Performance Modeling of Maximal Sharing

Michael J. Steindorfer
Centrum Wiskunde & Informatica
Science Park 123
1098 XG Amsterdam, The Netherlands
michael.steindorfer@cwi.nl

Jurgen J. Vinju
Centrum Wiskunde & Informatica
Science Park 123
1098 XG Amsterdam, The Netherlands
jurgen.vinju@cwi.nl

ABSTRACT

It is noticeably hard to predict the effect of optimization strategies in Java without implementing them. "Maximal sharing" (a.k.a. "hash-consing") is one of these strategies that may have great benefit in terms of time and space, or may have detrimental overhead. It all depends on the redundancy of data and the use of equality.

We used a combination of new techniques to predict the impact of maximal sharing on existing code: Object Redundancy Profiling (ORP) to model the effect on memory when sharing all immutable objects, and Equals-Call Profiling (ECP) to reason about how removing redundancy impacts runtime performance. With comparatively low effort, using the MAximal SHaring Oracle (MASHO), a prototype profiler based on ORP and ECP, we can uncover optimization opportunities that otherwise would remain hidden.

This is an experience report on applying MASHO to real and complex case: we conclude that ORP and ECP combined can accurately predict gains and losses of maximal sharing, and also that (by isolating variables) a cheap predictive model can sometimes provide more accurate information than an expensive experiment can.

Keywords

Performance modeling; maximal sharing; hash-consing; profiling; memory; optimization; Java Virtual Machine.

1. INTRODUCTION

This paper is about performance modeling of Java libraries. "Premature optimization is the root of all evil", says Donald Knuth [11]. The reason is that optimization strategies are prone to make code more complex and perhaps for no good reason because they may backfire unexpectedly.

Our question is: how can we know, a priori, that a particular optimization strategy will pay off? For most optimizations there is only one way to find out: implement an optimization and compare runtime characteristics against an unoptimized version. In reality it will often take multiple rounds of profiling and tuning before the desired effect and the promised benefit of an optimization is attained. In this paper we present the MAximal SHaring Oracle (MASHO): a prototype profiling tool that predicts the effect of the maximal sharing optimization a priori, avoiding costly and risky engineering. We report on the experience of testing MASHO and trying it out on a real and complex case.

The "maximal sharing" optimization tactic, dubbed "hash-consing" [8], entails that selected objects that are equal are not present in memory more than once at a given point in time. To make this happen a global cache is used to administrate the current universe of live objects, against which every new object is tested. There are two main expected benefits of maximal sharing: avoiding all redundancy by eliminating clones in memory, and the ability to use constant time reference comparisons instead of deep `equals` checks that are in $O(size\ of\ object\ graph)$. This is because maximal sharing enforces the following invariant among selected objects: $\forall$ *objects* $x, y : x.equals(y) \Leftrightarrow x == y$, which allows any call to `equals` on shared objects to be replaced with a reference comparison. The expected overhead is the maintenance of a global cache, and for each object allocation, extra calls to the `hashcode` and `equals` methods.

Figure 1 illustrates the effect of maximal sharing on an object that is "embarrassingly redundant": a reduction from exponential to linear size (in the depth of the tree). In contrast, a tree with the same structure but all unique integer values in its leaf nodes would have no sharing potential.

Maximal sharing is associated with immutable data structures [17], since it requires objects to not change after allocation. It is applied in the context of language runtimes of functional languages [27, 10], proof assistants [5], and algebraic specification formalisms [23, 4, 6, 15, 2], compilers [26], or libraries that supply similar functionality. Especially when many incremental updates are expected during computation

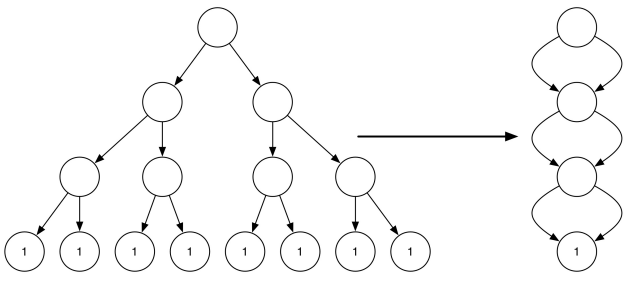


Figure 1: Good conditions for sharing: redundant objects.

ICPE'16, March 12-18, 2016, Delft, Netherlands

DOI: http://dx.doi.org/10.1145/2851553.2851566

(i.e., creating redundancy over time) we may expect big benefits. For example, implementations of term rewriting (reducing trees), type constraint solving (minimizing sets of solutions) and solving data flow equations (incrementally adding/removing graph edges) share these characteristics.

On the one hand, in the case of high performance implementations of term rewriting engine libraries, maximal sharing has proven to be a very successful strategy: *"It turns out the increased term construction time is more than compensated for by fast equality checking and less use of space (and hence time)"* [23]. In real cases memory savings between 52.20%–98.50% of term representations were observed [24]. On the other hand, maximal sharing can have a negative net effect on memory consumption in absence of enough redundancy, due to the overhead of the global cache.

The audience for the maximal sharing technique is library developers rather than application developers. Considering the effort associated with optimizing for maximal sharing, an often reused library is expected to have larger return on investment than a single application.

With the advance of immutable objects and functional language constructs in object-oriented languages —like Java 8 or Scala, and functional languages running on the Java Virtual Machine (JVM) like Clojure— it is now relevant to investigate if and how we can use maximal sharing to our benefit in JVM library implementations. Immutability may be a too strong requirement for any Java library in general, but if immutability comes naturally for different reasons, then the maximal sharing strategy is an important one to consider.

Potential adopters of maximal sharing suffer from a common problem: converting a library to use maximal sharing is hard and costly [22, 24, 25]: it is a cross-cutting design decision with difficult to tune implementation details. To illustrate one of many pitfalls, let us take a Java library for graph processing as example. It makes use of standard library classes for integers and sets. The hashcodes of empty sets are 0 in Java and for singleton sets the hashcodes are equal to the hashcode of the elements, because `hashCode()` of a `java.util.Set` is defined to be the sum of the hash codes of the elements in the set. Inserting such similar values in a global cache for sharing would trigger unexpected hash collisions. The success of maximal sharing depends on one hand on a broad spectrum of properties of a library, like its Application Program Interface (API) design, quality of hash codes, co-optimization of shared data structures, and on the other hand on runtime characteristics like data redundancy and the ratio between object allocations and equality checks of shared objects. Naive implementations of maximal sharing —that do not take these issues into account— are likely to slow down programs and increase their memory footprint.

1.1 Contributions and Outline

This paper does not contain an evaluation of the maximal sharing optimization technique; it does contain an evaluation of the accuracy and usefulness of a modeling and simulating technique for maximal sharing. The contribution of this paper is firstly the design of MASHO (Section 2), which includes:

- Object Redundancy Profiling (ORP): measuring the lifetime of redundant objects during a program execution, optimized to benefit from immutable data and to include the notion of data abstraction to accurately model the possible effects of maximal sharing;
- Equals-Call Profiling (ECP): capturing the recursive call-graph shapes of calls to `equals`, including a partial alias analysis;
- Maximal Sharing Model (MSM): a lightweight predictive model that uses ORP and ECP profiles to predict the behavior of a program after the application of maximal sharing.

