Static Detection of Loop-Invariant Data Structures

Guoqing Xu[1], Dacong Yan[2], and Atanas Rountev[2]

[1] University of California, Irvine, CA, USA
[2] Ohio State University, Columbus, OH, USA

Abstract. As a culture, object-orientation encourages programmers to create objects, both short- and long-lived, without concern for cost. Excessive object creation and initialization can cause severe runtime bloat, which degrades significantly application performance and scalability. A frequently-occurring coding pattern that may lead to large volumes of (temporary) objects is the creation of objects that, while allocated per loop iteration, contain values independent of specific iterations. Finding these objects and moving them out of loops requires sophisticated interprocedural analysis, a task that is difficult for traditional dataflow analyses such as loop-invariant code motion to accomplish.

Our work targets *data structures* that are loop-invariant, and presents a static type and effect system to detect loop-invariant data structures. For each loop, our analysis inspects each logical data structure in order to find those that have disjoint instances per loop iteration and contain loop-invariant data. Instead of automatically hoisting them to improve performance (which is over-conservative), we report *hoistability measurements* for each disjoint loop data structure detected by our analysis. Eventually these data structures are ranked based on these measurements and are presented to the user to help manual tuning. We have performed a variety of studies on a set of 19 moderate/large-sized Java benchmarks. With the help of hoistability measurements, we found optimization opportunities in most of the programs that we inspected and achieved significant performance improvements in some of them (e.g., 82.1% running time reduction).

1 Introduction

As a culture of object-orientation, Java programmers are taught to freely create objects for whatever tasks they want to achieve, without concern for cost. They often take for granted that the runtime system can optimize away all execution inefficiencies: the Just-In-Time (JIT) compiler can remove whatever redundancy exists in the code, and the Garbage Collector (GC) can quickly reclaim redundant objects created for simple tasks. However, creating an object in Java with a `new` operator, in most cases, is far beyond allocating memory space, and can be much more expensive than a programmer realizes.

For example, object creation may need to execute large volumes of code to construct and initialize a data structure, and this process may even involve many slow I/O operations. One especially important case is when these expensive objects have data that is invariant. Frequently constructing data structures with unchanged data may have significant effect on application running time and scalability. Large improvements can often be seen when these data structures are reused rather than recreated.

Loops are places where such data structures can cause significant harm and thus special attention needs to be paid to find and optimize them. We propose static analyses

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 738–763, 2012.

```
for(int i = 0; i < N; i++){
      SimpleDateFormat sdf = new SimpleDateFormat();
      try{
             Date d = sdf.parse(date[i]);
             ...
        }catch(...) {...}
}
```
(a)

```
Templates _template = factory.newTemplates(stylesheet);
while(...){
      XMLFile file = getNewInputFile();
      XMLTransformer transformer = _template.newTransformer();
      transformer.transform(file);
      ...
}
```
(b)

Fig. 1. Real-world examples of heavy-weight creation of loop-independent data structures. (a) A `SimpleDateFormat` object is created inside the loop to parse the given `Date` objects; (b) An `XMLTransformer` object is created within the loop to transform the input XML files.

that can find data structures that are created in a loop but are independent of specific iterations. This work is motivated by bloat patterns that are regularly seen in large-scale applications. Figure 1 shows two examples extracted from the real-world programs that we have studied. The code pattern in part (a) has appeared a great number of times in applications that were written by IBM's customers and tuned by a group from IBM Research [1]. The programmer may have never realized that creating one `SimpleDateFormat` object requires to load many resource bundles to get the current date, compile the default date pattern string, and load the time zone to create a calendar. The process involves many expensive operations such as object clones, hash table lookups, etc. Part (b) illustrates a problem detected by our tool in `DaCapo/xalan`. An `XMLTransformer` object is created in a loop to transform the input XML file. While the input file is updated per loop iteration, the transformer object is loop-invariant. A great amount of effort is needed to create a transformer and significant performance improvement can be achieved after hoisting the creation of this transformer. Details of this example can be found in Section 5.

Technical Challenges. While loop optimizations have been extensively studied and used in modern optimizing compilers [2], they are mostly intraprocedural and deal only with instructions that operate on scalar variables and simple data structures (e.g., arrays and linked lists). They are far from reaching our goal of finding large optimization opportunities in programs that make extensive use of object-oriented data structures. Techniques such as loop-invariant code motion target instructions whose input variables are not defined in the loop. Such techniques are usually ineffective at handling instructions involving objects: for an object created in the loop, even though one of its fields used in an instruction is not defined, it is not safe to move this instruction out of the loop, as other fields of the object may be modified elsewhere in the loop. In an object-oriented program, data abstractions are much more complex and data in different locations are tightly coupled based on logical object models.

Focusing on Logical Data Structures. In this work, we focus on the data side of the hoisting problem, that is, to find logical data structures that are loop-invariant, regardless of whether or not it is possible to hoist the actual code statements that access these data structures. If a logical data structure is loop-invariant, the programmer should modify its creating and accessing code statements in order to move it out of the loop. There are two important aspects in determining whether a logical data structure is hoistable. First, it is critical to understand *how this data structure is built up*. For example, all objects in a hoistable data structure have to be allocated together in one iteration of the loop. In addition, any object in a hoistable data structure must be owned only by this data structure, and it cannot escape to other data structures. These properties can be verified by checking *points-to relationships* among objects. Second, it is important to understand *where this data structure gets its values from*. For example, all values contained in (heap locations of) a hoistable data structure must *not* be computed from any loop-iteration-specific value. This aspect of the problem is naturally related to the *data dependence* problem, and thus, such (value origin) properties can be verified by checking *data-dependence* relationships.

These two kinds of relationships are formalized as two (points-to and dependence) effects by a type and effect system presented in Section 3. As the identification of loop-invariant data structures requires to reason about whether objects connected by these relationships are always created in the same iteration of a loop, our analysis computes, for each loop object, a *loop iteration count abstraction* that indicates whether or not an instance of the object created in one iteration of the loop can be carried over to the next iteration. A more detailed description of this abstraction can be found in Section 2. Section 3 presents a formalism that computes such abstractions.

Manual Tuning with the Help of Hoistability Measurement. Given logical loop-invariant data structures identified by our analysis, the second challenge lies in how to perform the actual hoisting. While it is attractive to design a transformation technique that automatically pulls out invariant data structures, we found that there is little hope that a completely automated approach can effectively hoist these data structures in practice. This is first because of the over-conservative nature of any transformation technique, which may prevent the technique from hoisting many real-world loop-invariant data structures due to their complex usage in large-scale applications. The chance of developing an effective transformation technique becomes even smaller in the presence of the many Java dynamic features such as dynamic class loading and reflection. Second, effectively optimizing real-world data structures requires developer insight. For example, a data structure with 100 fields cannot be transformed if it has even a single non-loop-invariant field. In fact, by manually inspecting and perhaps modifying the data model, it is highly likely that the data structure can be made hoistable (i.e., by introducing a separate object to store that loop-dependent field).

Our work advocates a semi-automated approach that is intended to identify larger optimization opportunities by bringing developer insight into the optimization process. Instead of eagerly looking only for *completely-hoistable* logical data structures, we also identify *partially-hoistable* logical data structures, by computing a *hoistability measurement* for each logical data structure, and rank all such data structures based on these measurements to help manual tuning. The higher measurement a data structure has, the more likely it is that this data structure can be manually hoisted.

One additional advantage of manual tuning using hoistability measurements is that these metrics can be easily modified to incorporate dynamic information obtained from a profile. Section 4 presents one such modification that includes loop frequencies in the metrics so that the "loop hotness" factor is taken into account when the rank of a data structure is computed. To optimize a non-hoistable data structure, the programmer can either split the data model (e.g., to separate the loop-invariant fields and non-invariant fields) and/or restructure the statements that access it (e.g., to eliminate dependences between hoistable and non-hoistable statements). In addition, highly-ranked data structures can often be indicators of other loop-related inefficiencies, such as inappropriate implementation choices. These problems may also be revealed during the inspection of the reported data structures.

Evaluation. We evaluated our technique using a set of 19 Java programs. With the help of hoistability measurements, we found optimization opportunities in most of these programs. We discuss the performance gains we have achieved for five representative programs: `ps`, `xalan`, `bloat`, `soot-c`, and `sablecc-j`. For example, we found a performance problem in `DaCapo/xalan`; removing it can improve the benchmark performance by 10.1%. As another example, we found a bottleneck in the core components of `ps`. After the optimization, the running time was reduced by 82.1%. Detailed description of the empirical evaluation can be found in Section 5. These results indicate that the proposed technique can be useful both in the coding phase (for finding small performance issues before they pile up) and in the tuning phase (for identifying performance bottlenecks).

2 Overview

Figure 2 shows a simple running example. This example contains 10 allocation sites (including string literals), and all of them are located in loops (i.e., either directly in a loop or in a method invoked in a loop). Our analysis inspects each object[1] located in a loop and discovers its structure (i.e., including objects that are calling-context-sensitively reachable from it) using context-free language (CFL)- reachability. Using new CFL-reachability algorithms, we develop novel techniques that inspect individual objects in a loop, identify their structures while taking into account the calling contexts of methods, and compute their hoistability, all without requiring a pre-computed whole-program points-to solution.

