Finding Reusable Data Structures

Guoqing Xu
University of California, Irvine
guoqingx@ics.uci.edu

Abstract

A big source of run-time performance problems in large-scale, object-oriented applications is the frequent creation of data structures (by the same allocation site) whose lifetimes are disjoint, and whose shapes and data content are always the same. Constructing these data structures and computing the same data values many times is expensive; significant performance improvements can be achieved by reusing their instances, shapes, and/or data values rather than reconstructing them. This paper presents a *run-time technique* that can be used to help programmers find *allocation sites that create such data structures* to improve performance. At the heart of the technique are three reusability definitions and novel summarization approaches that compute summaries for data structures based on these definitions. The computed summaries are used subsequently to find data structures that have disjoint lifetimes, and/or that have the same shapes and content. We have implemented this technique in the Jikes RVM and performed extensive studies on large-scale, real-world programs. We describe our experience using six case studies, in which we have achieved large performance gains by fixing problems reported by our tool.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management, optimization, run-time environments; F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages—Program analysis; D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids

General Terms Language, Measurements, Performance

Keywords object reuse, data structure encoding, memory management, performance optimization

1. Introduction

Large-scale object-oriented applications commonly suffer from run-time performance problems that can lead to significant degradation and reduced scalability. Experiences [28, 39, 44] show that many such bottlenecks in Java applications are caused by chronic run-time bloat, a term that is used to refer to excessive memory usage and computation to accomplish relatively simple tasks. Although modern compilers offer sophisticated optimization techniques, these techniques cannot effectively remove bloat, because large applications often lack hot spots [42]—inefficient operations exist throughout the program, making it extremely difficult for the (intraprocedural) compiler analyses to find and remove. In addition, traditional optimizers detect opportunities based primarily on control profiling, while large bottlenecks are often closely related to data activities [39]. Finding these bottlenecks requires the invention of novel techniques that can better understand how data is computed and aggregated during the execution of a real-world program.

In many cases, systemic bloat stems from performance-unconscious design/implementation choices, which are sometimes encouraged by the culture of object-orientation. For example, programmers are taught to freely create objects even for extremely simple tasks, taking for granted that the object creation and garbage collection (GC) are entirely free. However, this is only true for applications that need to process a very small amount of data. For large allocation-intensive applications (such as web graph processing systems and social networks) that often have massive-scale data to process, the excessive creation of objects can cause the heap to quickly grow, leading to significantly increased GC effort and reduced scalability. In a managed language (such as Java) that does not support explicit memory management, one way to effectively reduce the allocation and GC pressure is to *reuse existing objects in the heap for certain repeated tasks instead of reclaiming them and creating new objects*, because, in many cases, objects created for different tasks (e.g., iterations of an event loop, database transactions, etc.) have completely disjoint lifetimes and can never be used simultaneously during the execution.

Poor performance does not come only from excessive object creation and garbage collection; creating one single object often involves the creation of a set of other objects,

```
class SSAGraph{
  void findEquivalentNodes() {
     for(CFG cfg: cfgs){
       cfg.visit(new TreeVistior(){
         void visitBlock(Block b){b.visitBlockNodes();}
       });
     }  ...
  } ...
}
class Block{
  void visitBlockNodes(){
    for(Statement s: statements){
       s.visit(new NodeVisitor(){
         void visitExpression(Expr e){e.visitChildren();}
       });
    } ...
  } ...
}
class Expr{
  void visitChildren(){
     for(Expr child: children){
       child.visit(new ExprVisitor(){...});
     } ...
  } ...
}
                         (a)

for(String dateStr : dates){
    SimpleDateFormat sdf = new SimpleDateFormat();
    try{
         Date newD = sdf.parse(dateStr);
                 ...
    }catch(...) {...}
}
                         (b)
```

Figure 1. Real-world examples of reusable data structures: (a) Iteratively visiting a graph using the visitor pattern; and (b) creating a `SimpleDateFormat` object per iteration of the loop whose shape and data are completely independent of the loop.

the computation of their data content, and the combination of these objects into a valid data structure. Repeating this process is unnecessarily expensive if the reference structure and/or the data content are unchanged. Hence, additional performance benefits may be obtained if we reuse *not only object instances, but also their reference relationships and data content*, because the heavy value computation can be avoided.

Figure 1 (a) and (b) show two real-world examples that illustrate, respectively, the problems of excessive object creation and of frequent construction of data structures with the same shapes and content. The code snippet in Figure 1 (a) is adapted from bloat, a static bytecode optimization framework for Java. Method `findEquivalentNodes` in class `SSAGraph` identifies expressions of equivalent types in a set of control flow graphs (CFGs) by iteratively visiting program entities in them, such as basic blocks, statements, and expressions. To implement this traversal, bloat declares an anonymous visitor class for each type of program entity and creates an object of the class to visit each entity object. However, many of these visitor objects are created in nested loops. The numbers of their instances can grow exponentially with the layer of loop nesting, putting significant run-time pressure on the object allocator and GC.

In this example, it is easy to observe that the lifetimes of instances of a visitor class are entirely disjoint—the instances are never needed simultaneously during the execution. To optimize this case (including visitors not shown in Figure 1 (a)), we implemented a singleton pattern for each visitor class and used a single visitor object to visit all program entities of the same type throughout the execution. The content of this object is *reset* every time it is used to visit a different entity. This has led to a 37.3% running time reduction, a 16.7% reduction on the number of GC invocations, and a 11.1% reduction on the peak memory consumption (on Sun Hotspot 1.6.0_27).

Figure 1 (b) shows an example (extracted from an IBM application) where a more aggressive optimization can be applied. In this example, a new `SimpleDataFormat` object is created in each iteration of the loop to parse a string into a `Date` object. Note that the data structure rooted at this object is completely loop-invariant: it cannot escape the iteration where it is created, and its shape and data content never change. We can simply hoist this allocation site out of the loop in order to reuse (1) the object instance, (2) all objects reachable from it and their reference relationships, and (3) their data content. These *reusable data structures* are the focus of this paper—we develop a novel dynamic analysis technique that can help developers find such data structures in a program during its representative runs.

Focus on allocation sites The first challenge in this work is how to find an appropriate static object abstraction so that reuse opportunities can be detected among the set of run-time objects that map to the same abstraction. In this paper, we focus on allocation sites, that is, our technique compares data structures created by the same allocation site to determine whether or not there exist opportunities with this allocation site. Our tool eventually ranks and reports allocation sites where reuse opportunities can be found. For example, for the programs in Figure 1 (a) and (b), our analysis would report the allocation sites that create visitor objects and that create `SimpleDateFormat` object, respectively.

While there are many other types of abstractions that may also be considered for object reuse, we find that focusing on allocation sites achieves the right balance between the amount of optimization opportunities that can be detected and the difficulty of developing fixes. Considering a coarser-grained abstraction such as a class can miss reuse opportunities that exist in individual instantiations of the class, while considering a finer-grained abstraction such as an allocation site under a specific calling context [4, 11, 21, 35] may lead to the reporting of problems that are difficult to fix. It is often not easy to understand how to reuse a data structure only under certain calling contexts. Note that simply ranking allocation sites based on their execution frequencies cannot reveal reuse opportunities. For example, in a typical large application, the most frequently executed allocation site is one in `HashMap.put` that keeps creating `Map$Entry` objects to

```
for(int i = 0; i < N; i++) { A a= new A(); a.f = 1; B b = new B(); b.g = 1; a.link = b; …;     a.f(); //use a }
```
(a) Original program

```
for(int i = 0; i < N; i++) {
    A a= A.getInstance();
    a.f = 1;
    B b = B.getInstance();
    b.g = 1;
    a.link = b;
    …; a.f();
}
static A instance = null;
static A getInstance() {
    if(instance == null)
        instance = new A();
    return instance;
}
static B getInstance() {…}
```
(b) Optimization based on instance-reusability

```
for(int i = 0; i < N; i++) {
    A a= A.getInstance();
    a.f = 1;
    a.link.g = 1;
    …; a.f();
}
static A instance = null;
static A getInstance() {
    if(instance == null) {
        instance = new A();
        B b = new B();
        instance.link = b;
    }
    return instance;
}
```
(c) Optimization based on shape-reusability

```
for(int i = 0; i < N; i++) {
    A a= A.getInstance();
    …; a.f();
}
static A instance = null;
static A getInstance() {
    if(instance == null) {
        instance = new A();
        instance.f = 1;
        B b = new B();
        instance.link = b;
        b.g = 1;
    }
    return instance;
}
```
(d) Optimization based on data-reusability

Figure 2. A simple program and its optimizations based on the three reusability levels.

store newly-added keys and values. Objects created by this allocation site are not reusable at all. Hence, it is necessary to develop new metrics for allocation sites that can strongly correlate with their reusability.

Levels of reusability In order to fully exploit reuse opportunities in a program, we classify allocation sites of interest into three categories, each at a different reusability level.

