Profiling Field Initialisation in Java

Stephen Nelson, David J. Pearce, and James Noble

Victoria University of Wellington
Wellington, New Zealand
{stephen,djp,kjx}@ecs.vuw.ac.nz

Abstract. Java encourages programmers to use constructor methods to initialise objects, supports **final** modifiers for documenting fields which are never modified and employs static checking to ensure such fields are only ever initialised inside constructors. Unkel and Lam observed that relatively few fields are actually declared final and showed using static analysis that many more fields have final behaviour, and even more fields are *stationary* (i.e. all writes occur before all reads). We present results from a runtime analysis of 14 real-world Java programs which not only replicates Unkel and Lam's results, but suggests their analysis may have under-approximated the true figure. Our results indicate a remarkable 72-82% of fields are stationary, that **final** is poorly utilised by Java programmers, and that initialisation of immutable fields frequently occurs after constructor return. This suggests that the **final** modifier for fields does a poor job of supporting common programming practices.

1 Introduction

The notion of immutability has been well-studied in the programming language community (e.g. [1,2,3]). Modern statically typed languages, such as Java and C#, typically support immutable fields (e.g. **final** in Java) though, curiously, explicit support for immutable classes is often missing. Bloch advises Java programmers to "Favour Immutability" as "Immutable classes are easier to design, implement, and use than mutable classes" [4]. This leads to a natural question of how well immutability modifiers (such as Java's **final**) match common programming idioms.

To examine this question, Unkel and Lam developed the term *stationary field* to describe fields which are never observed to change, that is, all writes precede all reads [5]. Their intuition was that programmers are often forced to perform *late initialisation* of objects (i.e initialisation after the constructor has returned), meaning some fields cannot be declared **final**. A common idiom where this happens is with the initialisation of cyclic data structures [6].

Unkel and Lam performed a *static analysis* over a corpus of 26 Java applications, and found that 40-60% of Java fields were stationary. Their analysis was necessarily conservative and, hence, under-reported the number of stationary fields. In this paper, we report on an experiment to identify stationary fields

using *runtime profiling*. Our results from 14 Java applications indicates that 72-82% of fields are stationary — thereby supporting the conclusion of Unkel and Lam, but suggesting that it is an underestimate.

2 Background

Java provides a special modifier for immutable fields: **final**. Java itself ensures that fields annotated with **final** are only modified once and only within constructors. Java encourages programmers to use constructors for initialising fields and establishing invariants. Constructors are distinct from regular methods in one significant way: they can initialise **final** fields. Unfortunately, programmers are sometimes forced (or voluntarily choose) to initialise fields late (i.e. after the constructor has completed). This prevents such fields from being marked **final** *even when they are designed to be immutable*. The following illustrates a common patten which exemplifies this scenario:

```
abstract class Parent {
  private final Child child;
  public Parent(Child c) { this.child = c; }
}

abstract class Child {
  private Parent parent; // cannot be marked as final
  public void setParent(Parent p) { this.parent = p; }
}
```

The programmer intends that every Parent has a Child and vice-versa and, furthermore, *that these do not change for the life of the program*. He/she has marked the field Parent.child as **final** in an effort to enforce this. However, he/she is unable to mark the field Child.parent as **final** because one object must be constructed before the other. We have marked Parent and Child **abstract** to indicate the intention that different subclasses will be used.

In the above example, the method Child.setParent(Parent) is used as a *late initialiser*. This is a method which runs after the constructor has completed and before which the object is not considered properly initialised. Once the Child.parent field is late initialised by this method, the programmer does not intend that it will change again. Such a field — where all reads occur after all writes — is referred to as *stationary* [5]. Another common situation where this arises is for classes with many configurable parameters. In such case, the programmer is faced with providing a single large constructor and/or enumerating many constructors with different combinations of parameters. Typically, late initialisation offers a simpler and more elegant solution.

3 Implementation

We have developed a profiler called *rprof* to track (amongst other things) all reads and writes to object fields in an executing program. The key advantages of

rprof are that it runs on a commodity JVM (e.g. Oracle's HotSpot JVM), catches reads/writes to almost all objects (including those in the standard library), profiles large real-world applications with manageable overhead and processes event traces (containing potentially billions of events) using a parallel, distributed map/reduce computation. There are four main components:

- *Agent*: a JVMTI [7] C++ agent loaded by the JVM running the target application.
- *Profiler*: a Java application running in a separate JVM that performs bytecode rewriting, and provides other utility functions for the agent.
- *Worker*s: Java applications that aggregate the event stream and handle storing the results.
- mongodb: a commercial *nosql* database server that the profiler and worker applications use to store persistent data.

The profiler and workers perform tasks in parallel using multiple threads on multiple computers, minimising overheads on the profiled application. To enable profiling, *rprof* performs bytecode rewriting on the target application. When the JVM loads a class, it is intercepted by the agent which inserts instrumentation using the ASM bytecode modification library [8]. To increase the range of objects which can be profiled, bytecode rewriting is performed by the profiler in a separate JVM from the target application. The agent passes classes to be rewritten to the profiler (potentially across the network) running in another JVM which rewrites them, and passes them back. Without this, classes needed by the ASM library could not be profiled as they would have to be loaded before rewriting.

