhead on program execution. The instrumentation incurs a time penalty on every field access. Furthermore, the space overhead is also large, because every field of each object must have a corresponding tag.

We can apply stationary fields to thread monitoring to reduce this space overhead. Our analysis finds stationary fields for which initialization occurs before the containing object is referenced by any other object. This also implies that initialization of the stationary fields is carried out by the creating thread, because an object can only become accessible to another thread by having its reference placed in some shared objects. Therefore, while nonstationary fields each require a tag, all stationary fields in an object may share a single tag recording the creating thread. Furthermore, this tag is always required even if the object does not explicitly contain any stationary fields: all objects must contain some record of what class they are, for virtual method dispatch, reflection, etc. Typically there is a hidden field in each object's header that contains a pointer to the appropriate java.lang.Class object. Because the creating thread chose the type of the object, an operation such as virtual method dispatch that uses this information represents flow from the creating thread. The instrumentation therefore requires a tag for the creating thread. The instrumentation for stationary fields can piggyback on this required tag with no additional space penalty.

Stationary fields can also be used to reduce the time overhead of tagging techniques: if a thread reads from the same field of the same object repeatedly, it is not necessary to repeat recording the flow. Because the field cannot have changed, the tag for the field cannot have changed and a new flow cannot occur. This optimization is essentially a redundant load removal on the tag for the stationary field. This technique is even more effective if we do not need to record the field through which flow occurred. Because all stationary fields must have the same tag, once a thread has recorded flow from a stationary field of an object, it need not monitor accesses from *any* stationary field of that object.

There may also be reasons that we may wish to ignore information flow through stationary fields. For example, a programmer may be interested only in the communication flows that may cause a race. In that case, it is unnecessary to monitor stationary fields as they cannot be involved in races.

### 5.3.4 Experimental Setup

We created dynamic instrumentation to estimate the space overhead of a dynamic, tagging-based information flow monitor, such as flows2, with and without the optimization described above. We applied this instrumentation to the SPEC JVM98 benchmark programs. An alternative approach would have been to modify the flows2 framework to apply the optimization. The primary advantage of our approach is that the flows2 implementation is a proof of concept and does not instrument all code. Because our instrumentation is very simple, we can apply it to all executed Java code, including the standard and system libraries. We may therefore estimate the space overhead of a complete production system, rather than a demonstration system. Furthermore our simple dynamic instrumentation is more robust than the flows2 implementation. We use the SPEC benchmark programs rather than the applications used in the flows2 demonstration because the SPEC programs come with reproducible test suites.

Our dynamic instrumentation uses the JVMTI (Java Virtual Machine Tool Interface) to gain access to the Java classes during execution. The JVMTI is an interface for debugging and profiling that is provided by the JVM. One capability it provides is the option to be notified of each Java class that is about to be loaded, be provided the contents of the class file, and, if desired, to replace the contents of the class. Our framework intercepts each loaded class, and passes it to an instrumentation process. The instrumentation process runs outside the JVM, eliminating complex interactions between the instrumentation and the JVM bootstrapping procedure.

The instrumentation process uses the Apache Byte Code Engineering Library (BCEL) to add instrumentation to count the number of instances of each class that are created. At each *new* bytecode, it adds code to increment a counter corresponding to the class of the created object.

When the program exits, our instrumentation framework is notified through the JVMTI. It reads the counters from the instrumentation code (they are stored in a simple array stored in a static variable) and outputs them.

We then combine this information with the results of our stationary fields inference algorithm, and some basic information about the classes used by the program. Together this tells us what fields each class contains, the type of each field, and whether each field is stationary or not. From this we calculate the size of an object. We ignore field alignment and the overhead of the memory allocator and garbage collector. We assume an 8-byte object header, as used in recent versions of the Sun HotSpot JVM, and a 32-bit implementation, in which a reference is 4 bytes.

Let the base size $B$ of the object be the number of bytes of storage occupied by its fields: 4 bytes for each reference type, 2 bytes for a short, 8 bytes for a long, etc. Let $S$ be the number of stationary fields in the object and $N$ the number of nonstationary fields.

The uninstrumented size of the object is $8 + F$: header plus fields. The instrumented size without optimization is $12 + F + 4N + 4S$: head, object creator tag, fields, and tag for each field. The instrumented size with optimization is $12 + F + 4N$; stationary fields require no tag. By multiplying the size of each type of object by the number allocated, and summing the results, we obtain the number of bytes allocated by the program for the three cases.