The first category ($\mathcal{I}$) includes allocation sites that exhibit *instance reusability*. For each such allocation site, the number of its instances needed simultaneously during the execution is very small (e.g., bounded by a constant value). If this allocation site is frequently executed, we may cache its instances and reuse them to reduce the allocation/GC effort. The second reusability level is *shape reusability*. This level corresponds to a category ($\mathcal{S}$) of allocation sites such that not only their instances can be reused but also the run-time shapes of the data structures rooted at these instances are unchanged. By reusing both the instances and the reference relationships among them (i.e., object graph edges), we may get additional performance benefits from saving the effort to form the shape many times. The highest reusability level is *data reusability*. Each allocation site in this category ($\mathcal{D}$) creates data structures that are completely equivalent to each other—their instances, shapes, and data values can all be reused.

Figure 2 shows a simple program with redundancies and the possible optimizations based on these three reusability definitions. In the program, the loop keeps constructing the same data structure (rooted at an object of type `A`). At the level of instance reusability (shown in Figure 2 (b)), an optimization can employ a singleton pattern for each class, and reuses the instances of `A` and `B` to ameliorate the high object creation and GC pressure. As a further step, a more aggressive optimization can additionally reuse the reference edge between the two objects, based on the observation that the shape of the structure is unchanged (shown in Figure 2 (c)). Finally, once we find that the data values written into $a.f$ and $b.g$ are also independent of the loop, we can develop an optimization that reuses the entire data structure including the instance, the shape, and the data (shown in Figure 2 (d)). In this example, it is possible to further optimize the program by hoisting the call `A a = A.getInstance()`. However, how to hoist an allocation site is out of the scope of this paper. In addition, in a real-world program, an allocation site that exhibits high reusability may be far (e.g., many calls) away from the main event loop, making it difficult for the developer to hoist it. Our experience shows that a more common and practical way to reuse data structures is to employ singleton patterns, as illustrated in Figure 2.

The relationship among the three categories is $\mathcal{D} \subseteq \mathcal{S} \subseteq \mathcal{I}$. As the reusability level increases (from $\mathcal{I}$ to $\mathcal{D}$), the performance gains resulting from the data structure reuse may also increase. The proposed technique aims to expose opportunities at all the three levels to help a programmer maximize the possibility of improving performance through reusing data structures. Particularly, for each reusability category, we report a ranked list of allocation sites to the developer for manual inspection. Note that this technique does *not* fix problems automatically. The optimizations in Figure 2 are shown only for illustration purposes. They demonstrate typical fix patterns that can be employed by a real-world programmer to reuse data structures. In fact, our experience shows that true problems reported in each reusability category can always be fixed using one or a combination of these patterns. Details of our experiments can be found in Section 4.

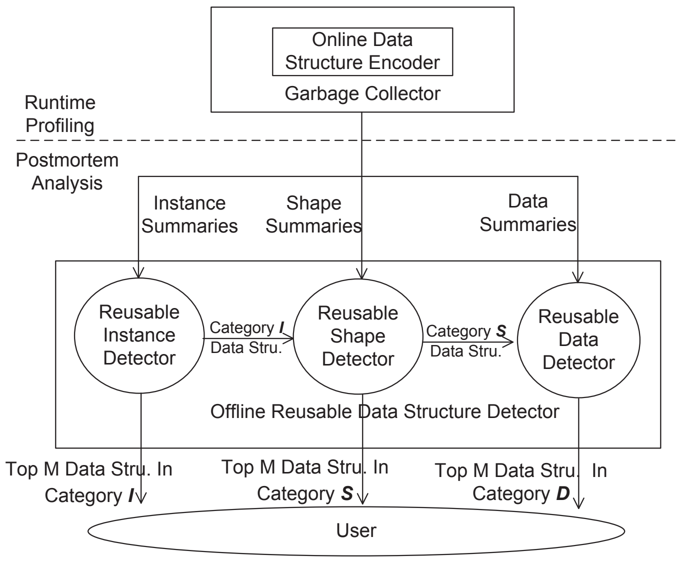

Figure 3. An overview of our technique.

Approximations It is not always possible to precisely understand the three reusability properties for each data structure at run time. For example, finding reusable instances requires the precise identification of object lifetimes, which is very difficult in a managed language like Java because objects do not die immediately after they are no longer used. Their lifetimes are determined by the GC points in the execution. As another example, understanding whether two data structures contain the same data values requires *value profiling* [12], an expensive technique that cannot scale to large applications. In order to improve scalability, we develop an approach to *approximate* each level of reusability instead of attempting to compute precise solutions. In particular, we develop a new metric to approximate instance reusability, and summarize data structure shapes and value content to approximate shape and data reusability. The detailed approximation algorithms are described in Section 2.

Overview The overview of our technique is shown in Figure 3. Our tool has two major components: (1) an *online* data structure encoder that uses various approximations to summarize data structures, and (2) an *offline* reusable data structure detector that compares summaries to find reuse opportunities at the end of the execution (but before the JVM exits). The encoder piggybacks on garbage collection to encode heap data structures based on their allocation sites. The detector consists of three stages to find reusable data structures after all summaries are generated. It starts with scanning all allocation sites and ranking them based on their instance reusability (e.g., instance summaries shown in Figure 3). The top M allocation sites on the list are then reported to the user for manual inspection. M can be given by the user as a parameter to our analysis. A part of this ranked list (whose length is much larger than M) is fed to the next stage—the reusable shape detector will scan only these allocation sites to find those that exhibit high shape reusability. This is because data structures at a certain reusability level must also exhibit reusability at all lower levels in order to be reused. For example, it can be very difficult, if not impossible, to reuse data structures that have the same data values but different shapes and overlapping instances.

We have implemented this technique in Jikes RVM 3.1.0 (http://jikesrvm.org), a high-performance Java-in-Java virtual machine, and successfully applied it to large-scale applications such as Eclipse. The implementation is described in Section 3. Our technique incurs an overall 10.8% running time overhead and a 30.3% space overhead. The detailed execution statistics are reported in Section 4.2. While the overhead is probably too high for production runs, we found it acceptable for performance tuning and debugging. Using our tool, we have identified reuse opportunities in all programs in our benchmark set. Section 4.1 presents six case studies on applications where large improvements (e.g., 37.3% running time reduction and 22% GC time reduction) were achieved from problem fixes. The experimental results strongly indicate that the proposed technique can be adopted in real-world development and tuning to help programmers quickly find optimization opportunities and reuse data structures for increased efficiency.

The main contributions of this work are:

- A three-level reusability definition that aims to help programmers maximize their chances of finding optimizations opportunities.
- A run-time technique that consists of three algorithms to approximate reusability at these levels.
- An implementation of this technique in the Jikes RVM that piggybacks on garbage collection to find reusable data structures.
- Six case studies demonstrating that our technique can help programmers quickly identify reuse opportunities and fix problems for large performance gains.

2. Encoding Data Structures to Approximate Reusability

In this section, we describe our key algorithms that encode data structures to find reuse opportunities. As observed in prior work on heap analysis (e.g., [1, 31]), garbage collection (GC) in a managed language execution is particularly suitable for computing heap-related program properties. Because GC requires a traversal of all live objects in the heap, it is a natural idea to associate an object graph traversal-based analysis with GC so that the analysis can be performed along with the regular GC traversal to avoid the additional run-time cost. In this paper, we employ a similar approach: the encoder (as shown in Figure 3) summarizes heap data structures for each allocation site during GC runs, and the detec-

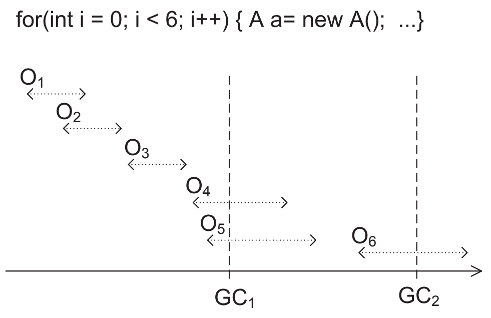

Figure 4. An example illustrating the lifetimes of different instances of the same allocation site.

tor eventually analyzes the generated summaries and finds reusable opportunities.

2.1 Approximating Instance Reusability

Finding an allocation site that exhibits instance reusability requires the understanding of whether the lifetimes of its instances can overlap. The smaller the number of instances whose lifetimes can overlap is, the more likely it is for a developer to cache and reuse instances for this allocation site. Because it is generally undecidable to determine object liveness in a managed language, approximations have to be employed. A typical handling is to use *reachability* to approximate *liveness*—a set of *checkpoints* is scattered over the execution and object reachability is inspected at each checkpoint. If an object is not reachable (from a set of root objects) at a checkpoint, its lifetime is treated as the time distance between this checkpoint and the point when it was created. Very often, GC runs are used as checkpoints because GC necessitates a reachability analysis.

