# Data Structure-Aware Heap Partitioning

Nouraldin Jaber Milind Kulkarni

School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN, USA
{njaber,milind}@purdue.edu

## Abstract

There are many applications of program (or heap) partitioning, such as computation offloading, region-based memory management, and OS-driven memory locality optimizations. Although these applications are conceptually different, fundamentally, they must generate code such that objects in the heap (and hence the code that operates on those objects) get partitioned depending on how those objects are used. Regardless of the intended application goal, the granularity at which the heap is partitioned is the key factor in partition quality, and hence it needs to be carefully chosen.

Previous work suggested two main granularities: class-based and allocation site–based, where objects from the same class (or those allocated at the same allocation site) are co-located. Both approaches share a critical drawback: data structures that are used in different ways can share the same class, or the same allocation sites for internal objects, and hence are forced to be co-located despite their different usage patterns.

We introduce the notion of *data structure–aware* partitioning to allow different data structures to be placed in different partitions, even by existing tools and analyses that inherently operate in a class-based or allocation site–based manner. Our strategy consists of an analysis that infers *ownership* properties between objects to identify data structures, and a code generation phase that encodes this ownership information into objects' data types and allocation sites without changing the semantics of the code.

We evaluate the quality of data structure–aware partitions by comparing it to the state-of-the-art allocation site–based partitioning on a subset of the DaCapo Benchmarks. Across a set of randomized trials, we had a median range of 5% to 25% reduction of cross-partition accesses, and, depending on partitioning decisions, up to a 95% reduction.

***Categories and Subject Descriptors*** D.3.4 [*Processors*]: Code generation; F.3.2 [*Semantics of Programming Languages*]: Program analysis

***Keywords*** Heap partitioning, Ownership types, Data structure identification

*CC'17*, February 5–6, 2017, Austin, TX, USA
ACM. 978-1-4503-5233-8/17/02...$15.00
http://dx.doi.org/10.1145/3033019.3033030

## 1. Introduction

Many applications in computer systems build on program (or heap) partitioning to achieve their goals. Such applications include computation offloading [14, 18, 19], region-based memory management [6], and OS-driven memory locality optimizations [9]. Regardless of how these applications are conceptualized, fundamentally, they are about figuring out a way to generate code such that objects in the program's memory get partitioned between multiple "locations". For example, in computation offloading [14, 18, 19], the objective is to send the resource-intensive parts of the program to be executed on remote locations while keeping a smaller portion of the program on the device. This involves partitioning the program's objects and shipping them to multiple servers to improve some metric such as energy consumption. Another example is region-based memory management [6], where objects are allocated to *regions*, and all objects in the same region are freed at the same time. Objects are partitioned so that those with similar lifetime end up in the same region. One last example is OS-driven memory locality optimizations [9] where the application layer gives hints to the operating system about object placement based on object usage patterns. The intuition is to partition objects to hot and cold regions with the hope of maximizing cold regions so larger portions of DRAM stay in low power mode. Regardless of the intended application goal, the granularity at which the locations or regions (which we generically call *partitions* in this paper) are created and managed is a key factor in performance. Hence, this granularity needs to be carefully chosen.

Previous work suggested two ways of partitioning a program: class-based [18, 19] and allocation site–based [9, 14]. In class-based partitioning, all objects of the same type are co-located at the same partition. However, such schemes poorly represent object-oriented programs, as objects sharing the same class definition (*e.g.* data structures) can have completely different uses. On the other hand, in allocation site–based partitioning, objects from the same type but different allocation sites can be placed separately. Although a step on the right direction, a purely allocation site–based partitioning suffers from a crucial drawback as we shall see.

### 1.1 Motivating Example

To illustrate the shortcomings of the previous partitioning techniques, consider a Java program that uses a `List` data structure as shown in Figure 1. The code is fairly simple: once the `main` function gets invoked, an object `m` of type `Main` is created and, in turn, creates two objects (`list1`, `list2`) of type `List`. Whenever a `List` is created, it allocates an object `header` of type `Entry`.

If a class-based partitioning scheme is used on this code, it will cause all objects sharing the same class definition (*i.e.* of the same type) to be co-located at the same partition. The resulting program representation is shown in Figure 2a where each class in the pro-

```
public class Main {
  List list1, list2;
  public static void main(String[] a){
    Main m = new Main(); // M
    m.go();
  }
  void go() {
    list1 = new List(); // L1
    list2 = new List(); // L2
  }
}
class List {
  private Entry header;
  List() {
    header = new Entry(); // E
  }
}
class Entry { }
```

**Figure 1:** Motivating example code

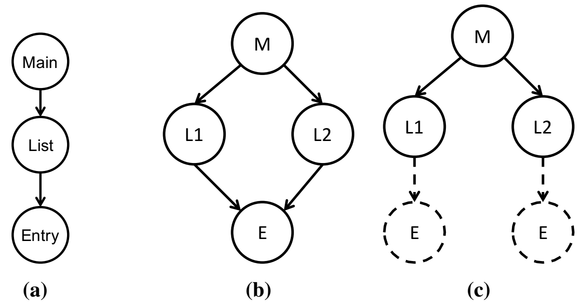


**Figure 2:** Program representations for (a) class-based (b) allocation site–based (c) data structure–aware

gram is represented as a vertex and interaction between classes are edges. This representation will force all the `List` objects created by allocation sites L1 and L2 (labeled in the figure) to be co-located even though they are virtually unrelated.

On the other hand, if allocation site–based partitioning is adapted, the program representation will be similar to the one shown in Figure 2b. Note that each allocation site in the program is represented as a node and interactions between these sites' objects are presented as edges. Allocation site–based schemes can successfully place objects of the same type but different allocation sites in different partitions (in our example `list1` is no longer co-located with `list2`). However, allocation site–based partitioning suffers from a crucial flaw: due to the prevalence of data structure libraries, different data structures often use the same allocation sites for their internal objects, forcing the internals of logically distinct data structures to share partitioning decisions and hence to be co-located. In our example, there is exactly one `Entry` allocation site, used to represent the internal nodes of the Lists, and hence all Entry objects for both lists will be co-located. This is a major limitation as, at runtime, this allocation site creates objects for both `list1` and `list2` yet it is almost guaranteed that each `Entry` object will always be accessed by its creator `List` object.

### 1.2 Data Structure-Aware Partitioning

This paper proposes a *data structure–aware* partitioning scheme that captures and accounts for this situation. If our scheme can prove that each `Entry` object will only interact with its corresponding `List` object, it will consider each `List` to have its own `Entry` allocation site as shown in Figure 2c. The net effect of this modification is that we can now place each `List` along with its `Entry` objects together as a single data structure. In other words, `Entry` objects are no longer co-located together but instead each of them is co-located with the `List` object that created it.