Due to lack of space, unfortunately we cannot discuss every aspect of the *rprof* profiler. A more complete discussion of the operation of *rprof* can be found in [9]. We will now give a high-level overview of the main issues.

3.1 Object Tracking

The first challenge faced in *rprof* is the unique identification of objects in the target application. Three options exist for uniquely identifying objects within the JVM:

- *Using Object References.* This approach is commonly used with weak references to ensure garbage collection proceeds as normal (see e.g. [10,11]). Since we store profiling data in the mongodb database, we require a concrete ID rather than a reference. Unfortunately, Java itself provides no easy mechanism for converting references into IDs[1].

[1] Some works (e.g. [12,13]) employ System.identityHashCode(Object) to generate object IDs. The value returned from this method is derived from the object's physical address, and then stored for subsequent calls. Consequently, it is unsuitable for uniquely identifying objects because, in standard VMs (using generational garbage collectors) objects initially reside within the nursery. This is a relatively small region of memory and we found many live objects which shared the same identityHashCode().

– *Using Physical Memory Addresses.* Since object references correspond to physical memory addresses in the JVM, a logical option is to use them as unique IDs. Through the JVMTI it is possible to convert an object reference into a physical address. Whilst this may seem straightforward, it is fraught with difficulty since an object can change its physical location during garbage collection. In other words, we would need to intercept garbage collection events to determine which objects were moved and now have a new physical address (hence, ID).
– *Storing unique ID's with every object.* The JVMTI provides a mechanism whereby agents can associate a 64bit (long) tag with any object. The JVM maintains this tag and handles all issues related to garbage collection, etc.

In our context, the only viable solution is to associate unique IDs with objects via the JVMTI — which is the approach taken in *rprof*.

Tracking System Objects. Before the JVM loads native agents, it performs some basic bootstrapping including loading classes such as java.lang.Object and java.lang.String. Agents have a chance to modify previously loaded classes, but JVMTI facilities such as object tagging remain unavailable until the JVM reaches the end of its bootstrapping phase. Once the JVM has completed bootstrapping, the agent uses JVMTI to iterate over all the Java objects, and adds unique ID tags to them.

3.2 Tracking Methods and Constructors

rprof is capable of generating events for all method and constructor calls and returns (inc. exceptional returns), although this analysis only requires method return tracking for constructors. Since the JVMTI does not provide any facility for tracking method calls, *rprof* uses bytecode modification to instrument classes as the JVM loads them. Consider the following Java method:

```
public boolean isHello(String message) { return message.equals("Hello!"); }
```

This takes a string as input and returns true if the message is "Hello!". *rprof* can generate three different events for this method: a *'Method Enter'* event, a *'Method Return'* event, and/or an *'Exceptional Return'* to catch any thrown exceptions (e.g. because message is null). Figure 1 shows how *rprof* instruments this method by inserting bytecodes to generate these three events. For example, the inserted code at the beginning pushes class and method identifiers onto the stack, constructs an array containing the arguments to the method, then invokes a static *rprof* method for tracking method entry events. *rprof* handles method returns similarly to method calls. *rprof* generates exceptional returns by wrapping the method body in a try block, and inserting a finally handler at the end of the method which signals to *rprof* an exceptional return. *rprof* inserts the try block last so that any other catch or finally blocks run first. If a block consumes the exception and returns normally then *rprof*'s exceptional return handler will not run.

