A Dynamic Analysis to Support Object-Sharing Code Refactorings

Girish Maskeri Rama[*]
Infosys Limited
girish_rama@infosys.com

Raghavan Komondoor
Indian Institute of Science, Bangalore
raghavan@csa.iisc.ernet.in

ABSTRACT

Creation of large numbers of co-existing long-lived isomorphic objects increases the memory footprint of applications significantly. In this paper we propose a dynamic-analysis based approach that detects allocation sites that create large numbers of long-lived isomorphic objects, estimates quantitatively the memory savings to be obtained by sharing isomorphic objects created at these sites, and also checks whether certain necessary conditions for safely employing object sharing hold. We have implemented our approach as a tool, and have conducted experiments on several real-life Java benchmarks. The results from our experiments indicate that in real benchmarks a significant amount of heap memory, ranging up to 37% in some benchmarks, can be saved by employing object sharing. We have also validated the precision of estimates from our tool by comparing these with actual savings obtained upon introducing object-sharing at selected sites in the real benchmarks.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.3.4 [**Programming Languages**]: Processors

Keywords

Memory optimization; object caching

1. INTRODUCTION

The advent of large system memories has not vetted the appetite for memory optimization tools for software. For instance, various researchers have noted [15, 16, 25, 24] that software commonly consumes unexpectedly high amounts of memory, frequently due to programming idioms that are used to make software more reliable, maintainable and understandable. In the case of modern object-oriented systems this problem is partly due creation of large numbers of co-existing *isomorphic* objects. Intuitively, two objects are isomorphic if they are of the same type, have identical values in corresponding primitive fields, and are such that corresponding reference fields themselves point to isomorphic objects. In other words, the portions of memory rooted at the two objects are isomorphic shape-wise as well as values-wise.

```
public CommonFont getFontProps(){
  ...
  CommonFont commonFont =
    new CommonFont(
        fontFamily, fontSelectionStrategy,
        fontStretch, fontStyle, fontVariant,
        fontWeight, fontSize, fontSizeAdjust);
    return commonFont;
}
```

Listing 1: Creation of isomorphic objects

1.1 Motivation

To illustrate this problem consider the example code in Listing 1, adapted from the real program *Apache FOP*. This code has been substantially simplified for ease of presentation. The `getFontProps` method in the code is used to create a `CommonFont` object (to encode display properties) for each FO (Formatting Objects) node in the input FO document. Each `CommonFont` object (which we simply call font object from here on) occupies 64 bytes in memory. For large documents the number of such font objects created is very large. However, in practice, because the fields of the `CommonFont` class can take on only a small number of distinct values, e.g., *Arial* or *TimesRoman* for `fontFamily` and 10pt or 12pt for `fontSize`, many of the font objects created are isomorphic to each other. For instance, with a 104-page real input document that we supplied as input, although the program created 9079 font objects, 8943 (i.e., 98.5%) of these objects belonged to only four distinct equivalence classes by isomorphism. Furthermore, all font objects are long lived (i.e., are live nearly until the end of execution of the program). A significant reduction in heap usage can therefore be achieved if the code is refactored to *de-duplicate* or *share* common font objects whenever possible instead of always creating distinct but possibly isomorphic objects. Such a refactoring, which employs a cache to keep track of objects created so far and to share them, is shown in Listing 2. We have elided the `equals` and `hashCode` methods of the `CommonFont` class from the listing, which ensure that a cache

[*]Currently pursuing part-time PhD in Indian Institute of Science, Bangalore.

ASE'14, September 15-19, 2014, Vasteras, Sweden.

ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2642992 .

```
private Map<CommonFont, CommonFont> map =
    new WeakHashMap<CommonFont, CommonFont>();

public CommonFont getFontProps(){
  ...
  CommonFont commonFont =
    new CommonFont(
        fontFamily, fontSelectionStrategy,
        fontStretch, fontStyle, fontVariant,
        fontWeight, fontSize, fontSizeAdjust);

    if(map.get(commonFont) != null)
        commonFont = map.get(commonFont);
        // return cached isomorphic object
    else
        map.put(commonFont,commonFont);
        // insert new object into cache
    return commonFont;
}
```

Listing 2: Caching isomorphic objects

lookup using a newly created object as key is guaranteed to return either *null* or an object that is isomorphic with the new object. Note that whenever a cached object is found, the newly created object implicitly becomes garbage and will be collected by the GC, thus not occupying space in memory that it otherwise would. Also, a `WeakHashMap` (rather than a normal hash-map) is appropriate, because any (*key, value*) pair gets automatically deleted from a weak hash-map when there are no references to the key object from any other location in memory. This ensures that any originally non-long-living objects that might happen to get introduced into the cache are not kept there any longer than they are needed, hence avoiding inadvertent increases in heap usage.

As a result of the above-mentioned refactoring, of the 9079 font objects, only 4 will be long living. This could potentially result in a reduction of memory consumption of 567 KB ((9079 - 4) * 64 bytes) by the time all the font objects have been created (ignoring space overheads due to the internal structures of the cache).

We have observed that in several open-source projects, for instance, *Apache FOP*, *PDFBox*, and *GanttProject*, there were several allocation sites that create long-living objects where sharing was not employed initially, but that were refactored subsequently by programmers to employ sharing by using a cache of some form. This is presumably in response to memory bottlenecks that were observed after deployment or usage of the initial (unoptimized) versions of the applications. In fact, in the case of FOP, we have measured that sharing that has been introduced at 14 different allocation sites by developers yields up to 11% reduction in heap footprint on real inputs. Correspondingly, a limit study on object duplication [15] has found that if an oracle could be used to eliminate all instances of isomorphic co-existing objects by resorting to object-sharing, then up to 60% reduction in peak heap usage can be achieved.

1.2 Challenges in introducing object sharing

However, once it has been recognized that an application suffers from high heap memory usage, there are many challenges in identifying object allocation sites that are good candidates for employing object sharing. A good allocation site is one that would give *good memory reduction* after a cache is introduced, and where it is *safe* to introduce a cache. Good memory reduction occurs if the site creates a reasonably large number of objects, with relatively few equivalence classes among these objects wrt isomorphism (thus increasing the scope for sharing), and with most of these objects being *long-living* objects. The need for the last condition mentioned above is as follows: If a site creates mostly short-lived (aka temporary) objects, then the portion of heap memory occupied by objects from this site will in general be low, because the garbage collector will collect most of the extant objects that were created at this site whenever it runs. Therefore, the savings in memory usage due to a cache at this site will also be low.

We now discuss the safety aspect. Introducing object sharing at a site is safe, intuitively, only if objects created at the site or reachable from these objects right after they are created are never modified in the rest of the program's execution (in any execution), and if the rest of execution is behaviorally independent of the actual addresses of these objects. Safety is a real concern. For instance, in the open-source project PDFBox (version 0.73) developers had employed a cache for object-sharing at the allocation site in line 106 in file PDFFontFactory.java. However, a user detected and reported a bug (Bug ID 610) on version 0.8.0 of the software, which was subsequently fixed in version 1.7.0. The bug fix involved disabling the cache, with the fix description mentioning that unsafe sharing was the cause of the bug.