The key insight is that, if we look at the internals of data structures, their behavior is really tied to the enclosing data structure and *not* to each other even though they are created by the same allocation site. Such objects are *owned* by their enclosing data structures. What truly separates a data structure from any random collection of objects is how its objects interact and the implicit notion of *ownership* between them.

Adopting data structure–aware partitioning can be beneficial to the applications mentioned earlier:

- In computation offloading [14, 18, 19], this will result in placing the internals of a data structure on the same server as the data structure reference, eliminating any unnecessary communication overhead across servers.
- In region-based memory management, where regions are deallocated as a unit, if a long-living data structure object located in region X had its components located in region Y with some short-living objects, then Y cannot be deallocated [6]. On the other hand, if each data structure were placed in the same region along with its components, that region can be freed easier since all of its objects have similar lifetimes.
- In OS-driven memory optimization, the classification of allocation sites into hot and cold is done by observing the count of created objects for those sites [9]. This classification scheme will cause all internal allocation sites of data structures (*e.g.* `Entry` allocation sites) to be marked hot since all objects created for possibly-cold data structures sharing that site are accumulated. With the aid of our data structure–aware partitioning, the classification scheme can take into account that such internal allocation sites produce logically distinct objects that follow their enclosing data structures in terms of usage and lifetime and, ultimately, marking them as cold by maintaining different count per creator data structure.

### 1.3 Contributions

This work makes two primary contributions:

- We introduce machinery to realize data structure–aware partitioning. This spans two major points: first, a static analysis that automatically identifies data structures and second, a compact program representation that facilitates data structure–aware partitioning by grouping together data structure components.
- Since many existing tools [9, 14, 18, 19] do heap partitioning based on allocation sites and classes, we contribute a code preprocessing phase that transforms the program so that such tools benefit from our data structure–aware partitioning for free. Our code generation strategy carefully encodes ownership properties into allocation sites so objects that behave differently in terms of ownership look different statically.

We evaluate the quality of data structure–aware partitions on a subset of the DaCapo benchmarks [2]. We generate data structure–aware partitions of the programs, and compare their quality—in terms of cross-partition accesses—to allocation site–based partitions of the same programs. We find, across a set of randomized trials, that data structure–aware partitions produce fewer cross-partition accesses: 5% to 25% less at the median, and up to 95% in some cases.

## 2. Identifying Data Structures

In order to provide a more meaningful partitioning, we must consider how programs tend to represent and manipulate their data. For that purpose, *data structures* are extensively used in any program to efficiently represent its data. Since data structures play a critical role in program performance, programming languages tend to provide libraries for most common data structures.

The first step towards a data structure–aware partitioning is to identify data structures as logical building blocks of the program. Although programmers tend to think of a data structure as an atomic logical unit, such structures boil down to a collection of objects—and hence, allocation sites—that interact in a systematic way to provide the required functionality. What truly separates a data structure from any random collection of objects is how its objects interact and the implicit notion of *ownership* between them. For example, an `Entry` object can only be *accessed* through its corresponding owner `List` object. As a result, the problem of identifying data structures is transformed to the problem of determining the access and ownership relations between objects and their allocation sites.

Previous work suggested Ownership Types [4, 7] as a way of statically enforcing object encapsulation using a typing system. Although this is a powerful way to specify which objects are encapsulated within other objects, it requires programmer annotation and hence is not appropriate when trying to tackle existing code.

Without the aid of external annotations, we try to approximate ownership relations between objects by looking into how such objects *interact* (or access) each other.

The following subsections introduce Escape and Creator analyses. The Escape analysis observes class definitions to identify objects contained (encapsulated) within other objects and are guaranteed to be only accessed locally by whoever creates them. Creator analysis is a whole-program analysis that decides whether the uncontained objects interact with objects other than their creators.

### 2.1 Escape Analysis

Most languages provide *access modifiers* (*e.g.* `private`, `public`, etc.) to control how the internals of one class can be accessed. *Escape analysis*[1] exploits those modifiers' semantics to draw conclusions about which parts of a given class definition can *escape*. Escape analysis classifies allocation sites of a given class into escaping and non-escaping allocation sites. An *escaping* allocation site creates objects that can leave the current class (*i.e.* some external code can obtain a reference to those objects).

External code can obtain a reference to an allocation site defined in class, say `Outer`, in three scenarios: First, if the allocation site was a non-private field of the class `Outer`, then any external code can dereference an object of type `Outer` and obtain a reference to objects created by that allocation site. For example, the Escape analysis will mark the allocation site in Figure 3 line 2 as escaping since it is a `public` field. Second, if the class `Outer` defines a non-private method that returns a reference aliased to an allocation site, then that site escapes. The allocation site in Figure 3 line 4 is a `private` field but was returned by the `public` method `foo` and hence can escape. Last, if a reference to an allocation site was passed to an already escaping object, then that allocation site escapes. The allocation site in Figure 3 line 5 escapes because it was passed (line 6) to the already-escaping objects of allocation site `new A()`. Allocation sites that are not marked as escaping by the end of the analysis, get marked as non-escaping. The allocation site in Figure 3 line 3 is an example of a non-escaping site.

[1] Note that this is basically a standard escape analysis (like the one in [12]) that we are describing for completeness.

```
 1 class Outer {
 2   public A a = new A();  // S1
 3   private B b = new B(); // S2
 4   private  C c = new C(); // S3
 5   private  D d = new D(); // S4
 6   a.pass(d);
 7   public C foo() {
 8     return c;
 9   }
10 }
```

**Figure 3:** Escaping and non-escaping allocation sites

**Algorithm 1** Escape analysis

```
for each class $c$ in the program do
    Mark static fields' allocation site as escaping
    Mark sites within static functions as escaping

    Define $EscapePoints$ to be:
    - Non-private fields
    - Returned references of non-private functions
    - Arguments to calls on escaping receiver objects
    - Static fields' references in the program
    for each allocation site $s$ in $c$ do
        for each reference $r$ in $EscapePoints$ do
            if $s$ is aliased to $r$ then
                mark $s$ as Escaped
            end if
        end for
    end for
end for
```

The ultimate goal of the Escape analysis is to identify which objects constitute a data structure. If an allocation site is marked as non-escaping, then we can conclude that the objects it creates only interact with the corresponding enclosing objects who created them. As a result, we can safely consider the non-escaping allocation site's objects to be part of the same data structure that contains the enclosing object. Algorithm 1 gives a high-level description of how Escape analysis works.

### 2.2 Creator Analysis

Concluding that any escaping allocation site is not a part of the same data structure of the enclosing class can be rather conservative. Having the necessary code to escape does not imply that the allocation site will indeed escape nor that it will be accessed by any external code. A whole-program view that considers all accesses in the program is necessary to decide if escaping allocation sites do get accessed in a way that violates encapsulation. To account for this case, we introduce a *Creator* analysis.