Secondly we contribute an experience report on the use MASHO for modelling the runtime environment of a programming language. From this we learned that:

- it predicts the impact of maximal sharing on memory usage and the use of equality very accurately and so it removes the need for direct experimentation with maximal sharing, producing equivalent information for making go/no-go decisions with a mean slowdown of 7x for ORP and ECP;
- it isolates the effects of introducing maximal sharing from the effects of JVM configuration (e.g., memory bounds, garbage collector heuristics) and accidental hash collisions that would occur due to a global cache.
- for the validating experiment, a set of realistic demonstrations, implementing maximal sharing will produce good memory savings but will not lead to performance speed-up without first applying major changes to the semantics of the library. We can decide a "no-go".

In general, this experience shows how cheap predictive performance modelling can produce more actionable information than an expensive real-world experiment can since it can soundly factor our confounding factors like the Java garbage collector and reason about otherwise infeasible design alternatives. Related work is further discussed in Section 4, before we summarize in Section 5.

2. DESIGN OF THE MODELING TOOL

In the following we describe the design decisions and most important implementation details of MASHO in this order: how it is used by a library developer, its architecture and implementation choices, what its preconditions are, and then how ORP, ECP and MSM work together to predict the effect of introducing maximal sharing. We identify possible sources of inaccuracy throughout the text and evaluate these in Section 3.

2.1 Library Developer Perspective

The user first configures MASHO with a list of interesting classes or interfaces which might hypothetically benefit from maximal sharing. MASHO then instruments the library (which does not implement maximal sharing). Next, the user runs programs that use the library, while the instrumentation logs information to be analyzed. After this, MASHO analyzes the logs producing charts and tables explaining the likely effect of maximal sharing on the library in the context of the executed programs. The user interprets the charts to decide go/no-go on investing in maximal sharing, or continues tweaking the experiment's setup or the test runs.

2.2 Instrumenting a Program for Profiling

Figure 2 depicts the architecture of MASHO. A client library is instrumented using both AspectJ and the Java

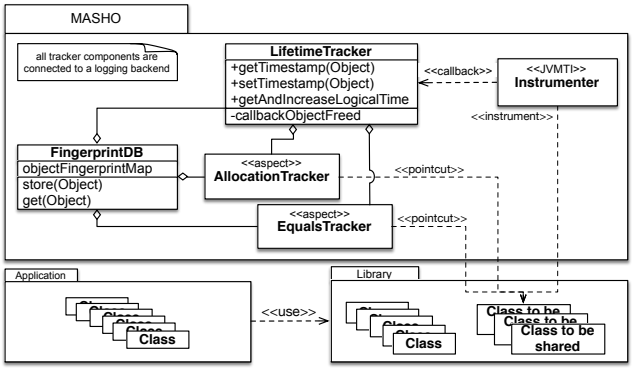


Figure 2: Class diagram and Aspect-Oriented Programming profile depicting MASHO's architecture.

Virtual Machine Tool Interface (JVMTI) for gathering the following events: object allocations, object garbage collections, and calls to `equals`. An AspectJ pointcut selects all constructor call-sites of to-be-shared classes. In an advice, which gets called whenever one of these constructors executes, MASHO performs at run-time ORP with fingerprinting (Section 2.4) and an alias-aware object graph size measurement (Section 2.5). Similarly, we use pointcuts for ECP to record the call-graph shape of `equals`-calls (Section 2.6). Bytecode Instrumentation (BCI) is used to track object allocations that are otherwise intangible for AspectJ, for example object construction via reflection. For lifetime tracking, we tag each newly allocated object with the aid of the JVMTI to get notified about an object's garbage collection.

2.3 The Precondition: Weak Immutability

Maximal sharing introduces a global cache for all designated objects and uses an object factory for creating new instances. Instead of `new Tuple(a, b)`, one would call a factory method like so: `factory.tuple(a, b)`. Whenever the factory encounters that an equal object already exists in the global cache, it returns the cached instance and forgets the temporary object. Otherwise it caches the new instance and returns it. Such a global cache introduces data dependencies between parts of the program which would normally be unrelated to each other. Consequently, maximal sharing does not work for mutable objects because it may break referential integrity: if a shared object would change, this would become observable in otherwise logically independent parts of the program.

One way to avoid breaking programs in the presence of maximal sharing is requiring full immutability of shared objects, but such a strong condition is not necessary. Therefore we define *weak immutability*, a sufficient condition under which maximal sharing can work, as follows: for any object o and its updated future value o' it holds that $o.\texttt{equals}(o')$, while observing that not necessarily all fields have to contribute to its `equals` method. Based on weak immutability, object identity can be defined by the transitive closure of immutable attributes of an object, also known as structural equality [1]. Similarly it follows that all object graphs generated from these classes, if we follow only references to fields that are used by the `equals` methods, are Directed Acyclic Graphs (DAGs).

Competitive tools [14] solely reason on the granularity of "physical" object-graph equality, while logical object equality may need some form of abstraction. For example, in case of unordered collection data structures such as hashtables, and lazily instantiated (caching) fields. We will detail in the next section, how to support those cases for better coverage.

2.4 Object Redundancy Profiling

The ORP part of MASHO takes advantage of the weak immutability of the selected classes and the fact that we are guaranteed to analyze data that could be represented as a DAG. Namely, we compute a fingerprint for each object, representing the structural identity of its value, using a bottom-up analysis of the object DAG. Fingerprinting allows us to avoid fully serializing heap objects or logging all changes of the heap to disk [20, 14]. Instead we serialize only the fingerprints that are expected to be a lot smaller in size.

The fingerprint function f, a cryptographic 256-bit SHA-2 hash function in our case, has similar goals as the normal standard 32-bit integer `hashcode` method but necessarily it has a much higher resolution to better represent object identity. For an optimal f (i.e., perfect hashing) it can be said that for any two objects o_1, o_2 it holds that $o_1.f() = o_2.f() \Leftrightarrow o_1.\texttt{equals}(o_2)$. The inevitable non-optimality of a cryptographic f may introduce inaccuracy in MASHO's profiles, while at the same time making the analysis feasible because we avoid a full serialization of every object.

Weakly-immutable object DAGs can only be created bottom-up, so MASHO computes a fingerprint at each allocation of a to-be-shared object. We use a fingerprint cache to efficiently refer to the fingerprints of already known objects. Therefore, fingerprinting a new composite object is always $O(\textit{shallow object size})$. We distinguish the following cases:

Leaf Objects: are objects that have no children in the immutable DAG. We serialize leaf objects and fingerprint them by applying f on the resulting byte-arrays.

Ordered Composite Objects: are objects that contain an ordered sequence of references to other shared objects. We first lookup and concatenate the fingerprints of all referenced shared objects. Then, we compute f over the concatenated hashes.

Unordered Composite Objects: are objects that contain an unordered sequence of references to other shared objects. We first lookup and concatenate the fingerprints of all referenced shared objects. Then, we reduce the set of fingerprints to a single fingerprint with the bitwise XOR operator. This commutative fingerprint computation is stable under arbitrary orderings.

In the case of unordered composite objects, arbitrary orderings of arrays, containing the same values, are to be expected for example in array-based implementations of hash-maps and hash-sets. We abstract from these arbitrary orderings in order to predict more opportunities for maximal sharing, as well as abstracting away from differences that are due to hash collision resolution tactics.

2.5 Object Graph Size Calculation

Modeling memory savings requires reasoning over which references already point to the same objects and which do not. Such aliasing is likely present in any Java application. MASHO computes the memory footprint of a to-be-shared

object efficiently at object allocation time, which is sufficient only due to weak immutability. It uses Java reflection to collect fields and compute the size of all referenced objects and contained primitive values. This traversal skips nested to-be-shared objects to solely measure the overhead incurred by redundant objects. Aliases of not to-be-shared objects are detected by maintaining a lookup table that maps object identities to their memory footprints. If an object reference is already present in the table, then we have detected an alias and should not count the same object again, but simply add the size of the 32 or 64-bit reference. Note that this alias analysis is *incomplete* by design due to efficiency considerations. We distinguish two cases: *visible* and *invisible* aliases. While the former is traced accurately, the latter may introduce inaccuracy because we only partly track the heap.