Since real applications can have thousands of object allocation sites, locating good candidate sites among these is proverbially like looking for needles in a haystack.

Related research in the area of refactoring programs by leveraging isomorphism opportunities is primarily focused on *reusing* short-lived data structures after they are dead rather than re-create new instances of these data structures repeatedly [3, 21, 17]. The objective of such a refactoring is primarily to reduce the GC overhead, and also to reduce the running time required to re-create and re-initialize data structures repeatedly. At an intuitive level these approaches look for sites such that (1) they create many short-lived objects with disjoint lifetimes, and (2) there is much isomorphism among these objects, so that a few representative objects can be saved in a cache and reused whenever required. While we share requirement (2) above, we actually have the *converse* of requirement (1): We need to find sites that create mostly long-lived objects, so that sharing can be introduced among these objects to reduce memory consumption. Another key difference is that reuse of a dead object does not introduce any risk of changing the program's behavior. In contrast, since we are looking to share live objects, we face the additional challenge of somehow checking whether a proposed refactoring to introduce sharing is safe or not.

1.3 Our approach and contributions

Our main contribution is a dynamic analysis technique for *estimating* for the allocation sites in a program, for a given input, the reduction in heap memory usage (in bytes, or as a percentage) to be obtained by employing object sharing at this site, as well as the safety of doing so. The quantitative estimates produced by our technique of a user-observable benefit (i.e., actual memory savings) make it easier for developers to select sites to refactor. In contrast, previous approaches [21, 17] compute a metric between 0 and 1 for each site, indicating the extent of isomorphism at the site. This metric can be used to rank sites heuristically, but does not directly indicate the extent of observable run-time benefit to be expected from eliminating isomorphism.

Computing a precise memory savings estimate for *every* allocation site in a program can be prohibitively expensive. Checking the safety of employing object-sharing at each site is also very expensive. At the same time, if imprecision in the estimates is high or if safety is not checked carefully then the utility of the approach will become compromised. The solution that we propose, therefore, is a two-phase analysis, both of which involve an instrumented run of the program on the given input (with different instrumentation in the two phases). The responsibility of the first phase is to identify unsafe sites and remove them from consideration as soon as possible during the run, and also to remove sites that are *possibly unprofitable* as early as possible. The profitability of each site is approximately encoded by a metric that is similar to the one used by Xu et al. [21]. The metric for each site is updated each time the site is visited. Once a site is removed from consideration during the run, visits to the site in the remaining part of the run are not analyzed in any way. In this way, the set of sites under consideration shrinks monotonically over the run.

The allocation sites that remain under consideration at the end of the first phase are the sites to be tracked in the second phase. The responsibility of this phase is to precisely partition the objects created at each of these tracked sites into equivalence classes wrt isomorphism. At the end of the run, for each tracked site, its equivalence classes are used to calculate a memory-savings estimate, as illustrated earlier in Section 1.1. A heuristic is employed during this calculation to eliminate from consideration non-long-lived objects in the equivalence classes. In our approach we estimate savings at the end of the run only, and not at other preceding points. We discuss why this is justified in Section 5 of the paper.

Reverting to the notion of safety of introducing object-sharing at a site, it is noteworthy that although our safety characterization for refactoring a site (as discussed in Section 1.2) basically insists on behavior preservation in *all* runs of the program, our current dynamic analysis approach checks for safety only in the given run of the program.

We have implemented our approach as a tool, *Object Caching Advisor* (OCA). A key aspect of our tools is that it is based purely on running an instrumented version of the program, with no modifications required to the underlying JVM. This increases the applicability of our approach in practice. We have experimented with our tool on a range of real Java programs, including the DaCapo 2006 benchmarks. Some of the key findings of our experiments include:

- Nearly all applications have potential for reduction of memory usage by object sharing, with up to 37% estimated reduction in heap footprint in the best case.
- Our tool was able to detect that the unsafe site in PDFBox that the developers had originally introduced sharing at is indeed an unsafe site.
- We considered an application, FOP, where developers had manually introduced object-sharing caches at 14 allocation sites, and ran our analysis on modified versions of these applications wherein we disabled the caching. We found that our tool placed 10 of these 14 sites within the top 27 ranks in its report of sites sorted by descending order of the estimated saving. Furthermore, we found that developers *had not* employed sharing at several sites that the tool reported and that would give good savings if sharing were to be employed at them.

```
void m() {
  ...
  T v =
  new C(a1,a2)
     ;
  ...
}
```

$\Longrightarrow$

```
private Map<T,T> cache=
   new WeakHashMap<T,T>();
 void m() {
  ...
  T v = new C(a1,a2);
  T w = cache.get(v);
  if (w != null)
     v = w;
  else
     cache.put(v,v);
  ...
 }
```

Figure 1: Caching using hash consing.

The rest of this paper is organized as follows. Section 2 introduces the object-sharing refactoring in generic terms. Section 3 introduces key notation and terminology that we use. The sufficient conditions for safely introducing caching at a site are presented in Section 4. Section 5 discusses some conceptual issues about when to estimate heap savings in a run. We describe our actual dynamic analysis approach, as well as our implementation of the approach, in Section 6. In Section 7 we discuss empirical results from our tool. Section 8 discusses certain limitations of our approach and experiments. Related work is presented in Section 9, while Section 10 concludes the paper.

2. THE REFACTORING PROBLEM

Given an allocation site s of the form "`v = new C(`$a_1, \ldots, a_i$`)`" in a method 'm' of a program, the problem that we address is to refactor the code around the allocation site to employ object sharing via the use of an object cache. In the rest of this paper we will use the phrase *introduce a cache* at s to denote this refactoring.

We spell out the code refactoring that our approach primarily supports using the templates shown in Figure 1, with the left side showing original code and the right side showing refactored code. This style of object sharing has been called *hash consing* in the literature. With hash-consing, after a new object is allocated, the cache of existing objects is searched for an object that is isomorphic to the new one. If an existing one is found a reference to it is used in place of the new one, thus allowing the new object to be garbage collected. Hash consing has been employed in compilers or run-time environments to save memory, in the context of languages such as Lisp [10] and ML [2], and even with Java (in the form of *string interning*). However, we are addressing the scenario of introducing hash-consing at the source-code level as a refactoring.