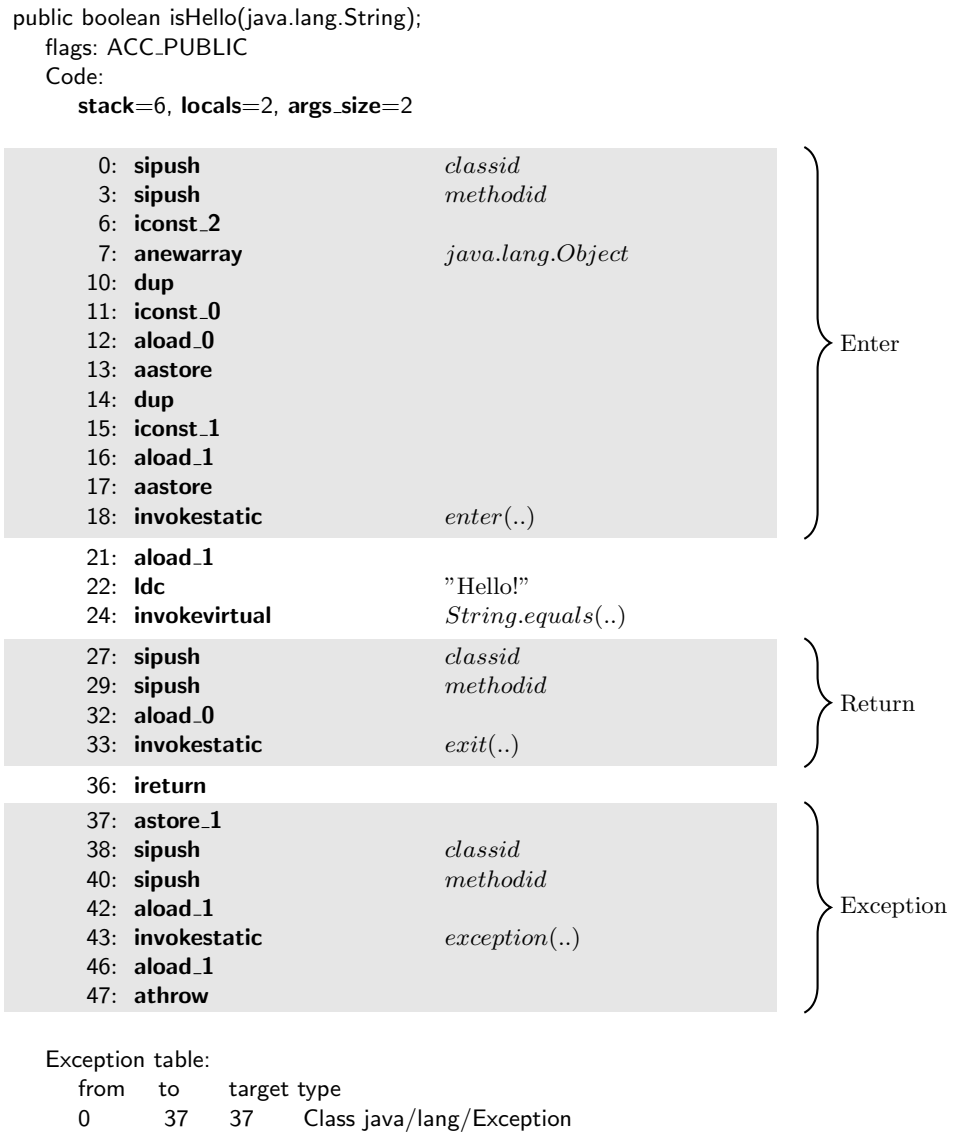

Fig. 1: The byte code resulting from modifying an example method to track method entry, exit, and exceptional return.

3.3 Tracking Fields

rprof supports tracking both field writes and field reads. This is simpler than for methods, since the JVMTI provides a callback mechanism to notify agents of these events. The callback provides a reference to the object that owns the field, the value of the read or write, and the JVM-internal field ID. Unfortunately, JVM field IDs are unique to a given class, but are not guaranteed unique across all classes. *rprof* requires program-wide unique field IDs for persistence and, hence, maintains a map between JVM-internal field IDs and *rprof*'s persistent field IDs. Finally, *rprof* tracks all objects created within the JVM, but it is not able to track all fields. *rprof* does not track fields of the following classes:

- java.nio.charset.CharsetDecoder, java.nio.charset.CharsetEncoder, java.util.zip.ZipFile. These three classes use some form of JVM optimisation which causes segfaults if *rprof* tracks them.
- java.lang.Throwable. The JVM generates an additional field at runtime which causes off-by-one errors in our tracking code.
- java.lang.String. Excluded because it is so common: Strings are actually immutable but they use internal fields to track access behaviour and generated properties, resulting in a disproportionate number of events which are not interesting for this experiment.

3.4 Data Aggregation and Analysis

rprof event streams contain billions of events which would take days or even weeks to store on disk. Reducing this event stream to a more compact form suitable for analysis or visualisation is a computational challenge. To address this, *rprof* processes the event stream as it is generated using a map-reduce style computation. *rprof* parallelises this computation across a cluster of machines, dramatically reducing the time and space taken to store the results.

The worker processes operate on the event stream as it is being generated, reducing it to a form amenable for analysis or visualisation. This reduction is specific to the experiment being performed, and discards information not directly relevant. For example, in the experiments discussed in this paper, it is not necessary to count how many field accesses there were — we only need to note when the first and last field reads and writes occurred.

As the profiled application runs the agent generates *event records*. These are mapped and reduced to *instance records* which, in turn, are then mapped and reduced to *result records*. Figure 2 presents the information contained in these records (roughly speaking) for the experiment presented in this paper. On the left of the figure, we see the event records emitted by the agent. One of these is emitted for every event being monitored in the profiled application which includes *constructor method exit* and *field read/write*. These events are then mapped and reduced to a set of complete instance records (shown in the middle), which capture information about a given object relevant to this experiment. The complete instance records are then mapped and reduced again to form the final result records (shown on the right).