Points-to Relationships. Figure 3(a) illustrates three data structures (a.1), (a.2), and (a.3) that are rooted at objects in the two loops shown in Figure 2. Each object is given a name o_i, where i is the number of the line in the code where the object is created. Each edge in a data structure represents a points-to relationship, and is annotated with a field name and a pair of integers (i, j). Field `elm` is a special field used to represent array elements. Integers in this pair are the loop *iteration count abstractions* (ICAs) for the two objects connected by this edge, and they can be used to determine whether these objects are created in the same iteration of the loop. Following the iteration abstraction [3] and the recency abstraction [4], an ICA can be one of three (abstract) values: 0, 1, or $\top$. Note

[1] "Object" will be used in the rest of the paper to denote a static abstraction (i.e., an allocation site), while "instance" denotes a run-time object.

```
1 class List{
2   Object[] arr; int index = 0;
3   List() { arr = new Object[1000];}
4   void add(Object o){ if(index < 1000) arr[index++] = o;}
5   Object get(i){ return arr[i]; }
6 }
7 class Pair{
8   Object f; Object g;
9   Pair(Object o1, Object o2){ this.f = o1; this.g = o2;}
10}
11
12 class Client{
13   static void main(String[] args){
14     for(int i = 0; i < 500; i++){
15        List l = new List();
16        Pair p = new Pair("hello", "world");
17        l.add(p);
18     }
19     Integer b = null;
20     for(int j = 0; j < 400; j++){
21        Integer a = new Integer(j);
22        if(j == 20) b = new Integer(10);
23        Pair q = new Pair(a, b);
24        Pair r = new Pair("good", a);
25        ... //use q and r
26     }
27  }
```

Fig. 2. Running example

that the use of ICA is not a contribution of this paper. The three major contributions are (1) a novel calling-context-sensitive algorithm for computing points-to and dependence relationships that are annotated with ICA information, (2) a new technique for detecting loop-invariant data structures with the help of such points-to/dependence information, and (3) the development of quantitative measurements that use these relationships to help programmers identify hoistable data structures.

For a particular loop l, an object whose ICA is 0 with respect to l must be created outside l. The ICA for an object being either 1 or $\top$ (with respect to l) means the object must be created inside l. In particular, let us consider a run-time iteration p of l and a run-time instance r created by allocation site o_r such that r is live during the execution of p. If the ICA for o_r is 1, r is guaranteed to be created during iteration p. In other words, the ICA for an object being 1 indicates that its instances must be "fresh" across iterations, that is, in any iteration where an instance of it is live, this instance must be created in that iteration (i.e., it must *not* be carried over from a previous iteration). An object that has a $\top$ in its ICA is created in a (previous) unknown iteration. For example, the ICA for o_{23} is 1, as it creates a fresh object in each iteration of the loop at line 20. The ICA for o_{22} is $\top$, as the instance it creates in one iteration can be carried over to the next iteration.

Hence, for a points-to edge annotated with (i, j), $i = j = 1$ guarantees that the two objects connected by the edge must be created in the same iteration of the loop, while either i or j being $\top$ indicates that the two objects may be created in different iterations. A data structure is obviously not hoistable if it contains objects that are created in different iterations.

Dependence Relationships. Figure 3(b) illustrates the dependence (def-use) chains that start at memory locations in each data structure shown in Figure 3(a). An edge $o_1.f \leftarrow o_2.g$ indicates that a run-time value contained in a heap location abstracted by

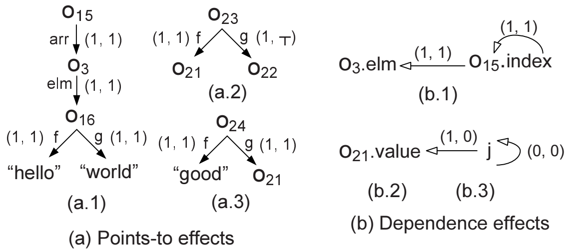

Fig. 3. Data structures identified for the running example and their effect annotations. (a) Points-to effects among objects; (b) Dependence effects among memory locations.

$o_2.g$ is required in computing a value written into (a heap location abstracted by) $o_1.f$. Note that o_{21}.*value* is a field of class `Integer` that stores the int value embedded in an `Integer` object. A stack location is also considered in a dependence chain, if this location (variable) has a primitive type and is in the method that contains the loop of interest (e.g., variable j in method `main`). Such a variable needs to be taken into account as it may contain iteration-specific values. Variables that are not in the loop-containing method are abstracted away from dependence chains, as they can get iteration-specific values only from heap locations or variables in the loop-containing method (e.g., variable o at line 4). Reference-typed variables (e.g., a and b) do not need to be considered as well (even though they are in the loop-containing method), because they can get loop-specific values only from heap locations, and thus, it is necessary to track only heap locations.

Note that each dependence edge $o_1.f \leftarrow o_2.g$ is also annotated with a pair of ICAs (for o_1 and o_2), which is used to determine whether o_1 and o_2 are always created in the same loop iteration. If a node in a dependence edge is a stack variable, such as j, its ICA is determined by whether or not it is declared in the loop. For example, j's ICA is 0, because it is declared before the loop starts. Its ICA would have been 1 if it were declared in the loop. The ICA for a stack variable can never be $\top$, as each variable declared in a loop must be initialized (i.e., get a new value) per iteration.

Hoistable Logical Data Structures. As discussed earlier in Section 1, the analysis identifies (completely or partially) hoistable logical data structures from a *purely data perspective*, regardless of the actual code and control flow. This can be done by reasoning about these two kinds of annotated relationships. A hoistable data structure has the following important properties.

(1) *(Disjoint).* Its run-time instances, created by different iterations of the loop, have to be disjoint. No object is allowed to appear in multiple instances of one single logical data structure. This property can be verified by checking whether all points-to edges in a data structure are annotated with $(1, 1)$. A data structure is not hoistable if any of its nodes has a non-1 ICA. This guarantees that any instance of a hoistable data structure does not have objects created outside the loop or in different iterations. For example, (a.2) is not hoistable, as edge $o_{23} \xrightarrow{g} o_{22}$ may connect objects created in different iterations.

(2) *(Loop-invariant).* Fields of objects in a hoistable data structure have to be loop-invariant. No data in any run-time instance of the data structure can be dependent on specific loop iterations. We check this property by formulating it as a data-dependence

problem. A sufficient condition for the statement "object o is loop-invariant" is that, for each field of o, (2.1) no edge on a dependence chain (e.g., shown in Figure 3(b)) that starts from the field can have $\top$ in its annotated ICA pair, and (2.2) for each memory location node $o.f$ (i.e., heap location) or j (i.e., stack location) on the chain, if the ICA for o or j is 0, this node must not be involved in a *dependence cycle*.

(2.1) enforces that all (stack and heap) locations from which a hoistable data structure instance (allocated in one iteration) gets its data are either created in this same iteration, or exist before the loop starts. No data in this instance can be obtained from an object created in a different iteration. In addition, as stated in (2.2), if one such location already exists before the loop starts (e.g., variable j), this node must not be in a dependence cycle. Otherwise, its value may be updated by each iteration and any data structure that is dependent on this value is not hoistable. For example, in Figure 3(b), o_{21}.*value* depends on variable j, whose ICA is 0. Because j is in a dependence cycle, o_{21}.*value* may have iteration-specific values, and thus, any data structure that contains o_{21} is not hoistable (e.g., structures (a.2) and (a.3) in Figure 3(a)). Note that field o_3.*elm* is loop-invariant: while it depends on field o_{15}.*index*, which is involved in a cycle, o_{15}'s ICA is 1. Hence, it is impossible for an iteration-specific value to propagate to this field.

Note that these two properties are sufficient (but not necessary) conditions for hoistable logical data structures. For example, the first condition (i.e., disjointness) is an over-conservative approximation of the shape of a hoistable data structure—it is perfectly possible for a hoistable data structure to contain objects that are created outside the loop (i.e., their ICAs are 0) but not mutated in the loop. We choose not to consider such objects in our hoistable data structure definition primarily for scalability purposes—these objects (created outside the loop) may have long dependence chains (as objects that they reference can come from arbitrary places). On the contrary, dependence chains for objects that are created in the loop and do not escape the loop (i.e., all their ICAs are 1) are generally much shorter. Therefore, the dependence analysis is much more scalable when considering only these chains.

Computing Hoistability Measurements. After inspecting these two conditions, it is clear that only data structure (a.1) is a completely hoistable logical data structure. However, there might still exist optimization opportunities with the other two (partially hoistable) data structures. For example, if we can move object o_{22} out of data structure (a.2), it may still be possible to hoist (a.2). In order to help programmers discover such hidden optimization opportunities, hoistability measurements are proposed to quantify the likelihood of manually hoisting data structures out of loops.

For example, for each data structures shown in Figure 3, we compute two separate hoistability measurements based on the two orthogonal (points-to and dependence) effects mentioned above: structure-based hoistability (SH) that considers how many objects in the data structure must be allocated in the same loop iteration (i.e., that comply with condition 1), and dependence-based hoistability (DH) that considers how many fields in the data structure must contain loop-invariant data (i.e., that comply with condition 2). Eventually, these three data structures are ranked based on the two measurements and are then presented to the user for further inspection. Detailed description of hoistability measurements can be found in Section 4.