There is another style of object sharing, called *memoization*, in which an attempt is made to discover before each object is created whether an existing object is available in the cache that would be isomorphic to the new object if it were to be created. If such an object is found the new object need not be created at all. This style of object sharing puts less pressure on the GC than the hash-consing style. However, it is less generally applicable than the hash-consing style. In particular, the key to be used for the cache lookup needs to be in terms of the parameters to the constructor corresponding to the allocation site where the object is to be created. Also, this constructor needs to be side-effect free, else the behavior of the program can change if calls to it are elided. While our memory savings estimation approach for a given allocation site is as precise for the memoization style as it is for the hash-consing style, we currently have not implemented the check for the side-effect freedom of the constructor. Therefore, the total savings estimates across all sites that we report in Section 7 include savings from sites that are potentially amenable only to hash-consing but not to memoization.

3. NOTATION AND TERMINOLOGY

In this section we present some notation and terminology that we will use throughout this paper. We say that two objects o_r and o_s are *isomorphic* at an instant of time t in a run, if the values of all corresponding primitive fields of o_r and o_s are equal at that instant, and the objects referred to by the corresponding reference fields of o_r and o_s are themselves isomorphic at the instant t. We say that an object o_j is *reachable* from an object o_i at some instant in a run, denoted as $o_j \in \mathit{Reach}(o_i)$, if either o_j is o_i itself or o_j is reachable from o_i at that instant by following one or more reference fields. We omit the instant t in the notation above as it will be obvious from the context of usage.

We say that an object o is a *value object* at an instant t in a run if object o as well as each object in the heap that is reachable from it at the instant t has the following two properties: (1) the object is not mutated in the rest of the run, and (2) the object's *identity*, in particular, its address, is not used in any computations or conditionals in the rest of the run (the notion of an object's identity affecting computations or conditionals is defined fully in previous work [15]). Intuitively, an object is a value object at an instant t if it itself and all objects reachable from it behave like true values from that instant onward, in the sense that they do not mutate and do not have any identity. Whenever we refer to a instant in a run just "after" an object o is created, we mean the instant when control has returned from the constructor corresponding to o's creation site. We will refer to an object o as a *value object* (without reference to an instant of time) if o is a value object at the instant right after its creation. We define an allocation site s in a program as a *value-object site*, or *VO site*, if in all runs of the program all objects created at the site are value objects.

A *long-lived* object is one that survives multiple rounds of garbage collection and is still extant (i.e., not garbage collected) at the end of the run. A *temporary* or *short-lived* object is one that dies soon after it is created. In a *Generational GC* scheme, which is used by default in the primary reference JVM implementation *HotSpot* [18], long-lived objects are the ones that are in the *tenured generation* at the end of the run, while short-lived objects are ones that never migrate to the tenured generation.

4. SAFETY

In this section we discuss our sufficient condition for safely introducing a cache at an allocation site s. The introduction of a cache at an allocation site (to implement hash-consing style object-sharing) can be said to be *safe* if the refactoring causes no change in the observable behavior of the program. That is, whenever a newly created object o_r is replaced by another object o_s that is in the cache, the rest of the run is identical to the rest of the run had the replacement not occurred, in terms of both the sequence of instructions executed, as well as the updates to the state of the program performed by corresponding pairs of instructions in the two runs.

Our sufficient condition is derived from the sufficient condition that was postulated by Marinov and O'Callahan [15] for safely *replacing* an object o_r in the heap of a program at an arbitrary instant of time t in an execution with another object o_s that also exists in the heap at that instant. Due to space limitations we are not able to present their sufficient condition. Our sufficient condition differs from theirs in two ways: (1) It is simplified and instantiated to the setting where a *freshly created* object is replaced immediately after creation by another object that is taken from a cache, and (2) it is stated in terms of *isomorphism* and *value objects*, which are notions that they did not use, and which are useful in our setting. The following theorem captures our sufficient condition for the safety of the hash-consing style refactoring.

THEOREM 1. *Introducing a cache at an allocation site s to employ hash-consing style object-sharing is safe if (1) the allocation site s is a VO site, and (2) if the method* `equals` *of the class whose instances are being created at the site is such that it returns* true *only if the two objects given to it are isomorphic.*

It can be shown in a straightforward manner that if two objects o_r and o_s are isomorphic at an instant t and are both value objects at that instant then they necessarily satisfy Marinov and O'Callahan's sufficient condition. Using this argument as the basis it is easy to show the correctness of our theorem above.

5. ESTIMATION OF HEAP SIZE SAVINGS

In this section we discuss some basic intuitions about how and when (in a run) to estimate heap savings. We present our actual dynamic analysis approach that is based on these intuitions in Section 6. A key prerequisite activity to doing memory-savings estimation is to determine what exactly to measure, and at what point or points of time in the run of the program to measure. A naive approach to estimating savings at any instant of time t in the run due to object sharing would be to take into account all the objects created so far from the allocation site s that are extant (i.e., not yet garbage collected) at this instant, and to calculate the extra memory occupied by objects that are isomorphic with other objects from this site. However, there are some problems with this suggestion. If at instant t a lot of temporary objects from site s are present in the heap, then these would skew the estimation and make it non-representative. The reason for this is that if hypothetically the GC had done a cycle of collection just before this instant then most of these temporary objects that are present could have as such gotten collected and would thus not have been present in the heap. In other words, estimates become dependent on (non-deterministic) GC timing, which is undesirable.

A possible fix to this problem is to somehow identify temporary objects that are present in the heap and exclude them from the calculation, or choose the instant t such that it is right after a GC collection cycle. However, these ideas are impractical to implement in a setting where we simply run an instrumented program on a stock JVM, with no help or support from a (modified) GC. Furthermore, even if savings could somehow be estimated by discounting temporary objects, in order to express this savings as a percentage (rather than simply in bytes) we would need to know the total amount of memory occupied by *all* non-temporary objects (from all allocation sites) at that instant. Again, this information is difficult to obtain without a modified JVM.

Therefore, the approach we take is to estimate the reduction in the size of the heap at the *end of the run* of the program only. This approach has the following merits:

- For the class of allocation sites that create predominantly long-lived objects, the space occupied in memory by objects from this site is likely to grow monotonically over the course of the program's run. Therefore, total memory requirement, and hence the savings potential due to sharing, peaks at the end of the run. Therefore, the end of the run is a natural point to estimate savings at.

- There exists a technique which can be applied at the end of the run to infer the likely fraction of objects allocated at each site that were not long-lived, without depending on JVM modifications. This information can be used to produce a memory-savings estimate that is not skewed by these non long-lived objects. We describe this technique in Section 6.

- The total amount of memory occupied by all long-lived objects (from all sites) at the end of the run, to be used as the denominator while calculating savings as a percentage, can be identified readily using logs from Generational GCs, without needing any modifications to the JVM.

- The efficiency of our approach is improved, since we do not need to calculate the estimate (which is an expensive operation) repeatedly for any site.

6. OBJECT CACHING ADVISOR: DESIGN AND IMPLEMENTATION

As mentioned in Section 1, we propose a two-phase approach. That is, given a program P and an input I, we run two different instrumented versions of P on the same input I sequentially. We now discuss these two phases in detail.