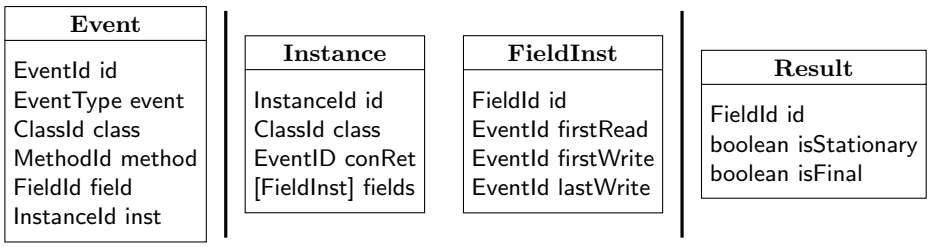

Fig. 2: Illustrating the event records emitted by the agent (left), which are mapped and reduced to instance records (middle) and then again into the final result records (right).

```
// Map event into instance record
void map(Event e) {
  InstanceId id = e.inst;
  Instance inst = new Instance();
  switch(e.type) {
    case FIELD_READ:
      FieldInst fi = new FieldInst(e.field);
      fi.firstRead = fi.lastRead = e.id;
      inst.fields.add(fi);
      break;
    case FIELD_WRITE:
      FieldInst fi = new FieldInst(e.field);
      fi.firstWrite = fi.lastWrite = e.id;
      inst.fields.add(fi);
      break;
    case METHOD_RETURN:
    case METHOD_EXCEPTION:
      if(isConstructor(e.clazz,e.method)) {
          inst.conRet = e.id;
          break;
      }
    default:
      return; // don't emit anything
  }
  emit(id, inst);
}
```

```
// Reduce two instances with same ID
Instance reduce(Instance l, Instance r) {
  // Update constructor return ID
  l.conRet = max(l.conRet,r.conRet);

  // Merge field instance records
  for(int i=0;i<l.fields.size();++i) {
    FieldInst fi = l.fields.get(i);
    for(int j=i+1;j<l.fields.size();++j) {
      FieldInst fj = l.fields.get(j);
      if(fi.id == fj.id) {
        fi.firstRead = min(fi.firstRead,fj.firstRead);
        fi.firstWrite = min(fi.firstWrite,fj.firstWrite);
        fi.lastWrite = max(fi.lastWrite,fj.lastWrite);
        l.fields.remove(j);
        j = j - 1;
      }
    }
  }

  return l;
}
```

Fig. 3: Illustrating how Event records are mapped to (incomplete) Instance records, which are then reduced to form complete records. Not every Event maps to an Instance record; for example, most method entry events are ignored, with only method return and exceptional return events on constructors emitting Instances. The methods min(EventID,EventID) and max(EventID,EventID) operated as expected — by return the earlier (resp. later) of the two parameters and handling null values correctly. Finally, please note that, in practice, these methods are further optimised for performance.

Consider the process of converting Event records into Instance records. Initially, each Event record is either ignored (if not relevant) or *mapped* to an (incomplete) Instance record. These Instance records are then *reduced* to form complete instance records. Here, Instance.conRet gives the EventID for the object's constructor return, whilst Instance.fields contains records for its fields. For each field, firstWrite and lastWrite give the EventIDs for the first and last write and, similarly, for firstRead. Figure 3 illustrates the map and reduce procedures.

Consider now the process for converting complete Instance records into complete Result records. This is similar to before. Given a complete instance record we can determine which of its fields were stationary (i.e. all writes before all reads) and/or final (i.e. one write which occurred before constructor return). We then reduce all Result records for a given class to determine which fields were stationary and final *across all instances*. The reduce procedure is thus:

```
// Reduce result records with same field ID
Result reduce(Result left, Result right) {
  left.isStationary &= right.isStationary;
  left.isFinal &= right.isFinal;
  return left;
}
```

Here, we see how two Result records with the same FieldID are reduced. All Result records for a given field are reduced to a single Result record capturing its stationary and final status across all instances.

4 Experimental Results

We now present our experimental results looking at final and stationary fields. We begin with a more detailed definition of these terms.

Final (F). A *Final* field is an *object instance field* which is modified once, before the object's constructor method returns. A field is not final if, for any object which reads the field, the field is written to after the object's constructor returns or the field is written to more than once. Final fields may be *Declared Final (dF)* or *Undeclared Final (uF)*. A declared final field is any field whose declaration in Java code is annotated with the final modifier. A field which is not annotated with this modifier but nevertheless conforms to this definition is undeclared final.

Stationary (S). A *Stationary* field is an *object instance field* which is not modified after it has been read. A field is not stationary if there exists an object which modifies that field after it has been read. The set of stationary fields has no relationship with that of declared or undeclared final fields. A field which has been declared final may have its state read before it is initialised (while it is in its default state). This is valid behaviour for declared final fields but not stationary fields. Likewise, stationary fields may be initialised after constructor return, which is not valid behaviour for final fields.