In our analysis, however, using this approximation could cause a large number of objects (created by the same allocation site) to have overlapping lifetimes, leading to the failure of detecting many truly reusable data structures. To illustrate, consider the example shown in Figure 4. We use O_i to denote the i-th object created by the allocation site. At the checkpoint GC$_1$, all the first five objects ($O_1, \ldots, O_5$) have overlapping lifetimes based on this approximation, even though three of them (O_2, O_3, O_4) are completely disjoint. Clearly, this approximation is too conservative to be used in our analysis. As a more precise handling, the Merlin [19] approach may be used to generate an object event trace that can be subsequently analyzed to compute lifetimes of objects. However, performing dynamic whole-heap object tracing and trace analysis can incur a prohibitively large overhead for allocation-intensive applications (e.g., 70-300× slowdown reported in [19] even for relatively small programs). Hence, it is more suitable for an offline analysis than an online analysis such as the one proposed in the paper.

2.1.1 A New Approximation

To alleviate the problem, we propose a new metric to approximate instance reusability. This metric considers, at each GC, the ratio between the number of dead objects and the number of live objects created by the same allocation site. This ratio is referred to as the *DL ratio* in the rest of the paper. For an allocation site, the larger its DL ratio is, the higher instance reusability it exhibits. This is because of the following two reasons: (1) the number of live objects at a GC is an (under-)approximation of the objects that are needed simultaneously from an allocation site. The larger this number is, the less likely the instances of this allocation site are to be reused; and (2) the number of dead objects at a GC is an (over-)approximation of the objects whose lifetimes are disjoint. The higher this number is, the more likely it is to reuse instances created by the allocation site.

To make more sense of this metric, consider the following two extreme cases. In the first case, there is an allocation site that creates completely disjoint instances during the execution. At any GC, the number of live objects for this allocation site is at most 1, and the number of dead objects is at least the number of objects created (since the last GC) - 1. If the number of live objects is 0, we cannot compute a valid DL ratio and thus this information is discarded; otherwise, the DL ratio is a big number that implies high instance reusability.

In the second case, all objects created by an allocation site are stored in a container and are needed simultaneously for a certain task. After the task is done, all of them die together. In this case, the number of live objects at a GC is either the total number of objects created by the allocation site or 0, and the number of dead objects is either 0 or the total number of objects created. If the number of live objects is 0, the information is discarded again; otherwise, the number of dead objects must be 0 and thus the DL ratio is 0, implying a very low chance for reuse. Eventually, the DL ratios computed at all GCs are averaged so that the information loss due to the lack of live objects at some GCs would not have big impact on our analysis outcome.

It is clear to see that using DL ratio computed at a GC to approximate liveness may lead to both false positives and false negatives. For example, in the first case (described above), if the number of live objects is 0, no DL ratio is computed, resulting in a false negative (because the objects are indeed disjoint). In the second case, if GC occurs in the middle of a resource release process where many objects (created by the allocation site) are dead but a few are still live, a big DL ratio may be generated for this allocation site leading to a false positive. However, despite the information loss and the imprecision at one single GC, we found that averaging DL ratios at all GC points for an allocation site can significantly reduce both false positives and false negatives—because GC points are generally independent of each other, it is much less likely that DL ratios computed at different

GCs for an allocation site are affected by the same mistreatment of its object lifetimes. This is the case especially for large-scale and long-running applications that often have a great number of GC runs. A detailed measurement of false positives reported by our tool can be found in Section 4.1.

Example Consider again the example in Figure 4. At GC_1 and GC_2, the DL ratios for the allocation site are 3/2 and 2/1, respectively, making its final DL ratio 1.75. This ratio does not indicate any problem by itself, unless it is compared with DL ratios for other allocation sites. In this program, however, reuse opportunities do exist. For example, in an ideal case, creating 2 instances would be sufficient—at any point in the execution, the maximum number of instances needed simultaneously is 2.

2.1.2 Computing DL Ratios

In order to compute the DL ratio for each allocation site, we need to identify both its live objects and dead objects at GCs. This is done through tagging each object with its allocation site ID. Finding live objects is straightforward: during the GC heap traversal, the number of reachable objects tagged with the same ID is counted and stored into a DL table. This table records, for each allocation site, the numbers of its live objects and dead objects at each GC. These numbers are used to calculate a final DL ratio when a report is about to be generated.

Finding dead objects It is much more difficult to identify dead objects at a GC. These objects cannot be found in a regular object graph traversal because they are unreachable. A naive way of finding such objects is to create a separate whole-heap traversal pass in GC that works after the reachable object graph traversal. In this pass, each heap cell is visited to check if it contains a reference to a valid object that has not been marked as live. However, visiting all cells in a big heap can be expensive and may thus incur a large run-time overhead.

We use a modified reference-counting algorithm to efficiently detect dead objects. Performing this algorithm requires an additional space in each object to store its reference counter. Generally, at each heap store $a.f = b$ (or $A.f = b$), the old object originally contained in $a.f$ (or $A.f$) is retrieved and its reference counter is decremented, and then the reference counter of the object pointed to by b is incremented. If an object's reference counter becomes 0, it is added to a *dead object queue* for further processing.

There are many objects whose references are never assigned to heap locations. They are referenced only by stack variables and die immediately after their containing methods return. If we instrument only heap write statements, these objects would not be added into the queue. To effectively find them, we additionally instrument each heap load $b = a.f$ to check the reference counter of the accessed object: if the reference counter of the object pointed to by a is 0, this object is also added to the queue. It will be removed if its reference is written into a heap location later.

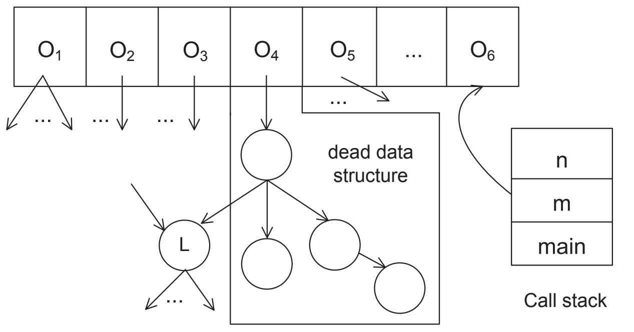

Figure 5. An example of dead object queue.

The only kind of (dead) objects that may still be missing in the queue are those that are never read and written during their lifetimes. Such objects are extremely rare and can be easily optimized away by a compiler optimization (e.g., via dead code removal).

At each GC, the dead object queue contains root (dead) objects from which (almost) all dead objects in the heap can be reached. The queue may also contain some live objects (that are referenced only by stack variables). These objects are filtered out and will not be processed. In the garbage collector, we create a separate pass after the regular object graph traversal to iteratively identify dead objects. When this pass executes, all live objects in the heap have already been visited and marked. This pass then traverses the object graph starting from the (root) dead objects in the queue to find those that are not marked (as live). An example dead object queue is shown in Figure 5. Object O_6 is a live object referenced by a stack variable and is not processed in this pass. There are many objects reachable from O_4, among which only objects not marked 'L' (live) are identified.

2.1.3 Objects v.s. Data Structures

Reporting reuse opportunities for data structures (with multiple levels of objects) can be more useful than doing so for individual objects. To do this, for each GC, we compute DL ratios only for allocation sites whose objects are in the dead object queue. These objects are usually the roots of data structures containing many other objects. To account for the size of each data structure in our metric, the DL ratio for an allocation site a is modified to S * (D/L), where D/L is the original DL ratio described earlier and S is a size measurement of the dead data structures created by a. The value of S is computed by calculating the average number of dead objects (directly and transitively) reachable from a's objects in the queue. Hence, S measures the size of the data structure that has the same liveness property as the root object. This part of object graph may be reused together with the root. It is also the target for our shape and data summarization described later in this section. In the example shown in Figure 5, the dead data structure rooted at O_4 is highlighted in the box, and the value of S for O_4's allocation site is 5.

These new DL ratios are used as instance summaries (shown in Figure 3) to rank allocation sites in the end.

2.2 Encoding Shapes

The second reusability level is shape reusability. The goal of our analysis at this level is to find allocation sites that keep generating data structures with the same shapes (i.e., reference relationships). As described earlier in this section, data structures we are interested in are the *dead* data structures rooted at objects in the dead object queue (shown in the box in Figure 5), because objects in these data structures die together and thus may have the same liveness property. In this stage, we compare the shapes of all dead data structures rooted at objects created by the same allocation site throughout the execution to determine the shape reusability for this allocation site. Two heap data structures are considered to have the same shape if (1) their object subgraphs are isomorphic and (2) the corresponding objects in the two subgraphs are created by the same allocation sites. Allocation sites are considered because it may not be possible to reuse the shape if objects constituting the shape in different data structures are created by different allocation sites.

Determining whether two graphs are isomorphic is known to be NP-complete. What makes the problem even more complicated is that two data structures created by the same allocation site are not always available for comparison, because at the time one data structure exists in the heap, the other one may have already been garbage collected. In order to enable efficient comparison, we compute a shape summary for each dead data structure and record it with its root allocation site in a shape table. This summary encodes both the shape of a subgraph and its allocation site information. Summaries for data structures created by the same allocation site are compared later to determine the shape reusability for the allocation site.