6.1 Phase 1: Filtering out unsafe and unprofitable allocation sites

The objective of this run of the program is to identify unsafe allocation sites as well as each possibly unprofitable allocation sites, and to emit the remaining sites for processing by Phase 2 (which actually calculates estimated savings).

The following are the key data structures that are employed in the instrumented program. *Tracked* is the set of IDs of allocation sites in the program that are under consideration (by "allocation site ID" we mean the static location of an allocation site). This set is initialized to contain all the allocation sites in the program; as the run proceeds allocation sites are subsequently removed from this set as and when they are found unsafe or potentially unprofitable.

```
switch (current event)
  case (object $o_j$ is created at site $s_i$):
    if ($s_i \in$ Tracked)
      foreach $o_k \in$ Reach($o_j$): Add $s_i$ to O2Sites[$o_k$]
      $fp = \varphi(o_j)$
      F2Counts[$s_i$, $fp$] = F2Counts[$s_i$, $fp$] + 1;
    endif
    if (calc_profit_now($s_i$))
      if (poor_profit($s_i$)): Tracked = Tracked - $s_i$
    endif
  case (object $o_j$ is written to):
    foreach $s_k \in$ O2Sites[$o_j$]: Tracked = Tracked - $s_k$
  case (addresses of objects $o_i$ and $o_j$ are compared):
    foreach $s_k \in$ O2Sites[$o_i$] $\cup$ O2Sites[$o_j$]:
     Tracked = Tracked - $s_k$
end-switch
```

Figure 2: Event-processing pseudo-code for Phase 1

O2Sites is a map from objects to sets of allocation site IDs. Intuitively, if o_k is an object and s_i is an allocation site, s_i needs to be added to *O2Sites*[o_k] if at some point during the run an object o_j is created at site s_i and o_k is reachable from o_j right after o_j's creation.

F2Counts is a map from (allocation site ID, fingerprint) pairs to frequencies. A fingerprint of an object is a number that encodes the shape of the region of heap reachable from the object, as well as values in primitive fields of the objects in this region. Fingerprints are such that two isomorphic objects will have the same fingerprint (but not necessarily vice versa). *F2Counts*[s_i][fp] indicates the number of objects that have fingerprint fp that were created by instances of the allocation site s_i up to the current point in the run.

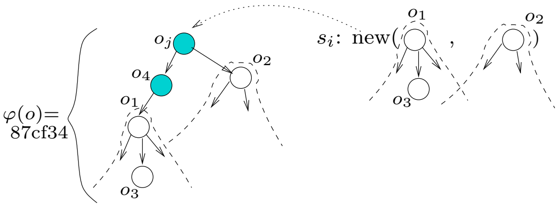
The run-time events of interest that are instrumented in this run of the program are object creations, writes to objects, as well as identity-based operations.

Object allocations: When an object o_j is created at an allocation site s_i, this event is processed by the instrumentation code as shown in the first "case" block in Figure 2. If site s_i is still in under consideration (i.e., is in the set *Tracked*) then the following actions are taken: (1) The site s_i is added to the sets of all objects reachable from o_j in the *O2Sites* map. (2) The fingerprint fp of o_j is computed, and then the count *F2Counts*[s_i, fp] is incremented by 1. We use the function φ proposed by Xu [21] to compute fingerprints; this function basically involves a recursive descent from the given root object, and an aggregation of the values in the primitive fields of all objects encountered using additions and multiplications.

The activities mentioned above are illustrated in Figure 3. The call to the constructor at site s_i is shown to the right in the middle, while the returned 'root' object o_j as well as objects reachable from it are shown to the left. Objects o_1 and o_2 have been passed to the constructor, and are referred to by the new objects (shown shaded). Note that object o_4 would have been created at an allocation site that is inside the constructor corresponding to site s_i. Sample pre-existing maps *O2Sites* and *F2Counts*[s_i] before s_i is executed are shown at the top of the figure. The bottom part of the

O2Sites	
o_1	$\{s_b, \ldots\}$
o_2	$\{s_c, \ldots\}$
o_3	$\{s_d, \ldots\}$
$\ldots$	$\ldots$
o_6	$\{s_h, \ldots\}$

F2Counts$[s_i]$	
d568f3	12
87cf34	8
830ce4	32



O2Sites	
o_j	$\{\underline{s_i}\}$
o_4	$\{\underline{s_i}\}$
o_1	$\{s_b, \underline{s_i}, \ldots\}$
o_2	$\{s_c, \underline{s_i}, \ldots\}$
o_3	$\{s_d, \underline{s_i}, \ldots\}$
$\ldots$	$\ldots$
o_6	$\{s_h, \ldots\}$

F2Counts$[s_i]$	
d568f3	12
87cf34	$\underline{9}$
830ce4	32

Figure 3: Handing of allocation events

figure shows these same maps after they get updated due to the creation of o_j. The underlined entries are the newly added or updated entries.

We now come back to the pseudo-code in Figure 2. The predicate *calc_profit_now*(s_i) indicates whether it is time to check heuristically whether s_i is a potentially unprofitable allocation site. In our current implementation this predicate returns *true* every 100th time site s_i is reached in the run. The predicate *poor_profit*(s_i) checks whether site s_i has poor potential profit by checking whether there is not much isomorphism among the objects created so far at site s_i. This predicate is modeled after the *reusability* metric of Xu [21], although they use their metric as a final output rather than simply to filter away sites from further processing. We currently use the following definition of *poor_profit*(s_i):

$$((\max_{fp \in F} \mathit{F2Counts}[s_i][fp] \,/\, \sum_{fp \in F} \mathit{F2Counts}[s_i][fp]) < 0.1) \vee$$
$$((|F| \,/\, \sum_{fp \in F} \mathit{F2Counts}[s_i][fp]) > 0.2)$$

where is F is $dom(\mathit{F2Counts}[s_i])$, or in other words, the set of fingerprints in $\mathit{F2Counts}[s_i]$. If *poor_profit*(s_i) is true we remove s_i (once for all) from the set of sites under consideration. This saves run-time expense in the rest of the run, because during future visits to this site s_i all objects reachable from the root object that was just created need not be traversed.

Write operations: When a field of an object o_j is written to, this event is processed by the instrumentation as shown in the second "case" block in Figure 2. Basically, all objects o_k from which o_j was reachable just after o_k was created become non value objects. Therefore, the sites at which such objects o_k were created are inferred to be unsafe, and are hence removed from consideration (once for all).

Identity-based operations: In our current implementation we check only for address comparisons (which are the most common form of identity-based operations). Objects whose addresses are compared become non value objects, and cause allocation sites to be considered unsafe, as shown in the pseudo-code in the third "case" block in Figure 2.

6.2 Phase 2: Calculating estimated savings

In this instrumented run only the allocation sites that remain in the *Tracked* set at the end of the first phase are tracked. These sites are known to create only value objects (when the program is run on the input I); therefore, in this phase, write operations and identity-based operations need not be instrumented.