Name	Description	Classes	Methods
avrora	Simulates a number of programs run on a grid of AVR microcontrollers.	999	10685
batik	Produces a number of Scalable Vector Graphics (SVG) images based on the unit tests in Apache Batik.	1814	21710
eclipse	Executes some of the (non-gui) jdt performance tests for the Eclipse IDE.	2653	37949
fop	Takes an XSL-FO file, parses it and formats it, generating a PDF file.	1703	20133
h2	Executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application.	934	14622
jython	Inteprets the pybench Python benchmark.	2953	34167
luindex	Uses lucene to indexes a set of documents; the works of Shakespeare and the King James Bible.	788	10887
lusearch	Uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible.	701	9640
pmd	Analyzes a set of Java classes for a range of source code problems.	1328	18281
sunflow	Renders a set of images using ray tracing.	907	12854
tomcat	Runs a set of queries against a Tomcat server retrieving and verifying the resulting webpages.	2373	32369
tradebeans	Runs the daytrader benchmark via a Jave Beans to a GERONIMO backend with an in memory h2 as the underlying database.	8155	96250
tradesoap	Runs the daytrader benchmark via a SOAP to a GERONIMO backend with in memory h2 as the underlying database.	8246	97026
xalan	Transforms XML documents into HTML.	1125	14260

Table 1: List of programs in the *Dacapo* benchmarks suite (dacapo−9.12−bach) including a brief summary (from [14]). Also included are statistics on each benchmark obtained using *rprof* giving the number of classes loaded during the experiment run (including interfaces) and the number of methods they contained (including static).

4.1 Benchmarks

Unkel and Lam analysed a selection of Java programs and also the SpecJVM98 benchmark suite. Unlike their static analysis, our dynamic analysis needs to execute each program with a set of inputs that will exercise the program's functionality in a reproducible manner. We decided to analyse the *dacapo* benchmark suite, a compilation of non-trivial real world Java applications designed for benchmarking that includes non-trivial inputs for each application [14]. The dacapo suite consists of the 14 applications listed in Table 1. Each benchmark was executed using the default input size for a single iteration. We used the following command to execute benchmarks:

```
java [rprof−opts] −Xint −Xmx1024m −jar dacapo−9.12−bach.jar −n 1 −t 1 [benchmark]
```

4.2 Experimental Setup

The benchmarks were profiled executing on an Opteron 254 (2.8GHz) dual-CPU machine with 4GB of memory running Ubuntu 10.04.3 LTS (64 bit server) using OpenJDK 1.8.0-ea-b37[2]. We used the preview Java 8 build because our analysis is not stable on previous JDK versions. OpenJDK 1.8.0-ea-b37 includes a bug fix for a problem with JVMTI which we identified and reported[3].

4.3 Results

The results in this section are directly comparable to the Unkel and Lam's work on stationary fields so we use a consistent format to present our results [5]. Figure 4 presents the results of our analysis for all fields. The first column shows the name of the benchmark, the second shows the total number of unique fields contributed by each benchmark (i.e. which were read or written at least once during the run), all other columns show the percentage of the total number of fields in that category (rounded to whole numbers). Results are separated broadly into stationary and non-stationary fields, then into *declared final* (dF), *undeclared final* (uF) and *not final* (¬F). The final three columns show summary information: the total number of *declared final* (dF) fields, *final* fields (F = dF ∪ uF), and *stationary* fields (S).

Figure 4 shows that between 70% and 86% of the fields declared in dacapo benchmarks are *stationary* (S) and between 50% and 68% are *final* (F). The number of final fields which are declared final (dF) varies between benchmarks but, in most cases, is less than half the number of fields whose behaviour was observed to be final.

Finally, Figures 5 and 6 show the results across fields of reference type and fields of primitive type. The format is largely the same as before, comparing final field behaviour between stationary and non-stationary fields in each case.

4.4 Discussion

Comparing our results to Unkel and Lam's [5], we can make the following main observations:

1. **Declared v Undeclared Final**. Consistent with the findings of Unkel and Lam, we find many undeclared final fields. This suggests that programmers are not making good use of the final modifier. The most obvious reason it that some programmers may simply be "lazy" and choose not to use it even when they could. However, other possibilities exist. For example, such undeclared final fields may be protected, where the programmer anticipated subclasses that would mutate them but which never eventuated in the given application.

[2] The batik benchmark which relies on a proprietary jpeg class which is not included in the pre-release JDK 8 build. For this benchmark we used Oracle JDK 1.7.0_03-b04.