We start by defining the *creation relation* between objects. We say that object X created object Y if X invoked the code that resulted in the allocation of Y. Figure 4 shows an example of such relation where an object of type A, say `a`, is allocated then invoked the function `create` that results in the allocation of an object, say `b1`, of type B. We consider object `a` to be the creator of object `b`. Any access from `a` to `b` is called a creator access and any other access to `b` is called a non-creator access. Note that this relation is between runtime objects and not allocation sites.

The Creator analysis uses the creation relations to decide which allocation sites create objects that get only accessed by their creators. Such *owned* sites create objects that are part of the creator's

```
class A {
  public static void main(String[] x){
    A a = new A();
    a.create();
  }
  void create() {
    B b = new B();
  }
}
class B {}
```

**Figure 4:** Creation relation example

```
public class Main {
  public static void main(String[] a){
    Main m1 = new Main();
    m1.go();
    Main m2 = new Main();
    m2.go();
  }
  void go() {
    List L1 = new List();
    List L2 = new List();
  }
}
class List {
  List() {
    Entry e1 = new Entry();
  }
}
class Entry { }
```

**Figure 5:** Example code

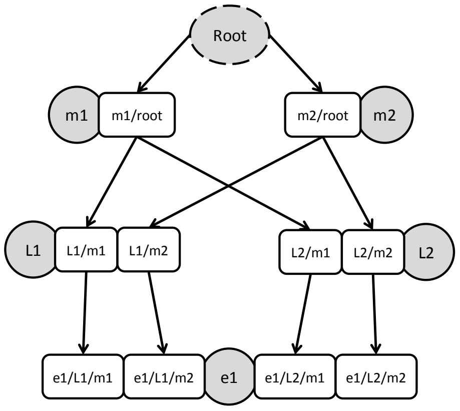


**Figure 6:** Example Creator Graph

**Algorithm 2** Creator analysis

```
▷ Initially, all allocation sites are marked as owned
for each access in the program do
    Let the access be from object $a$ to object $b$
    if $a$ is not the creator of $b$ then
        if $a$ and $b$ trace back to different creators then
            $b\_alloc \leftarrow b$'s allocation site
            Mark $b\_alloc$ as assigned (not owned)
        end if
    end if
end for
```

data structure. Later we shall see how to exploit this to achieve data structure–aware partitioning.

We define the Creator Graph as a graph that enumerates all creation relations in the program. Each vertex represents a runtime object and each edge is a creation relation between its vertices. Figure 5 shows a simple program and Figure 6 shows the corresponding Creator Graph. The Root node is a virtual creator for objects allocated by a code that was not invoked by another object. In our example, the JVM invoked the `main` function and that resulted in the creation of objects `m1` and `m2`. The graph node X/Y/Z reads as follows: object X that was created by object Y that, in turn, was created by object Z. For purposes of illustration we include allocation sites in this figure as circles and objects they create as rounded rectangles attached to them. Note that objects `m1` and `m2` each create two lists `L1` and `L2` making a total of four lists. Each time an object of type `List` is allocated, it creates an object `e1` of type `Entry`. Note that a complete Creator Graph cannot be statically computed so we build an approximation of the graph. The mechanics of building the Creator Graph is discussed in Section 5.

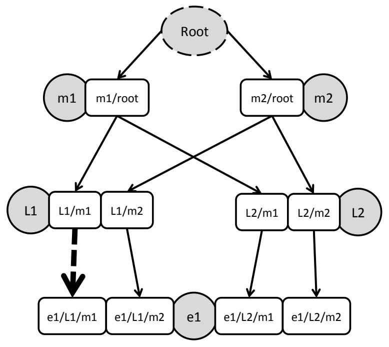
With this graph at hand, any access in the program can be classified as creator access or non-creator access. For example, an access from L1/m1 to e1/L1/m1 (dashed arrow in Figure 7a) is a creator access since it corresponds to an edge in the Creator Graph. An access from L2/m1 to e1/L1/m2 (dashed arrow in Figure 7b) is an example of a non-creator access.

An allocation site creating objects that only get accessed by their creator can be safely considered a part of a data structure. We call such allocation sites *owned*. For example, if the `Entry` allocation site was marked as owned (*i.e.* each `Entry` object only interacts with its creator `List` object), then we consider this allocation site's objects to be part of a bigger data structure along with their creator `List` objects. If at least one object broke this and interacted with a non-creator object, then the whole allocation site is not owned.

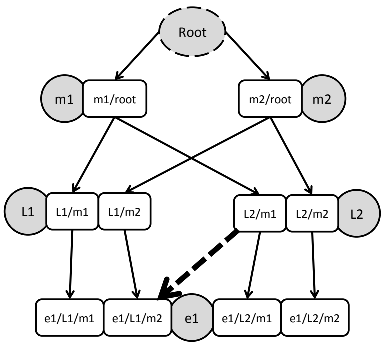
Although the creation relation is *not* transitive, it still makes sense to allow two objects belonging to the same data structure to interact while keep marking the accessed object's allocation site as owned. This allows the internals of data structures to interact safely. For example, although the access from L2/m1 to L1/m1 (Figure 7c) is a non-creator access, both objects *trace back* to the same creator (m1/root) and hence, we can keep the allocation site L1 as owned. Two objects trace back to the same creator if, starting from each object, following the chain of creators upwards will reach that shared creator. Trace back logic is also used to handle constructs like `Iterators` that, in order to operate efficiently, have access to the internal objects of a data structure although such objects are created by the data structure and not by the `Iterator` object. Algorithm 2 gives the high-level procedure for the Creator analysis.

Both algorithms are fixpoint algorithms (simplified here for presentation purposes). Termination is guaranteed because there is an implicit lattice in both cases: objects can go from being owned to assigned (Similarly: from being non-escaping to escaping), but not the other way around.

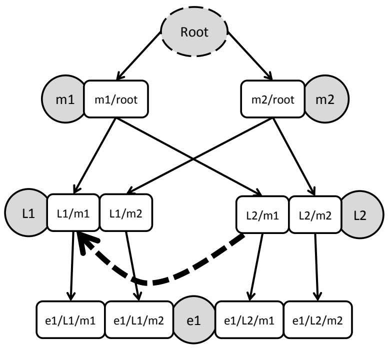
Both Escape and Creator analyses try to identify allocation sites that create the building parts of data structures. The Escape analysis exploits access modifiers to check object containment and conclude which objects can escape. Non-escaping objects are considered part of the enclosing object's data structure. The Creator analysis identifies which of the escaping sites is still owned and only creates



**(a)** Creator access



**(b)** Non-creator access



**(c)** Transitivity via traceback

**Figure 7:** Computing creator relations

objects that get accessed by their creators. Such sites' objects are also considered parts of the creator objects data structures.