The main data structure maintained in this phase is a map *O2Info*, which is a map from (allocation site ID, object) pairs to pairs of natural numbers. *freq*$(\mathit{O2Info}[s_i][o_j])$ is the first of the two numbers that (s_i, o_j) is mapped to, and indicates the number of objects that have been allocated so far in the run from site s_i that are isomorphic to object o_j. An object o_j is used as an index in the map in conjunction with s_i iff (1) o_j is a root object that was returned from site s_i, and (2) o_j is the *first* element in its equivalence class of objects (wrt isomorphism) to have been created at s_i. In other words, corresponding to each equivalence class of objects from site s_i there is only one entry in *O2Info*$[s_i]$. *size*$(\mathit{O2Info}[s_i][o_j])$ is the second of the two numbers that (s_i, o_j) is mapped to, and indicates the number of bytes in memory that would have been saved had o_j been garbage collected right after it was created. We count this as the sum of the sizes of the objects that are reachable from o_j right after it was created, excluding the sizes of the objects that were pre-existing before o_j was created. In our implementation, we treat all objects reachable from the parameters to the constructor invoked at site s_i as pre-existing objects. In Figure 3 the objects to be counted towards the size of o_j are shown shaded.

Actions during object allocations: When an object o_j is created at an allocation site s_i, the key activity is to see if there is any object that has already been created at this site that is isomorphic to o_j. That is, we look for an entry *O2Info*$[s_i][o_k]$ such that o_j and o_k are isomorphic. This is inherently an expensive operation, but is optimized to some extent by using the fingerprint of o_j and the fingerprints of the objects that are used as indexes in the map *O2Info* as a filter to reduce unnecessary isomorphism checking. If such an entry *O2Info*$[s_i][o_k]$ is found, then we simply increment *freq*$(\mathit{O2Info}[s_i][o_k])$. Otherwise, we create a new entry *O2Info*$[s_i][o_j]$, set its *freq* component to 1, and set its *size* component to the total sizes of fresh objects reachable from o_j as discussed earlier.

Emitting estimates at the end of the run: For any allocation site s_i, let R_i represent the set of root objects allocated at site s_i that are the first elements of their respective equivalence classes. At the end of the run, the instrumentation code calculates the estimated savings (in bytes) for each tracked allocation site s_i as follows:

$$\sum_{o_k \in R_i} \begin{array}{c} (\mathit{freq}(\mathit{O2Info}[s_i][o_k]) - 1) * \\ \mathit{ratioLL}_i * \mathit{size}(\mathit{O2Info}[s_i][o_k]) \end{array}$$

ratioLL$_i$ (read as "ratio of long-lived objects") is an estimate of the ratio of objects created at s_i that are long-lived. This information can be readily obtained at the end of (an uninstrumented version of) the program using profilers such as *hprof*. The calculation that was presented above can be summarized as follows. For each equivalence class of objects (wrt isomorphism) that were created at s_i, the savings due to sharing of long-lived objects in this class is the product of (1) the number of objects in the class except its representative element times the proportion of these objects that are estimated to be long-lived, and (2) the amount of memory to be saved if these objects were to be collected soon after it is created.

The estimate for a site can also be expressed as a percentage of the total size of the *tenured generation* at the end of the run. The tenured generation at the end of the run is expected to consist mostly of long-lived objects.

6.3 Implementation details

Our implementation is based on the *JavaAssist* [12] bytecode instrumentation framework. We instrument the application code as well as third-party libraries (i.e., *jars*) on which the application is dependent. Therefore, we compute estimates for allocation sites in the application as well as in third-party libraries. For the sake of efficiency we do not instrument the JDK core libraries. This could cause some writes or identity operations on application-level objects to be missed, and hence could potentially introduce unsoundness. To mitigate this partially we use a manually constructed list of JDK core library methods that can cause safety issues, and treat any object that is being passed to any of these methods as a non value object. We use the *Redis* in-memory key-value store to maintain the data structures used by the instrumented run efficiently. For the data structures that are indexed by objects (i.e., *O2Sites* and *O2Info*), we use *object IDs* to represent objects. We construct an ID for an object by combining the values returned by the *identityHashCode()* method when applied on this object as well as on the corresponding "class" object.

While we generally treat address comparisons as causes of unsafeness, we ignore address comparisons in user-defined `equals` methods. In our observation most of these address comparisons turn out to be benign.

Another common idiom that we treat specially is the one wherein certain fields of newly-allocated objects are initialized in statements that follow the allocation site that creates the object. This idiom causes many benign allocation sites to be called non VO sites (due to the mutation in the above-mentioned statements). In order to partially offset this limitation we ignore mutations to any object that are carried out in the method that contains the allocation site that created the object. These mutations are frequently not a hindrance to introducing a cache, for the following reason: in the refactored code one could check whether the newly allocated object is in the cache at the point *after* the statements that set these fields, rather than at the point after the allocation site. Finally for the sake of efficiency we do not do deep isomorphic check for objects of type belonging to the *java.lang* package. Instead we rely on the equals method which we assume is implemented correctly for the classes in this core package.

7. EVALUATION

In this section we evaluate our tool on a number of software benchmarks, as shown in Table 1, and answer several research questions.

In Table 1 the first column indicates the name of the benchmark, the second column indicates the total number of allocation sites in the benchmark, including ones in the third-party libraries (excluding the JDK core libraries), the third column indicates the subset of these allocation sites that are in the third-party libraries, while the fourth column indicates the total number of bytecode instructions in the application code as well as third-party libraries. We distinguish the application code from the third-party library code using simple patterns on the package names, which may not be fully accurate.

System Name	Total allocation sites	Allocation sites in libraries	Bytecode instructions
antlr	2496	0	105596
chart	5355	4395	451719
bloat	2958	0	121912
fop	17897	14591	184699
luindex	1298	0	77816
pmd	11201	9476	545766
hsqldb	4590	443	197981
xalan	7110	5250	151134
Apache fop (version 1.1)	14293	0	537711
pdfbox (version 0.73)	2265	89	76002

Table 1: Benchmarks considered for evaluation.

The first eight benchmarks (i.e., upto xalan) belong to the DaCapo 2006 [4] benchmark suite. There are two reasons why did not use the more recent 2009 version of this suite: (1) A previous related approach [21] used this version, and we wanted to compare our results with theirs. (2) We had technical difficulties in instrumenting and running the 2009 benchmarks. We have omitted *eclipse*, which is a part of the DaCapo suite, from our studies as we could not instrument and run it successfully.