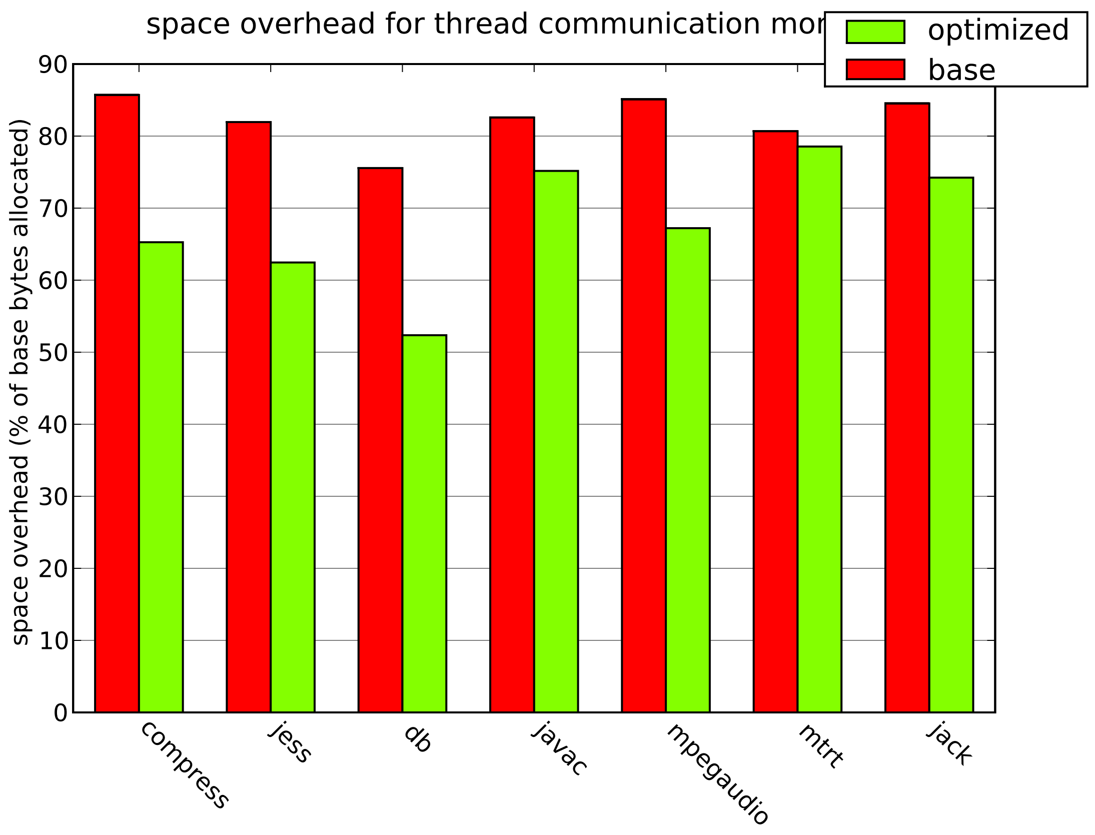

Figure 5.5: Space overhead of monitoring thread communication with and without optimization using stationary fields

### 5.3.5 Experimental Results

Figure 5.5 shows the reduction in overhead that results from applying this optimization to monitoring thread communication in the SPEC JVM98 benchmarks. We express overhead as additional bytes allocated over the uninstrumented case, as a percentage of bytes allocated in the uninstrumented cases. The overhead for the base instrumented case is 75–85%: the tagging method requires nearly as much tag data as the size of objects. Applying the optimization reduces the overhead on average by 14% of the uninstrumented bytes, with the maximum being 23% (the db benchmark) and the minimum 2% (mtrt). The results demonstrate that these benchmarks do allocate objects containing stationary fields, and that the optimization can be profitable.

It is also interesting to compare these results with those in Section 3.4, which reported the fraction of reads that were of stationary fields. By both metrics the db application uses stationary fields the most. Overall stationary fields comprise a smaller percentage of bytes allocated than percentage of reads, which suggests that the stationary reads are from a smaller number of hot, perhaps long-lived objects, rather than from a larger number of little-used objects.

## 5.4 The Java Hash Code Method

In this section, we hypothesize that programmers use stationary fields to ensure that they are correctly implementing Java's `hashCode` method. If this hypothesis holds, it creates an expectation that specific fields will be stationary. Cases that fit the pattern provide evidence that programmers use stationary fields to maintain program invariants. Conversely, by examining fields for which this expectation is violated, we may discover reasons the definition of stationary fields excludes fields we would expect it to include.