## 3. Data Structure-Aware Representation

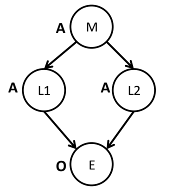
In partitioning applications, program heaps are represented using graphs. In class-based partitioning each node in the graph represents a class in the program and each edge is an interaction between instances of nodes it connects. In allocation site–based partitioning, nodes in the graph represent all objects allocated at a particular allocation site. The granularity of partitions in the program is tied to the granularity of this heap representation: static partitioning decisions (*e.g.* which region to place allocated objects in) require that all objects abstracted by the same node make the same decision.

We use the results of both Escape and Creator analyses to refine a purely allocation site–based program representation to account for the presence of data structure. Specifically, we refine the Object Interaction Graph (OIG) proposed by Sinha and Kulkarni for computation offloading[14]. The OIG is a purely allocation site–based representation where each node corresponds to an allocation site in the program. If two objects interact, an edge between those objects' allocation sites is added to the OIG[2].

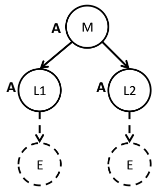
We refine the OIG in two phases, first: we classify each allocation site into either owned or assigned. Recall that an *owned* allocation site is a site whose objects are only ever accessed by their creator objects. Owned allocation sites are those marked as owned by the Creator analysis *and* those marked as non-escaping by the Escape analysis. On the other hand, an *assigned* allocation site is a site that created at least one object that interacts with an object other than its creator. Assigned allocation sites are those sites that escaped *and* the Creator analysis concluded that their objects indeed interact with non-creator objects. A given allocation site is owned unless proven otherwise by the analyses. Static fields allocation sites and sites within static functions are marked as assigned since, by definition, static code is shared between all objects of the same type and not owned by only one of them.

The second phase in the OIG refinement is to systematically omit owned allocation sites by collapsing them to their creator sites. As a result, owned objects will be co-located with their owners and will reside on the same destination partition providing local access to those objects. To see the significance of this, consider the OIG shown in Figure 8a that corresponds to the example from

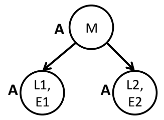
[2] The OIG is a particular representation used for computation offloading where edges represent interactions; the techniques here can easily be adapted to, *e.g.*, heap abstractions where edges represent reachability.



**(a)** Unrefined OIG



**(b)** Logical view



**(c)** Refined OIG

**Figure 8:** OIG refinement (A: assigned, O: owned)

Section 1. If a placement algorithm were to assign each node to a destination partition, *all* `Entry` objects will reside on the partition. This will require `List` objects on different partitions to communicate to their `Entry` objects remotely causing unnecessary cross-partition communications (*e.g.* messaging overhead between servers or across-region memory accesses). If the `Entry` allocation site was marked as owned, then we know that each `Entry` object is only ever accessed by its creator/owner `List` object. As a result, we can consider the `Entry` allocation site to be two distinct allocation sites each of which is serving its owner `List` as shown in Figure 8b. Only assigned allocation sites will be visible for the placement tool to *assign* them to destination partitions while owned allocation sites (allocation sites of components of data structures) are *hidden* from the placement tool and will be always co-located with their owners. The refined OIG, containing only the assigned allocation sites, is shown in Figure 8c.

One crucial thing to notice here is that the resulting program representation *encodes* ownership information into an allocation site–compatible representation. This enables any tool that represents a program as allocation sites to utilize ownership information for free and ensures all objects constituting a data structure are shipped to the same partition. In order to make this complete, our tool comes equipped with a code generation mechanism that rewrites the program so that creators have actual static separate copies of their owned allocation sites.

## 4. Code Generation

Most tools that resort to partitioning a program abstract the heap as a graph of allocation sites [9, 14]. Placement decisions get implemented *statically* at each allocation site [3], directing its objects to the right partition. In order to exploit ownership properties, we need code that behaves differently in terms of ownership to look different statically.

In this section, we describe a code generation strategy that takes an input program (along with its Creator Graph) and outputs a program where owned allocation sites look different statically (*i.e.* have different types). To preserve the semantics of the program, the generated class definitions of owned allocation sites should have exactly the same functionality but come from different class definitions (to have different types). To achieve this, one option is to define new classes with the same logic as the ones we tend to clone and rewrite the whole program using the new types. However, this requires a lot of rewriting as the new types are not related to the old ones. Another complication is that if some foreign library expects an old type, it will not accept a new unrelated type. As a result, there should be a tighter relation between the class and its clone.

We use the concept of *inheritance* to clone classes. If two `List` objects need to be placed in different locations (say Partition1 and Partition2), we create two separate classes `List_Partition1` and `List_Partition2` both extending `List`. The clone classes override all inheritable methods and fields (*e.g.* `public`) and simply copy all non-inheritable methods and fields. The intuition behind using inheritance is that we can change the type of an object from, say `List`, to `List_Partition1` and the program will still type-check. Static code is not copied to the clones because, in the original program, static code of a class is shared between all objects regardless of where they reside or what allocation sites create them. There are many corner cases that need to handled as we shall see in Section 5.

Ultimately, we want to force owned objects to be co-located with their owners and give the freedom to the placement tool to decide the best place to locate assigned objects. The code rewriting mechanism enforces that by first generating classes using the following rules (Assuming we have partitions P*1*, P2, ..., P*n*):

- For each assigned allocation site `new Foo()` on partition P*i*, create a new class (if it does not already exist) `Foo_A_P`*i* extends `Foo` where $i \in [1\text{-n}]$.
- For each owned allocation site `new Bar()`, create a new class (if it does not already exist) `Bar_O_P`*i* extends `Bar` $\forall$ *i*=1,2,...,n.

Second, the rewriting mechanism populates the new clones and changes the original allocation sites as follows (assuming the current clone is `Baz_A/O_P`*c*):

- If the original allocation site `new Foo()` is assigned on partition P*i*, change it to: `new Foo_A_P`*i*`()`.
- If the original allocation site `new Bar()` is owned, change it to: `new Bar_O_P`*c*`()` where P*c* is the current clone's partition.

Note that if the placement decisions of the assigned allocation sites are not known prior to generation, each assigned allocation site is assumed to be in its own partition. This will result in creating a new owned class type per creator instead of having creators on the same partition sharing the same owned class type.

***Bound on generated code size*** Since we need, for each class, at most two copies per partition (one assigned and one owned), the number of classes generated is linear in the number of partitions. If a full duplication of the code is done (*i.e.* a unique class per node in the Creator Graph), then an exponential blowup will result, which we avoid.