Visible aliases.

I.e., references that are reachable from a to-be-shared object. For example, consider two different Java fragments which construct a tuple and its content, an atom. Both classes are to be maximally shared:

- Tuple elements are aliases:
  `Atom a = new Atom("S"); new Tuple(a, a);`
- Tuple elements are unique:
  `new Tuple(new Atom("S"); new Atom("S"));`

ORP should predict savings for the latter because it uses duplicates, whereas the former already shares the atom.

Invisible aliases.

I.e., references to library objects that are outside the interfaces that the library developer chose to track. Consider the following Java fragment: `Atom atom(String s) { return new Atom(s); }`. `Atom` is to be shared, whereas `String` is not. We attribute the size of `s` to the size of the first tracked object that references `s`. Note that `s` might be referenced by any other object: either from an object to be shared, or from an object that is not meant to be shared. The accuracy of MASHO is influenced by this effect (addressed by one of our evaluation questions in Section 3).

2.6 Equals-Call Profiling

The goal of ECP is to record the shape of (recursive) calls to `equals` on to-be-shared objects. Tracking the calls to `equals` requires detailed consideration to be applicable to maximal sharing. After objects are maximally shared, all calls to `equals` can be replaced by reference comparisons, but also already existing aliases have to be taken into account to not over-approximate the potential benefits of maximal sharing.

In Java it is common that `equals` implementations first check for reference equality on both arguments to short-circuit the recursion in case of aliased arguments. Using AspectJ we cannot easily capture the use of == or !=, but we can measure the difference between root calls to `equals` and recursively nested calls to `equals`. By root calls we mean invocations of `equals` that are not nested in another `equals`-call. In case `equals` implementations do not return on aliases directly, MASHO pinpoints these optimization opportunities by warning the user about them.

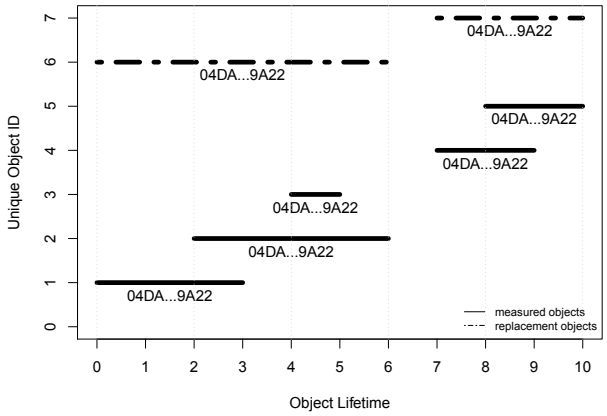


Figure 3: Overlapping lifetimes for objects with identical fingerprints. Two families of redundant alive objects are visible (solid lines) and also their possible replacement maximally shared alternatives (dashed lines).

2.7 Creating a Maximal Sharing Model

By combining the results from redundancy and `equals`-call profiling, we are able to hypothetically model the impact of maximal sharing, including changes to the heap structure, overhead introduced by a global cache, and substitutions of (recursive) `equals`-calls by reference comparisons.

Modeling memory usage.

MASHO analyzes the profiled data as follows. It reasons about redundancy that is present at each point in time. Time is measured by allocation and deallocation event timestamps. Figure 3 illustrates several objects that map to the same fingerprint *04DA...9A22*. The objects with identifiers 1, 2, and 3 consecutively overlap, as well as objects 4 and 5. We call a sequence of overlapping lifetimes an "object family". In an optimal situation each family would reduce to a single object, with an extended lifetime (see dashed lines in Figure 3).

First, we replay the trace to compute the current memory usage of the profiled program for each point in time. We start from a set of object lifetime triples, $allocationTime \times deallocationTime \times uniqueSize$. We compute two lists: one with objects sorted by allocation time, and another with objects sorted by deallocation time. Then we traverse the two sorted lists and compute their running sums. At each timestamp the difference between the running sums denotes the current memory usage.

Second, we compute an estimated memory profile of the same program run as-if objects would be maximally shared. Again, we sort the aforementioned object lifetime triples on timestamps but now we also group them by their fingerprints. This artificially removes duplicate objects and extends the lifetimes of the remaining objects. The final memory usage at each point in time is computed exactly as before, but on this filtered and re-ordered set of lifetime triples. This computation predicts what memory is theoretically minimally necessary to store the observed objects. In practice of course more memory will be used because objects are now even more unlikely to be garbage collected immediately after they become unreferenced. This effect will be observable in the evaluation later.

Modeling the global cache memory overhead.

MASHO assumes a fixed bytes-per-record overhead per object reference stored in the global cache that is to be introduced. Predicting the overhead is a matter of multiplying a constant —currently 42— by the number of unique objects at any point in time. To choose its default value, we analyzed the memory overhead of an object repository that is implementable with the standard Java collections API (i.e., `WeakHashMap` with `WeakReference`s as values) and an existing and thoroughly optimized implementation from the ATerm Java library [22]. ATerm's global cache imposes a 42 bytes-per-record memory overhead, while a standard `WeakHashMap` implementation requires 79 bytes-per-record.

Modeling the global cache runtime overhead.

The expected number of newly introduced calls to the `equals` method is exactly equal to the number of redundant object allocations. The new implementation of `equals` will not have to be recursive anymore under the assumption of maximal sharing. Note that these predictions are under the assumption of optimal hash code implementations and a collision free global cache implementation.

We may also predict the maximal number of new executions of == by counting at each call to `equals` the number of immutable fields, i.e., the arity, of the object. Note that this arity depends on the definition of the original `equals` method. This is the number of fields that contribute to its implementation.

Suppose an implementation of a to-be-shared class uses arrays for storing nested objects. In this case the arity of the object is open and `equals` is in principle in $O(arity)$ even after introducing maximal sharing. The higher this arity, the lower the benefit of maximal sharing will be. This is why MASHO reports also the expected number of newly introduced reference comparisons to the library engineer.

3. EVALUATION

Does MASHO allow library engineers to model and simulate what they might get out of maximal sharing without actually implementing and tuning it? Our evaluation questions are:

Q-Accurate: does MASHO predict memory gains and the effect on `equals`-calls after maximal sharing accurately?

Q-Actionable: does MASHO give a library engineer enough information to decide upon further time investments in the maximal sharing strategy?

First, we set up a controlled experiment (Section 3.4) where we can theoretically explain the shape of the input and the shape of the resulting statistics. This is to test whether the experimental setup works reliably and accurately.

To answer **Q-Accurate** we will then compare MASHO's models to profiles obtained from realistic cases that implement maximal sharing (Section 3.5). The hypothesis is that the memory and `equals`-calls models are very accurate, i.e., within a 1% margin of error. The hypothesis assumes that the introduction of the global cache —that holds weak references to shared objects— does not (or only marginally) change the overlapping lifetimes of the object families; we will report whether or assumption holds. The output of these realistic experiments is discussed in detail as a prerequisite to assess **Q-Actionable** qualitatively.