### 5.4.1 Background

Maps, sometimes called dictionaries, are a commonly used data structure, and the Java standard libraries provide a standard interface for maps, as well as implementations based on hash tables and binary trees. A table-based implementation, such as `java.util.HashMap`, requires a way of obtaining a hash value in order to determine in where in the table a key should be located. To facilitate this, the Java base object class contains a method `hashCode` that returns an integer. The implementation in `java.lang.Object` returns an hash code based on the identity of the object. Subclasses can override the hashCode method if they want their use in hash maps to be based on some logical notion of the object's value, rather than its physical identity.

When an object is placed as a key in a hash table, the hash table class will place it in a particular bucket. If the same object is used in a subsequent lookup, the hash table will attempt to locate based on its hash code. If the hash code, as returned by the object's hashCode method, has changed, the the hash map would be unable to locate the object, leading to confusing results. This possibility is therefore prohibited by the contract for hashCode and for maps.

The contract for hashCode [25] states that:

> Whenever [`hashCode`] is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` is modified.

The contract for `java.util.Map` further states that "The behavior of a map is not specified if the value of an object is changed in a manner that affects `equals` comparisons while the object is a key in the map." In combination these two requirements imply that an object must return the same hash code throughout its tenure as a key in a map.

### 5.4.2 Stationary Fields and hashCode

A programmer implementing a hashCode method will need a method of ensuring that the contract is obeyed, and that the method returns consistent results. One approach would be to compute the result by using only stationary fields. Because a stationary field is guaranteed to have the same value each time it is read, a method constructed in such a fashion will return the same value each time it is invoked on the same object.

### 5.4.3 Algorithm

We present an algorithm for verifying that hashCode methods conform to the relevant portion of the hashCode contract, by verifying that the compute their result using only stationary fields. The algorithm is as follows. Given a hashCode method $m$:

- Calculate all transitive callees of $m$, by traversing the call graph to find every method is reachable from $m$.

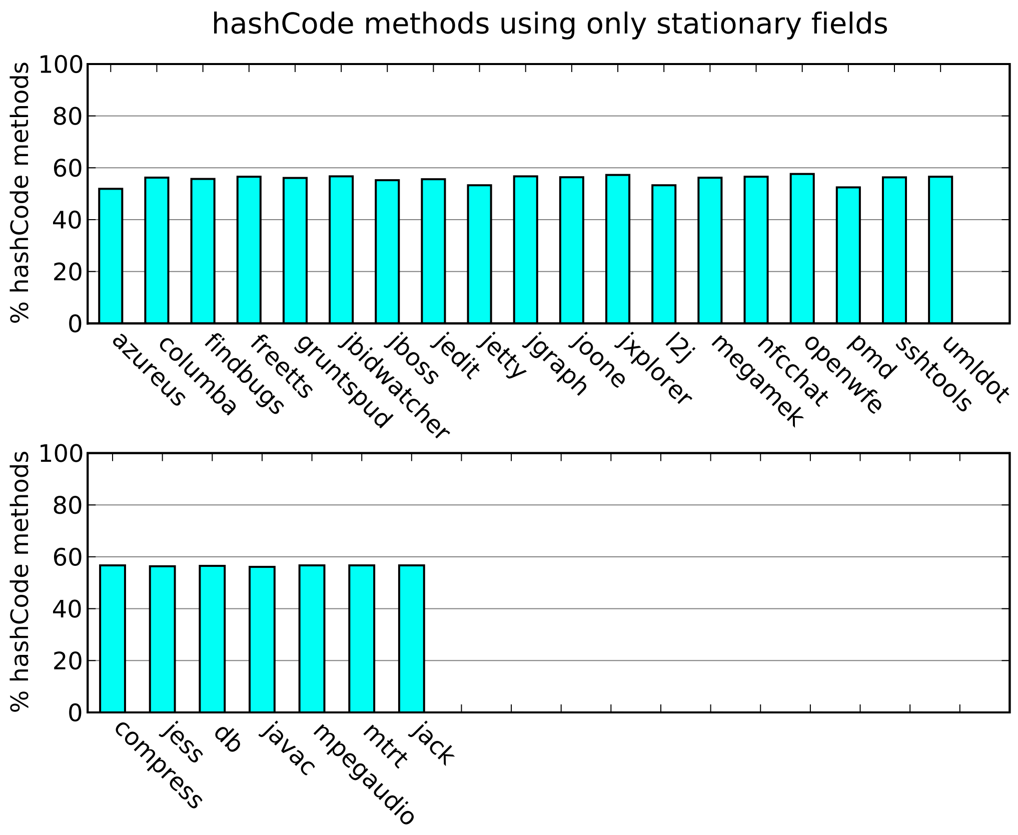

Figure 5.6: Percentage of hashCode methods using only stationary fields

- Obtain the set of all fields referenced by the callees by a simple scan of each method.
- If every instance field accessed is stationary, and every class field accessed is final, the hashCode method returns the same value each time it is invoked.