[3] Placement decisions in class-based partitioning is implemented at a class level, which is a subset of allocation site–based partitioning where allocation sites of the same type have the same placement policy.

```
public class Main_A_P1 extends Main {
  void go() {
    List L1 = new List_A_P1();
    List L2 = new List_O_P1();
  }
}
public class Main_A_P2 extends Main {
  void go() {
    List L1 = new List_A_P1();
    List L2 = new List_O_P2();
  }
}
```

**Figure 9:** Part of the rewritten code

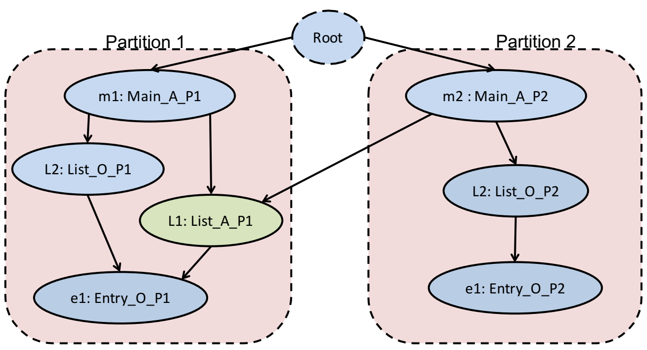


**Figure 10:** Heap partitions visualized

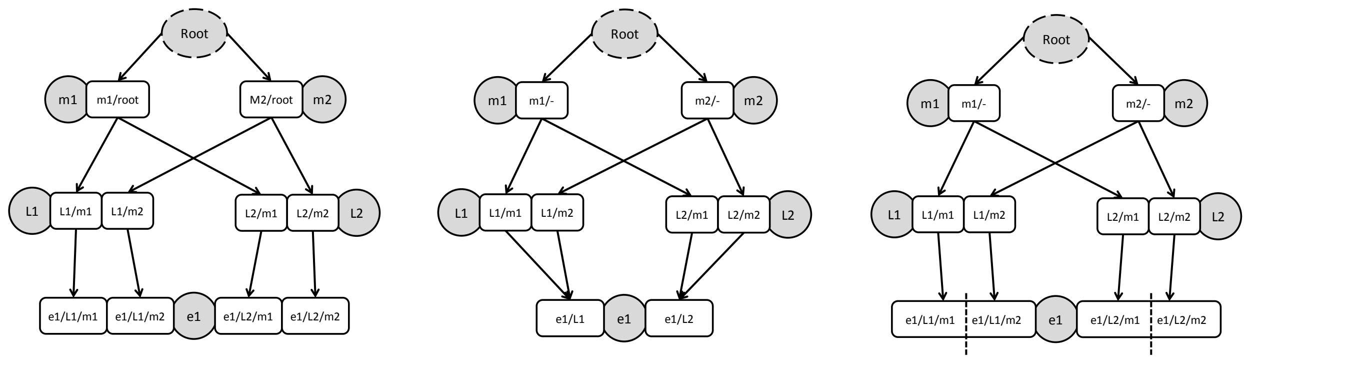
***Example*** In order to see how all pieces of this work fit together, consider the same input program code as in Figure 5. After running our analyses, M1, M2, and L1 allocation sites are marked as assigned[4] and the other sites as owned. Suppose a placement tool assigned both M1 and L1 to Partition P1 and M2 to partition P2. Figure 9 shows the the effect of applying the former rules to the allocation sites of type `Main`. Note that, since L1 site is assigned, both of the new `Main` types (`Main_A_P1` and `Main_A_P2`) are using the same `List_A_P1` class to create L1 objects regardless of the owners location. However, since L2 site is owned, each `Main` class uses the copy residing on its partition. Figure 10 illustrates objects placement for this example where each owned object is co-located with its owner.

## 5. System Implementation

We implemented our analyses and transformations in Soot [16]. The input program is transformed to Jimple (an intermediate representation of Soot). The code analyses and transformations were conducted on Jimple and the re-written output code is modified Java Bytecode.

***Building the Creator Graph*** The pointer analysis choice is one of the key factors to tune the precision of the Creator Graph. In order to determine the creator of an object, we need to know which object called the function that allocated it. For this purpose, any context-insensitive pointer analysis will be very imprecise as it will simply merge the points-to sets of all the callers of the function making it impossible to determine the actual creator. If we were

[4] Even though the analysis described in Section 2 will mark L1 as Owned, for the sake of this example, we will assume it is Assigned.



**(a)** **(b)** **(c)**

**Figure 11:** (a) Perfect Creator Graph (b) Approximated Creator Graph (c) Approximated Creator Graph with Escape analysis

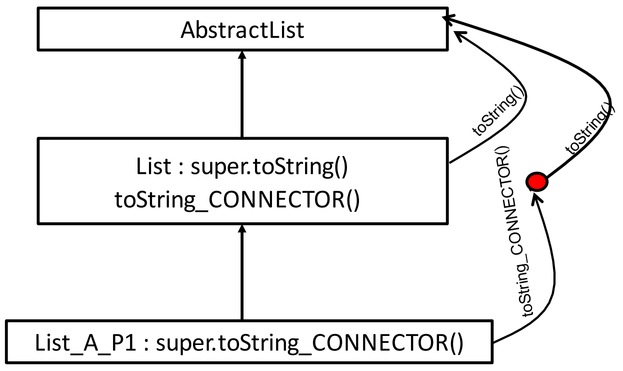
to conservatively consider all of them to be possible creators, then a lot of accesses will be falsely considered creator accesses and hence disturb the process of identifying data structures. A notion of context-sensitivity is needed to get a more precise results.

We use Paddle [1], a pointer analysis with *object-sensitivity* [10, 15]. Paddle's object sensitivity captures the context of a method called on an object with the object that *allocated* that object; longer chains of these allocating objects correspond to more object sensitivity. This notion of object sensitivity exactly matches the creator chain from the Root to any node in the Creator Graph. In order to build a complete Creator Graph, we would need to set the object sensitivity to a very large number which is impractical.

We use an object sensitivity of 1 (*i.e.*, we use a 1-object sensitive analysis). This will result in a loss of precision that causes the Creator Graph at Figure 11a to look like the approximated one in Figure 11b. Since $k$=1, we can only abstract one creator parent towards the Root. Using this graph, we can distinguish `Entry` objects created by L1 from those created by L2 but we cannot tell if it is L1 of M1 or L1 of M2. However, with the presence of Escape analysis we can conclude that non-escaping allocation sites' objects only get accessed by their creators even though we cannot prove that using a pointer analysis with $k$=1. For example, if the `Entry` allocation site was marked as non-escaping, the approximated Creator Graph will be similar to the one shown in Figure 11c. This effectively enhances the decisions made by the Creator analysis.

***Challenges of cloning classes using inheritance*** Implementing our code generation strategy using inheritance poses several challenges, such as handling the `super` keyword, accesses to `private static` and `protected` fields, and `private` constructors. We discuss handling calls using the `super` keyword and private code in some detail and the other challenges are handled using similar tricks.