2.2.1 Balanced-Parentheses Tree Encoding Algorithm

While there exist many techniques to encode trees and graphs (primarily in the theory community), their focus is the space efficiency and the ability of quickly performing common data structure operations (such as subtree, children, root, etc.) in the encoded form. On the contrary, our top concern is how to encode the allocation site IDs of the objects in a data structure with its run-time shape.

We have studied a set of related encoding algorithms (e.g., [5, 17, 20, 29]), and found that the balanced-parentheses (BP) encoding algorithm is particularly suitable for our shape summarization. The BP algorithm is proposed by Munro and Raman [29, 30] to efficiently represent binary trees, rooted ordered trees, and balanced parenthesis expressions. This algorithm uses an amount of space within a lower order term of the information theoretic minimum and supports a rich set of navigational operations in constant time. The key idea is to represent a tree containing n nodes with a string of balanced parentheses of length $2n$. A node is represented by a pair of matching parentheses '(' . . . ')', which denote, respectively, the starting point (i.e., '(') and the finishing point (i.e., ')') of a depth-first traversal of the subtree rooted at this node. All descendants of the node are encoded in order between its matching parentheses. A detailed example of this encoding is illustrated in Figure 6 (a). It is straightforward to see that the resulting string records the depth-first traversal of the entire tree and it can be efficiently stored in a bit vector.

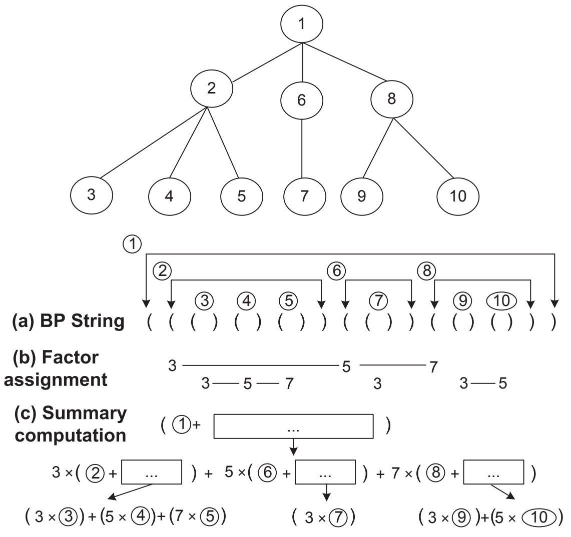

Figure 6. An example showing our data structure shape encoding algorithm: (a) the balanced-parentheses encoding of an ordered tree; (b) assigning factors to nodes; and (c) the actual shape summary computation.

This algorithm is suitable for our analysis for the following three reasons. First of all, each node is explicitly represented in the summary, making it easier for us to incorporate allocation site IDs (note that in a heap object graph, the number annotated with each node is its allocation site ID). Second, it can be computed efficiently by a single tree traversal, which fits well into the pass that we create for computing DL ratios (described in Section 2.1). Finally, the BP encoding respects the order of nodes in a tree. This is important for our shape encoding, because different fields in an object are ordered based on their offsets. Two data structures that reference the same objects in different orders should have different summaries.

2.2.2 Our Encoding Algorithm

In order to adapt the BP algorithm that encodes only acyclic data structures, we first have to break cycles in our data structures to turn them into trees. This can be easily done by computing a spanning tree of a data structure using a depth-first traversal, which is needed anyway to perform the BP encoding. While this causes information loss, we may miss only a very small number of (back) edges. Because our summaries are used for comparison purposes (i.e., not for recovering the shape), these back edges are not critically important. In fact, we did not find any significant false report due to the lack of such edges in our experiments.

The original BP bit vector representation is insufficient when allocation sites are taken into account. The major challenge here is how to incorporate allocation site IDs into the BP parentheses. Explicitly recording an allocation site ID with its corresponding parentheses would not be scalable, because, to do this, a bit vector has to be expanded to an integer vector, which consumes orders of magnitude more space. It is too expensive to record one such integer vector per run-time data structure for the postmortem comparisons. To solve this problem, instead of using a vector to represent a data structure, we develop a new algorithm that computes a (probabilistically) unique value for the data structure shape, together with its allocation site IDs, and uses this value as the shape summary of the data structure. This is conceptually similar to encoding a dynamic calling context with a (probabilistically) unique value [11]. Our encoding function φ for a node i in the tree is recursively defined as:

$$(1)\ \varphi\,(i) = \text{‘(’}\ N_i + \Sigma_{j\in[0,\#children-1]} f_j \times \varphi(child(j))\ \text{‘)’}$$

where N_i is the number associated with node i. For a heap object, N_i is its allocation site ID, and $child(j)$ denotes the j-th reference-typed field of the object. ‘(’ and ‘)’ are the parentheses for node i in the BP bit vector representation. They are not explicitly represented in the summary, and are shown here only for illustration purposes. For each node in the tree, this recursive definition allows us to compute a summary for each subtree of the node separately (i.e., rooted at $child(j)$) and compose them to form the summary for the node. Node i's summary is computed as the sum of its own ID and each of its children's summary $\varphi(child(j))$ multiplied by a factor f_j. Different child (i.e., subtree) is assigned a different f_j, and thus the order of fields is respected in the summary. For the j-th child (j starts from 0), f_j is simply defined as $2 \times j + 3$. It is guaranteed to be an (non-1) odd number, which is less likely (compared to an even number) to cause different $f_j \times \varphi(child(j))$ to have the same result. Obviously, the function is non-commutative because of the mixture of addition and multiplication. Figure 6 (b) shows the factor assignment for each node (except the root). The actual summary computation based on the BP string and the assigned factors is shown in Figure 6 (c).

As discussed earlier, the summary computed for each data structure is recorded in a shape table for further comparison. Because an allocation site can have a great number of distinct data structures, recording encoded values for all of them is not scalable. To make our analysis scale to real-world applications, we reserve a fixed-size array (e.g., s) of slots for each allocation site in the table to store summaries. The summary of each data structure created by this allocation site is mapped into a slot using a simple mod operation. In other words, for each summary φ computed, we increment the counter stored in $s[\varphi\ \%\ |s|]$. Eventually, the shape reusability for this allocation site is calculated as

$$(2)\ \max_{i\in[0,|s|-1]} s[i]\ /\ \Sigma_{i\in[0,|s|-1]} s[i]$$

The higher this value (whose maximum is 1) is, the more data structures created by the allocation site may have the same shape. In our experiments, we have tried a number of different sizes (from 4 to 11) for this array. We found that (1) a large number (i.e., more slots) preserves more information than a small number (for obvious reasons) and (2) a prime number preserves more information than a composite number. We chose 7 as the size of this array in our experiments, because it is the largest prime number for which all programs in our benchmark set could correctly run. OutOfMemory error was seen in some large programs (such as Eclipse) when the next prime number (i.e., 11) was used.

2.2.3 Computing Shape Summaries

As dead data structures are our focus, the shape summary computation is done along with the DL ratio computation in the additional GC pass (described earlier in Section 2.1) that traverses dead objects. One challenge here is that the object graph traversal implemented in GC is often a worklist-based breadth-first algorithm, while our summary computation requires a depth-first traversal, which, if implemented naively (e.g., using recursion), cannot scale to large object graphs that are often hundreds of layers deep.

We develop an efficient depth-first traversal algorithm to compute summaries. This algorithm is conceptually similar to the one used in [1] to check the `assert-ownedBy` assertions. The algorithm is still based on worklist but does depth-first traversal by coloring objects. Algorithm 1 shows the details of this algorithm.

We maintain three worklists in parallel—an object worklist O that stores objects for processing, a factor worklist F that contains factors (f_i) for the corresponding objects in O, and a summary worklist Φ that contains the encoded summaries for the subtrees rooted at the corresponding objects in O. There are three colors that can be used to mark objects: WHITE, GREY, and BLACK. Each object is marked WHITE initially. The first time an object is reached by the depth-first traversal, it is marked GREY, indicating its subtree is currently being visited. When this traversal is done, the object's color is changed to BLACK, indicating this object has been visited.