Variables	$a, b \in$	V
Allocation sites	$o \in$	O
Instance fields	$f \in$	F
Labels	$l \in$	L
Statements	$e \in$	E

$$e ::= \; a = b \mid a = new\ ref^{o} \mid a = b.f \mid a.f = b \mid a = \mathsf{null} \mid$$
$$e\ ;\ e \mid \text{if } (*) \text{ then } e \text{ else } e \mid \text{while}^{l}\ (*) \text{ do } e$$

(a)

Iteration count	i	$::= 0 \mid 1 \mid 2 \mid \ldots$	$\in \mathsf{N}$
Iteration map	ν	$\in \mathsf{L} \rightarrow \mathsf{N}$	
Loop status	π	$::= \langle l\ ,\ i \rangle$	$l \in \mathsf{L} \cup \{0\}$
Labeled object	$\hat{o}$	$::= o^{\pi}$	$\in \Phi$
Heap	σ	$\in \Phi \times \mathsf{F} \rightarrow \Phi \cup \{\perp\}$	
Environment	ρ	$\in \mathsf{V} \rightarrow \Phi \cup \{\perp\}$	
Data origin	μ	$\in \mathsf{V} \rightarrow 2^{\Phi \times \mathsf{F}}$	
Heap points-to effect	H	$::= \emptyset \mid H \cup \{\hat{o}_1 \rhd^{f} \hat{o}_2\}$	
Heap data dep. effect	Ω	$::= \emptyset \mid \Omega \cup \{\hat{o}_1.f \prec \hat{o}_2.g\}$	

(b)

Fig. 4. A simple `while` language: (a) abstract syntax; (b) semantic domains

3 Loop-Invariant Logical Data Structures

This section formalizes the notion of loop-invariant logical data structure, and in this context, formally defines our analysis that identifies hoistable data structures. The presentation proceeds in three steps. First, we define a simple imperative language and present its abstract syntax and operational semantics, which we will use to formalize our analysis algorithms. For the ease of presentation, function calls are not considered in this language. Our implementation supports full context-sensitivity using a CFL-reachability formulation.

Second, we present a type and effect system that abstracts concrete objects and effects. Finally, the analysis that detects hoistable data structures is described based on the abstract heap effects generated by the type and effect system.

3.1 Language, Semantics, and Effect System

Language. The abstract syntax and the semantic domains for the simple `while` language that we use are defined in Figure 4. A program in this language has a fixed set of global variables with reference types. While primitive-typed variables are considered in our analyses, they are excluded from this language for the simplicity of presentation. Each allocation site is labeled with an ID o. Each loop is annotated with a natural number label l ($l > 0$), which will be used as the ID of the loop. * denotes a side-effect-free boolean expression that contains only local variables and constants.

We develop a concrete operational semantics for the language in order to detect hoistable data structures. A loop iteration count i records the number of iterations that a loop has executed. A global loop iteration map ν maps each loop (label) to its current iteration count. Each object instance is represented as its allocation site o annotated with a pair $\langle l, i \rangle$, where l is the label of the loop in which o is located (always > 0), and i is the count of the iteration of l that creates this instance. If an object is not located in any loop, the loop status π for its instances is always $\langle 0, 0 \rangle$. Our analysis does not consider hoisting objects out of nested loops, and in the presentation we assume that all loops in the abstract language are not nested. While nested loops can be handled easily in our framework (e.g., by creating and associating with each object an *iteration count map* that records an iteration count for each loop in which the object is located), we found that it is not useful in hoisting data structures for real-world Java programs: it is extremely rare that a data structure can be hoisted out of multiple loops.

A heap σ records object reference relationships, and an environment ρ maps variables to objects in the heap. They are defined in standard ways. A data origin map μ records, for each stack variable v, a set of *heap locations* such that values in these locations are required (i.e., either as a pointer for dereferencing, or being copied through a sequence of intermediate stack locations) in order to obtain a value written to v. This map tracks dependences between variables and their relevant heap locations, and will be used to compute *dependence effects* as described shortly. For example, after the execution of a sequence of statements $c = d.f; b = c; a = b$, we have $\mu(a) = \mu(b) = \mu(c) = \{o_d.f\} \cup \mu(d)$, where o_d represents the object that d points to. $\mu(d)$ is included here because d is required as a reference to an object from which the value is obtained. Dependences via the intermediate stack locations (e.g., b and c) are abstracted away as we are interested only in fields of objects that form data structures.

Note that μ records only one-hop heap location dependence—if the value in $d.f$ is obtained from another heap location, $\mu(a)$, $\mu(b)$ and $\mu(c)$ remain the same. Multi-hop heap location dependences can be obtained by computing the transitive closure of μ.

As discussed earlier in Section 2, we use *points-to* and *dependence* relationships to reason about (1) how data structures are built up and (2) where they get values from, respectively. These relationships are modeled by the following two kinds of effects in our system. A heap *points-to effect* $\hat{o}_1 \rhd^f \hat{o}_2 \in H$ is generated if, at a certain point, object $\hat{o}_2$ becomes reachable from $\hat{o}_1$ through field f. A *data dependence effect* tracks the flow of data. One such effect $\hat{o}_1.f \prec \hat{o}_2.g \in \Omega$ indicates that $\hat{o_2}.g$ is required in order to compute a value written into $\hat{o_1}.f$. This effect captures a transitive data dependence relationship between two heap locations, abstracting away a possible sequence of dependences via intermediate stack variables. Data dependence effects can be computed efficiently by using the data origin map μ.

Note that in this language, it is safe for a dependence chain to *not* include any stack variable (like j in Figure 3 (b)). This is because the language supports only reference-typed variables, which can never form a dependence cycle themselves (without a heap location involved). While we choose not to include primitive types in this language (for the simplicity of presentation), our implementation handles both primitive and reference types. It is also important to note that the effects shown in Figure 4 are *concrete effects*. We will present an approach to project them into *abstract effects*, which are essentially the annotated edges shown in Figure 1.

Concrete Instrumented Semantics. Figure 5 presents a big-step operational semantics for our language. A judgment of the form

$$e, \nu, \sigma, \rho, \mu \Downarrow \nu', \sigma', \rho', \mu', H, \Omega$$

$$a = \text{null}, \nu, \sigma, \rho, \mu \Downarrow \nu, \sigma, \rho[a \mapsto \bot], \mu[a \mapsto \emptyset], \emptyset, \emptyset \quad (\textsc{Assign-Null})$$

$$\frac{\rho' = \rho[a \mapsto \hat{o}] \qquad \sigma' = \sigma[\forall f.(\hat{o}.f \mapsto \bot)] \qquad \hat{o}.o = alloc \qquad \hat{o}.\pi = \langle l\ ,\ \nu(l)\rangle}{a = \text{new ref}^{alloc}, \nu, \sigma, \rho, \mu \Downarrow \nu, \sigma', \rho', \mu[a \mapsto \emptyset], \emptyset, \emptyset} \quad (\textsc{New})$$

$$a = b, \nu, \sigma, \rho, \mu \Downarrow \nu, \sigma, \rho[a \mapsto \rho(b)], \mu[a \mapsto \mu(b)], \emptyset, \emptyset \quad (\textsc{Assign})$$

$$\frac{\rho(b) = \hat{o} \qquad \mu' = \mu[a \mapsto \mu(b) \cup \{\hat{o}.f\}]}{a = b.f, \nu, \sigma, \rho, \mu \Downarrow \nu, \sigma, \rho[a \mapsto \sigma(\hat{o}.f)], \mu', \emptyset, \emptyset} \quad (\textsc{Load})$$

$$\frac{\begin{array}{c}\rho(a) = \hat{o}_1 \qquad \rho(b) = \hat{o}_2 \\ H = (\hat{o}_2 = \text{null} ~?~ \emptyset : \{\hat{o}_1 \rhd^f \hat{o}_2\}) \qquad \Omega = \bigcup\{\hat{o_1}.f \prec \hat{o_i}.g_i \mid \hat{o}_i.g_i \in \mu(a) \cup \mu(b)\}\end{array}}{a.f = b, \nu, \sigma, \rho, \mu \Downarrow \nu, \sigma[\hat{o}_1.f \mapsto \hat{o}_2], \rho, \mu, H, \Omega} \quad (\textsc{Store})$$

$$\frac{e_1, \nu, \sigma, \rho, \mu \Downarrow \nu', \sigma', \rho', \mu', H_1, \Omega_1 \qquad e_2, \nu', \sigma', \rho', \mu' \Downarrow \nu'', \sigma'', \rho'', \mu'', H_2, \Omega_2}{e_1; e_2, \nu, \sigma, \rho, \mu \Downarrow \nu'', \sigma'', \rho'', \mu'', H_1 \cup H_2, \Omega_1 \cup \Omega_2} \quad (\textsc{Comp})$$

$$\frac{e_1, \nu, \sigma, \rho, \mu \Downarrow \nu', \sigma', \rho', \mu', H, \Omega}{\text{if } (*) \text{ then } e_1 \text{ else } e_2, \nu, \sigma, \rho, \mu \Downarrow \nu', \sigma', \rho', \mu', H, \Omega} \quad (\textsc{If-Else-1})$$

$$\frac{e_2, \nu, \sigma, \rho, \mu \Downarrow \nu', \sigma', \rho', \mu', H, \Omega}{\text{if } (*) \text{ then } e_1 \text{ else } e_2, \nu, \sigma, \rho, \mu \Downarrow \nu', \sigma', \rho', \mu', H, \Omega} \quad (\textsc{If-Else-2})$$

$$\frac{\begin{array}{c}e, \nu[j \mapsto \nu(j) + 1], \sigma, \rho, \mu \Downarrow \nu', \sigma', \rho', \mu', H_1, \Omega_1 \\ \text{while}^j\ (*) \text{ do } e, \nu', \sigma', \rho', \mu' \Downarrow \nu'', \sigma'', \rho'', \mu'', H_2, \Omega_2\end{array}}{\text{while}^j\ (*) \text{ do } e, \nu, \sigma, \rho, \mu \Downarrow \nu'', \sigma'', \rho'', \mu'', H_1 \cup H_2, \Omega_1 \cup \Omega_2} \quad (\text{W})$$

Fig. 5. Concrete instrumented semantics

starts with a statement e, which is followed by loop iteration map ν, heap σ, environment ρ, and value origin map μ. The execution of e terminates with a final iteration map ν', heap σ', environment ρ', origin map μ', heap points-to effect set H, and heap data dependence effect set Ω.