### 5.4.4 Program Study

We applied this algorithm to our benchmarks; the results are shown in Figures 5.6 and 5.7. About half of all hashCode methods are shown to use only stationary fields. The percentage is somewhat higher outside the standard libraries; however few of the hashCode methods occur outside the standard libraries (typically around a dozen of

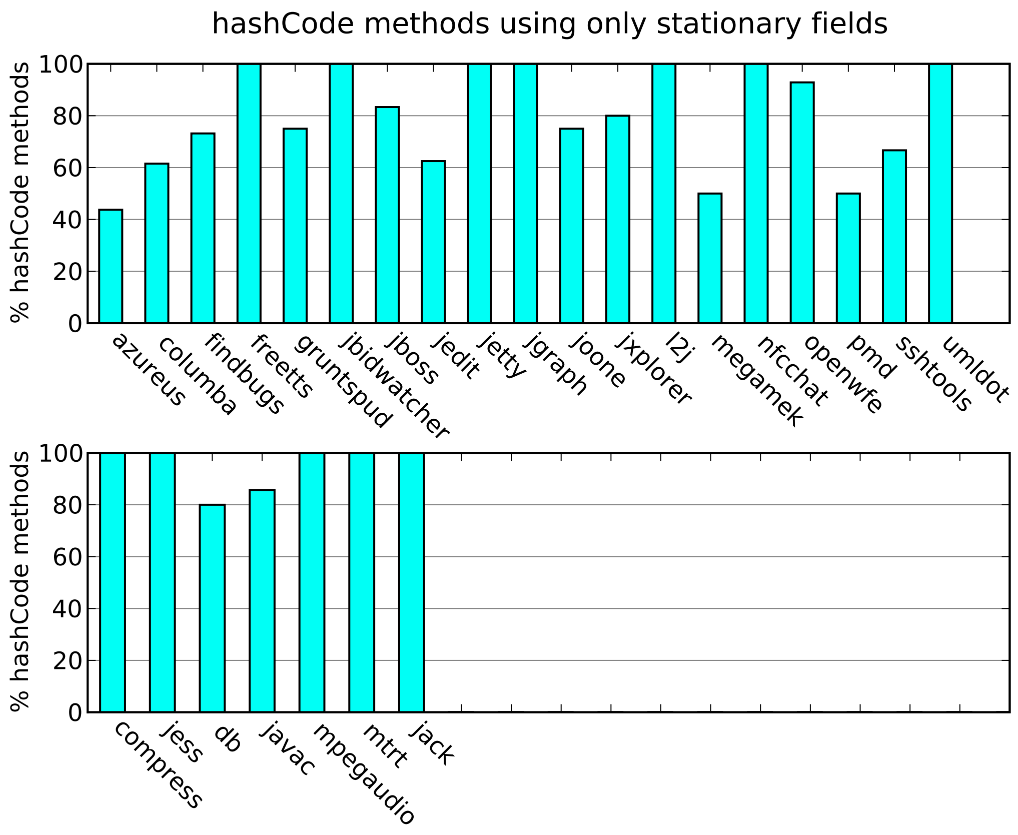

Figure 5.7: Percentage of hashCode methods using only stationary fields (application only)

three hundred total hashCode methods.)

This indicates that half of the hashCode methods may be using nonstationary fields. One reason for this result is the call graph obtained from CHA, which contains large strongly-connected components. Many hashCode methods are contained in these large SCCs, which are very likely to contain some method that uses a nonstationary field.

However, examining the results also revealed three major reasons that hashCode methods may be expected to use nonstationary fields.

#### hashCode and equals

There are requirements on hashCode other than that it return the same value throughout its tenure in a hash map. One requirement is that the hashCode method be consistent with the equals method. The equals method, defined in java.lang.Object and therefore present on every object, can be used to define a logical notion of object equality. The equals method of java.lang.Object, which is used for any class that does not override it, defines equality based on object identity. It considers two objects to be equal if and only if they are the same object.