3.1 Experience: the Program Data Base Case

We report on our experience testing and evaluating MASHO on two open-source projects: the Program Data Base (PDB),[1] a library for representing immutable facts about programs, and Rascal [10], a Domain-Specific Language (DSL) for meta-programming. Both projects are actively developed and maintained since 2008. Rascal has 110K Source Lines of Code (SLOC) and PDB 23K SLOC. PDB is the run-time system of Rascal. All values produced and consumed by Rascal programs are instances of PDB library classes. This ranges from primitive data types, to collections, and more complex compositional data structures like arbitrary Algebraic Data Types (ADTs). PDB's basic primitive is the `IValue` interface, which every weakly-immutable data type adheres to. Thus, analyzing the usage of `IValue` in Rascal programs is comparable to analyzing how `Object` is used in Java. For the experiments below, we configured MASHO to share all objects of library classes that implement `IValue`.

The PDB classes support two forms of equality. This is common for weakly-immutable libraries with structural equality (cf. Clojure). One is implemented by `equals` satisfying weak immutability by implementing strict structural equality. The other is called `isEqual` and ignores so called "annotations". Annotations are extensions to a data value which should not break the semantics of existing code that does not know about them. The `isEqual` method does satisfy weak immutability, but if maximal sharing would be applied based on the `isEqual` semantics instead of on the `equals` semantics it could break client code: annotations would quasi-randomly disappear due to accidental order of storing (un)annotated values in the global cache. With maximal sharing in mind, annotations should not be ignored for equality.

To be able to analyse `isEqual` as well, we configured ECP to track `isEqual` calls like it tracks `equals`-calls. Note that `equals` and `isEqual` do not call each other. Furthermore, we inserted an additional rule that checks at run-time if both arguments to `isEqual` map to the same fingerprint, by performing fingerprint cache lookups. If yes, we imply that they are strictly equal to each other and do not contain annotations in their object graphs. As a consequence we model such an `isEqual` call as a reference equality, because both arguments will eventually become aliases under maximal sharing. Otherwise, if fingerprints do not match, we continue to recursively descent into the `isEqual` call.

To conclude, PDB represents a well-suited but challenging case for maximal sharing: it is not set up for tautological conclusions of our evaluation of MASHO.

3.2 A Minimal Maximal Sharing Realization

For answering the **Q-Accurate** question we should verify the predictive model of MASHO against actual data from a real maximal sharing implementation. To gather memory data and to profile `equals`-calls, a fully optimized implementation is not necessary, and also absolute numbers about runtime performance are not comparable anyway due to interference of the profiler and JVM configuration. Therefore, we should abstract from absolute runtime numbers and instead evaluate the absolute reductions/increases in terms of `equals`-calls.

Figure 4 shows a class diagram of how we used AspectJ to obtain a maximally shared version of PDB. Our global cache

[1] https://github.com/cwi-swat/pdb.values

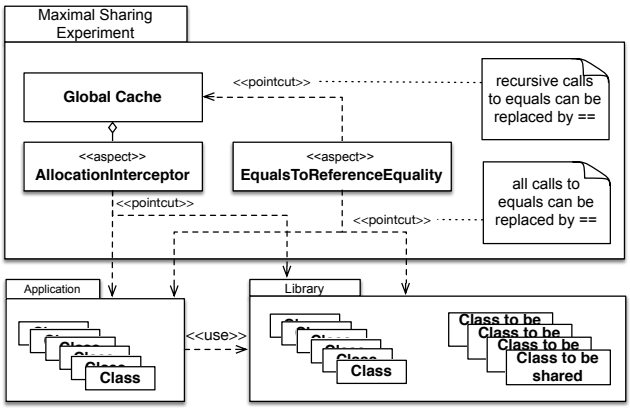


Figure 4: Using AspectJ to experiment with maximal sharing.

is implemented using a `WeakHashMap` with `WeakReference` values. We use a pointcut with *around*-advises to intercept and instrument object allocation call-sites (both in library and in application code), substituting new objects with references from the global cache, if present. We also replace all calls to `equals` outside of the central repository by reference equalities, as well as all recursive calls to `equals` called by the central repository. The recursive calls can be replaced because nested values have already been maximally shared.

3.3 Shared versus Non-Shared Measurement

We reuse MASHO's measurement facilities to measure both a shared and a non-shared version of each experimental run. In the shared version we reuse a strict subset of MASHO's measurement facilities, namely for ECP, object size calculation, and timestamping of allocation and deallocation events. The latter are services of the JVMTI. Reuse of MASHO's measurement facilities entails a threat-to-validity that is mitigated by the controlled experiment, where we can theoretically explain the shape of the data and the expected outcome. We do completely turn off ORP profiling for these experiments to avoid interference. In the presence of the maximal sharing aspect we can observe real global cache behavior and identify redundant objects based on cache hit counts.

Our first naive memory measurement method (*method 1*) is to aggregate and compare the mean memory usage from a shared library experiment against the model that is calculated from the non-shared profiles. If the difference is small, then MASHO predicts accurately. Otherwise, either the evaluation method is flawed, fingerprinting has too many collisions, or MASHO misses an important aspect in modeling maximal sharing. The hypothesis is that based on this analysis we will see only minor differences because MASHO is expected to produce an accurate model.

The previous method is naive, because we know that the Garbage Collector (GC) will influence the mean memory usage as often and perhaps as much as the optimization strategy does. It is a confounding factor. We should expect sawtooth patterns in MASHO's memory profiles caused by short-living temporary objects that could all be discarded immediately after allocation —in case of a hit in the global cache— but instead will remain in memory until the GC starts to operate. So, from the comparison of mean memory usage we should hypothesize significant inaccuracy in predicting memory usage.

To mitigate the confounding factor introduced by the delays in garbage collection we may set the heap size setting of the JVM to a benchmark specific global minimum.[2] This would trigger the GC more often and force it to collect temporary objects. The mean memory usage then starts approaching the global minimum in memory usage. While identifying globally minimal heap sizes per benchmark could be automated with bisection search, we argue it is not precise enough for our memory measurements. Therefore we also set up a second method of comparison (*method 2*). This method is similar to the previous, but additionally we tag all short-living temporary objects —by measuring whether they cause cache hits— and subtract their sizes from the actual memory usage. The effect is that we filter noise introduced by the GC. Instead of only considering one global minimum, we now reason over a series of local minima in time. If the difference in memory usage between this minimal size and the predicted size is still large, then MASHO is inaccurate, as caused by fingerprint collisions or an unsound modeling. Otherwise it is accurate. For the sake of transparency we will discuss both the results of the naive method and the mitigated method of comparison.

Setup of JVM *Parameters.*

We use AspectJ 1.7.3 a 64-bit JVM from Oracle, Java version 1.7.0_51, running on an Intel Core i7 3720QM CPU under Mac OS X.[3] We configured the JVM with the following settings additional to the `-server` flag: with `-Xms4G -Xmx4G` we set the heap to a fixed size of 4GB and prohibit resizing; `-XX:+DisableExplicitGC` deactivates manual invocation of the GC; `-XX:+UseParallelOldGC` uses a parallel collector for new and old generations.

3.4 Controlled Experiment

Here we test-drive our evaluation method. We use two scenarios that are based on the introductory example from Figure 1: with the PDB library we first build binary trees of depth d where all leaf nodes are equal with respect to each other, and second binary trees of depth d where all leafs nodes are different from each other. We hypothesize the results of the experiment and observe whether or not these numbers are produced. This check of both an optimal case and a worst case scenario for maximal sharing would reveal obvious problems with our measurements.