In order to test whether our tool finds allocation sites that developers consider important, we searched the check-in commit comments of various open-source projects using *Krugle* [14] to identify employment of caches by developers. We then manually identified allocation sites in these systems that use caching to share long-lived objects. Following this process we found two relevant benchmarks, fop (version 1.1) and pdfbox (version 0.73). There are 14 sites where developers have employed (hash-consing style) caching for sharing objects in fop (version 1.1), and one site in pdfbox. Note that this version of fop is different from the one that is part of the DaCapo suite, which is the version 0.20.5. Since we use both these versions in our evaluation, for convenience we refer to the DaCapo version simply as "fop" and version 1.1 as "Apache fop".

For the DaCapo benchmarks we used the "default" test input provided for each benchmark. We did not use the "large" inputs, because in some of the benchmarks the "large" run turns out to be nothing but a sequence of separate runs over smaller inputs. Since at the end of each of the smaller runs all long-lived objects become dead, for our purposes the "large" run is not appropriate.

For each of the two non-DaCapo systems we needed to select a test input. For Apache fop we used a real 104-page document [9]. For pdfbox we used *cweb.pdf*, which is a 28-page document provided along with the pdfbox distribution, as the test input. We ran pdfbox in the "text extraction" mode.

We ran our experiments on a Linux desktop with an Intel Core 2 2.4 GHz Quad Core processor and 4 GB of RAM. We use the standard Oracle HotSpot JVM with the default options. Detailed artifacts corresponding to all our experi-

System	H	E_b	$E_{\%}$	T_o	T_1	T_2
	(KB)	(KB)		(sec)	(min)	(min)
antlr	7093	1035	16%	11	13	5
chart	12199	986	8%	27	31	14
bloat	8197	530	6%	14	580	8
fop	7215	2658	37%	3	6	4
luindex	9437	1248	13%	39	300	21
pmd	15979	2167	14%	19	517	18
hsqldb	76970	6762	9%	9	52	4
xalan	16523	1503	9%	8	40	4
Apache fop	75680	5102	7%	6	73	5
pdfbox	5119	1667	33%	2	18	3

Table 2: Estimated savings in tenured heap for top-20 sites

ments are available from the home page of the second author of this paper.

7.1 *RQ 1: What is the estimated total savings in heap by introducing caches in real benchmarks?*

Table 2 shows the total estimated savings produced by our tool considering the top-20 sites (ranked by estimated savings per site) for each benchmark. The second column (H) of the table indicates the size of the tenured generation at the end of the run, the third column E_b shows the total estimated savings for the top 20 sites (which we calculate as the sum of the estimates for the individual sites), while the fourth column $E_{\%}$ expresses the value in the third column as a percentage of the value in the second column. It can be observed that all systems present significant opportunities for memory reduction, ranging from 6% of the tenured heap (bloat) to 37% (fop).

An interesting observation we made is that of the top-20 sites in each benchmark, the number of sites inside third-party libraries varies from zero in several benchmarks to 19 sites in the case of pmd.

7.2 *RQ 2: What is the overhead of the analysis?*

The fifth column (T_o) of Table 2 indicates the running time (in seconds) of the benchmark without any instrumentation, the sixth column (T_1) indicates the running time (in minutes) of Phase 1 of our analysis, while the seventh column (T_2) indicates the running time (in minutes) of Phase 2 of our analysis.

The running time overhead of the analysis is very high, ranging from approximately 98x for antlr and chart to approximately 2520x for bloat. Part of the overhead is due the need to recursively visit all reachable objects from the "root" object at allocation sites; however, the overhead due to this reduces as sites get removed from the *Tracked* set. However, there is instrumentation overhead at all *write* operations, and this, unfortunately, does not reduce as the run progresses.

We feel that the overall running time, which in the worst case corresponds to an overnight run (e.g., for bloat and pmd), is tolerable. We also feel it is commensurate with the benefit provided, namely, an actionable report that contains estimates of user-observable memory savings potential at highly profitable allocation sites. Isomorphism analysis is inherently expensive; for instance, the Cachetor tool [17], which measures isomorphism (among temporary objects), incurs overhead of 567x on antlr, 317x on pmd, and 287x on luindex. Their overheads are on average lower than ours as they don't actually construct precise equivalence classes for all objects from a site, which is required to produce direct quantitative estimates of memory savings.

Site manually cached	Rank
CommonBorderPaddingBackground.java:351	1
CommonHyphenation.java:114	3
CommonFont.java:123	4
EnumNumber.java:54	5
EnumProperty.java:97	8
StringProperty.java:107	9
FixedLength.java:84	13
FontFamilyProperty.java:55	14
NumberProperty.java:158	26
LengthProperty.java:56	27
NumberProperty.java:178	NP
NumberProperty.java:148	NP
ColorProperty.java:84	NP
CharacterProperty.java:72	NP
PDFontFactory.java:106	Non-VO

Table 3: Positions of developer-cached sites in tool output

We also evaluated our tool in a different mode, wherein the developer would like to estimate savings for a *specific* allocation site that they suspect might give good memory reduction upon caching. In this mode, in Phase 1 of the tool's run, the set *Tracked* is initialized to contain only the site selected by the developer. If a developer happens to select a site that is either unsafe or not profitable, Phase 1 takes between 5 to 70 times less time than when all sites are being tracked (depending on the benchmark). This is because such sites get removed early from the set *Tracked*, leading to immediate termination. However, if the selected site happens to be a profitable VO site, our observation was that the running time was not much less than when all sites are being tracked.

7.3 *RQ 3: Does the approach detect sites that developers want to cache?*

Table 3 shows information about the 14 sites in Apache fop where developers have introduced caches to share long-lived objects (the first 14 rows in the table), and the one site in pdfbox where the same was done (the final row). For each site we first indicate in the table the file name and line number at which it is located.

In order to determine whether our tool identifies these very sites in its output we made code changes to disable all the caches mentioned in Table 3, and than ran our analysis on the thus modified versions of the two benchmarks. The second column of Table 3 shows the rank of each of these sites in our tool's output (when sorted in descending order of estimated savings).

We discuss the sites in Apache fop first. Ten of the 14 sites (the ones in the first Ten rows of the table) were determined to be profitable by Phase 1 of our tool, and were computed an estimate for by Phase 2. Four other sites were removed from consideration in the Phase 1 itself. These five sites are marked "NP" in the table. This indicates that our tool has high "recall" of sites that developers think are worth caching.

Interestingly, the developers have not employed caches at the sites that occupy ranks 2, 6, and 10 in our tool's output. These sites have estimated savings of 1.7%, 0.7%, and 0.5%, respectively. This is evidence that developers can potentially miss profitable sites in the absence of good tool support for identifying such sites.

Coming to pdfbox, the only site that was cached by the pdfbox developers (see the last row in Table 3) was identified as a non-VO by our tool. The objects created at this site were being mutated during the run. As we noted in Section 1 developers subsequently removed this caching (in the subsequent version 1.7.0 of the project) in response to a user's bug report. This instance illustrates clearly the importance of ensuring safety while introducing caches, and also the difficulty of doing so without good tool support.