Hash maps use a key object's hash code to select a bucket in which to locate its entry. However, it uses the key's notion of equality, defined by equals(), to determine whether keys located in that bucket match. Because a hash map needs to use the hashCode and equals method in coordination, hashCode and equals must be consistent. Specifically, the contract for equals and hashCode specifies that, for all objects x and y, if x.equals(y) returns true, then x.hashCode() and y.hashCode() must return the same value. The inverse need not be true: two non-equals objects may have the same hash code (that is, hashing collisions are not forbidden.)

One implication of this consistency requirement is that any class that overrides equals must also override hashCode in order to satisfy the contract. Many tools for Java will check that any class that defines equals also defines hashCode, and vice-versa. (They generally do not check that the two methods as implemented are actually consistent.)

However, desiring to use objects as keys in hash maps is rare relative to wanting to use a logical notion of equality. Many classes contain an overridden hashCode method simply to be consistent with their equals method as a matter of good practice, rather than for actual intended use as a key in a map. Mutable objects that define equality in terms of nonstationary fields will have a corresponding hashCode method defined in terms of nonstationary fields.

**hashCode in the Java Standard Libraries**

The Java standard libraries contain many cases where a hashCode method is defined to be consistent with an equals method. Many of these objects are mutable in general. For example, the library specification defines notions of equality for container types defined by interfaces such as java.util.List, java.util.Set, and even java.util.Map. This allows two differently implemented lists, such as one using linked list and another using an array, to be logically equal if they represent the same sequence of objects. In order to obey the contract regarding hashCode and equals, these interfaces also specify how to compute hash codes. Every class implementing any of these interfaces must implement both equals and hashCode.

Most data structures that implement these interfaces are mutable. For example java.util.LinkedList, java.util.HashSet, and java.util.HashMap are all mutable. They all implement equals and hashCode. It is possible to use these data structures as keys in maps simply by refraining from modifying them once they have been entered into a map. However, because these implementations are very widely used, it is likely that they will be used in a mutable fashion somewhere in any program, and will therefore contain some nonstationary fields.

This highlights a very important limitation of the definition of stationary fields: it requires fields to be *monomorphically* stationary. That is, the field must be stationary in *all instances*. It is likely that some fields will have the property that all the reads occur before all the writes on certain instances, but not on others. Relaxing the definition of stationary fields, such that a field in a particular object could be said to be stationary even if the same field in another object cannot, would likely significantly expand its applicability.

The Java standard libraries are the dominant implementer of hashCode, commonly in container implementations. This is one reason there are many classes that use nonstationary fields within hashCode methods, even when the effects of the imprecise call graph are considered.

### Cached Hash Codes

Computing a hash code may sometimes be expensive. For example, it may entail traversing a long string or large linked list. Some objects cache the hash code once computed so that it may be returned on successive calls without repeating the costly computation. Typically they do not precompute the value, instead waiting until hashCode is first invoked by the hash map, and then saving the returned value.

The field caching the hash code is found to be nonstationary by our analysis, because the hashCode may be invoked once the object is a key in the map, and is therefore heap-referenced. Furthermore, our stationary fields analysis does not understand the guard that prevents the cache from being written more than once.

There were several hashCode methods in our program study that fit this pattern. These methods are also examples of lazy initialization, as discussed in Section 4.1, although these fields are typically primitives and so not included in the experiment described in that chapter.

# Chapter 6

# Related Work

## 6.1 Immutability in Java

We begin with previous explorations of immutability in Java. Porat et al. [30] presented an analysis that infers final fields in Java; their implementation focused on static (class) fields rather than instance fields. Their analysis operated under an open-world assumption, severely limiting its ability to infer that non-private fields were final. Nonetheless, they found that static fields not declared final could be inferred to be so. Petchanski and Sarkar [29] used annotations about the immutability of Java fields to enable optimizations; their implementation used programmer-supplied annotations, rather than inference. Concurrently to our work, the JQual system performed inference of final fields as one instance of a type qualifier in Java [16]. They also found many opportunities to add the `final` keyword.

Several proposals have been made for adding a "read-only" qualifier to Java, including JAC [20], Universes [26], ModeJava [36], and Javari [39]. These all propose creating a qualifier to reference types; programs would be forbidden from using a read-only reference to modify fields. That is, with a read-only reference, the *referenced* object cannot be changed. This is orthogonal to `final` and stationary, which indicate that the qualified field may not be altered to reference another object. This distinction is also present in C and C++, where the `const` keyword may indicate either a pointer must continue to point to the same object, or that it may not be used to mutate the object pointed to.