Rules Comp, If-Else1, If-Else2, and W are defined in expected ways. In rule Assign-Null and New, the data origin for a (in μ) is assigned $\emptyset$, as the value is freshly generated and does not depend on any heap value. In rule New, for a labeled object $\hat{o}$, $\hat{o}.o$ and $\hat{o}.\pi$ denote the allocation site and the loop status pair for $\hat{o}$, respectively. In $\hat{o}.\pi$, the loop l where the allocation site o is located is determined statically, and is associated with each instance created by o. The iteration count for l is retrieved from the global iteration count map ν. Rule Assign propagates both the object reference and the data origin of this reference value from b to a. In rule Load, the data origin map μ for variable a is updated in a way so that both $\hat{o}.f$ and the origin of the value in b are recorded as a's origin. Hence, dependences via both the value copy (from $b.f$ to a) and the pointer dereference (i.e., dereferencing b) are captured.

To handle a store $a.f = b$ where b's value is written to the heap, a points-to effect $\hat{o}_1 \rhd^f \hat{o}_2$ is first generated. The rule next generates a set of heap dependence effects $\{\hat{o}_1.f \prec \hat{o}_i.g_i \mid \hat{o}_i.g_i \in \mu(b) \cup \mu(a)\}$, by consulting the data origin map μ. Each dependence effect states that a value read from field g_i of object $\hat{o}_i$ has been used to produce a value written to $\hat{o}_1.f$ during the execution.

▶ **Example** For illustration, consider the following example:

a = `new ref`^{o_1}; e = `new ref`^{o_2}; $a.f = e$; $j = a.f$;
`while`^{1} (j) `do`$\{$ $b = a.f$; $d = b$; c = `new ref`^{o_3}; $c.g = d$;$\}$

Iteration count abstr.	$\tilde{i}$	$::= 0 \mid 1 \mid \top$	$\in \mathbb{N}$
Loop status abstr.	$\tilde{\pi}$	$::= \langle l, \tilde{i} \rangle$	$l \in \mathsf{L} \cup \{0\}$
Type	$\tilde{\tau}$	$::= o^{\tilde{\pi}} \mid \top$	$\in \mathbb{T}$
Type environment	Γ	$\in \mathsf{V} \to \mathbb{T} \cup \{\bot\}$	
Data origin abstr.	$\tilde{\mu}$	$\in \mathsf{V} \to 2^{\mathbb{T} \times \mathsf{F}}$	
P.T. effect abstr.	$\tilde{H}$	$::= \emptyset \mid \tilde{H} \cup \{\tilde{\tau_1} \trianglerighteq \tilde{\tau_2}\}$	
Dep. effect abstr.	$\tilde{\Omega}$	$::= \emptyset \mid \tilde{\Omega} \cup \{\tilde{\tau_1}.f \preceq \tilde{\tau_2}.g\}$	

Fig. 6. Syntax of types and abstract effects

At the end of the first iteration of the loop, the semantic domains contain the following values:

$\nu = [1 \mapsto 1]$,
$\sigma = [\hat{o_1}.f \mapsto \hat{o_2}, \hat{o_3}.g \mapsto \hat{o_2}]$,
$\rho = [a \mapsto \hat{o_1}, b \mapsto \hat{o_2}, c \mapsto \hat{o_3}, d \mapsto \hat{o_2}, e \mapsto \hat{o_2}, j \mapsto \hat{o_2}]$,
$\mu = [a \mapsto \emptyset, b \mapsto \{\hat{o_1}.f\}, c \mapsto \emptyset, d \mapsto \{\hat{o_1}.f\}, e \mapsto \emptyset, j \mapsto \{\hat{o_1}.f\}]$,
$H = \{\hat{o_1} \rhd^f \hat{o_2}, \hat{o_3} \rhd^g \hat{o_2}\}$,
$\Omega = \{\hat{o_3}.g \prec \hat{o_1}.f\}$. ◀

Abstract Semantics. The concrete semantics uses an unbounded number of objects and unbounded loop iteration counts. We next develop a type and effect system that describes an abstract semantics, which conservatively approximates the concrete semantics with a bounded set of objects and bounded loop iteration counts. The syntax of types and abstract effects are illustrated in Figure 6. The abstraction of each concrete domain (e.g., π) shown in Figure 4 is represented by its corresponding tilded symbol (e.g, $\tilde{\pi}$). Environment ρ is abstracted by the type environment, denoted by Γ. A type $\tilde{\tau}$ abstracts a labeled object instance $\hat{o}$ by projecting its concrete iteration count $\hat{o}.\pi.i$ to an iteration count abstraction (ICA) $\tilde{\tau}.\tilde{\pi}.\tilde{i}$, which can have three abstract values: 0, 1, and $\top$. The meaning of these values was explained in Section 2. Using this type and effect system, we can identify data structures whose objects are guaranteed to be created in the same iteration by reasoning about object ICAs. Note that each abstract effect in $\tilde{H}$ and in $\tilde{\Omega}$ corresponds to an edge in Figure 3 (a) and in Figure 3 (b), respectively.

Figure 7 shows the type rules, which are parallel with the operational semantics in Figure 5. Auxiliary operations used in the type rules are shown in Figure 8. Some abstract semantic domains in Figure 6 are extended with $\top$ and/or $\bot$ elements, as necessary.

Since the type and effect system abstracts the concrete semantics in Figure 5, most of the rules in Figure 7 are similar to their corresponding operational semantics rules. Here we explain only a few of them that differ significantly from their concrete counterparts. In rule TNEW, the ICA for a newly created object is always 1, and this value will be changed later (by rule TW) if this object is carried over to the next iteration. (For objects created outside of loops, the ICA is 0; for brevity, this variation of TNEW is not shown in Figure 7.) Rule TLOAD types variable a with an unknown type $\top$. This handling is over-conservative for the purpose of soundness. Our implementation improves this by consulting a points-to graph that is computed on demand.

Type environment join ($\uplus$) needs to be performed in order to handle different control flow paths of an `if-else` statement (in Rule TIF-ELSE). Joining two environments

$$\Gamma, \tilde{\mu} \vdash a = \text{null} : \Gamma'[a \mapsto \bot], \tilde{\mu}[a \mapsto \emptyset], \emptyset, \emptyset \quad (\text{TASSIGN-NULL})$$

$$\frac{\Gamma' = \Gamma[a \mapsto \tilde{\tau}] \qquad \tilde{\tau}.o = alloc \qquad \tilde{\tau}.\tilde{\pi} = \langle l\ ,\ 1 \rangle}{\Gamma, \tilde{\mu} \vdash a = \text{new ref}^{alloc} : \Gamma', \tilde{\mu}[a \mapsto \emptyset], \emptyset, \emptyset} \quad (\text{TNEW})$$

$$\Gamma, \tilde{\mu} \vdash a = b : \Gamma[a \mapsto \Gamma(b)], \tilde{\mu}[a \mapsto \tilde{\mu}(b)], \emptyset, \emptyset \quad (\text{TASSIGN})$$

$$\frac{\tilde{\tau} = \Gamma(b) \qquad \tilde{\mu}' = \tilde{\mu}[a \mapsto \tilde{\mu}(b) \cup \{\tilde{\tau}.f\}]}{\Gamma, \tilde{\mu} \vdash a = b.f : \Gamma[a \mapsto \top], \tilde{\mu}', \emptyset, \emptyset} \quad (\text{TLOAD})$$

$$\frac{\begin{array}{ll} \tilde{\tau}_1 = \Gamma(a) & \tilde{\Omega} = \bigcup\{\tilde{\tau}_1.f \preceq \tilde{\tau}_i.g_i \mid \tilde{\tau}_i.g_i \in \tilde{\mu}(b) \cup \tilde{\mu}(a)\} \\ \tilde{\tau}_2 = \Gamma(b) & \tilde{H} = \{\tilde{\tau}_1 \trianglerighteq^f \tilde{\tau}_2\} \text{ if } \tilde{\tau}_1 \neq \bot \text{ and } \tilde{\tau}_2 \neq \bot,\ \emptyset \text{ otherwise} \end{array}}{\Gamma, \tilde{\mu} \vdash a.f = b : \Gamma, \tilde{\mu}, \tilde{H}, \tilde{\Omega}} \quad (\text{TSTORE})$$

$$\frac{\Gamma, \tilde{\mu} \vdash e_1 : \Gamma', \tilde{\mu}', \tilde{H}_1, \tilde{\Omega}_1 \qquad \Gamma', \tilde{\mu}' \vdash e_2 : \Gamma'', \tilde{\mu}'', \tilde{H}_2, \tilde{\Omega}_2}{\Gamma, \tilde{\mu} \vdash e_1; e_2 : \Gamma'', \tilde{\mu}'', \tilde{H}_1 \cup \tilde{H}_2, \tilde{\Omega}_1 \cup \tilde{\Omega}_2} \quad (\text{TCOMP})$$

$$\frac{\Gamma, \tilde{\mu} \vdash e_1 : \Gamma', \tilde{\mu}', \tilde{H}_1, \tilde{\Omega}_1 \qquad \Gamma, \tilde{\mu} \vdash e_2 : \Gamma'', \tilde{\mu}'', \tilde{H}_2, \tilde{\Omega}_2}{\Gamma, \tilde{\mu} \vdash \text{if } (*) \text{ then } e_1 \text{ else } e_2 : \Gamma' \uplus \Gamma'', \tilde{\mu}[\forall v.(v \mapsto \tilde{\mu}'(v) \cup \tilde{\mu}''(v))], \tilde{H}_1 \cup \tilde{H}_2, \tilde{\Omega}_1 \cup \tilde{\Omega}_2} \quad (\text{TIF-ELSE})$$

$$\frac{\Gamma[\forall v.(v \mapsto (\Gamma(v).o)^{(\Gamma(v).\tilde{\pi}^j \oplus 1)})], \tilde{\mu} \vdash e : \Gamma, \tilde{\mu}, \tilde{H}, \tilde{\Omega}}{\Gamma, \tilde{\mu} \vdash \text{while}^j\ (*) \text{ do } e : \Gamma, \tilde{\mu}, \tilde{H}, \tilde{\Omega}} \quad (\text{TW})$$