7.4 *RQ 4: What is the estimation accuracy of our tool?*

For this study, we considered 19 sites from four benchmarks. Fourteen of these sites are in Apache fop, and are the ones in the first fourteen rows in Table 3. The caches at these sites already exist (were introduced by the developer). Therefore, we (1) ran the tool with caches disabled at all sites to produce an estimate for each site, and (2) for each site ran the original (uninstrumented) program, with the cache disabled first and then enabled, to determine the actual savings in final tenured heap due to the cache at that site. Of the remaining five sites, two are from antlr, two are from pdfbox, and one is from chart. At these sites there were no developer-introduced caches. Therefore we had to introduce caches ourselves at these sites.

Due to space constraints we do not show the estimated savings and actual savings at all these sites. At one extreme there was a site where the estimated savings was less than the actual savings by as much as 2.8% of the final tenured heap. On 16 out of the 19 sites the estimate was either less than the actual savings, or was above the actual savings by up to 0.1% of tenured heap. On the remaining three sites, the largest deviation was on a site where the estimate was above the actual savings by 1.53% of tenured heap. The *root mean square* (RMS) of these deviations when they are expressed as percentages of the tenured heap works out to 1.08%. We feel that expressing these deviations as a percentage of final tenured heap is more useful than expressing them as a percentage of actual savings, because developers ultimately care about savings in the final tenured heap.

We investigated more closely the two extreme sites mentioned above. The site where the estimate was less than the actual savings by 2.8% of the final tenured heap is the site *CommonFont.java:123* in Apache fop, which we happened to show in Listing 1. The final tenured heap size in the run of the original code with caches at all fourteen sites disabled is 78269 KB. After introducing the cache at the above site, the heap size reduced to 75507 KB, which is a reduction of 3.5% in tenured heap. As illustrated in Section 1.1, the estimated savings for this site is 567 KB (0.7% of tenured heap). The estimate is low because, as discussed in Section 6.2, objects that were passed in as parameters to the constructor and that are reachable from the root object returned from the site are not included in the estimate. However, in this particular case the parameter objects are not referred to by any other object or variable after the root object gets constructed. Therefore, if this site is cached, these parameter objects would also get garbage collected whenever a root object is thrown away (due to a cache hit).

The other extreme case is the site *FontFamilyproperty.java:55* in Apache fop, where the estimated savings is above the actual savings by 1.53% of the tenured heap. On this site the estimated savings was 92.6 KB, while the actual savings was -1107 KB, indicating an *increase* in memory requirement due to the overhead of caching.

Input	Estimated savings	Actual savings
104 page	5.6%	11.1%
10 page	3.9%	7.5%
3 page	8.5%	6.5%

Table 4: Variation of savings across different inputs

System	# Intersection	# Unsafe sites	Estimated savings due to safe sites (KB)	Estimated savings (%)
antlr	0	11	8	0%
bloat	0	10	0	0%
luindex	3	5	0.5	0%
pmd	2	9	175.7	1%
xalan	0	3	0	0%
fop	0	4	3.7	0%

Table 5: Comparison with Xu's[21] approach

7.5 *RQ 5: What is the sensitivity of estimated and actual savings to program input?*

The objective of this RQ is to measure the sensitivity of the estimated as well as actual savings from a selected set of sites to the test input that is used. For the 14 sites in Apache fop that are listed in Table 3, Table 4 shows both the sum of the estimated savings from these sites (with caches at all these sites disabled) as well as actual savings when the caches are enabled (simultaneously) at all these sites, on three different test inputs. The first input happens to be the input that was used to answer all other RQs above. This table indicates that both the estimated and the actual savings for a given set of sites can vary across different test inputs. The estimation error (i.e., estimate compared to actual) also varies across inputs. As part of future work we intend to explore how to select a suitable set of test inputs for each program and produce aggregated estimates from these inputs.

On a related note, we also measured the running time of Apache fop with all the caches disabled and enabled, respectively. On the 104-page document the running time with the caches enabled decreased by about 1%, while on the other two documents it increased by about 5%.

7.6 *RQ 6: How does our tool compare with a previous tool?*

We requested the authors of the previous work [21, 17] on finding allocation sites that create a lot of isomorphic objects, but predominantly temporary objects, to share their tool output with us. Our objective was to see if our top-20 lists differed significantly from their top-20 lists for the DaCapo 2006 benchmarks. We are thankful to the authors for sharing with us the lists from their first approach [21]. We provide a summary of this comparison in Table 5. The second column of this table shows the sites that are in the intersection of their list and our list. It is notable that these numbers are very low. The third column indicates the number of sites in their top-20 list which our approach detects as unsafe (for sharing). It is notable, though, that since most of these sites create temporary objects, which anyway don't give savings upon sharing, the inability to employ sharing here is not a concern in practice. The fourth column indicates the total estimated savings calculated by our tool if sharing were to be employed at the safe sites in their list. Note that these estimates are very low, indicating these sites indeed create mostly temporary objects. The last column shows the same total estimate as a percentage of the tenured heap size.

In summary these results show clearly that our technique fully complements previous techniques that aim predominantly at finding sites that create isomorphic temporary objects.

8. LIMITATIONS

One of the threats to validity of our empirical studies is that our estimates could include savings from potentially unsafe sites that were not found unsafe under the test inputs that were considered. To mitigate this partially, for the DaCapo benchmarks, we did check if any of the top-20 sites in any of the benchmarks that was found safe in the "default" run was found unsafe under the corresponding "small" or "large" runs. This turned out to be not the case. Some related statistics about the coverage of these runs are as follows: chart was the benchmark that had the least coverage, with 7%, 7%, and 10% *line coverage* on the small, default, and large inputs, respectively, while antlr had the highest coverage at 24%, 46%, and 46%, respectively. Nevertheless, to mitigate this threat, as part of future work we plan to employ static analysis to verify safety of sites across *all* runs.

The total estimated savings that we show in Table 2 could be biased or inaccurate, for the following reasons: (1) These are obtained using a single test run per benchmark, (2) Total actual savings when a set of sites are cached simultaneously could be less than the sum of individually estimated savings for these sites, and (3) The various assumptions made in Sections 5 and 6 potentially impact the accuracy of the estimates produced by our tool.

The scalability of the tool needs to be improved; as part of this, static analysis could potentially be used to reduce instrumentation overhead wherever feasible.

9. RELATED WORK

The previous literature that is most closely related to our work is that by Marinov and O'Callahan [15]. Marinov and O'Callahan recognized the problem of excessive usage of memory by co-existing isomorphic objects. They introduced the notion of safely replacing one object by another at run time (i.e., object sharing), and gave a dynamic analysis approach to find the maximum extent of co-existing isomorphic objects during a run. Their results are in effect a limit study, because it is in general not possible to refactor code to remove all the isomorphism redundancy that they detect. Our work is a logical extension of their work, in that our analysis identifies the extent of redundancy that is *practically remediable* by introducing caches at allocation sites. In other words, we provide actionable reports to developers. Our empirical results also give a feel for the extent of remediable isomorphism redundancy in real benchmarks.