Initialization in object-oriented languages can be quite complex; as we have shown, these restrictions limit the applicability of final. Others have also developed techniques to cut through the complexity of initialization. For example, Fähndrich and Xia introduced delayed types [11], which are types for which a specified property will hold in the future; in the interim, a type system proves that the fields are not referenced. They apply them to proving that fields are non-null even in the presence of complex initialization, especially as of circularly referential structures. Their technique could also be used to show that fields are stationary in the presence of circular references, which our analysis is unable to do.

## 6.2 Immutability in Other Languages

Many other languages include some method for indicating value immutability. C [17] and C++ [37] have `const`, which depending on use can indicate either a property similar to final and stationary, or to read-only. C# uses both keywords `const` and `readonly`, depending on whether the value is known at compile-time (`const`) or not (`readonly`.)

The Ruby programming language [38] includes the ability to "freeze" an object, after which no field of the object may be altered. Enforcement is entirely dynamic; attempts to modify a frozen object result in an exception.

Languages or libraries may supply both a mutable and immutable version of a particular data structure. For example, Java provides both strings (immutable) and string buffers (mutable). Python's standard types include both `tuple`, an immutable sequence type, and `list`, a mutable one; and both a mutable set and an immutable `frozenset` [41]. In Python, only the immutable versions of these types may be used as keys in dictionaries; this shields the programmer from dangers of changing hash codes similar to those in Java.

Foster et al. [13] presented an inference of type qualifiers for C, in which they discovered that many more `consts` can be used in C programs than are actually present; this corroborates our experience that programmers often neglect declarations of this kind.

## 6.3 Purity Analysis

Method purity analysis [35, 32] is in some sense a dual stationary fields analysis: a pure method changes no fields, and a stationary fields is changed by no methods. Techniques for finding the two will logically share some similarities, in that both will be computing what objects a piece of code may affect. A large difference is when the two techniques may give up. Once our analysis has found an effect that renders a field nonstationary, it can ignore that field throughout the program. Depending on the exact definition of purity, a purity analysis may be able to stop analyzing a section of code when it sees the first side effect within it.

## 6.4 Escape Analysis

In the description of our algorithm we noted a relationship between our algorithm's lost objects and the notion of an escaped object defined by escape analysis [34, 9, 28, 4]. An object is said to escape a method if its lifetime exceeds the runtime of that method. In general escaped does not imply lost, and lost does not imply escaped. An object may escape a method by being returned directly by the method, in which it is escaped but not lost. An object may be lost by the creation of a reference in an object that is local to a method, in which case it is lost but does not escape.

However, many of the ways that an object may escape involve the creation of a reference. An object may be returned indirectly by a function after the creation of a reference in some other object that is returned directly. An object may escape when a reference to it is stored into an object that existed before the method was invoked. Escape analyses vary in the precision with which they handle the heap. Some escape analyses assume that any object referred to by other objects may escape; such analyses would assume that all of our lost objects could escape.

Some escape analyses attempt to track some objects into the heap, to prove that they do not escape despite the existence of references. Similar techniques could be applied to strengthen our stationary fields analysis, finding more stationary fields at the cost of increased computational expense.

## 6.5 Stationary Fields and Concurrency

Existing work on reducing synchronization overhead in Java typically focuses on overall properties of objects or the locks they use, rather than on individual fields. The use of a thread escape analysis to eliminate synchronization on objects that are private to a thread is a common technique [1, 43, 5, 7]. Even if objects are shared, synchronization may be eliminated if they are only synchronized by a single thread [33]. These approaches determine whether an entire object need not be synchronized, without regard to whether individual fields may not need to be. Other approaches focus on the locks rather than the objects, for instance eliminating synchronization when a lock is reentered or when one lock always encloses another [2].

Goetz [14] notes that an object that is composed entirely of `final` fields may be shared freely between threads with no need for synchronization operations, and proposes the use of a Java annotation `@Immutable` to indicate such objects. Our data on classes that use only stationary fields suggests that many classes could potentially be thread-safe, and that an automated tool could effectively infer the annotation.