Fig. 7. Typing

(rules 1-4 in Figure 8) needs to consider both allocation sites and abstract loop iteration counts contained in types. If two types $\tilde{\tau}_1$ and $\tilde{\tau}_2$ do not have the same allocation sites o (rule 2), performing join on them yields $\top$. Otherwise, their loop status abstractions $\tilde{\tau}_1.\tilde{\pi}$ and $\tilde{\tau}_2.\tilde{\pi}$ are forced to join (rule 3). Loop labels ($\tilde{\pi}.l$) in the two status pairs have to be the same because they are associated with the same allocation site. Joining ICAs $\tilde{i}_1$ and $\tilde{i}_2$ is shown in rule 4: if $\tilde{i}_1 \neq \tilde{i}_2$, the result is $\top$, meaning that nothing is known about the iteration where the object is created. A finite-height type lattice can be defined based on the operations in Figure 8, with $\top$ and $\bot$ as the maximum and minimum types in the lattice. Types with different allocation sites are not comparable.

In the beginning of each loop iteration (shown in rule TW), the ICA of each type (whose allocation site is under loop j) in the type environment is incremented by using operator $\oplus$, which is defined in Figure 8 (rule 5). The goal of this is to "clear the loop status" of the objects that are carried over from the last iteration, so that these (old) objects and the fresh objects created in the current iteration can be distinguished. Note that a fixed point is computed for the handling of loops: while each iteration of the loop may yield a different solution, the fixed-point solution must not be smaller than this solution.

Next, we explain how to detect data structures whose objects are guaranteed to be allocated in the same loop iteration, using the type and effect system.

Lemma 1. (Connected objects created in the same iteration). *For each heap points-to effect $\tilde{\tau}_1 \trianglerighteq^f \tilde{\tau}_2 \in \tilde{H}$, if $\tilde{\tau}_1.\tilde{\pi}.\tilde{i} = \tilde{\tau}_2.\tilde{\pi}.\tilde{i} = 1$ for a particular loop j (i.e., $\tilde{\tau}_1.\tilde{\pi}.l = \tilde{\tau}_2.\tilde{\pi}.l = j$), in each iteration of j where an instance of $\tilde{\tau}_1.o$ and an instance of $\tilde{\tau}_2.o$ are connected by a store operation, these instances must be allocated in this same iteration.*

[Join of Γ]

(1) $\Gamma_1 \uplus \Gamma_2 = \Gamma_3$, where $\forall a \in \text{DOM}(\Gamma_3)$, $\Gamma_3(a) = \begin{cases} \Gamma_1(a) & \text{if } a \in \text{DOM}(\Gamma_1) \text{ and } a \notin \text{DOM}(\Gamma_2) \\ \Gamma_2(a) & \text{if } a \in \text{DOM}(\Gamma_2) \text{ and } a \notin \text{DOM}(\Gamma_1) \\ \Gamma_1(a) \uplus \Gamma_2(a) & \text{if } a \in \text{DOM}(\Gamma_1) \cap \text{DOM}(\Gamma_2) \end{cases}$

(2) $\tilde{\tau_1} \uplus \tilde{\tau_2} = \begin{cases} \tilde{\tau_1} & \text{if } \tilde{\tau_2} = \bot \\ \tilde{\tau_2} & \text{if } \tilde{\tau_1} = \bot \\ (\tilde{\tau_1}.o)^{\tilde{\tau_1}.\tilde{\pi} \uplus \tilde{\tau_2}.\tilde{\pi}} & \text{if } \tilde{\tau_1}.o = \tilde{\tau_2}.o \\ \top & \text{otherwise} \end{cases}$

(3) $\tilde{\pi_1} \uplus \tilde{\pi_2} = \langle \tilde{\pi_1}.l\ ,\ \tilde{\pi_1}.\tilde{i} \uplus \tilde{\pi_2}.\tilde{i} \rangle$

(4) $\tilde{i_1} \uplus \tilde{i_2} = \begin{cases} \tilde{i_1} & \text{if } \tilde{i_1} = \tilde{i_2} \\ \top & \text{otherwise} \end{cases}$

[Operator $\oplus$]

(5.1) $\tilde{\pi}^j \oplus 1 = \begin{cases} \langle \tilde{\pi}.l\ ,\ \tilde{\pi}.\tilde{i} \oplus 1 \rangle & \text{if } \tilde{\pi}.l = j \\ \tilde{\pi} & \text{otherwise} \end{cases}$

(5.2) $\tilde{i} \oplus 1 = \begin{cases} 1 & \text{if } \tilde{i} = 0 \\ \top & \text{otherwise} \end{cases}$

Fig. 8. Auxiliary operations

Proof sketch. Consider a specific iteration k of j. If both objects are allocated in this iteration, their corresponding abstract iteration counts $\tilde{\pi_1}.\tilde{i}$ and $\tilde{\pi_2}.\tilde{i}$ are both updated to 1 upon their creation (rule TNEW). In the very beginning of the next iteration $k + 1$, $\tilde{\tau_1}.\tilde{\pi}.\tilde{i}$ and $\tilde{\tau_2}.\tilde{\pi}.\tilde{i}$ will be incremented to $\top$ (rule TW) because these objects are carried over from the last iteration. If in this iteration, both of their allocation sites are executed again, the two ICAs (for the two new instances) are set back to 1 (rule TNEW). This process (of setting the ICAs to $\top$ and then 1) is repeated if these allocations are executed during every iteration of j until j terminates. However, if one of the allocation sites (say o_1) is not executed in iteration $k+1$, its corresponding ICA $\tilde{\tau_1}.\tilde{\pi}.\tilde{i}$ will keep the value $\top$. Hence, at the end of iteration $k+1$, $\tilde{\tau_1}.\tilde{\pi}.\tilde{i} = \top$ and $\tilde{\tau_2}.\tilde{\pi}.\tilde{i} = 1$. Because the final solution Γ is a fixed point and $\top$ is greater than any other abstract value, $\top$ will be recorded in Γ for $\tilde{\tau_1}.\tilde{\pi}.\tilde{i}$ even though o_1 may allocate instances again later in the loop.

Note that $\tilde{\tau_1}.\tilde{\pi}.\tilde{i} = \tilde{\tau_2}.\tilde{\pi}.\tilde{i} = 1$ does not necessarily indicate that $\tilde{\tau_1}.o$ and $\tilde{\tau_2}.o$ are executed in *every iteration* of loop j. Their ICAs are 1 as long as their instances cannot escape from the iteration where they are created to the next iteration of the loop. This feature allows the analysis to report potentially-hoistable data structures even though their construction code (i.e., stores that connect objects in them) is guarded by conditionals.

Similarly, given a dependence effect $\tilde{\tau_1}.f \preceq \tilde{\tau_2}.g$, if $\tilde{\tau_1}.\tilde{\pi}.\tilde{i} = \tilde{\tau_2}.\tilde{\pi}.\tilde{i} = 1$ for a loop j, we can safely conclude that this whole dependence (i.e., computation) chain from $\tilde{\tau_1}.f$ to $\tilde{\tau_2}.g$ occurs in the same iteration of j, because there are only stack variables between the two end heap locations of the chain.

3.2 Hoistable Logical Data Structures

In this subsection, we introduce the notion of hoistable logical data structures based on the points-to and dependence effect abstractions computed by the type and effect system. As discussed earlier, here we address the question "what data is hoistable in the best scenario"—that is, to find hoistable logical data structures that meet the two criteria discussed in Section 2. Whether and how they can actually be hoisted will be decided by the user upon inspection. This subsection presents mathematical properties of hoistable data structures.

Definition 1. (Constrained closures of $\tilde{H}$ and $\tilde{\Omega}$) *Constrained closures of $\tilde{H}$ and $\tilde{\Omega}$ are represented by relations $\trianglerighteq^*_{j,\tilde{i}_1,\tilde{i}_2}$ and $\preceq^*_{j,\tilde{i}_1,\tilde{i}_2}$, where parameters j, $\tilde{i}_1$, and $\tilde{i}_2$ denote a loop label, a lower bound, and an upper bound of ICAs, used to compute transitive relationships. We define order $\leq$ on the ICA domain $\tilde{i}$ as $0 \leq 1 \leq \top$.*

(1) Closure $\trianglerighteq^*_{j,\tilde{i}_1,\tilde{i}_2}$ (on $\tilde{H}$) selects a set of data structures (whose nodes are types), in which each edge has the form $o_1^{\tilde{\pi}_1} \trianglerighteq o_2^{\tilde{\pi}_2} \in \tilde{H}$, s.t. $\tilde{\pi}_1.l = \tilde{\pi}_2.l = j$, $\tilde{i}_1 \leq \tilde{\pi}_1.\tilde{i} \leq \tilde{i}_2$, $\tilde{i}_1 \leq \tilde{\pi}_2.\tilde{i} \leq \tilde{i}_2$.