3.4.1 Expectations

While profiling we expect from our setup that no object is garbage collected until the program ends and that both trees consume the same heap size. Zero redundancy should be measured in the redundancy-free case, and for depth d in the redundant case $2^{d+1} - d$ duplicates. When running PDB with the maximal sharing aspect, memory savings should be visible for the redundant case, and growing with increasing depth. The controlled experiment only allocates objects,

[2]Minimum heap size is a function of time: each program state has its own minimum. With global minimum we refer to the maximum of all minima, i.e., the lower memory bound that is sufficient to run the program.

[3]At the time of performing our experiments, the latest stable AspectJ Development Tools (AJDT) version included AspectJ 1.7.3, which only supported Java Development Kit (JDK) 7.

but does not invoke `equals`. However the maximal sharing introduces `equals`-calls by performing global cache lookups. We expect one `equals`-call per cache hit, and furthermore for each binary tree node two reference comparisons, one for the left and one for the right subtree.

3.4.2 Results

Figure 5 shows the results of profiling the creation of trees with depths from 1 to 20. The plots use logarithmic scales.

Redundant Trees.

Figure 5a focuses on redundant trees. The x-axis highlights the profiled allocation count at each depth d. Surprisingly, the measurement at $d = 0$ exhibits four allocations instead of the one expected: Manual inspection revealed that PDB's integer implementation contains an `integer(1)` constant, and further, the two boolean constants `TRUE` and `FALSE` were pre-initialized by the library.

The *profile* line shows memory usage obtained by the profiles, while the *maximum sharing model* line illustrates the predicted minimal heap usage under maximal sharing. At low object counts ($d <= 2$) the *maximum sharing model* signals a higher memory usage with maximal sharing than without, However, at $d = 5$ the measurements break even, denoting a saving potential of 66%. The saving potential stabilizes around 100% from $d = 10$.

The *sharing run (with default heap size)* line shows the heap profile with the maximal sharing aspect applied. For $d < 20$ there is no measurable difference from the *profile* line. Only at $d = 20$ with about 2M object allocations, we see a difference because temporary objects are partially collected. Performing another *sharing run (with tight heap size)*, yields results that are clearly different from the original memory profile, yet a significant error remains. The results confirm that the GC largely influences the naive *method 1*; we obtained a mean accuracy of 27%, with a range of 3–93%.

Measurements with *method 2* are not visible in the graph, because the data aligns exactly with our *maximum sharing model*. It performed with 100% accuracy at experiments with an allocation count bigger than 66; at smaller counts the three unexpected allocations reduce accuracy marginally.

The measured global cache hits (that are not listed here for brevity) are exactly off by one due to the `integer(1)` constant. Measured `equals`-calls that are caused by the global cache match exactly with the number of cache hits, as expected. Estimated reference equalities are also accurate: each cache hit of a tree node object yields two reference comparisons, one for each sub-node.

Redundancy-free Trees.

Figure 5b shows the results for trees with no shareable data. The *maximum sharing model* and *sharing run (with default heap size)* correlate. The plot illustrates the overhead of the global cache that grows linearly with each unique object. The only unexpected observation is one additional cache hit, caused by the previously mentioned `integer(1)` constant.

No hash collisions were recorded due to global cache lookups, with the exception of a single experiment ($d = 20$) that yielded 420 `false` equality comparisons in a cache with 2M cached objects. We list the number of equality checks that yielded `false` rather than full collisions to abstract from global cache implementation details.

3.4.3 Analysis

First, we observed particularities of PDB in terms of pre-allocated constants. Second, even under optimal conditions hash collisions became visible at 2M cached objects. We suspect this becoming a dominant factor in further experiments. This indicates also that the hash code quality should be an engineering priority in case of a "go" decision for maximal sharing for this kind of data.

The naive *method 1* of comparing mean memory is not able to show the effect of maximal sharing due to GC noise. In contrast, our alternative *method 2* shows accurate results that matched our model.

We may confirm **Q-Accurate** for this case: MASHO precisely analyzes potential savings and losses, our second method of memory comparison works, and also `equals`-calls are predicted accurately.

3.5 Realistic Demonstrations

In this section we report on our experience with predicting the effect of maximal sharing in the context of PDB being embedded into Rascal. We will evaluate the following benchmarks:

A: Start the Read–Eval–Print Loop (REPL) of the Rascal language interpreter, and load its prelude module.

B: Start the REPL of the Rascal language interpreter, and generate a parser for an expression grammar.

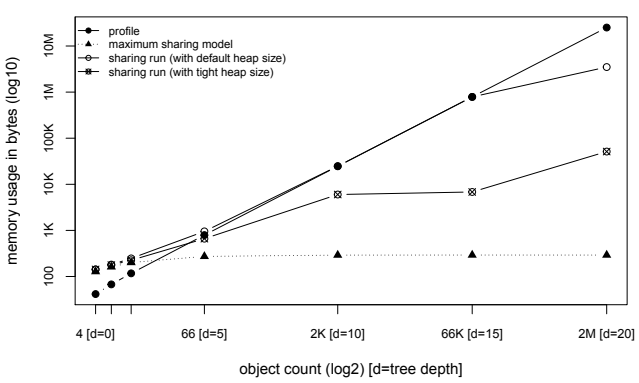
C: Start the REPL of the Rascal language interpreter, and type check a large module (5–10k lines of code).

D–H: Load serialized call-graphs and calculate the transitive closure for *JHotdraw*, *JWAM16FullAndreas*, *Eclipse-202a*, *jdk14v2* and *JDK140AWT*. These benchmarks are supposed to stress the influence of data shape, and the effect of redundancy in algorithmic computation.

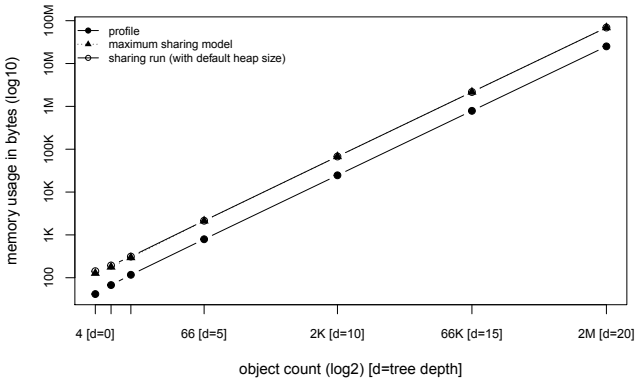
M{E,S,T}: Peano arithmetic modulo 17 in three variations, i.e. *expression*, *symbolic*, and *tree*. These are standard benchmarks for term rewriting engines and are previously used to measure the effect of maximal sharing [23].

3.5.1 Results

First of all, we report that the experimental runs with the maximal sharing aspect of benchmarks B and C timed out after 30 minutes. The cause of the problem, after some manual investigation, was an enormous amount of hashing collisions in the global cache of the shared version. Using MASHO's hashcode logging feature and a Java debugger we found out that the "annotations" feature of PDB was causing trouble. For every annotated value there is a non-annotated value with the same `hashcode`, leading to as many collisions as there are annotated values. In benchmarks B and C there are many parse trees that are annotated with their source code position. To continue our experiments, we then provided an alternative `hashcode` implementations for annotated values, which only the global cache invokes for lookups. Note that altering the problematic `hashcode` method itself is not an option, because it would break the semantic of any program that uses the annotation feature. Applying the fix was necessary for continuing the evaluation, to be able to compare MASHO's models against data from a real maximal sharing implementation. However, the fix was not necessary for a priori performance modeling. We also



(a) results for redundant trees



(b) results for redundancy-free trees

Figure 5: Calibration data: Memory usage for various test runs (without compensation for GC noise). Figure 5a illustrates that compensating for GC noise is necessary to obtain accurate memory footprint models.

noticed another `hashcode` related problem —the `hashcode` of a singleton set collides with the `hashcode` of its contained object— and fixed it analogous to the previous problem. Figure 6 finally visualizes the results for all benchmarks. We first interpret this data to subsequently answer our evaluation questions. In obtaining the results, ORP and ECP yielded a mean slowdown of 7x (range 2.5–32x).