The race detector rccjava [12] requires that fields that point to locks be `final`, in order to ensure that the same lock is always used; this is an example of how the must-alias property given by stationary fields may be used. Our program study indicates that few fields are marked `final`, which is a downside of this requirement. Using inferred stationary fields would expand the number of fields available to refer to locks.

## 6.6 Correctness of hashCode

Several tools, such as FindBugs [3] and the Java compiler within the Eclipse IDE, attempt to find errors relating to the implementation of hashCode. The errors searched for by these tools are typically limited to such things as arithmetic errors in the computation of hashCode, or implementing equals without implementing hashCode. No tool that we are aware of attempts to verify that hashCode returns a consistent value. The implementations of hashCode and equals may also be generated automatically, for example by an editor such as Eclipse or JBuilder. Doing so minimizes the risk of a mismatch between the two methods, or errors such as null pointer exceptions, but again does not address the problem of a potentially changing hash code.

# Chapter 7

# Conclusion

This thesis introduced the notion of a stationary field, a field whose value never changes once its value has been read. We have developed an efficient algorithm to identify stationary fields. It was surprising that this simple algorithm identifies approximately half of the fields in Java programs to be stationary.

While the inference of stationary fields is relatively simple compared to other pointer alias analysis, it yields an important invariant property about object fields. Previous context-sensitive points-to analyses forgo flow sensitivity and objects are often named by the allocation site, without context sensitivity. A field in a heap object usually points to a multitude of objects; such analysis cannot tell if two accesses of the same field, even if they are right next to each other, yield the very same object. In contrast, our results show that for about half of the fields, reading the same field off an object always yields the same result throughout the program.

Stationary fields are very common, but they are certainly not all the fields in programs. Some programming idioms may interfere with fields being stationary. We gave two examples: lazy initialization of fields, and modification to fields as objects are disposed of. Recognizing such idioms and developing tools to identify them allows us to realize the maximum benefit from stationary fields. There are many programming idioms and practices, and it is likely that there are many more that interact with stationary fields.

We showed several applications. Clearly these examples are just first steps in the application of stationary fields to program analysis; most of the potential applications are complex research topics in their own right. We are encouraged that, since the prior publication of portions of this work [40], others have continued the task of determining how stationary fields can be used [24, 31, 6]. It is not a coincidence that three of the four applications we present are to concurrent programs: the properties of stationary fields seem to be very helpful in reasoning about concurrent programs. This is likely to be a fruitful avenue for future research, especially given that concurrent programs appear to be increasingly important now that commodity processors are delivering much of their performance increases in the form of multiple cores.

Finally, our result suggests new approaches to tackling pointer alias analysis in Java. We can devise different analysis techniques for stationary and nonstationary fields. We already showed that having a bit more precision during the initialization phase provides a lot of information for stationary fields. Further analysis of the initialization code of stationary fields will provide valuable information about the identities of what the stationary object fields point to. Similarly, better understanding of nonstationary fields may lead to specialized and more efficient analysis for them too.

# Bibliography

[1] Jonathan Aldrich, Craig Chambers, Emin Gün Sirer, and Susan J. Eggers. Static analyses for eliminating unnecessary synchronization from java programs. In *SAS '99: Proceedings of the 6th International Symposium on Static Analysis*, pages 19–38, London, UK, 1999. Springer-Verlag.

[2] Jonathan Aldrich, Emin Gün Sirer, Craig Chambers, and Susan J. Eggers. Comprehensive synchronization elimination for Java. *Sci. Comput. Program.*, 47(2-3):91–120, 2003.

[3] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8, New York, NY, USA, 2007. ACM.

[4] Bruno Blanchet. Escape analysis for object-oriented languages: application to Java. *SIGPLAN Not.*, 34(10):20–34, 1999.

[5] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in java. *SIGPLAN Not.*, 34(10):35–46, 1999.

[6] Nathan G. Bronson, Christos Kozyrakis, and Kunle Olukotun. Feedback-directed barrier optimization in a strongly isolated stm. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 213–225, New York, NY, USA, 2009. ACM.

[7] 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.

[8] 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.

[9] Alain Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 157–168, New York, NY, USA, 1990. ACM.

[10] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Bejamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of Eighteenth ACM Symposium on Operating System Principles*, pages 57–72, October 2001.

[11] Manuel Fahndrich and Songtao Xia. Establishing object invariants with delayed types. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 337–350, New York, NY, USA, 2007. ACM.

[12] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. *SIGPLAN Not.*, 35(5):219–232, 2000.

[13] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 192–203, New York, NY, USA, 1999. ACM Press.