(2) Similarly, closure $\preceq^*_{j,\tilde{i}_1,\tilde{i}_2}$ (on $\tilde{\Omega}$) selects a set of computation chains, in which each edge has the form $o_1^{\tilde{\pi}_1} \preceq o_2^{\tilde{\pi}_2} \in \tilde{\Omega}$, s.t. $\tilde{\pi}_1.l = \tilde{\pi}_2.l = j$, $\tilde{i}_1 \leq \tilde{\pi}_1.\tilde{i} \leq \tilde{i}_2$, $\tilde{i}_1 \leq \tilde{\pi}_2.\tilde{i} \leq \tilde{i}_2$.

Note that constraint $\tilde{i}_1 \leq \tilde{\pi}.\tilde{i} \leq \tilde{i}_2$ is used to compute these closures: $\tilde{\tau}_1 \trianglerighteq^f \tilde{\tau}_2$ (or $\tilde{\tau}_1.f \preceq \tilde{\tau}_2.g$) is added into the closure $\trianglerighteq^*_{j,\tilde{i}_1,\tilde{i}_2}$ (or $\preceq^*_{j,\tilde{i}_1,\tilde{i}_2}$) only when the ICAs $\tilde{\tau}_1.\tilde{\pi}.\tilde{i}$ and $\tilde{\tau}_2.\tilde{\pi}.\tilde{i}$ are "between" the specified parameters $\tilde{i}_1$ and $\tilde{i}_2$. For example, the general closures $\trianglerighteq^*$ and $\preceq^*$ are special cases of their constrained closures when $\tilde{i}_1 = 0$, $\tilde{i}_2 = \top$, and j is an arbitrary loop label. It is also easy to see that $\trianglerighteq^*_{j,0,0}$ selects data structures whose objects are all created outside loops. Similarly, a data structure selected by $\trianglerighteq^*_{j,0,1}$ is such that its objects can be created both inside and outside loop j, and the set of inside objects in any run-time instance of the data structure are always allocated in the same iteration. Using constrained closures, we give the following definitions.

Definition 2. (Disjoint Data Structure *(DDS)) For an allocation site p located in loop j, a data structure (denoted as δ_p^j) rooted at p with respect to loop j is a graph whose edge set is a subset of $\tilde{H}$. Its run-time instances are guaranteed to be disjoint if, for any edge $\tilde{\tau}_1 \trianglerighteq^f \tilde{\tau}_2$ of the data structure, there exists a type $\tilde{\tau}$ for p, s.t.*

$$\tilde{\tau}.o = p \;\wedge\; \tilde{\tau}.\tilde{\pi}.\tilde{i} = 1 \;\wedge\; \tilde{\tau} \trianglerighteq^*_{j,1,1} \tilde{\tau}_1 \;\wedge\; \tilde{\tau} \trianglerighteq^*_{j,1,1} \tilde{\tau}_2$$

A DDS contains objects that are reachable from root p and that are created in the loop. Each run-time instance of a DDS is guaranteed *not* to contain any object instance created (1) outside the loop and (2) inside the loop but in different iterations. This is achieved by using constraint $(j, 1, 1)$ for the closure computation.

Lemma 2. (Disjointness of DDS instances) *Given two run-time instances of a DDS δ_p^l created by two iterations of a loop, no run-time object exists in both instances.*

Proof sketch. The lemma can be proved by contradiction. Suppose there is a run-time object that exists in both instances of the data structure. At the point it is added into the second data structure instance (created by a later iteration), the abstract loop iteration count for j contained in its type must be $\top$, which is recorded in the abstract points-to effect that represents this addition. This contradicts the fact that δ_p^l is constructed using closure $\trianglerighteq^*_{j,1,1}$, which limits the abstract iteration count for each type to be 1. $\square$

Definition 3. *(*Iteration-Dependent Field*) A field of the form $\tilde{\tau}.f$ is loop-iteration-dependent (LID) with respect to loop j if*

$$(a)\ \exists \tilde{\tau}'.g : \tilde{\tau}.f \preceq^*_{j,0,\top} \tilde{\tau}'.g \ \wedge\ (\tilde{\tau}' = \top \ \vee\ \tilde{\tau}'.\tilde{\pi}.\tilde{i} = \top)$$
$$\vee\ (b)\ \exists \tilde{\tau}'.g : \tilde{\tau}'.\tilde{\pi}.\tilde{i} = 0 \ \wedge\ \tilde{\tau}.f \preceq^*_{j,0,1} \tilde{\tau}'.g \ \wedge\ \tilde{\tau}'.g \preceq^*_{j,0,1} \tilde{\tau}'.g$$

Determining whether the value of an object field depends on a specific loop iteration requires to inspect abstract dependence effects. As discussed in (condition 2 of) Section 2, a field can be iteration-dependent if (1) it depends on a value read from a field of an unknown object or an object created in an unknown (different) iteration, or (2) it depends on a field of an object created outside the loop (e.g., $\tilde{\tau'}.g$), and this field is involved in a *dependence cycle* (i.e., it can transitively depend on itself).

Lemma 3. *(*Loop-Invariant Data Structure*) A data structure is loop-invariant if for each type τ in the data structure, $\forall \tilde{\tau}.f \in \textsc{Dom}(\tilde{\Omega})$: $\tilde{\tau}.f$ is not an LID field.*

Proof sketch. Let us negate the two conditions in Definition 3, that is, a loop-invariant field $o.f$ can depend only on (1) fields of objects guaranteed to be created in the same iteration with o, or (2) fields of objects created outside the loop and not involved in dependence cycles. For (1), the proof can be done by induction on the chain of dependence edges leading to $o.f$. In the base case, fields without any incoming dependence edge must be assigned newly created objects or `null`, and thus must be loop-invariant. For the inductive step, consider the n-th edge along the chain. If the source field of the edge is loop-invariant, the target field of the edge must also be loop-invariant.

For (2), let us first consider a simplified situation where there is only one outside object field $p.q$ (i.e., p is created outside the loop) involved in the dependence chain. Here are two subcases. First, field $o.f$ (o is an object created inside the loop) depends on $p.q$ and $p.q$ is never written in the loop. It is straightforward to see that $p.q$ does not carry any iteration-specific values and thus $o.f$ is loop-independent.

Second, suppose field $p.q$ is written in iteration i with a value v produced in iteration i'. Here i must equal i', because otherwise $o.f$ could depend on a value computed in a different iteration, which contradicts the statement in (1). Since value v cannot depend on $p.q$ (otherwise $p.q$ would depend on itself), it must come only from objects freshly created in this iteration. Based on the proof of case (1), we know that v must be loop-invariant. If $p.q$ is read later in iteration $k > i$ to produce another value v', v' must also be the same across iterations because $p.q$ is invariant. This reasoning can be easily generalized to handle the more complex situation where multiple outside object fields exist in the dependence chain. □

Definition 4. *(*Hoistable Logical Data Structure *(HDS)) If a data structure δ^j_p is a hoistable logical data structure if it is (1) disjoint and (2) loop-invariant.*

A HDS can exhibit exactly the same behavior at run time when it is located inside the loop and outside the loop, under the assumption that the code statements that access this data structure can be safely hoisted. In fact, instead of reporting only completely-hoistable data structures, our analysis identifies, for each logical data structure, its hoistable part (that is both disjoint and loop-invariant). The analysis eventually ranks all loop data structures based on their hoistability measurements, in order to quantify the likelihood of successful manual hoisting.

3.3 Analysis Algorithm

This subsection briefly discusses our analysis algorithm, which, for the first time, uses a context-free-language (CFL)-reachability formulation to compute the context-sensitive dependence and ICA information. The CFL-reachability formulation enables a demand-driven analysis that can work on each individual object in the loop and explore the points-to and the dependence relationships only for the fields of this object without performing a whole-program analysis. The analysis has three logical components. The first component is a data structure analysis. In order to discover the data structure rooted at an object, this analysis employs a variation of the CFL-reachability formulation of points-to analysis [5], which models both context sensitivity via method entries and exits and heap accesses via object fields read and writes. For each loop in an actual Java program, data structure root objects are first located. To find such root objects, we first consider objects that are created directly in the loop body. Objects that are created in a method (e.g., used as a factory) invoked by a call site in the loop and that can be returned to the loop-containing method are also considered.

Next, for each root object o, our analysis identifies the set of reachable objects and their points-to relationships. In particular, the analysis looks for chains of stores of the form $a_0.f_0 = new\ X; a_1.f_1 = b_0; a_2.f_2 = b_1; \ldots; a_n.f_n = b_{n-1}; b_n = o$, such that (1) the two variables in each pair (a_i, b_i) for $0 \leq i \leq n$ are aliases and (2) the CFL path for this chain contains balanced method entries and exits. If such a chain can be found, the object represented by $new\ X$ is in the data structure rooted at o, because it could potentially be reached from o at runtime through a sequence of field dereferences $f_n.f_{n-1} \ldots f_1.f_0$. Using this formulation, our hoisting analysis can be performed on demand: it can work directly on each loop object, and performs only the work necessary to detect its data structure and to check its hoistability.