Object Redundancy and Memory Gains.

Figure 6a illustrates *object redundancy* in relative numbers, that is how many newly allocated objects yield a hit in the global cache. Over all benchmarks, we can report a mean redundancy of 64%, with a minimum of 33%, and a high of 100% in case of the Peano arithmetic benchmarks. However, the amount of object redundancy does not imply equal gains in *mean memory reduction*. Allover, observed mean memory reductions are below the object redundancy numbers, emphasizing that the size of redundant objects matters and not only their count. In case of the algorithmic transitive closure benchmarks (D–H) we even see a negative net impact on mean memory consumption, albeit 33–58% object redundancy. The loss is attributed to the overhead of the global cache and that redundancy is mostly present in terms of small objects. Figure 6b presents another view on the *Mean Memory Reduction* data points from Figure 6a by displaying the mean memory usage of the benchmarks before and after applying maximal sharing.

Cache Hits and Negative Comparisons due to Chaining and Hash Collisions.

In Figure 6c we illustrate the number of `false equals`-calls that occur on average when performing a global cache lookup that eventually yields a hit. We do not further distinguish and discuss the causes of `false equals`-calls, which could be either attributed to implementation details of the global cache (e.g., chaining due to modulo `size` operations), or to hashcode implementations causing collisions. A high ratio should alert a library engineer to systematically explore these possible causes.

Figure 6d shows the absolute numbers for object allocations, cache hits, and collisions for all benchmarks. Benchmarks ME20, MS20, and MT20 created a high cache load —causing many negative equality checks— that in the case of ME20 and MT20 led to substantial memory savings (cf. Figure 6b).

In our data set, on average a global cache hit triggers 1.4 nested reference comparisons. This illustrates how maximal sharing transforms the shape of `equals`-call-graphs: frequent comparisons in the global cache are shallow, and recursive `equals`-calls in the program collapse to one comparison.

Equality Profile of the Original Library.

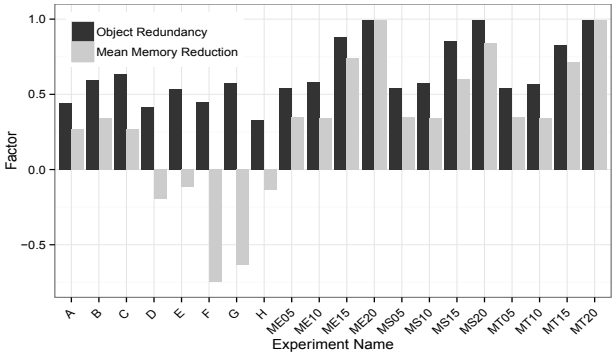
Figure 6e highlights the mixture of equalities encountered. Surprisingly, calls to `equals` with aliased arguments occurred more frequently than calls to `equals` and `isEqual` with distinct arguments. The transitive closure benchmarks D, E and H solely perform reference comparisons. Consequently, the alias-aware analysis of ECP is necessary in our case, otherwise we would have clearly over-approximated savings under maximal sharing. With respect to the recursive call-graph shape of `equals` and `isEqual`, we observed on average 2.7 nested equality calls (other than reference equalities).

Equality Profile with Maximal Sharing.

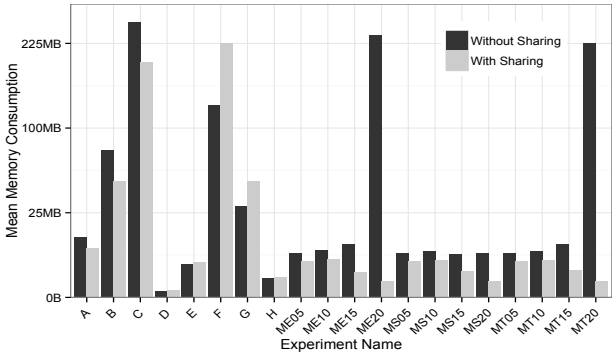
Figure 6f shows the equality profile of the experiments with maximal sharing enabled and highlights the changes to Figure 6e. Absolute numbers of calls decrease, because each recursive `equals`-call is replaced by a single reference comparison. Recursive call-graphs for `isEqual` remain, if two objects are objects are equivalent (according to `isEqual`) but not strictly equal.

Benchmarks M{E,S,T} — Peano Arithmetic.

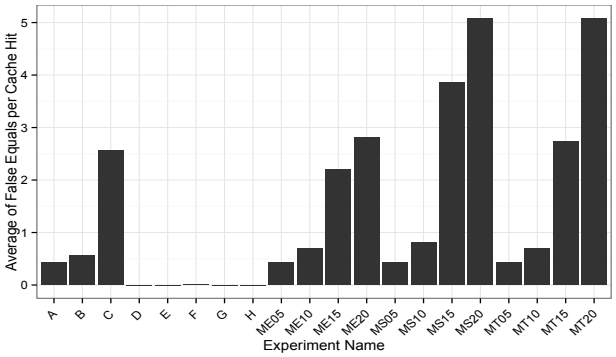
These benchmarks are designed to bring out best behavior for maximal sharing by generating an enormous amount of redundant terms. The results are shown for different sizes of the problem of symbolically computing 2^n **mod** 17 in Peano arithmetic. The results show that redundancy was accurately predicted for all three benchmark versions. MS exhibited a saving potential up to 86% with increasing input size, the others up to 100%, which is in line with related work [23]. However, we do not see significant gains in use of equality. The reason is that our implementation of the benchmark uses deep pattern matching instead of direct calls to `equals` and therefore loses the benefit of $O(1)$ reference comparisons.



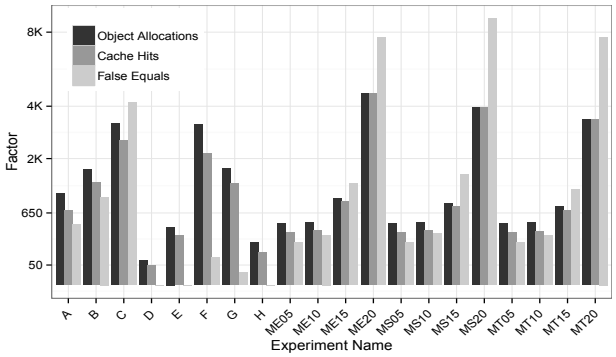
(a) Relation of Redundancy and Mean Memory Savings



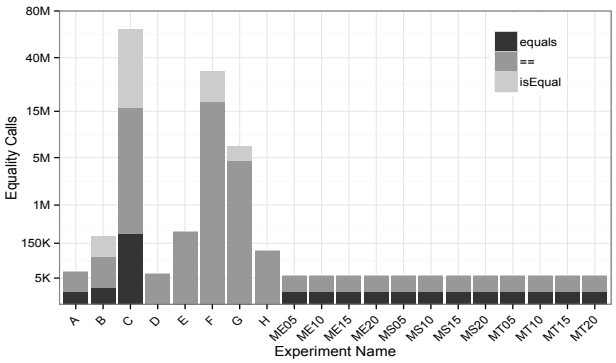
(b) Effect of Maximal Sharing on Memory Consumption



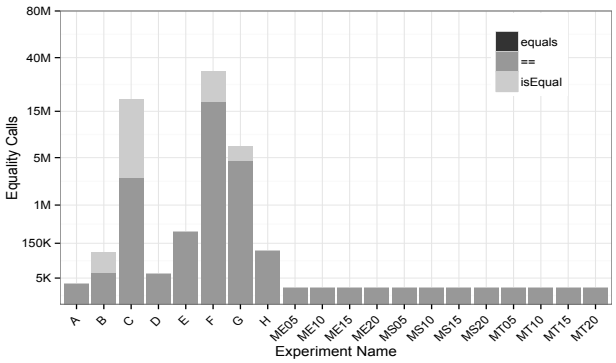
(c) Ratio of False Equality Comparisons per Cache Hit



(d) Relation of Object Allocation, Cache Hits, and Collisions



(e) Equality Profile without Maximal Sharing



(f) Equality Profiles with Maximal Sharing

Figure 6: Realistic data: Various memory and equality aspects of the applications under test. Figures 6b and 6d use a square root scale on the y-axes, and figures 6e and 6f a double square root scale, to better accommodate the wide range of values.