Each iteration of the main loop (line 5) retrieves an object from worklist O (line 6). This object is not processed if it has already been visited before or it is still live (line 7). Live objects cannot be part of a dead data structure. If it is the first time to see this object during the traversal (line 11-18), we mark it GREY and push it back onto the worklist

Algorithm 1: Computing shape summary for a dead data structure.

```
   Input: Object $o$ in the dead object queue
   Output: Shape summary $\varphi$ for the data structure rooted at $o$
1  $mark(o, 'WHITE')$
2  Object worklist $\mathsf{O} \leftarrow \{o\}$
3  Factor worklist $\mathsf{F} \leftarrow \{1\}$
4  Summary worklist $\Phi \leftarrow allocID(o)$
5  while $\mathsf{O} \neq \emptyset$ do
6      $a \leftarrow pop(\mathsf{O})$
7      if color($a$) = 'BLACK' or isLive($a$) = 'TRUE' then
8          $pop(\mathsf{F})$
9          $pop(\Phi)$
10     else
11         if color($a$) = 'WHITE' then
               // The first time we see it
12             mark($a$,'GREY')
13             push($\mathsf{O}$, $a$)
14             foreach Non-null object b referenced in the i-th field of a do
15                 mark($b$,'WHITE')
16                 push($\mathsf{O}$, $b$)
17                 push($\mathsf{F}$, $2 * i + 3$)
18                 push($\Phi$, $allocID(b)$)
19         else
               // The traversal of its subtree is done
20             $mark(a, 'BLACK')$
21             $f_a \leftarrow$ pop($\mathsf{F}$)
22             $\varphi_a \leftarrow$ pop($\Phi$)
23             $index \leftarrow$ findObjectWithColor($\mathsf{O}$, 'GREY')
24             $\mathsf{F}(index) \leftarrow \mathsf{F}(index) + f_a * \varphi_a$
25             if $\mathsf{O} = \emptyset$ then
26                 $\varphi = \varphi_t$
27 return $\varphi$
```

(line 12-13). All its children (i.e., objects it references) are pushed onto the worklist O (line 14-16). In addition, for each child i, we compute its factor f_i based on its index and push the factor onto the factory worklist F (line 17). Its own allocation site ID is pushed onto the summary worklist Φ as its initial summary (line 18). This value will be updated once the summaries for its subtrees are computed. It is clear to see that the sequence of GREY objects in O identifies the path currently being explored by the traversal.

Seeing this GREY object again (line 19-26) implies that its entire subtree has been visited and the summary for the subtree has been appropriately computed. We mark it BLACK (line 20), and retrieves its corresponding factor f_a and summary φ_a (line 21-22). Next, we need to attribute this node's $f_a \times \varphi_a$ to the summary of its parent. The index of its parent node can be easily obtained by finding the next GREY object in O (line 23). The summary of the parent node (i.e., $\mathsf{F}(index)$) is then updated accordingly (line 24).

Example To illustrate, Figure 7 contains the first seven steps of computing the shape summary for the root node in Figure 6. O, F, and Φ are the three worklists in Algorithm 1. While in reality O contains object references, their allocation site IDs are used here for illustration purposes. Figure 7 (a) shows the initial state of the worklists (corresponding to lines 2–4 in Algorithm 1): O contains the root object, the initial factor for the root object is 1 (in F), and Φ contains the allocation site ID of the root object, which is 1. The first step of the algorithm pops the object out of O, changes its color to GREY, and pushes back onto O (lines 6, 12, and 13 in Algorithm 1). All objects directly referenced by the first object are found and pushed onto O, as shown in Figure 7 (b). At this point, the factors for objects 2, 6, and 8 are determined (i.e., they are 3, 5, and 7) and pushed onto factor worklist F. Φ contains their initial allocation site IDs.

Next, object 8 is processed, and its children (objects 9 and 10) are pushed onto O. Figure 7 (c) shows the state of the worklists during the processing of object 10. Note that objects whose colors are GREY in O form the current exploration path (i.e., $1 \rightarrow 8 \rightarrow 10$) in the depth-first traversal. At this moment, object 10 does not have any children and its color is GREY, so it is popped out of O (line 21 in Algorithm 1) and marked BLACK. BLACK nodes are not displayed in the example, because they are not part of any worklist. Object 10's $f * \phi$ is calculated (line 23–24 in Algorithm 1) and added to the summary of its parent (i.e., the next GREY object in O), making the summary of object 8 58 ($= 8 + 5 * 10$), as shown in Figure 7 (d). Similarly, object 9 is popped and its $f * \phi$ ($= 3 * 9 = 27$) is added to the summary of object 8. In step (e), the processing of the subtree rooted at object 8 is done, and its shape summary is 85. Object 8 is then popped and its $f * \phi$ ($= 7 * 85 = 595$) is attributed to the summary of its parent, which is the root object. The last two steps show the worklist updates when the subtree rooted at object 6 is traversed. When the depth-first traversal finishes and object 1 is popped, worklist Φ will contain the shape summary for the entire data structure.

2.3 Encoding Data

The third stage of the analysis is to find allocation sites that produce data structures with the same data content. Finding such allocation sites requires the encoding of data values contained in primitive-typed fields of each dead data structure. We develop an algorithm similar to the shape summarization approach to encode data values. Based on a depth-first traversal, all primitive-typed data in a dead data structure are encoded into a (probabilistically) unique value, and then the value is mapped to a slot in a fixed-size array for the allocation site. The data summary for a data structure rooted at object o is defined as:

$$(3)\ \psi\ (o) = \Sigma_{j \in [0, \#fields-1]} f_j \times p_j$$

$$p_j = \begin{cases} child(j) & \text{The } j\text{-th field has a primitive type} \\ \psi(child(j)) & \text{otherwise} \end{cases}$$

Unlike the shape summary computation that considers allocation site IDs and reference-typed fields, this definition focuses on values in primitive-typed fields and summarizes all such values in a recursive manner. Allocation site IDs of objects are *not* considered in this algorithm. Similarly to the

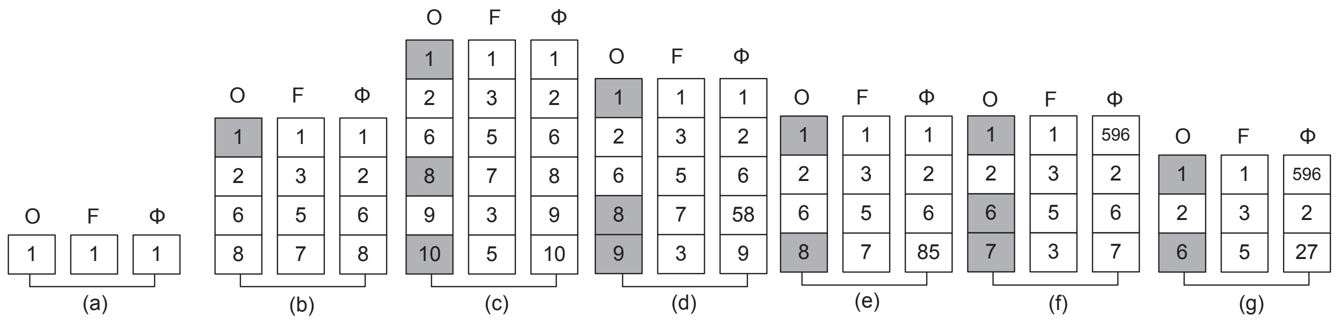

Figure 7. A step-wise example illustrating the shape summary computation for the tree in Figure 6.

shape summarization, a factor f_j is assigned to each field, regardless of its type. If the field has a primitive type, its value is retrieved directly and multiplied with the factor; otherwise, we recursively compute data summary for (the object referenced by) this field and then attribute the resulting summary to the final summary for the data structure. The same function (i.e., $2 \times j + 3$) is used to determine f_j for a field.

Data summary computation is performed together with shape summary computation in one single depth-first traversal of the objects in the dead object queue. Our technique summarizes all types of data values. The final data summary ψ is a 64-bit double value and this value is converted to a long value for the mod operation. For each primitive-typed array, we summarize all its elements, and for each reference-typed array, we recursively summarize all its containing objects. Similarly to the shape summary computation, the final data summary for an allocation site is a ratio (between 0 and 1) computed by formula (2).

Example Figure 8 shows an example of computing the data summary for a data structure that contains 4 objects. Each object is represented by an array and each cell in the array represents a field. For each primitive-typed field, its value is directly shown in the cell, while a cell for a reference-typed field contains a link pointing to another object. The factor assigned to each field is shown under the cell for the field. Similarly to the shape summarization, each factor is an odd number starting from 3. The data summary for each object is computed based on formula (3) and then used to compute the data summary for its parent. The detailed computation steps are listed aside. If a primitive-typed field contains a boolean or a char value, it is first converted to an integer before the computation is performed. Finally, summary DS_0 is converted to a long value (i.e., 35479) on which the mod operation (i.e., `mod` 7 in our experiments) is performed. The counter in the 3-rd ($= 35479\%7$) slot of the array reserved for the allocation site in the data table is incremented.

2.4 Ranking and Reporting

At the end of the execution, each allocation site in the program has three summaries computed by the encoder—the average data structure DL ratio (discussed in Section 2.1.3) as its instance summary, and the ratios computed by formula (2) on the encoded shapes and the encoded data values as its shape and data summary. To report reuse opportunities, all allocation sites are first ranked based on their instance summaries. There are two ways to use this ranked list. The top N allocation sites are forwarded to the next stage (i.e., reusable shape detector) for re-ranking. The top M allocation sites ($M < N$) are reported to the user directly for manual inspection. Regardless of whether or not larger opportunities can be found in later stages, these M allocation sites may point to interesting problems themselves and are thus worth inspecting. In our experiments, M and N are set to 20 and 200, respectively. It appears that these are the appropriate choices—M is a small enough so that it does not overwhelm the user and N is big enough so that the forwarded allocation sites retain most of the optimization opportunities.