The second component of the analysis is a form of data flow analysis that analyzes each loop to perform type inference. An abstract heap (points-to and data dependence) effect is actually the join of data flow facts on all valid paths from the loop entry to the assignment that connects the two entities in the effect. Aliasing relationships are determined by querying the CFL-reachability-based points-to analysis. The third part is a form of data dependence analysis that detects loop-invariant object fields. This analysis traverses backward the def-use chains from each store that writes to a field of an object in a loop data structure, and checks whether the two conditions in Definition 3 hold for the field. A key challenge in computing precise data dependences lies in the handling of data flow via heap locations. Our analysis initially works on top of a context-insensitive points-to analysis: for each load $a = b.f$, we find the set of all assignments $c.f = d$ such that c and b can alias context-insensitively. We next perform refinement on this candidate set using the CFL-reachability formulation of pointer-aliasing to find whether b and c can indeed alias, and if they can, the calling context for $c.f = d$ under which the value flow occurs. This calling context is used to guide the future graph traversal to follow the appropriate entry/exit edges.

4 Computing Hoistability Measurements

In practice, we have observed that only a small number of data structures and statements in a large program can meet both criteria described in Section 3. This is primarily due to their complex usage and the conservative nature of any static analysis algorithm, which must model this complex usage soundly. While fully-automated transformations are desirable and sometimes possible, the usefulness of the static analyses can be increased even further by generalizing them to provide valuable support for programmer-driven manual code transformations.

Previous studies such as [6,7] have demonstrated that, in many cases, manual tuning with developers' insight can be much more effective than fully-automated compiler optimizations. For instance, a programmer may quickly identify that it is problematic to create a 100-field data structure in a loop with only 1 field iteration-dependent, while the sound transformation would give up and terminate silently. To enlist human effort, we must present to them highly-relevant information that can quickly direct them to a problematic area. In this section, we present two metrics that we use to measure *hoistability* of data structures. These measurements are computed based on the two orthogonal relationships (i.e., points-to and dependence) that are described earlier in the paper.

Dependence-Based Hoistability. (DH) The first metric we consider measures the amount of data in a data structure that is constant during the execution of a loop (i.e., the part that complies with rule 2 in Section 2). This dependence hoistability metric is simply defined as an exponential function $DH = s^{n/s}$, which considers two factors: the total number of fields s in a data structure and the number of its loop-invariant (i.e., non-LID, discussed in Def. 3) fields n. The larger s is, the more performance improvement could potentially be achieved by hoisting it. The larger n/s is, the easier it is for a programmer to hoist this data structure. If s is 1, the data structure contains a single field. Even though this field is invariant (i.e., n/s is 1), hoisting it may not have a large impact on performance. If n/s is a small number (i.e., most of its fields are not invariant), the result of $s^{n/s}$ can be very small (i.e., close to 1) regardless of how large s is, which also indicates it is not worth spending time as the data structure may be too difficult to hoist. In addition, we choose an exponential function instead of a polynomial function as the metric because the exponential function "penalizes" cases where n is small, while a polynomial function would be "fair" for all cases of n. For example, if n=s/2 (half the fields are invariant), our exponential function will give the square root of s, while a polynomial function may give a much larger number.

Structure-Based Hoistability. (SH) Similarly to the first metric, the second metric considers, for each data structure, how many objects in it are guaranteed to be allocated in the same iteration (i.e., the part that complies with condition 1 in Section 2). This structural hoistability metric is defined as $SH = t^{m/t}$, where t is the total number of objects in the data structure and m is the number of such objects whose ICA is 1. The value of t is computed by counting the number of objects that are context-sensitively reachable from the root object of a data structure. It is clear to see that SH considers both the size of the data structure and the size of its disjoint part. Note that when m/t is 1, this data structure is a DDS (as discussed in Def. 2), as it is guaranteed to have disjoint instances in all loop iterations.

During our studies, we found that DH is much more useful than SH in distinguishing data structures that are easy to hoist manually from those that are not. First of all, s is a more accurate measurement of the size of a data structure than t, as the data structure can still be large if it contains fewer objects but each object has more fields. Second, we found that for a large number of data structures in our benchmarks, their m/t is 1, which means they are all DDS. It would be quite labor-intensive to inspect all of them and check if they are hoistable. To help the programmer quickly identify truly optimizable data structures, we focus on DDS (whose m/t is 1) and compute dependence-based hoistability (DH) only for these data structures. Finally, only DDS are ranked (based on their DH measurements) and presented to the user for inspection.

Incorporating Dynamic Information. For performance tuning, dynamic (frequency) information is necessary to help programmers focus on "hot" entities (e.g., calling contexts, data structures, etc.). For example, it could be more beneficial to hoist a small, but frequently-allocated data structure than a big, occasionally-occurring data structure. Frequency information can be easily incorporated into the two hoistability metrics. For example, for DH, its definition can be simply extended to $DDH = (f * s)^{n/s}$, where f represents the allocation frequency of the root object of the data structure.

While these metrics are simple, we show that they are effective in locating data structures that are mostly hoistable and easy to optimize. In this work, we focus on demonstrating that the static analyses are useful—even with these simple metrics, the reports can quickly guide us to data structures that are truly optimizable.

5 Evaluation

We implemented the analysis on the Soot analysis framework [8] and evaluated it on the 19 Java programs shown in Table 1. All experiments were conducted on a quad-core machine with an Intel Xeon X3363 2.83GHZ processor, running Linux 2.6.18. The setup for static analysis (similarly to [9]) used the library classes from Sun JDK 1.5.0_06 and 4GB of max heap space. Programs in the table were from the `SPECjvm98`, `Ashes`, and `DaCapo` (its 2006 release and a pre-release) benchmark suites. We did not choose the recent release of DaCapo, because it contains applications making heavy use of class-loading/reflection, which can prevent any static analysis from producing precise information. For `SPECjvm98`, we included only large programs that have loop objects.

5.1 Static Analysis and Hoisting

Table 1 reports statistics of the benchmarks, the analysis, and the dependence-based hoistability measurements. For a GUI application `muffin`, we could not find an appropriate test case to perform profiling, and thus, "-" is used to fill out columns that report dynamic-information-based measurements. We could not perform profiling for `eclipse` either, as different plugins use their own class loaders, making it difficult for them to access our profiling library without modifying their customized class loaders. The cost of the analysis is generally proportional to the number of loop objects processed because of its demand-driven nature. The analysis running time can also be influenced by the size of the code base, as the analysis is context-sensitive and the

Table 1. Shown in the first seven columns of the table are the general statistics of the benchmarks and the analysis: the benchmark names, the numbers of methods (in thousands) in Soot's Spark context-insensitive call graph (*M*), the numbers of loops inspected (*Loops*), the numbers of loop objects considered (*Obj*), the running times of the analysis (*Time*), the total numbers of disjoint data structures (*DDS*), and the total numbers of hoistable logical data structures (*HDS*). Columns *SF* and *SIF* in section DH (i.e., dependence-based hoistability) show the total numbers of fields (*SF*) and the numbers of loop-invariant fields (*SIF*), averaged among the top 10 DDS that we chose to inspect. These data structures are ranked based on the dependence-based hoistability measurement (DH). Columns *DF* and *DIF* report the same measurements as *SF* and *SIF*, except that the inspected data structures are ranked using DDH that incorporates dynamic frequency information.

Benchmark	(a) Analysis statistics						(b) DH			
	#M(K)	*#Loops*	*#Obj*	*Time (s)*	*#DDS*	*#HDS*	*#SF*	*#SIF*	*#DF*	*#DIF*
jack	12.5	88	13	1224	5	3	797	62	797	62
javac	13.4	270	89	1745	33	8	45	31	42	28
soot-c	10.4	475	17	3043	7	3	56	36	56	36
sablecc-j	21.4	202	228	7910	82	53	429	194	221	61
jess	12.8	119	32	304	7	1	1135	51	1135	51
muffin	21.4	318	96	10766	47	8	1503	198	-	-
jflex	20.2	209	17	2325	9	0	55	17	55	17
jlex	8.2	108	9	5549	4	0	36	6	36	6
java-cup	8.4	99	19	474	4	0	107	57	107	57
antlr	12.9	154	3	77	2	1	3	0	3	0
bloat	10.8	562	141	3476	36	10	1536	136	674	46
chart	17.4	482	102	12746	6	0	84	19	84	19
xalan	12.8	17	8	63	6	0	78	24	78	24
hsqldb	12.5	33	10	178	5	0	75	19	75	19
luindex	10.7	14	5	163	5	0	65	15	65	15
ps	13.5	117	21	1784	21	11	36	20	34	20
pmd	15.3	594	30	168	15	2	127	68	127	68
jython	27.5	614	48	423	24	3	77	25	190	26
eclipse	41.0	3322	93	21557	80	52	1182	180	-	-

number of contexts often grows significantly when the size of the program increases. It is clear that the analysis can scale to large applications, including the `eclipse` framework, which has millions lines of code in its implementation.

Across all applications we observe large numbers of disjoint data structures (DDS) and hoistable logical data structures (HDS). This is a strong indication of the existence of optimization opportunities that can be exploited by human experts, which motivates our proposal of computing hoistability measurements to help manual tuning. To compare the effectiveness of *DH* (i.e., dependence-based hoistability measurement proposed in Section 4) and *DDH* (i.e., the profile-based version of it) in finding optimization opportunities, we inspected the top 10 data structures (or the total number of data structures if it is smaller than 10) that appear in both reports. The total numbers of fields / the numbers of loop-invariant fields for these inspected data structures are shown in columns *SF/SIF* and *DF/DIF*, for these two kinds of reports. Profiling is implemented by Soot-based bytecode instrumentation that records the execution frequency for each loop. The goal of this comparison is to understand how much impact the dynamic information can have on the interpretation of reports. Specifically, can *DDH* (i.e., the incorporation of the run-time frequency f) lower the ranks of data structures that are highly-likely to be optimized (i.e., have larger n/s but smaller f)?