3.6 Analysis

Q-Accurate.

None of the experiments showed significant differences between the predicted and the actual memory usage; the mean accuracy of *method 2* was about 99%. For all but one benchmark, calls to `equals` methods were predicted with an accuracy of at least 99%. The only outlier was benchmark B, the parser generator benchmark, that exhibited 19% more calls to `equals`, caused by an corresponding increase in global cache hits. The additional cache hits were caused by equivalent objects that were re-generated at a higher rate than the collection of the weak references from the global cache. In the latter case, we actually under-approximated the potential savings, because the longer living weak references caused an overlap between previously disjoint object families.

We conclude that MASHO accurately models lower bounds for hypothetical heap evolution and calls to `equals` under maximal sharing for all our benchmarks. This means that 256-bit SHA-2 hashes were good enough at least for this (heterogeneous) data, and that the MASHO model is complete.

Q-Actionable.

The redundancy data clearly suggests that PDB could benefit from maximal sharing for most benchmarks. However, during profiling we figured out that PDB's annotation fea-

ture causes a substantial number of hash-collisions. Furthermore, the effectiveness of maximal sharing diminished under `isEqual` that ignores annotations: many calls to `isEqual` cannot be replaced by reference comparisons. It follows that the library would require severe reengineering before maximal sharing can be applied optimally.

As compared to general memory profilers, which do not consider the specifics and preconditions of maximal sharing, we showed that the GC can hide memory savings of maximal sharing. A memory profiler will not even see substantial savings unless by trial and error global minimum heap bounds are found. Since MASHO ignores the effect of the GC, this confounding effect has become a non-issue.

The information provided by MASHO as compared to our simple maximal sharing aspect is comparable. More importantly, the maximal sharing aspect suffers from arbitrary hash collisions in terms of accuracy (more `equals`-calls will be made as hash buckets become deeper) and speed (the benchmarks will run longer and longer). MASHO provides filtered information, isolating the effect of maximal sharing from the confounding effect of hash collisions.

In additional experiments simulating the semantics of related memory profilers, which check for isomorphic heap graphs [18, 14], we measured that MASHO uncovers up to 14%, and at median 4%, more unique caching opportunities in the aforementioned benchmarks than the related work can provide —due to the additional abstraction facilities.

4. RELATED WORK

We position our contribution with respect to memory profiling tools and studies, programming language features, and maximal sharing libraries.

Memory profiling tools and studies.

Sewe et al. [21] investigated the differences in memory behavior between Java and Scala programs. The key findings were that objects in Scala are more likely to be immutable, small, and to have a short lifetime, compared to pure object-oriented Java programs. Ergo, the Scala community may benefit from MASHO.

Dieckmann and Hölzle [7] originally published a study about the allocation behavior of six SPECjvm98 Java programs, and compared the results to Smalltalk and ML. The authors obtained allocation data by instrumenting the source code of a virtual machine and built a heap simulator.

Sartor et al. [20] discuss the limits of heap data compression by examining types and sources of memory inefficiencies. Techniques were investigated that work on a wide spectrum of granularity, ranging from object equality to stripping off empty bytes, or compressing fields and arrays. Their analysis approximates saving potentials by analyzing a series of timed heap dumps. The authors observed that deep object equality together with array sharing reduces the size of applications by 14% on average. These results also motivate our research but timed heap dumps do not provide enough detail to assess the impact of maximal sharing accurately.

Resurrector [29] is an object lifetime profiler that supports a tuneable cost settings per allocation site. For frequent calls to allocation sites, Resurrector works more precisely than garbage collector heuristics and can avoid expensive reachability analyses to identify dead objects, as used by like Merlin [9] or Elephant Tracks [19]. These more advanced lifetime profiling techniques are usually implemented inside a Virtual Machine (VM). In contrast, MASHO uses garage collector timestamps as heuristic for object lifetime obtained standard interfaces and techniques (JVMTI and BCI) and thus works across different JVMs. MASHO predicts with almost perfect accuracy (see Section 3), so these more precise and much more expensive techniques are not necessary here.

Bhattacharya et al. [3] reduce unnecessary allocations of temporary strings or container objects inside loops, by analyzing which objects can be reused after each loop iteration. MASHO reasons over redundancy of a whole program run and therefore also covers these cases, necessarily.

Nguyen and Xu [16] detect cacheable objects at allocation sites with variants of data dependence graphs, and memoizable functions at their call sites. Their tool, Cachetor, is implemented inside a VM and targets arbitrary mutable programs and thus leading to a 200x overhead. Redundancy profiling, as implemented by MASHO, in contrast exploits the preconditions of immutable object graphs and can thus operate at lower runtime overheads.

To optimize compilers, Lattner and Adve [12] researched a *macroscopic* approach for reasoning over pointer-intense programs, by focusing on how programs use entire logical data structures, rather than individual objects, to then segregate these objects automatically into separate memory pools.

With Object Equality Profiling (OEP), Marinov et al. [14] pinpoint groups of equivalent objects that could be replaced by a single representative instance. OEP considers every single object created during a program run. The authors use BCI to track heap activity dynamically. A post-mortem analysis calculates mergeability of objects, by checking isomorphism of labelled graphs. OEP uses an off-line graph partitioning algorithm to process data sets that might exceed main memory size in $\mathcal{O}(n\ log\ n)$ time. One of the key contributions of OEP—that makes it scalable for mutable objects but difficult to apply for modeling maximal sharing—is pre-partitioning heap graphs based on the primitive values of objects as a discriminator. In our context this causes OEP not being able to abstract from implementation details such as arbitrary ordering of elements in arrays, specialized sub-classes, and lazily initialized or cached hashcodes. For libraries based on immutable objects, this can make objects look different while they should be the same. Our experiments showed that MASHO uncovers up to 14%, and at median 4%, more unique sharing opportunities than OEP on the same data would. The focus on immutable objects gives MASHO both the opportunity to abstract and the ability to optimize the necessary high granularity memory profiles.

Rama and Komondoor [18] worked on an extension of OEP and introduced a tool, the Object Caching Advisor (OCA), to support introducing hash-consing at the source-code level as a refactoring. The authors reuse a fingerprinting function, introduced by Xu [28], that runs in $O(size\ of\ object\ graph)$ and yields a runtime overhead ranging from 98–2520x. In contrast, MASHO's fingerprinting, which is based on Merkle trees, operates in $O(shallow\ object\ size)$.

Language support for obviating equals and hashCode.