The N ($= 200$) allocation sites are then re-ranked based on their shape summaries. Similarly to the first step, the top M ($= 20$) allocation sites are reported to the user directly while a longer list (whose length is 150) is forwarded to the reusable data detector, which, in turn, re-ranks the list based on their data summaries and reports the top 20 allocation sites for manual inspection. Note that although these specific numbers are chosen for our experiments, they can be easily changed by a different user via JVM command-line options.

3. Implementation

We have implemented our reusable data structure detector in Jikes RVM 3.1.0, a high-performance Java Virtual Machine. We add one word (32-bit) to the header of each object. This space is shared by the allocation site ID (the lower 16 bits) and the reference counter (the upper 16 bits). We found that this space is sufficient to store these two pieces of information even in large applications such as Eclipse. During the dead object graph traversal, the upper 16 bits

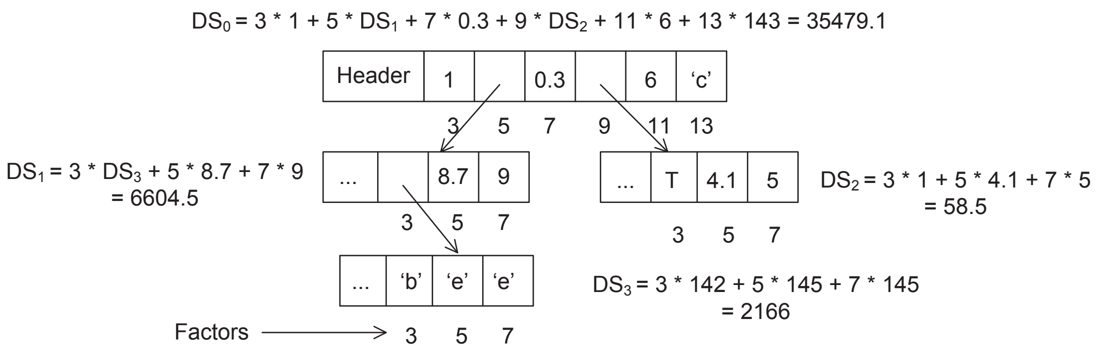
DS0 = 3 * 1 + 5 * DS1 + 7 * 0.3 + 9 * DS2 + 11 * 6 + 13 * 143 = 35479.1
Header
1
0.3
6
'c'
3 5 7 9 11 13
DS1 = 3 * DS3 + 5 * 8.7 + 7 * 9 = 6604.5
...
8.7
9
3 5 7
...
T
4.1
5
3 5 7
DS2 = 3 * 1 + 5 * 4.1 + 7 * 5 = 58.5
...
'b'
'e'
'e'
DS3 = 3 * 142 + 5 * 145 + 7 * 145 = 2166
Factors
3 5 7

Figure 8. An example of data summary computation.

are also used to store the color of each visited object (see Algorithm 1) because the reference counter of a dead object is no longer needed.

We have modified both the baseline compiler and the optimizing compiler to do the instrumentation. Our tool adds instrumentation at each allocation site that stores its ID into the allocated object's header space. This ID can be used to find the source code location (class, method, and line number) of the allocation site. As described earlier in Section 2.1, our tool also instruments each heap access statement to perform appropriate reference counter updates. Objects whose reference counters are 0 are added to the dead object queue while objects that are already in the queue but are written to heap locations are removed from the queue. An optimization here is to use one bit to mark an object when it is added into the queue so that we do not enqueue the object again if it is encountered in a heap read. The bit is cleared if it is removed from the queue.

Although the technique piggybacks on garbage collection, it requires only a very small set of changes to an existing garbage collector. In addition to adding a pass to summarize dead data structures, we need to modify the regular object graph traversal to count the number of live objects for each allocation site. Our current implementation supports all non-generational tracing garbage collectors (e.g., MarkSweep, MarkCompact, and Immix). The algorithms may not work well with a generational GC because a nursery GC scans only part of the heap, which may prevent our shape and data summarization algorithm from correctly identifying dead objects.

4. Evaluation

We have performed a variety of studies with our reusable data structure detector primarily using the DaCapo benchmark set [8]. Our benchmarks (shown in Table 2) include 11 programs in the DaCapo 2006 release, an additional set of 2 programs in its recent (9.12-bach) release, and the SPECJbb2000 benchmark. Some large (server) programs in the DaCapo 9.12-bach release were not chosen, because they could not run on the version of the Jikes RVM we used (i.e., 3.1.0). DaCapo programs were executed with their large workloads, and SPECJbb2000 was executed under its standard configuration (i.e., ramp_up_seconds = 30 and measurement_seconds = 120). All experiments were run on a quad-core machine with an Intel Xeon E5620 2.40GHz processor, running Linux 2.6.18. The maximum heap size specified for each program run was 1GB.

4.1 Case Studies

We have carefully inspected the tool reports and found optimization opportunities in all of the 14 benchmarks. In this subsection, we report our studies on 6 benchmarks: bloat, chart, luindex, lusearch, xalan, and SPECJbb2000. Problems in these programs are chosen to report because they point to large optimization opportunities—by reusing the reported data structures, we have achieved either large total running time reduction (e.g., 37.3% in bloat) or large GC time reduction (e.g., 22% in xalan). It took us about 1.5 weeks to find the problems and implement the fixes for these 6 applications we were not familiar with. More insightful changes could have been made if their developers had seen the reports and come up with fixes themselves. Although we used the Jikes RVM to find reusable data structures, performance statistics (before and after problem fixes) were collected on Hotspot 64-bit Server VM build 1.6.0_27. Jikes RVM appeared to be unstable—we often saw inconsistent performance reports for different runs of the same application on it. In order to avoid the compilation cost and the execution noise, each application was run 5 times and the median of the running times is reported in this subsection.

chart chart is a graph plotting toolkit that plots a number of complex line graphs and renders them as pdf via itext. The No. 1 allocation site in all the three reusability reports was at line 767 of class `dacapo.chart.Datasets`, which creates an array of `XYSeries` objects in method `createPtrAgeHistData`. The average size of the (dead) data structures created by this allocation site was 94663 and all their shapes were the same (i.e., its shape reusability was 1). Although their data values were different (its data reusability is 0.66), we found a way to reuse their shapes and instances. Because different instances of the array (as well as the `XYSeries` objects in them) are never needed simultaneously in the program, we moved this allocation site out of the method `createPtrAgeHistData` and made it referenced by a static field. As such, not only the array object but also its containing `XYSeries` objects are cached. We inserted code into the method to reinitialize an `XYSeries` object (by resetting it with the new content) only if it is requested. In fact, we found that in many executions of the method, a number of `XYSeries` objects were not used at all, and thus, the effort to recreate and reconstruct these objects was completely saved by our fix. The fix led to a running time reduction of 24.2% (from 7073ms to 5364ms). The number of GC runs and the total GC time were reduced, respectively, by 6.5% and by 15.3%. No reduction was seen on the peak memory consumption.

luindex luindex is a text index tool that indexes a set of documents. The allocation site related to the problem was No. 2 in the reusable instance report and No. 1 in the reusable shape report. It created an array of `Posting` objects in method `sortPostingTable` of class `DocumentWriter`. The average size of the data structures created by this allocation site was 2094. The method takes a Hashtable as input, sorts the elements of this table using a quicksort algorithm, and returns a list containing the sorted elements. Because its implementation of quicksort works only on arrays, this allocation site creates an array simply to store the elements of the Hashtable to be processed by quicksort. After the sorting finishes, a new list is created. The sorted elements are copied from the array to the list, which is finally returned. We implemented two fixes: we first pulled out the list allocation site and used a static field to cache its instance, because one instance of the list would suffice for all executions of the method. Second, we eliminated this array allocation and used an insertion sort algorithm to gradually copy elements from the Hashtable to this list. We saw a 17.6% running time reduction (from 8298ms to 6783ms) and a 21.8% reduction on the total number of objects created (from 36019657 to 28183309). The number of GC runs and the total GC time were reduced from 35 to 28 (20%) and from 1048ms to 929ms (11.4%), respectively. The peak memory consumption was reduced from 46468KB to 41384KB (12.3%).

bloat bloat is a bytecode-level optimization tool for Java that analyzes and optimizes some of its own classes. Almost all allocation sites in the three reports point to reuse opportunities. One major category of problems was the pervasive use of anonymous classes (implementing the visitor pattern), as described in Section 1. By reusing instances and content of these allocation sites, we achieved a reduction of 37.3% in running time (from 28783ms to 18053ms). The number of GC runs and the total GC time were reduced from 66 to 55 (16.7%) and from 4132ms to 3694ms (10.6%), respectively. The peak memory consumption was reduced from 813264KB to 732140KB (11.1%).