In Java, the `super` keyword is used to call functions and constructors of the super class. However, since we clone a given class by inheriting from it, the original calls to `super` now present in the clone point to the class being cloned not to the actual super class. For example, the reference `super` in the class `List_A_P1` refers to the `List` class itself instead of referring to `List`'s superclass. For `List` and `List_A_P1` to be behaviorally equivalent, a call on `super` in both classes should refer to the original superclass of `List`. This is achieved by injecting new connector functions in the class to be cloned that calls the original superclass, then altering calls on the clones to use those functions. Figure 12 gives an illustration where the injected function `toString_CONNECTOR()` is used to enable the class `List_A_P1` to reach the original superclass `AbstractList`.



```
public String toString_CONNECTOR(){
  return super.toString();
}
```

**Figure 12:** Handling calls to `super` using connector functions

Since non-inheritable code (*e.g.*, `private` fields) is simply copied into the clones in the code generation process, careful code rewriting must take place to preserve the semantics of the program in the presence of two copies of such code. For example, a `private` field appears in the original class and the cloned class with no direct relation between them (as opposed to `public` fields that are inheritable and essentially point to the same data). Hence, although accesses to such copied `private` fields originating from an inherited function work fine, accesses to that field from, say, the non-inheritable static functions will be directed to the original, obsolete field. In order to ensure correct functionality, any such access must be redirected to the new copied `private` field. We achieve this by synthesizing `public` accessor functions (setters and getters) and replace any access to the original field by the corresponding accessor, ensuring that the new field ends up being used instead.

## 6. Evaluation

To evaluate the efficacy of our analyses and transformations for enabling data structure–aware partitioning, we examine a specific metric: given a data structure–aware partition of a program, how many *cross-partition accesses* are there compared to a data structure–unaware partition? Our intuition is that by allowing the internals of distinct data structures to be co-located with their enclosing structures, rather than all being placed in the same partition due to allocation-site limitations, programs that make heavy use of data structures can reduce how frequently (dynamic) accesses are made from an object in one partition to an object in another.

Reducing cross-partition accesses is useful across a wide range of program-partitioning problems. If partitioning is used for computation offloading, cross-partition accesses represent communication, which should be minimized. For region-based memory management, cross partition accesses represent reachability between objects of different lifetimes, which can prevent region deallocation. For OS-based memory partitioning, cross partition accesses can represent accesses from hot regions to cold regions, which bring cold regions into the working set.

We illustrate three aspects of the evaluation: First, we present the results of running both the Escape and Creator analyses. Second, we discuss the results of applying data structure–aware partitioning, and its behavior compared to data structure–unaware partitions. Finally, we investigate the effects of our code transformations on code size.

***Evaluation methodology*** We partition a subset of the Dacapo Benchmarks [2]. We chose benchmarks with less reflection [3] since the current infrastructure (both our analyses and the underlying Soot infrastructure) does not handle reflection very well. We run the experiments on an Intel Xeon E5-4650 system with four 8-core processors (total of 32 cores) running at 2.7 GHz.

### 6.1 Analysis Results

Figure 13 shows the results of running the Escape and Creator analyses. The top two segments of each bar represent owned allocation sites, with the first segment capturing those sites marked owned by the Escape analysis, and the latter are sites marked owned by the Creator analysis. For all of our benchmarks, the majority of owned sites are *not* captured by the Escape analysis; the Creator analysis is necessary to correctly identify these sites as owned.

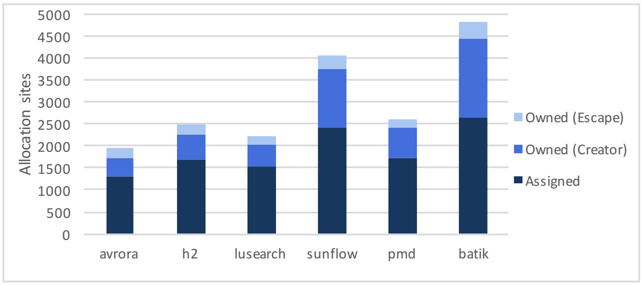


**Figure 13:** Interleaved results of Escape and Creator analyses

***Analysis performance*** We present the analysis wallclock runtime for each benchmark in the last column of Table 1. The pointer analysis takes a considerable amount of time as we use a 1-object-sensitive analysis. Escape and Creator analyses use complicated points-to queries, such as "return the points-to set of x within the context of allocation site y." As a result, these queries can take a substantial amount of time. The code generation is a matter of outputting the ".class" files, so it needs less time than other components.

Note that we expect the most of the benefits from our technique to arise in data structure code, which largely resides in libraries

| Benchmark | Assigned | | Owned | | Time |
|---|---|---|---|---|---|
| | objects | accesses | objects | accesses | (mins) |
| avrora | 40966 | 58325 | 398608 | 2113699 | 94 |
| h2 | 4144904 | 9671248 | 1884159 | 11434289 | 112 |
| lusearch | 510182 | 4348741 | 64972 | 354671 | 81 |
| sunflow | 440547 | 471292 | 166 | 5628 | 130 |
| pmd | 1590 | 7117 | 442 | 2630 | 118 |
| batik | 20079 | 71646 | 15995 | 75664 | 106 |

Table 1: Statistics of one candidate data-structure-aware configuration for each benchmark

that are analyzed and used regardless of the benchmark. For example, while lusearch, the smallest benchmark, has 660 user allocation sites, the analyzed code encompasses almost nine thousand sites. Within those sites, our technique automatically identifies data structures and their uses. Moreover, analysis time is dominated by analyzing the libraries, and we would not expect analyzing larger programs to take substantially longer.

### 6.2 Partitioning Results

***Methodology*** Evaluating the quality of data structure–aware partitioning is challenging, because different partitioning tools may make different decisions regarding object locations, which can have different impacts on performance or other metrics. Hence, we perform a *randomized evaluation* to measure partition quality by generating a series of random partition decisions, to understand the range of benefits that data structure–aware partitioning can provide. This randomized evaluation follows four steps.

1. Using the data structure–aware program representation described in Section 3, we create a *partition configuration* of the program by randomly partitioning the program into two pieces[5]. This partition configuration automatically places owned objects with their creators, and hence ensures that data structures are co-located with their internal objects.
2. We use our code generation strategy to generate an executable version of this partitioned program. Object accesses are instrumented to determine whether the access crosses partition boundaries or not. Note that this instrumentation is straightforward because our code generation strategy lets us statically distinguish between objects placed in different partitions. The results from this execution form the *baseline* for this partition configuration.
3. We then derive *data structure–unaware* variants of the program. To do this, we start with the initial partition configuration. We then assume that owned objects, rather than being able to be co-located with their owners, are represented by their original allocation sites (hence, all objects from one allocation site must be in the same partition). We then generate a *random derived configuration* that assigns these formerly-owned allocation sites to the partition of one of their owners. For example, in Figure 8a, the allocation site marked E would be randomly assigned to either the allocation site of L1 or the allocation site of L2.
4. For each partition configuration, we evaluate several such random derived configurations. For each configuration, we use our code generator (with all sites marked "assigned") and then use the same instrumentation to measure cross-partition accesses. By comparing the number of cross-partition accesses in these derived configurations to the number in the baseline configura-

[5] Note that, regardless of the number of partitions, an object can either be with its creator or away from its creator, so two partitions are sufficient to explore the costs of partitioning.