Vaziri et al. [26] proposed a declarative, stricter form of object identity called *relation types*. By requiring that the identity of an object never changes during runtime, the authors obviated potential error-prone `equals` and `hashcode` methods. A subset of immutable key fields, referred to as tuple, designates object identity. These tuples match our

weak-immutability requirement (see Section 2.3). The authors formalized their programming model and proved that hash-consing preserves semantics and is a safe optimization in their model. Our contribution is nicely orthogonal: Whereas MASHO investigates maximal sharing for libraries and requires that `equals` and `hashcode` are user provided, relation types are meant to be an equality substitute at language level. Vaziri et al. contribute the language supported semantics of weak immutability, which is our a priori assumption.

Scala counters fragile `equals` and `hashcode` implementations with the concept of *case classes*. Scala shows that immutable data types that adhere to structural equality can obviate hand-written `equals` and `hashcode` implementations. The compiler —like with relation types— synthesizes their implementation, but since maximal sharing is not always beneficial it does not generate shared implementations. A recast of MASHO to Scala may help finding optimal solutions for libraries that heavily rely on case classes.

The ATerm library.

is a prime source of inspiration [22, 24, 25]. Both in C and in Java this is a successful library that employs maximal sharing for representing atomic data-types, lists and trees. Key design considerations of the ATerm library are to specialize garbage collection (in C), and (de)serialization as well based on the condition of maximal sharing and structural equality. In this paper we use benchmarks from the ATerm experience to evaluate MASHO. The reported use cases of ATerm library (specifically in the term rewriting and model checking context) indicate the possibility of great savings in memory consumption and great increases in performance, but in the general case it is unlikely that maximal sharing is always a good idea.

ShadowVM [13].

ShadowVM is a recent generic Java run-time analysis framework. It separates instrumentation from the client VM and adds asynchronous remote evaluation to increase isolation and coverage. Analyses can be written on a high level of abstraction using an open pointcut model and support bytecode instruction granularity as well. MASHO would be a good usage scenario for ShadowVM, since tracking == bytecode instructions needs bytecode instrumentation beyond the capabilities of AspectJ.

5. CONCLUSION

We introduced a new predictive performance modeling tool named MASHO— for assessing the effects of introducing the maximal sharing optimization strategy into a Java library without changing the library or client code of the library.

MASHO profiles object redundancy and calls to `equals` efficiently using object fingerprints. Under the assumption of *weak immutability*, fingerprinting leads to an accurate model efficiently. MASHO can abstract from accidental implementation details in the current version of a library, such as arbitrary array orderings which also enhances its accuracy.

The experience report focused on the accuracy of the predictions, since fingerprinting and feedback loops with garbage collection heuristics may introduce noise. This showed on a controlled case and on realistic cases that MASHO's predictions are accurate.

Predictive performance analysis with MASHO isolates the effect of maximal sharing from other noise in the measurements, in contrast to a full blown experiment where confounding effects (like garbage collection) may be prohibitive for decision making.

6. ACKNOWLEDGMENTS

The authors thank the anonymous reviewers and further Ali Afroozeh, Bas Basten, Martin Glabischnig, Davy Landman, Tijs van der Storm, and Pablo Inostroza Valdera for their valuable feedback on earlier draft versions of this paper.

7. REFERENCES

[1] H. G. Baker. Equal rights for functional objects or, the more things change, the more they are the same. *SIGPLAN OOPS Messenger*, 4(4):2–27, 1993.

[2] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on Java. In *Rewriting Theory and Applications (RTA 2007)*, LNCS. Springer, 2007.

[3] S. Bhattacharya, M. G. Nanda, K. Gopinath, and M. Gupta. Reuse, recycle to de-bloat software. In *ECOOP '11: Proceedings of the 25th European Conference on Object-oriented Programming*. Springer, 2011.

[4] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In C. Kirchner and H. Kirchner, editors, *International Workshop on Rewriting Logic and its Applications*, volume 15 of *ENTCS*. Elsevier, 1998.

[5] T. Braibant, J.-H. Jourdan, and D. Monniaux. Implementing hash-consed structures in Coq. In *Proceedings of the 4th International Conference on Interactive Theorem Proving*, LNCS. Springer, 2013.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

[7] S. Dieckmann and U. Hölzle. A study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. In *ECOOP '99: Proceedings of the 13th European Conference on Object-oriented Programming*. Springer, 1999.

[8] E. Goto. Monocopy and Associative Algorithms in Extended Lisp. University of Toyko. Technical report, 1974.

[9] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Generating object lifetime traces with Merlin. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(3):476–516, 2006.

[10] P. Klint, T. van der Storm, and J. J. Vinju. EASY meta-programming with Rascal. leveraging the Extract-Analyze-SYnthesize paradigm for meta-programming. In *Proceedings of the 3rd International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'09)*, LNCS. Springer, 2010.

[11] D. Knuth. Structured programming with Goto statements. In E. N. Yourdon, editor, *Classics in Software Engineering*. Yourdon Press, 1979.

[12] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. *ACM Sigplan Notices*, 40(6):129–142, 2005.

[13] L. Marek, S. Kell, Y. Zheng, L. Bulej, W. Binder, P. Tůma, D. Ansaloni, A. Sarimbekov, and A. Sewe. ShadowVM: robust and comprehensive dynamic program analysis for the Java platform. In *GPCE '13: Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*. ACM, 2013.

[14] D. Marinov and R. O'Callahan. Object equality profiling. In *OOPSLA '03: Proceedings of the 2003 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. ACM, 2003.

[15] P. D. Mosses. *CASL Reference Manual, The Complete Documentation of the Common Algebraic Specification Language*, volume 2960 of *LNCS*. Springer, 2004.

[16] K. Nguyen and G. Xu. Cachetor: Detecting cacheable data to remove bloat. In *ESEC/FSE '13: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013.

[17] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[18] G. M. Rama and R. Komondoor. A dynamic analysis to support object-sharing code refactorings. In *ASE '14: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2014.

[19] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant tracks: portable production of complete and precise GC traces. *ACM Sigplan Notices*, 48(11):109–118, 2013.

[20] J. B. Sartor, M. Hirzel, and K. S. McKinley. No bit left behind: The limits of heap data compression. In *ISMM '08: Proceedings of the 7th International Symposium on Memory Management*. ACM, 2008.

[21] A. Sewe, M. Mezini, A. Sarimbekov, D. Ansaloni, W. Binder, N. Ricci, and S. Z. Guyer. new Scala() instance of Java: a comparison of the memory behaviour of Java and Scala programs. *ACM Sigplan Notices*, 47(11):97–108, 2012.

[22] M. G. J. Van den Brand, H. A. De Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Software: Practice and Experience*, 30(3):259–291, 2000.

[23] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *TOPLAS*, 24(4):334–368, 2002.

[24] M. G. J. van den Brand and P. Klint. ATerms for manipulation and exchange of structured data: It's all about sharing. *Information and Software Technology*, 49(1):55–64, 2007.

[25] M. G. J. Van Den Brand, P.-E. Moreau, and J. Vinju. A generator of efficient strongly typed abstract syntax trees in Java. *IEE Proceedings - Software Engineering*, 152(2):70–87, 2005.

[26] M. Vaziri, F. Tip, S. Fink, and J. Dolby. Declarative object identity using relation types. In *ECOOP '07: Proceedings of the 21th European Conference on Object-oriented Programming*. Springer, 2007.

[27] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *LNCS*. Springer, 2004.

[28] G. Xu. Finding reusable data structures. In *OOPSLA '12: Proceedings of the 2012 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. ACM, 2012.

[29] G. Xu. Resurrector: A tunable object lifetime profiling technique for optimizing real-world programs. In *OOPSLA '13: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. ACM, 2013.