[3] http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=7162645

Program	Total	S (%)			¬S (%)			%	%	%
		dF	uF	¬F	dF	uF	¬F	dF	F	S
avrora	1,702	36	31	16	0	0	17	36	67	82
batik	3,272	8	52	17	0	0	23	8	60	77
eclipse	6,779	10	37	25	0	0	28	10	46	72
fop	3,390	9	43	28	0	0	19	9	53	81
h2	2,014	17	39	22	0	0	22	17	56	78
jython	2,501	14	46	19	0	0	22	14	60	78
luindex	1,674	18	38	16	0	0	26	18	57	73
lusearch	1,303	14	43	21	0	0	21	14	58	78
pmd	1,962	14	41	24	0	0	20	14	55	79
sunflow	1,831	14	43	20	0	0	23	14	57	77
tradebeans	15,404	25	35	21	0	0	19	25	60	81
tradesoap	15,611	25	35	21	0	0	19	25	60	81
tomcat	6,279	9	40	25	0	0	26	9	50	74
xalan	2,063	11	41	27	0	0	21	11	52	78
Total	65,785	18	38	22	0	0	21	18	57	78

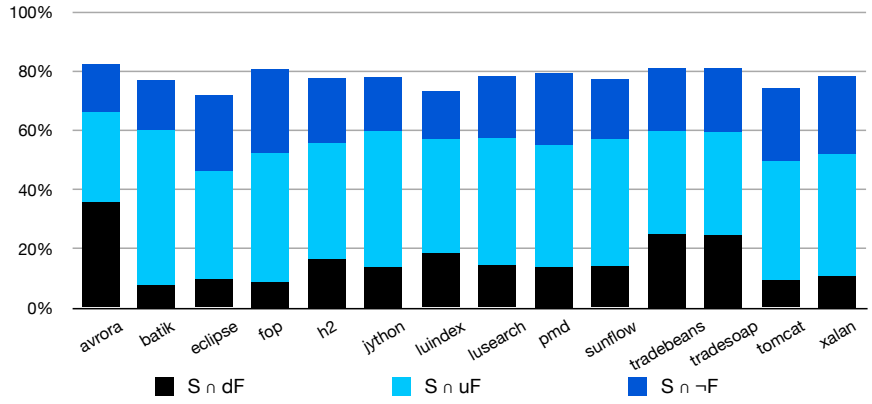

Fig. 4: Stationary vs Final for All Fields

2. **Stationary v Final**. Our results show a significantly higher proportion of stationary fields than reported by Unkel and Lam. Furthermore, we detect far fewer fields which are final but not stationary[4] — indeed, fewer than 0.5% in Figure 4. These results suggests that the final modifier is doing a poor job of supporting common programming practices.

3. **Reference v Primitives**. Our results show that reference fields are more likely than primitive fields to be declared final, undeclared final, and/or stationary. This is consistent with Unkel and Lam, though the differences

[4] Such behaviour is possible in Java if a field read occurs indirectly via dynamic dispatch from a super-constructor [6].

Program	Total	S (%)			¬S (%)			%	%	%
		dF	uF	¬F	dF	uF	¬F	dF	F	S
avrora	880	47	31	12	0	0	9	47	78	90
batik	1,593	9	54	17	0	1	19	9	64	81
eclipse	3,324	15	43	20	0	0	22	15	58	78
fop	1,760	12	45	28	0	0	14	12	58	86
h2	990	21	41	20	0	0	18	21	62	82
jython	1,354	16	49	18	0	0	16	16	66	83
luindex	812	28	42	16	0	0	14	28	70	85
lusearch	598	20	48	18	0	1	14	20	69	86
pmd	1,048	18	45	22	0	1	15	18	63	85
sunflow	831	18	47	20	0	0	15	18	66	85
tradebeans	9,451	33	34	19	0	0	14	33	67	86
tradesoap	9,619	33	34	19	0	0	14	33	67	86
tomcat	3,393	12	43	26	0	0	18	12	55	82
xalan	1,095	14	46	24	0	1	16	14	60	83
Total	36,748	25	39	20	0	0	16	25	64	84

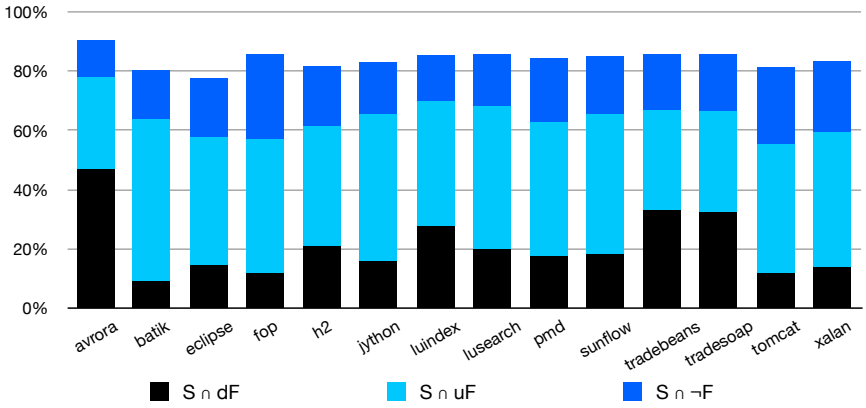

Fig. 5: Stationary versus Final for fields with reference type.

between groups is more modest in our results. This suggests a distinct difference in the way programmers treat reference and primitive fields.