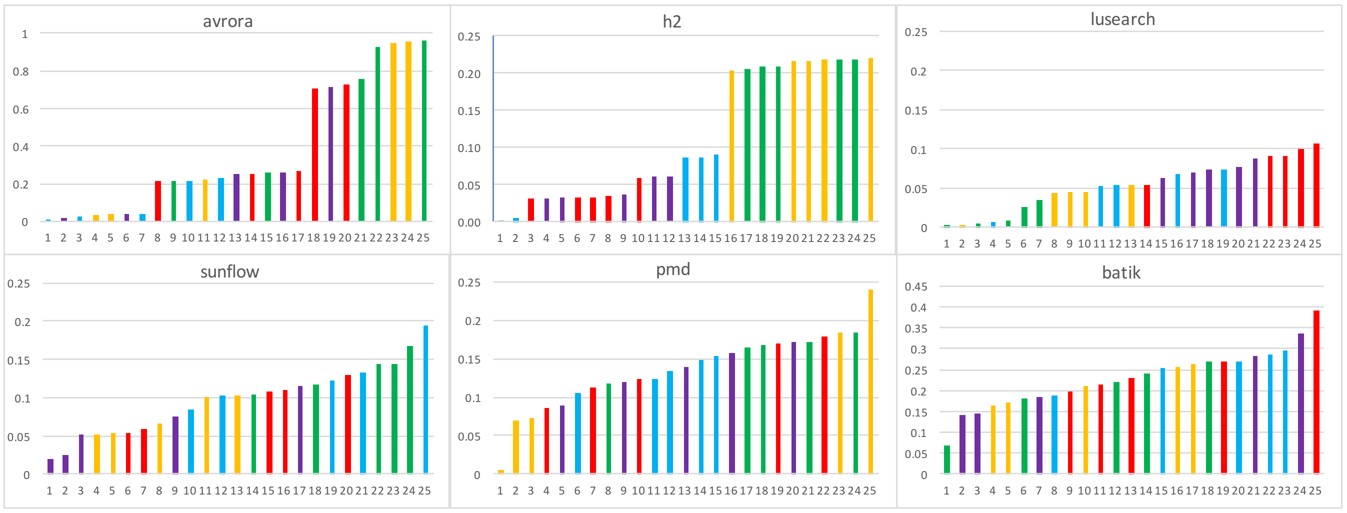


**Figure 14:** Reduction in cross-partition accesses for each run (x-axis: run number, y-axis: reduction ratio: higher is better). Y-axis scale is different for each figure.

tion, we can evaluate how much data structure–aware partitions (the baseline) can reduce cross-partition accesses.

Our fundamental goal is to isolate the improvement of data structure–aware partitioning from confounding effects such as partition decisions (which can vary based on objective). To this end, we randomize partitioning decisions (step 1). Further randomization (step 3) is used to establish a compatible baseline of data structure–unaware partitions.

***Results*** Figure 14 shows applying the procedure described above with five random initial configurations for step one and five derived random placements for each one resulting in 25 runs per benchmark (we sort the improvement ratios for ease of digestion). As shown from the figure, sometimes the base configuration (step one) places all data structures in the same partition so there is no savings from data structure–awareness. Other times if the base configuration separated data structures to different partitions, then there are a lot of cross-partition accesses that get eliminated using our techniques. The median improvement ranges from 5% to 25% reduction of cross-partition accesses.

Using data structure–aware partitioning is never worse and usually much better, and that depends on the heuristic for placement and the particular application. The bar colors in each graph represent different baseline configurations. For some baseline configurations data structure–awareness is unnecessary: there exist similarly-performing derived configurations. For other baseline configurations, though, any assignment of sites to partitions results in increased cross-partition accesses in the data structure–unaware case. Table 1 shows a candidate data structure–aware partition for each benchmark. The table shows how many assigned objects[6] were allocated and how many accesses to such objects. The same statistics are shown for owned objects.

It is important to notice that what matters is not the number of created objects but *how frequently* those objects are accessed. Figure 15 access-per-object ratio for both assigned and owned objects. By inspecting the instrumentation logs, the runtime types of the owned objects included:

- `java.util.[Weak,Linked]HashMap$Entry`
- `java.util.TreeMap$Entry`
- `java.util.LinkedList$Node`
- `java.util.Hashtable$Entry`

This illustrates that our analyses can identify the building blocks of data structures, and designate them “owned.”

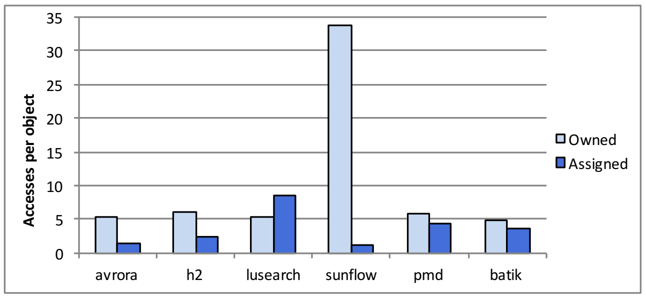


**Figure 15:** Accesses per object

## 6.3 Code Size

This section evaluates how much code bloat our code generation process incurs. Figure 16 shows, for each benchmark, the ratio of bytecode instructions in the (transformed, partitioned) program to the original program. Across all the benchmarks, the cloning process produces roughly 2.5 times as much code as the baseline (the first bar). Note that in a distributed setting, each partition only needs the code associated with itself as shown in the second and third bars.

We note, however, a program performs the same amount of work, so the extra code does not correspond to additional overhead. Figure 17 shows the normalized runtime of the original and partitioned code. As shown in the figure, the overhead introduced by partitioning is not statistically significant in most cases.

[6] This includes static fields and objects, which are, by nature, assigned, since they only exist once.