Bench	Categories		
	I	*S*	*D*
chart	5	5	4
bloat	0	0	0
luindex	8	6	2
lusearch	6	4	2
xalan	0	0	0
jbb	4	4	1

Table 1. Numbers of false positives in the top 20 allocation sites of each reusability category for the programs we have studied.

lusearch lusearch is a text search tool that looks for keywords over a corpus of data. The first allocation site in the reusable instance report was at line 119 in method `parse` of class `QueryParser`. This allocation site creates a `QueryParser` object each time a new query string is generated in order to parse the string into a `Query` object. Because the parser object never escapes to the heap, each thread needs only one instance of this data structure at any point during the execution. To solve the problem, we created a static `QueryParser` array that maintains one `QueryParser` object per thread, and added a `reset` method in class `QueryParser`. Each time a `QueryParser` object is needed, this method is invoked to reset its content. While this simple fix did not lead to significant running time reduction on Hotspot (only from 1872ms to 1867ms), it reduced the number of GC runs from 34 to 31 (9%) and the peak memory consumption from 78.6MB to 75.0MB (4.7%).

xalan xalan is an XSLT processor for transforming XML documents. The first allocation site in the reusable instance report was in the constructor of class `XPath` that created an `XPathParser` object to parse each expression string into an `XPath` object. Similarly to the handling of `QueryParser` in lusearch, we created a static field to cache its instance and reset it upon request. In addition, a few allocation sites in the reusable data report indicated that objects of type `TransformerImpl` might have the same content. Upon code inspection, we found that these objects were transitively created by a call in `dacapo.xalan.XalanHarness`. This call site is located in a while loop and creates an XML transformer per iteration of the loop to transform an incoming XML file. Because these transformer data structures are exactly the same, we hoisted this call site out of the loop. These fixes reduced the total number of GC runs from 50 to 37 (26%), and the total GC time from 2819ms to 2200ms (22.0%). No significant reduction was seen on the running time and the peak memory consumption.

SPECJbb2000 SPECJbb2000 simulates an online trading system. The first five allocation sites in the three reports were the same. Each of these allocation sites creates an object of a transaction type (i.e., `DeliveryTransaction`, `OrderStatusTransaction`, `PaymentTransaction`, `NewOrderTransaction`, and `StockLevelTransaction`) per iteration of the main loop, while all transaction objects of the same type are completely disjoint, and have the same shapes and data content. We employed a thread-singleton pattern in the implementation of each transaction class, and this fix improved the overall throughput from 148128 opr/sec to 155414 opr/sec (4.7%). No reduction was seen on GC running time and memory consumption, because the performance of SPECJbb2000 is evaluated based on a fixed-time execution. A more efficient implementation should process a larger workload in a specified period of time (reflected by the improved throughput), but does not necessarily reduce the GC effort and the memory requirement.

Summary and discussion Despite the approximations used in our analysis, we did not find many false positives in the tool reports. Table 1 shows the numbers of false positives we identified during the inspection of the top 20 allocation sites for each program. An allocation site is considered as a false positive if either it is clearly not a problem or we could not develop a solution to reuse its objects. We found that an important source of false positives is the use of linked data structures. For example, both luindex and lusearch create a great number of `Token` objects during the parsing of expressions. These objects are linked through their `next` field and any regular operation of the list can break a link and make many such objects become unreachable. The allocation sites creating them often have big DL ratios while their objects are not truly reusable. We did not find any false positives resulting from the hash collisions in the shape and data summarization algorithms. Data structures (among the top 20 reported allocation sites) whose shape and data summaries are 1 are indeed completely invariant. False positives found in the second and third stage reports are all inherited from the first stage report. This is not surprising because precisely approximating object lifetimes is the most difficult part in the detection of reusable data structures.

While it is interesting to understand the collision rates in the shape/data summarization, they are difficult to measure for large programs. To verify whether run-time data structures created by the same allocation site have the same shape or data content would require a whole program execution trace that records all heap accesses and values during the execution. Such a trace can only be obtained through whole program dynamic slicing [2, 48, 49, 50] and value profiling [12], a task that is impossible to scale to real-world applications.

We found that true problems are often very easy to fix. One solution or a combination of solutions shown in Figure 2 is always sufficient for us to reuse the identified data structures. Another important observation is that shape reusability often couples tightly with data reusability. In each program we studied, more than half of the allocation sites in the shape reusability report also appear in the data reusability report. This in fact makes it easier for us to implement fixes because the overlap often points to data structures that are completely invariant during the execution. For a few allocation sites in SPECJbb2000, we classified them as false positives because we could not understand why they are reusable by inspecting only the allocation sites. These allocation sites are located in factory methods that create objects, arrays, and strings for many different components of the program, and therefore, it is difficult to understand under what contexts these objects can be reused without more detailed information. Future work may consider to add context profiling into this analysis to provide developers with more useful debugging information.

4.2 Reusability and Overhead Measurements

All overhead statistics reported in this subsection were collected from Jikes RVM 3.1.0, running a high-performance configuration FastAdaptiveImmix. This configuration uses the optimizing compiler to compile both the JVM code and the application code, and the Immix garbage collection algorithm [7]. Section (a) in Table 2 reports the measurements of reuse opportunities. Each column in Section (a) shows, for each program, the size of the intersection of the reported allocation sites in different categories. The higher these numbers are, the easier it is for human developers to find optimization opportunities and implement fixes. Note that many allocation sites appear in all of the three reports (shown in $I \cap S \cap D$), which strongly indicates reuse opportunities. Column *#Inv* reports the numbers of invariant data structures—both their shapes and their data values are unchanged throughout the execution. Even if their instances may not be reusable, these allocation sites may point to deep design/implementation issues (e.g., designing an algorithm that is unaware of the characteristics of its input data) and fixing these issues can often lead to larger performance improvement (than just reusing data structure instances).

Section (b) of the table shows the overhead of the technique. The running time measured for our tool (shown in column T_1) includes both the time for the program execution (including the online summarization) and the time for the postmortem analysis, because the analysis is performed before the JVM completely exits. Overall, our tool slows the programs down by 10.8%. The space overhead is measured by identifying the maximum post-GC memory consumption during the execution. The overall space overhead is 30.3%, which is primarily due to the additional header space per object and the dead object queue. In one case (i.e., jython), the peak memory consumption for our tool is even lower than that for the original run, presumably because GC is triggered at a different set of program points (in the modified run)

Bench	(a) Reusability measurements				(b) Overhead measurements			
	$I \cap S$	$S \cap D$	$I \cap S \cap D$	*#Inv*	$T_0(s)$	$T_1(s)$	$S_0(MB)$	$S_1(MB)$
antlr	6	14	4	1	10.8	11.7 (8.7%)	42.3	59.1 (39.7%)
bloat	3	13	3	1	41.2	43.2 (4.8%)	63.7	89.1 (39.9%)
chart	8	4	3	13	43.9	44.9 (2.5%)	37.2	63.1 (69.6%)
eclipse	8	14	7	1	15.3	17.0 (11.0%)	35.3	78.9 (123.0%)
fop	11	14	6	22	1.1	1.2 (14.7%)	66.2	97.3 (30.3%)
hsqldb	3	17	3	18	6.5	8.0 (23%)	32.6	36.5 (12.0%)
jython	5	18	5	19	25.7	30.4 (18.3%)	108.6	85.2 (-21.5%)
luindex	8	12	3	10	11.9	13.6 (13.8%)	48.9	85.7 (75.3%)
lusearch	9	16	5	10	5.7	13.9 (143%)	74.7	97.3 (30.3%)
pmd	4	15	4	12	11.6	12.7 (9.6%)	90.8	111.0 (22.3%)
xalan	3	17	3	25	13.1	29.7 (127.4%)	17.8	23.4 (31.1%)
avrora	15	18	13	28	22.1	22.9 (3.4%)	66.3	70.0 (5.6%)
sunflow	10	11	5	21	43.5	43.9 (1.0%)	104.9	129.7 (23.7%)
SPECJbb	11	13	7	15	110583⋆	104936⋆ (5.1%)	513.7	513.9 (0%)
GeoMean						10.8%		30.3%

Table 2. Reusability and overhead measurements: Section (a) shows the numbers of allocation sites that appear in both the report of instance reusability and that of shape reusability ($I \cap S$), the numbers of allocation sites that appear in the reports of instance and data reusability ($S \cap D$), the numbers of allocation sites that appear in all the three reports ($I \cap S \cap D$), and the numbers of allocation sites whose shape summaries and data summaries are 1 (*#Inv*); section (b) reports the running times of the original (T_0) and the instrumented (T_1) programs, and their peak memory consumptions (S_0 and S_1 respectively). ⋆We measure throughput instead of running time.

that happens to have a lower maximum reachable memory size. While these overheads may be too high in a production setting, we found they are acceptable for performance tuning and debugging purposes—they have not prevented us from collecting data from any real-world application. Future work could use sampling to reduce overhead. We may also define a tradeoff framework between the quality of the reported information and the frequency of running the additional (dead data structures scanning) pass, and find a balance point where sufficient information can be reported at acceptably low cost.

5. Related Work

GC-based heap analysis There exists a body of work that piggybacks on garbage collection to discover heap-related program properties, such as object staleness [10, 45], types with growing instances [22], and object reachability properties [1, 4, 37]. Merlin [19] is an efficient algorithm that can provide precise time-of-death statistics for heap objects by computing when objects die using collected timestamps. While our work also falls into this category, our goal is different from all existing techniques—we use garbage collection to find reusable data structures.