[14] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice.* Addison-Wesley Professional, 2006.

[15] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2005.

[16] David Greenfieldboyce and Jeffrey S. Foster. Type qualifier inference for Java. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 321–336, New York, NY, USA, 2007. ACM.

[17] Samuel P. Harbison and Guy Steele. *C, A Reference Manual*. Prentice-Hall, Inc., New York, NY, 1987.

[18] David L. Heine and Monica S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 168–181, New York, NY, USA, 2003. ACM Press.

[19] David L. Heine and Monica S. Lam. Static detection of leaks in polymorphic containers. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 252–261, New York, NY, USA, 2006. ACM Press.

[20] Gunter Kniesel and Dirk Theisen. JAC — acess right based encapsulation for Java. *Software — Practice and Experience*, 31(6):555–576, 2001.

[21] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: inferring the specification within. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 12–12, Berkeley, CA, USA, 2006. USENIX Association.

[22] Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 17–32, New York, NY, USA, 2002. ACM Press.

[23] Benjamin Livshits and Thomas Zimmermann. DynaMine: finding common error patterns by mining software revision histories. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 296–305, New York, NY, USA, 2005. ACM Press.

[24] Alexey Loginov, Eran Yahav, Satish Chandra, Stephen Fink, Noam Rinetzky, and Mangala Nanda. Verifying dereference safety via expanding-scope analysis. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 213–224, New York, NY, USA, 2008. ACM.

[25] Sun Microsystems. Java platform, standard edition 6, API specification. http://java.sun.com/javase/6/docs/api/.

[26] Peter Müller and Arnt Poetzch-Heffter. A type system for controlling representation exposure in Java. In *2nd ECOOP Workshop on Formal Techniques for Java Programs*, 2001.

[27] Matthew Nasielski and Julien Wetterwald. Flows2. http://cs.stanford.edu/people/wetterwa/cs343/.

[28] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. *SIGPLAN Not.*, 27(7):116–127, 1992.

[29] Igor Pechtchanski and Vivek Sarkar. Immutability specification and its applications. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 202–211, New York, NY, USA, 2002. ACM Press.

[30] Sara Porat, Marina Biberstein, Larry Koved, and Bilha Mendelson. Automatic detection of immutable fields in Java. In *CASCON '00: Proceedings of the 2000 Conference of the Centre for Advanced Studies on Collaborative research*, page 10. IBM Press, 2000.

[31] Ian Rogers, Jisheng Zhao, Chris Kirkham, and Ian Watson. Constraint based optimization of stationary fields. In *PPPJ '08: Proceedings of the 6th international symposium on Principles and practice of programming in Java*, pages 95–104, New York, NY, USA, 2008. ACM.

[32] Atanas Rountev. Precise identification of side-effect-free methods in java. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 82–91, Washington, DC, USA, 2004. IEEE Computer Society.

[33] Erik Ruf. Effective synchronization removal for Java. *SIGPLAN Not.*, 35(5):208–218, 2000.

[34] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–293, New York, NY, USA, 1988. ACM.

[35] A. Salcianu and M. Rinard. Purity and side effect analysis for Java programs. *Lecture Notes in Computer Science*, 3385:199–215, 2005.

[36] Mats Skoglund and Tobias Wrigstad. A mode system for readonly references. In Akos Frohner, editor, *Formal Techniques for Java Programs*, number 2323 in Object-Oriented Technology, ECOOP 2001 Workshop Reader, pages 30–, Berlin, Heidelberg, New York, 2001. Springer-Verlag.

[37] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[38] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby: The Pragmatic Programmer's Guide, Second Edition*. Addison-Wesley, 2005.

[39] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 211–230, San Diego, CA, USA, October 18–20 2005.

[40] Christopher Unkel and Monica S. Lam. Automatic inference of stationary fields: a generalization of Java's final fields. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 183–195, New York, NY, USA, 2008. ACM.

[41] Guido Van Rossum and Fred L. Drake. *The Python Language Reference Manual (version 2.5)*. Network Theory Ltd., September 2003.

[42] John Whaley. Joeq: A virtual machine and compiler infrastructure. In *Proceedings of the SIGPLAN Workshop on Interpreters, Virtual Machines, and Emulators*, pages 58–66, June 2003.

[43] 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.

[44] Chadd C. Williams and Jeffrey K. Hollingsworth. Recovering system specific rules from software repositories. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[45] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 282–291, New York, NY, USA, 2006. ACM Press.