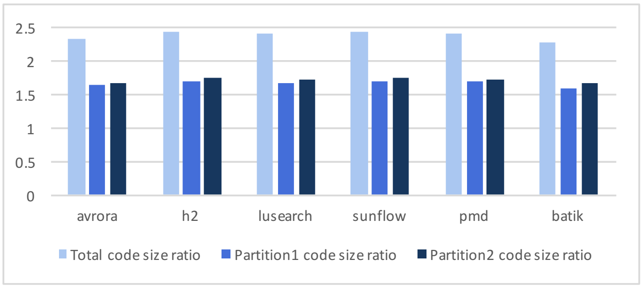


**Figure 16:** Code size increase ratio

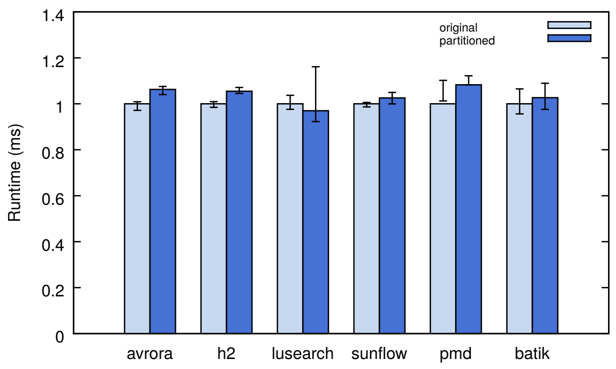


**Figure 17:** Normalized runtime for 50 runs: bars for the median, error margins around the mean

## 7. Discussion

In this section we briefly go over some design decisions we made including the absence of a runtime component and baking ownership information into the allocation sites.

In spirit of the separation on concerns principle, our goal is to interact with partition tools that statically partition the program just by using allocation sites or classes. Because we want to be orthogonal to those tools, we cannot just annotate the allocation sites (since the tools would not understand those annotations) or use any kind of runtime information (since the tools work at compile time). By generating class-based code, our work is compatible with all these tools.

## 8. Related Work

Program partitioning is used in various fields including computation offloading and region-based memory management. In this section we explore program representation techniques used in those fields. Programs are usually represented on the granularity of allocation sites, classes, or tasks.

Much work on program partitioning for computation offloading use class- or task-based representations [8, 11, 17–19]. The program is represented as a graph where nodes are classes (or tasks) and edges are interactions of those classes (or tasks). In class-based program representations, all objects of the same type must be allocated to exactly one partition. As discussed in Section 1, this representation is rather constraining and will incur more cross-partition accesses. Allocation site–based program partitioning, where all objects of the same allocation site end up of the same partition, is a finer, more precise granularity [9, 14]. Sinha and Kulkarni [14] use OIG (mentioned in Section 3) to represent a program. However, their representation does not consider ownership, resulting in data structure internals' being co-located.

Salagnac *et al.* [13] proposed a region-based memory management scheme that tries to allocate connected structures into the same partition. They start by running a static context-insensitive analysis to approximate connectivity between variables of the program. Then when creating an object at runtime, the system uses the analysis results to check for related (connected) objects, and place that object at the same region with them. Although the placement decisions are done at runtime, it depends on the result of a rather conservative context-insensitive analysis that, as shown in Section 5, can lead to grouping all components of logically-unrelated data structures at the same partition.

Boyapati *et al.* [5] use ownership types to perform data structure–based region allocation. However, their approach requires user-provided annotations in the form of ownership types, including in library code. Our tool does not require any annotations to operate.

## 9. Conclusions

This paper presents analyses to achieve data structure–aware program/heap partitioning. The analyses exploit ownership relations between objects to conclude which objects are fully enclosed within other objects and hence can be co-located with the enclosing objects without requiring a separate placement decision. As we showed in Section 6, there was an improvement over the purely allocation site–based partitioning, with data structure–aware partitions exhibiting significantly fewer cross-partition accesses than data structure–unaware partitions.

A code generation strategy was implemented to encode ownership information into the allocation sites. As a result, objects with different ownership properties look different statically. This code generation strategy, coupled with our data structure–aware heap representation, means that existing partitioning tools that expect to work at the allocation site or class granularity gain the benefits of ownership for free. Indeed, even in the absence of a placement tool, our analysis and code generation will let *any* allocation site–based tool get more precision based on ownership information.

## References

[1] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. *ACM SIGPLAN Notices*, 38(5):103–114, 2003.

[2] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[3] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 241–250. ACM, 2011.

[4] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *ACM SIGPLAN Notices*, volume 38, pages 213–223. ACM, 2003.

[5] C. Boyapati, A. Salcianu, W. Beebee Jr, and M. Rinard. Ownership types for safe region-based memory management in real-time java. In *ACM SIGPLAN Notices*, volume 38, pages 324–337. ACM, 2003.

[6] W.-N. Chin, F. Craciun, S. Qin, and M. Rinard. Region inference for an object-oriented language. In *ACM SIGPLAN Notices*, volume 39, pages 243–254. ACM, 2004.

[7] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *ACM SIGPLAN Notices*, volume 33, pages 48–64. ACM, 1998.

[8] T. Guan, E. Zaluska, and D. De Roure. A grid service infrastructure for mobile devices. In *Semantics, Knowledge and Grid, 2005. SKG'05. First International Conference on*, pages 42–42. IEEE, 2005.

[9] M. R. Jantz, F. J. Robinson, P. A. Kulkarni, and K. A. Doshi. Cross-layer memory management for managed language applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2015.

[10] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1):1–41, 2005.

[11] H. Rim, S. Kim, Y. Kim, and H. Han. Transparent method offloading for slim execution. In *Wireless Pervasive Computing, 2006 1st International Symposium on*, Jan 2006. doi: 10.1109/ISWPC.2006.1613608.

[12] G. Salagnac, S. Yovine, and D. Garbervetsky. Fast escape analysis for region-based memory management. *Electronic Notes in Theoretical Computer Science*, 131:99–110, 2005.

[13] G. Salagnac, C. Rippert, and S. Yovine. Semi-automatic region-based memory management for real-time java embedded systems. In *13th IEEE RTCSA 2007*. IEEE, 2007.

[14] K. Sinha and M. Kulkarni. Techniques for fine-grained, multi-site computation offloading. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Computer Society, 2011.

[15] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *ACM SIGPLAN Notices*, volume 46, pages 17–30. ACM, 2011.

[16] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a Java bytecode optimization framework. In *Conference proceedings of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1999.

[17] C. Wang and Z. Li. A computation offloading scheme on handheld devices. *Journal of Parallel and Distributed Computing*, 64(6):740–746, 2004.

[18] L. Wang and M. Franz. Automatic partitioning of object-oriented programs for resource-constrained mobile devices with multiple distribution objectives. In *Parallel and Distributed Systems, 2008. ICPADS '08. 14th IEEE International Conference on*, pages 369–376, Dec 2008. doi: 10.1109/ICPADS.2008.84.

[19] K. Yang, S. Ou, and H.-H. Chen. On effective offloading services for resource-constrained mobile devices running heavier mobile internet applications. *Communications Magazine, IEEE*, 46(1):56–63, 2008.