4.5 Threats to Validity

Any experiment of this nature has limitations with respect to the scope of the experiment itself. We now identify the main limitations:

– **Benchmark Inputs**. As discussed in Section 4.1, each of our benchmarks was profiled using the workflow provided by Dacapo. This ensures that our results are reproducible, but effectively constitutes running the benchmark using a single set of inputs. Clearly, we cannot generalise program behaviour from one set of inputs to all possible inputs, and it is possible that the Dacapo inputs are not representative of the benchmark program's general behaviour. In particular, fields identified as undeclared final or stationary may receive

Program	Total	S (%)			¬S (%)			%	%	%
		dF	uF	¬F	dF	uF	¬F	dF	F	S
avrora	822	23	30	20	0	0	26	23	54	74
batik	1,679	6	50	17	0	0	26	6	56	74
eclipse	3,455	5	30	31	0	0	34	5	35	66
fop	1,630	6	42	28	0	0	25	6	47	75
h2	1,024	12	38	23	0	0	26	12	50	73
jython	1,147	11	41	19	0	0	28	11	53	72
luindex	862	10	35	17	0	0	38	10	45	62
lusearch	705	10	38	24	0	0	28	10	48	72
pmd	914	9	36	27	0	0	27	9	46	73
sunflow	1000	11	40	20	0	0	29	11	50	70
tradebeans	5,953	12	36	25	0	0	26	12	49	73
tradesoap	5,992	12	36	25	0	0	27	12	48	73
tomcat	2,886	6	37	22	0	0	34	6	43	65
xalan	968	8	36	30	0	0	27	8	43	73
Total	29,037	10	37	25	0	0	29	10	47	71

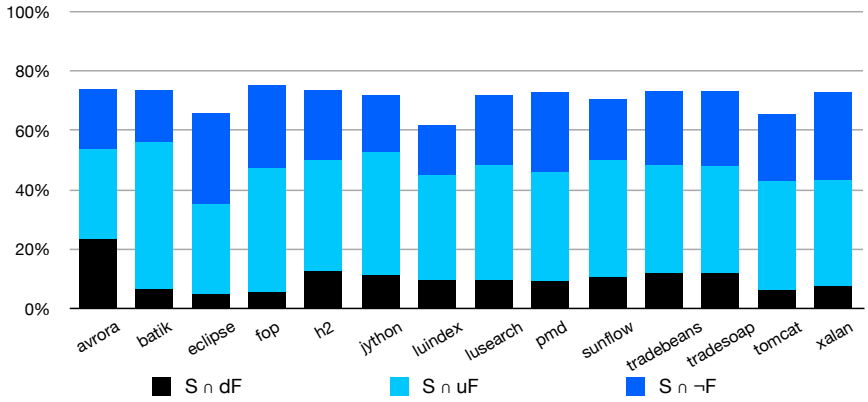

Fig. 6: Stationary versus Final for fields with primitive type.

different classifications for different inputs. This contrasts with the work of Unkel and Lam, whose static analysis was a conservative approximation of all possible program behaviours.

– **Benchmark Scope**. The Dacapo benchmark suit is well-known and widely used for experiments such as this. However, it remains unclear how representative Dacapo is of the general population consisting of all programs. Indeed, there is work which suggests Dacapo programs do have observably different behaviour from other benchmark suites [15].

Despite these limitations, we believe our work compliments that of Unkel and Lam. Being a conservative static analysis, their results necessarily *under approximate* the true number of stationary fields. In contrast, being a runtime analysis

our results necessarily *over approximate* the true number of stationary fields. This provides insight into how conservative the results of Unkel and Lam were.

5 Related Work

Various OO languages have support for immutability via, for example, *final* or *const* fields. CLU [16] also supports immutable versions of primitive data structures — although clusters (classes) are always mutable. A similar design has been adopted in Scala, where the library provides mutable and immutable versions of most collections [17].

As discussed already, Unkel and Lam also examined stationary fields [5]. Unlike us, they employed a static analysis which is necessarily conservative. For a corpus of 26 Java applications, they found that 40-60% of Java fields were stationary which is a similar, but consistently lower, figure than we have found. Given that their result is an under-approximation and ours an over-approximation, it seems reasonable to conclude that the true figure lies somewhere inbetween. Earlier, Porat et al. [18] conducted a similar analysis looking for "deeply immutable" fields (where neither the field itself nor any object reachable from that field is modified after the object's constructor completes) and found that around 60% of `static` fields were immutable. These results compare with our (dynamic) profile finding that a large fraction of Java objects are immutable after full construction. Previously, we examined object behaviours and found significant differences depending on whether or not they entered a Java collection [19]. This is particularly relevant for collections such as e.g. HashSet and HashMap which restrict how contained objects may be modified.

Pechtchanski and Sarkar present an interesting study of field immutability [20]. Their work includes a framework for specifying and verifying both shallow and deep field immutability, as well as a runtime study that found that at least 61% of field accesses were immutable, a similar property to stationary fields. Their analysis computed exhaustive lists of field and array read and write operations using a modified Jikes JVM, but their analysis was limited to much smaller programs than the Dacapo suite. Nevertheless, they find similar results to ours and additionally use those results for performance optimisations, yielding 5-10% speedups for some benchmarks.