Heap optimization for Java programs Object Equality Profiling (OEP) [23] is a run-time technique that discovers opportunities for replacing a set of equivalent object instances with a single representative object to save space. Unlike our approach that encodes data structure shapes and values to approximate their reusability, OEP records an execution trace and uses it to detect equivalent objects offline. Hence, OEP can incur a significantly higher overhead than our summarization-based approach. In addition, by focusing on allocation sites and comparing objects created by the same allocation site, our analysis is able to produce more specific diagnostic information than OEP, which attempts to find opportunities among arbitrary objects of the same type. Sartor *et al.* [33, 34] propose run-time techniques to compress heap data, particularly arrays. Instead of optimizing programs at such a low (system) level, our technique targets logical data structures and attempts to find both space and time optimization opportunities by detecting reusable data structures.

Software bloat analysis As large-scale object-oriented applications are pervasively used and their performance problems become significant, a body of work has been devoted to software bloat analysis [3, 25, 27, 28, 36, 40, 41, 42, 43, 44, 47] that attempts to find and remove performance problems due to inefficiencies in the code execution and the use of memory. Prior work [24, 25] proposes metrics to provide performance assessment of use of data structures. Mitchell *et al.* [26] propose a manual approach that detects bloat by structuring behavior according to the flow of information, and their later work [25] introduces a way to find data structures that consume excessive amounts of memory. Work by Dufour *et al.* [15] uses a blended escape analysis to characterize and find excessive use of temporary data structures. This work approximates object lifetimes using control flow regions such as a method invocation or a sequence of method invocations, whereas our work is more concerned about whether lifetimes of different objects created by the same allocation site can overlap, which is much more difficult to find using static analysis.

Shankar *et al.* propose Jolt [36], which makes aggressive method inlining decisions based on the identification of regions that make extensive use of temporary objects. Work by Xu *et al.* [42] detects memory bloat by profiling copy chains and copy graphs. Other work [35] dynamically identifies inappropriately used Java collections and recommends to the user those that should really be used. Recent work [13] identifies 11 common patterns of memory inefficiencies and proposes a ContainerOrContained model to detect such patterns in heap snapshots. Different from all existing work, our technique is a new type of bloat analysis that aims to find reuse opportunities in the program using GC-based heap analysis.

Static liveness approximation Escape analyses [9, 14, 16, 38] are designed to identify objects whose lifetimes are within the lifetime of the stack frame of the method that allocates the objects. These objects can be stack allocated for increased performance. Work by Ruggieri and Ruggieri [32] attempts to use static dataflow analysis to approximate object lifetimes in order to enable various optimizations on object allocation and deallocation. Gheorghioiu *et al.* propose a static analysis [18] to identify *unitary* allocation sites whose instances are completely disjoint so that these instances can be pre-allocated and reused. While this is similar to the detection of reusable instances in our work, we can find more opportunities such as reusable shapes and reusable data. Recent work such as [6, 46] uses static analysis to identify reusable data structures created in a loop. However, in a large-scale application, reuse opportunities may be located in methods far away from a loop, limiting significantly the real-world usefulness of these analyses. In addition, static techniques can find only data structures that are reusable for all possible runs and thus may miss opportunities that exist only for certain executions. Our work overcomes the problem by finding reusable data structures completely online, leading to the detection of more opportunities and the improved usefulness.

6. Conclusions and Future Work

The paper presents the first dynamic technique to find data structures that can be reused for better performance. In order to fully expose optimization opportunities, we define reusability at three different levels: instance reusability, shape reusability, and data reusability, each providing a unique perspective in finding reuse opportunities. It is impossible to compute precise reusability information, and thus, for each reusability category, we develop a corresponding approximation to find data structures that fall into this category. Particularly, we compute Dead/Live ratios to approximate instance reusability, and summarize data structure shapes and data values to approximate shape and data reusability, respectively. We have implemented this tool in the Jikes RVM and applied it to a set of large-scale applications. Our experimental results demonstrate that the tool incurs a reasonable overhead and reports problems that can be easily fixed for large performance gains.

The positive results from this work would serve as the motivation for the further investigation of the problem of reusing objects/data structures. For example, the existence of a large number of reusable data structures strongly calls for a new runtime system that can automatically cache and reuse data structures during the program execution. We plan to develop such a system in the future.

Acknowledgments

We would like to thank Michael Bond for his helpful comments on an early draft of the paper. We also thank the OOPSLA reviewers for their valuable and thorough comments.

References

[1] E. E. Aftandilian and S. Z. Guyer. GC assertions: Using the garbage collector to check heap properties. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 235–244, 2009.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 246–256, 1990.

[3] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 739–753, 2010.

[4] M. Arnold, M. Vechev, and E. Yahav. QVM: An efficient runtime for detecting defects in deployed systems. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 143–162, 2008.

[5] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43:275–292, 2005.

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

[7] S. M. Blackburn and K. S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 22–32, 2008.

[8] 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 *ACM SIGPLAN International Conference on Object-Oriented Pro-*

gramming, Systems, Languages, and Applications (OOPSLA), pages 169–190, 2006.

[9] B. Blanchet. Escape analysis for object-oriented languages. Applications to Java. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 20–34, 1999.

[10] M. D. Bond and K. S. McKinley. Bell: Bit-encoding online memory leak detection. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 61–72, 2006.

[11] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 97–112, 2007.

[12] B. Calder, P. Feller, and A. Eustace. Value profiling. In *International Symposium on Microarchitecture (MICRO)*, pages 259–269, 1997.

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

[14] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–19, 1999.

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

[16] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *International Conference on Compiler Construction (CC)*, LNCS 1781, pages 82–93, 2000.

[17] R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1–10, 2004.

[18] O. Gheorghioiu, A. Salcianu, and M. Rinard. Interprocedural compatibility analysis for static object preallocation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 273–284, 2003.

[19] 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*, 28(3):476–516, 2006.

[20] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 575–584, 2007.

[21] R. E. Jones and C. Ryder. A study of Java object demographics. In *International Symposium on Memory Management (ISMM)*, pages 121–130, 2008.

[22] M. Jump and K. S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 31–38, 2007.

[23] D. Marinov and R. O'Callahan. Object equality profiling. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 313–325, 2003.

[24] N. Mitchell. The runtime structure of object ownership. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 74–98, 2006.

[25] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 245–260, 2007.

[26] N. Mitchell, G. Sevitsky, and H. Srinivasan. Modeling runtime behavior in framework-based applications. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 429–451, 2006.

[27] N. Mitchell, E. Schonberg, and G. Sevitsky. Making sense of large heaps. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 77–97, 2009.

[28] N. Mitchell, E. Schonberg, and G. Sevitsky. Four trends leading to Java runtime bloat. *IEEE Software*, 27(1):56–63, 2010.

[29] J. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 118–126, 1997.

[30] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.

[31] C. Reichenbach, N. Immerman, Y. Smaragdakis, E. Aftandilian, and S. Z. Guyer. What can the GC compute efficiently? A language for heap assertions at GC time. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 256–269, 2010.

[32] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 285–293, 1988.

[33] J. B. Sartor, M. Hirzel, and K. S. McKinley. No bit left behind: the limits of heap data compression. In *International Symposium on Memory Management (ISMM)*, pages 111–120, 2008.

[34] J. B. Sartor, S. M. Blackburn, D. Frampton, M. Hirzel, and K. S. McKinley. Z-rays: divide arrays and conquer speed and flexibility. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 471–482, 2010.

[35] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 408–418, 2009.

[36] A. Shankar, M. Arnold, and R. Bodik. JOLT: Lightweight dynamic analysis and removal of object churn. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 127–142, 2008.

[37] M. Vechev, E. Yahav, and G. Yorsh. PHALANX: Parallel checking of expressive heap assertions. In *International Symposium on Memory Management (ISMM)*, pages 41–50, 2010.

[38] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 187–206, 1999.

[39] G. Xu. *Analyzing Large-Scale Object-Oriented Software to Find and Remove Runtime Bloat*. PhD thesis, The Ohio State University, 2011.

[40] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *International Conference on Software Engineering (ICSE)*, pages 151–160, 2008.

[41] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 160–173, 2010.

[42] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: Profiling copies to find runtime bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 419–430, 2009.

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

[44] 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 *FSE/SDP Working Conference on the Future of Software Engineering Research (FoSER)*, pages 421–426, 2010.

[45] G. Xu, M. D. Bond, F. Qin, and A. Rountev. Leakchaser: Helping programmers narrow down causes of memory leaks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–282, 2011.

[46] G. Xu, D. Yan, and A. Rountev. Static detection of loop-invariant data structures. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 738–763, 2012.

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

[48] X. Zhang. *Fault Localization via Precise Dynamic Slicing*. PhD thesis, University of Arizona, 2006.

[49] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 94–106, 2004.

[50] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *International Conference on Software Engineering (ICSE)*, pages 319–329, 2003.