Xu et al. [21, 17] focus on reducing running time of programs by enabling *reuse* of (dead) temporary objects. For each allocation site they calculate a metric that represents the proportion of all objects allocated at that site that belong to its single largest equivalence class (wrt isomorphism). This metric indicates the potential for reuse at the site. (They also compute reuse metrics at instruction level and at method level.) Our objective, in contrast, is to save memory by *sharing* (live) existing objects. Our outputs are estimates of a user-observable benefit, namely, quantitative memory savings, and are hence readily actionable by developers. Finally, our analysis approach is quite different from theirs, and specifically does not required a modified JVM.

In addition to the work of Marinov and O'Callahan [15], there are other previous approaches about optimizing memory usage of programs. Mitchell et al. [16] and Chis et al. [7] propose approaches to help developers identify uses of data structures that are overly memory intensive, and that could possibly be remediated using idiomatic code improvements. On the same topic of memory inefficiency, there is a large body of work on using modified JVMs during final deployment of code to compress the heap or reduce the space overhead of objects [6, 1, 5, 19]. Kawachiya et al. [13] propose a scheme to use a modified JVM at deployment time to de-duplicate co-existing string literals. A recent approach by Infante [11] addresses the problem of identifying memory savings opportunities by employing object caches in the context of the Smalltalk programming language. However, this approach is described only at a high-level, making it hard to be compared and contrasted with our approach. The approach does not produce quantitative estimates. Finally, it has been evaluated only in the context of a limited case study.

Finally, there is a rich body of literature on analyses for detecting as well as remediating performance inefficiencies due to factors such as creation of too many temporary objects [8, 20, 3, 21], creation and use of low utility data structures [26, 23], chains of pure copies [22], and use of only a small subset of objects created [27]. In contrast to these approaches the problem we address is to remediate excessive use of memory.

10. CONCLUSION

In this paper we presented an approach to identify opportunities to reduce the memory requirement of Java programs by introducing object sharing. The approach addresses the challenges in identifying suitable opportunities, namely, determining the safety of introducing sharing at an allocation site, identifying sites that potentially create large numbers of isomorphic long-lived objects, and estimating the memory saving without actually having to introduce caches. We have empirically validated the approach and answered several research questions pertaining to it by applying an implementation of the approach on a set of real benchmarks.

11. REFERENCES

[1] C. S. Ananian and M. Rinard. Data size optimizations for Java programs. In *Proc. ACM Conference on Language, Compiler, and Tool for Embedded Systems (LCTES)*, pages 59–68, 2003.

[2] A. W. Appel and M. J. R. Conçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, 1993.

[3] S. Bhattacharya, M. G. Nanda, K. Gopinath, and M. Gupta. Reuse, recycle to de-bloat software. In *European Conf. on Object-Oriented Programming (ECOOP)*, pages 408–432, 2011.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. Conf, on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, 2006.

[5] G. Chen, M. Kandemir, and M. J. Irwin. Exploiting frequent field values in java objects for reducing heap memory requirements. In *Proc. Int. Conf. on Virtual Execution Environments (VEE)*, pages 68–78, 2005.

[6] G. Chen, M. Kandemir, N. Vijaykrishnan, M. Irwin, B. Mathiske, and M. Wolczko. Heap compression for memory-constrained Java environments. In *Proc. Conf. on Object-oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 282–301, 2003.

[7] A. E. Chis, N. Mitchell, E. Schonberg, G. Sevitsky, P. O'Sullivan, T. Parsons, and J. Murphy. Patterns of memory inefficiency. In *European Conf. on Object-Oriented Programming (ECOOP)*, pages 383–407, 2011.

[8] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *Proc. Int. Symp. on Foundations of Software Engineering (FSE)*, pages 59–70, 2008.

[9] P. Friere. Test document. www.arvindguptatoys.com/arvindgupta/oppressed.doc, July 2014.

[10] E. Goto. Monocopy and associative algorithms in extended Lisp. Technical Report TR 74-03, University of Tokyo, 1974.

[11] A. Infante. Identifying caching opportunities, effortlessly. In *Companion Proc. Int. Conf. on Software Engineering (ICSE)*, pages 730–732, 2014.

[12] Javassist, www.javaassist.org.

[13] K. Kawachiya, K. Ogata, and T. Onodera. Analysis and reduction of memory inefficiencies in java strings. In *Proc. Conf. on Object-oriented Programming Systems Languages and Applications (OOPSLA)*, pages 385–402, 2008.

[14] Krugle. http://www.krugle.com/.

[15] D. Marinov and R. O'Callahan. Object equality profiling. In *Proc. Conf. on Object-Oriented programing, Systems, languages, and applications*, 2003.

[16] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *Proc. Conf. on Object-oriented programing, Systems, languages, and applications (OOPSLA)*, pages 245–260. ACM, 2007.

[17] K. Nguyen and G. Xu. Cachetor: detecting cacheable data to remove bloat. In *Proc. Foundations of Software Engineering (FSE)*, pages 268–278, 2013.

[18] Oracle. Java SE 6 Hotspot Virtual Machine Garbage Collection tuning. http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html, March 2014.

[19] J. B. Sartor, M. Hirzel, and K. S. McKinley. No bit left behind: the limits of heap data compression. In *Proc. International Symposium on Memory management*, pages 111–120. ACM, 2008.

[20] A. Shankar, M. Arnold, and R. Bodik. Jolt: lightweight dynamic analysis and removal of object churn. In *Proc. Conf. on Object-oriented Programming Systems Languages and Applications (OOPSLA)*, pages 127–142, 2008.

[21] G. Xu. Finding reusable data structures. In *Proc. Int. Conf. on Object oriented Programming Systems Languages and Applications (OOPSLA)*, pages 1017–1034, 2012.

[22] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: profiling copies to find runtime bloat. In *Proc. Conf. on Programming Language Design and Implementation (OOPSLA)*, pages 419–430, 2009.

[23] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *Proc. Conf. on Programming Language Design and Implementation (OOPSLA)*, pages 174–186, 2010.

[24] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky. Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proc. FSE/SDP Workshop on Future of Software Engineering Research*, pages 421–426, 2010.

[25] G. Xu and A. Rountev. Precise memory leak detection for java software using container profiling. In *Int. Conf. on Software Engineering (ICSE)*, pages 151–160, 2008.

[26] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *Proc. Conf. on Programming Language Design and Implementation (OOPSLA)*, pages 160–173, 2010.

[27] D. Yan, G. Xu, and A. Rountev. Uncovering performance problems in Java applications with reference propagation profiling. In *Proc. Int. Conf. on Software Engineering (ICSE*, pages 134–144, 2012.