Several works have looked at permitting type-safe late initialisation of objects in a programming language. Summers and Müller presented a lightweight system for type checking delayed object initialiation which is sufficiently expressive to handle cyclic initialisation [6]. Fähndrich and Xia's Delayed Types [2] use dynamically nested regions in an ownership-style type system to represent this post-construction initialisation phase, and ensure that programs do not access uninitialised fields. Haack and Poll [1] have shown how these techniques can be applied specifically to immutability, and Leino et al. [3] show how ownership transfer (rather than nesting) can achieve a similar result. Qi and Myers' Masked Types [21] use type-states to address this problem by incorporating a list of uninitialised fields ("masked fields") into object types. Gil and Shragai [22]

address the related problem of ensuring correct initialisation between subclass and superclass constructors within individual objects. Based on our results, we would expect such type systems to be of benefit to real programs.

6 Conclusion

We have reported the results from an experiment examining final and stationary fields across 14 real-world benchmarks. Our work compliments the earlier work of Unkel and Lam which employed static analysis, and supports their general conclusions. However, our findings indicate a larger proportion of stationary fields which, in part at least, stems from the differences between our approaches.

Like Unkel and Lam, we conclude that final fields annotations are used far less often than they could be, while a *stationary* annotation could be used even more. The extremely high number of stationary fields that we found (around 80%) suggests that language authors should make fields stationary by default, while VM authors should optimise for immutability. These results also support the use of type systems for immutability, e.g. Masked Types [21] could be used to track fields requiring additional initialisation.

Finally, there are many additional studies that are motivated from these results. For example, it would be interesting to examine whether protection modifiers (e.g. public, protected, private) had any bearing on the likelihood of a field being declared or undeclared final. It would also be interesting to extend our analysis to detect deep stationary behaviour, similar to the smaller analysis of Pechtchanski and Sarkar [20].

References

1. Haack, C., Poll, E.: Type-based object immutability with flexible initialization. Technical Report ICIS-R09001, Radboud University Nijmegen (January 2009)
2. Fähndrich, M., Xia, S.: Establishing object invariants with delayed types. In: OOPSLA. (2007) 337–350
3. Leino, K.R.M., Müller, P., Wallenburg, A.: Flexible immutability with frozen objects. In: VSTTE. (2008) 192–208
4. Bloch, J.: Effective Java. Prentice Hall PTR (2008)
5. Unkel, C., Lam, M.S.: Automatic inference of stationary fields: a generalization of Java's final fields. In: POPL. (2008) 183–195
6. Summers, A.J., Müller, P.: Freedom before commitment: a lightweight type system for object initialisation. In: OOPSLA, ACM (2011) 1013–1032
7. : Jdk 6 java virtual machine tool interface (JVMTI) (2008)
8. Bruneton, E.: Asm 3.0 a java bytecode engineering library. URL: http://download. forge. objectweb. org/asm/asmguide. pdf (2007)
9. Nelson, S.: Measuring Equality and Immutability in Object-Oriented Programs. PhD thesis, School of Engineering and Computer Science, Victoria University of Wellington, NZ (2012, Submitted)
10. Agesen, O., Garthwaite, A.: Efficient object sampling via weak references. In: Proc. ISMM. (2000) 121–126

11. Pearce, D.J., Webster, M., Berry, R., Kelly, P.H.J.: Profiling with AspectJ. Software: Pracice and Experience **37**(7) (2007) 747–777
12. Goldberg, A., Havelund, K.: Instrumentation of java bytecode for runtime analysis. In: FTfJP. (2003)
13. Xu, G.H., Rountev, A.: Precise memory leak detection for java software using container profiling. In: ICSE, ACM (2008) 151–160
14. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The dacapo benchmarks: java benchmarking development and analysis. (2006) 169–190
15. Mitchell, N.: The runtime structure of object ownership. In: Proc. ECOOP. Volume 4067 of Lecture Notes in Computer Science., Springer (2006) 74–98
16. Liskov, B., Guttag, J.V.: Abstraction and Specification in Program Development. MIT Press/McGraw-Hill (1986)
17. Odersky, M.: Programming in Scala. Artima, Inc (2008)
18. Porat, S., Biberstein, M., Koved, L., Mendelson, B.: Automatic detection of immutable fields in Java. In: Proc. CASCON. (1990)
19. Nelson, S., Pearce, D.J., Noble, J.: Understanding the impact of collection contracts on design. In: TOOLS Europe. (2010)
20. Pechtchanski, I., Sarkar, V.: Immutability specification and its applications. Concurrency and Computation: Practice and Experience (2005) 639–662
21. Qi, X., Myers, A.C.: Masked types for sound object initialization. In: POPL. (2009) 53–65
22. Gil, J., Shragai, T.: Are we ready for a safer construction environment? In: ECOOP. (2009)