The ratio between *SIF* (or *DIF*) and *SF* (or *DF*) indicates, to a large degree, the difficulty of hoisting data structures manually by inspecting the analysis reports. The larger it is, the easier it may be for a performance expert to modify the data models to hoist them. It is clear that the ratios of *DIF*/*DF* are generally close to those of *SIF*/*SF*. In many cases, the former are even greater than the latter (e.g., `bloat`). This observation indicates that *DDH* can expose not only hot spots (i.e., frequently-allocated objects), but also *optimizable* data structures.

5.2 Case Studies

We have carefully inspected the generated analysis reports (with dynamic information incorporated) for these 19 benchmarks and found optimizable data structures in almost every one of them. This subsection presents five representative case studies, in which either large performance improvement was seen, or interesting bloat patterns were found. These applications are `ps`, `xalan`, `bloat`, `soot-c`, and `sablecc-j`, all with large code base and a great number of loop objects. The performance problems we show in this paper are new and have never been reported by any previous work. Performance improvements are measured on Sun Hotspot 64-bit Server VM build 1.6.0_11.

Through these studies, we found that the analysis is quite useful in helping programmers find mostly-loop-invariant data structures and the execution inefficiencies due to these data structures. It took us about three days to find and fix problems in these five applications, among which we had studied only `bloat` before. Note that most of this time was spent on developing fixes rather than finding data structures that can be easily hoisted: for each benchmark, we looked at only the top 10 data structures in the reports (due to the limited time we had), and found that most of them were indeed hoistable. Even larger optimization opportunities could have been possible if we had inspected more warnings generated by the tool.

It is important to note that it would not be possible to find such problems by using any existing profiling tool: to detect loop-invariant data structures, a purely dynamic analysis has to perform whole-program *value profiling*, a task that is prohibitively expensive for large-scale, long-running Java programs. This is the reason why we have not compared our results with dynamic analysis reports.

ps. `ps` is a postscript interpreter. The top data structure in the list ranked by *DDH* is rooted at a `NameObject` object created in a loop in method `execute` of class `makeDictOperatorDB`. The loop is used to traverse a stack: for each stack element (i.e., a `NameObject` object containing a key and a value), the goal is to remove the '/' character in the key of the element. The way the loop is implemented is that it creates another (backup) stack, pops the original stack, creates a new `NameObject` object with most values copied directly from the original object, and pushes this new object onto the backup stack. Eventually all the new objects in the backup stack are pushed back onto the original stack. The creation of such `NameObject` objects directs us to think about this implementation, and especially about the way the stack is operated. In fact, this process can be done entirely *in place* so that these objects do not even need to be created. A further inspection of code found an even more interesting problem. The programmer seems to ignore the fact that class `Stack` is a subclass of `List` in JDK and uses `push` and `pop` to implement everything related to stack. For example, this same pop-push pattern is used even for element retrieval. For almost each occurrence of this problematic stack usage pattern, there is a corresponding (mostly loop-invariant) data structure in our report. We removed only two occurrences of such a pattern (in `makeDictOperatorDB.execute` and `DictStack.getValueOf`), and this resulted in a reduction of 82.1% in running time (from 28.3s to 5.3s) and 70.8% reduction in the total number of objects created (from 10170142 to 2969354).

xalan. `xalan` is an XLST processor for XML documents. The problem we found is in a test harness (`XalanHarness`) used by DaCapo to run the benchmark. This harness class creates multiple threads to transform input XML files. In method `run`, there is a `while(true)` loop that assigns jobs to different threads. Our report shows that a data structure rooted at a `Transformer` object created in the loop is a HDS (shown in Figure 1(b)). The same *Transformer* object is created every time the loop iterates, and then used by different threads for transforming the input files. It is highly unlikely to automatically hoist this data structure because this object is created by using a transformer factory object, which is obtained from a reflective call. After hoisting this allocation site, we observed a 10% reduction in running time and 1% reduction in the number of objects created. This problem has also been confirmed by the DaCapo maintainers [10] and will be handled in the next release of the DaCapo benchmark set.

bloat. `bloat` is a program analysis tool designed for Java bytecode-level optimizations. Many loop data structures reported by our analysis are objects of inner classes that are declared exactly at the point where their objects are needed. In `bloat`, most of these objects are created to implement visitor patterns. Hence, the objects are used only for method dispatch and do not contain any data related to the program context under which they are created. These objects commonly exist in loops, and in many cases we found them even located in loops with many layers of nesting. This problem exemplifies a typical object-oriented philosophy: the programmer should focus on patterns and abstractions when coding, and leave the mess to the run-time system. By simply hoisting the reported objects (and the declarations of their classes) out of the loops, we saw 11.1% running time reduction and 18.2% reduction in the number of created objects.

soot-c. `soot-c` is a part of the Soot analysis framework [8] benchmarked in the Ashes benchmarks [11]. One top data structure reported by our analysis is rooted at a `StmtValueBoxPair` object created in a loop (in a constructor of `jimple.SimpleLocalUse`) that builds def-use relationships as follows:

```
while(defIt.hasNext()){
    List useList = (List) stmtToUses.get(defIt.next());
    useList.add(new StmtValueBoxPair(s, useBox));
}
```

For each statement s that uses a variable, the program finds a set of statements that define the variable, creates a `StmtValueBoxPair` object, and adds it to the list. These `StmtValueBoxPair` objects, while containing the same values, are created for safety purposes: if one such pair is changed later, other pairs should not be affected. After inspecting the code, we found that the use list associated with each statement is never changed after the jimple statement chain is constructed for a program. Even if a client analysis could change it by inserting statements, Soot always creates a new object to represent this (newly-established) def-use relationship rather than change the original object. This problem shows a typical example of an over-protective implementation, where several different mechanisms are used simultaneously to enforce the same property while one (or a few) of them may be sufficient to do so. By sharing one `StmtValueBoxPair` object among multiple def statements, we achieved 2.5% running time reduction and 3.5% reduction in the number of created objects. In this example, we can see once again the advantage of tool-assisted manual tuning: this data structure can never be eligible for hoisting from the perspective of any fully-automated analysis. However, the human insight did make hoisting happen as it is unnecessary to have these instances simultaneously.

sablecc-j. `sablecc-j` is a version of the Sable Compiler Compiler that produces the sablecc files (parser, lexer, etc.) for a preliminary version of the jimple grammar. Similarly to the problems found for `bloat`, a large number of HDS reported are related to inner classes: two such classes are declared in `sablecc.GenParser` to perform depth-first traversal of syntax trees, and one such class is declared in `sablecc.DFA` to represent an interval in a char set. Creating multiple objects for each such class is completely unnecessary. Hoisting these class declarations and their objects resulted in 6.7% running time reduction and 2.5% reduction in the number of objects created.

Evaluation Summary. While all loop-invariant data should be hoisted out of loops, the tight data coupling in an object-oriented program makes it impossible for us to do so (either automatically or manually). To help programmers focus on data structures that are (1) easy to hoist and (2) worth hoisting, we propose to compute hoistability measurements. Through these case studies, we demonstrate that our measurements are effective in pinpointing such data structures. In fact, by inspecting reported data structures, we found many performance problems and achieved significant performance improvement. Some invariant data structures that we have managed to hoist are due to (deeper) design issues such as inefficient implementations of design patterns (e.g., visitors in `bloat`) or over-protective implementation strategies (e.g., `soot-c`). Our measurements were also helpful in revealing these issues by exposing their symptoms (i.e., mostly-invariant data structures).

6 Related Work

The related work can be broadly classified into three categories: loop optimizations, runtime bloat detection techniques, and related static analyses.

Loop Optimizations. In the literature on compiler optimization [2], loop optimizations are important techniques that, for example, improve locality and make effective use of parallel processing capabilities. There is a large body of work on making the execution of loops faster. This set of techniques includes, for example, loop interchange, loop splitting, loop unrolling, loop fusion, loop-invariant code motion, etc. In high-performance computing, loop optimizations play a key role in automated parallelization for exploiting the parallelism capabilities of the hardware. Broader overview and more detailed descriptions of these techniques is available from a number of sources (e.g., [12,13,14]).

Bloat Detection. Mitchell and Sevitsky [15] introduce a way to find data structures that consume excessive amounts of memory. Work by Dufour *et al.* [16] uses a blended escape analysis to characterize the excessive use of temporary objects, which can also be used to help diagnose performance problems. JOLT [17] is a tool that makes aggressive method inlining decisions based on the identification of regions where a large volumes of temporary objects are observed. The approach from [18] dynamically identifies inappropriately-used Java collections to detect bloat. Recent work proposes dynamic analyses [6,7] that detect memory bloat by profiling copy chains and finding low-utility data structures, and static analysis [9] that finds inefficiently-used data structures. A detailed overview of the causes of runtime bloat can be found in [19,20].

Different from these existing techniques, our work focuses on loop-invariant data structures, and aims to help programmers identify them using a series of sophisticated static analyses. As discussed earlier, it may not be feasible to find such optimizable data structures using a dynamic analysis, which requires to profile all values generated during an execution, a task prohibitively expensive for real-world programs. Bhattacharya *et al.* propose an escape-analysis-based static technique [21] that can find reusable collections in a loop. Our analysis is more powerful: we can find general data structures that have disjoint instances as well as the same shapes and data content.

Related Static Analyses. The loop iteration abstraction is first proposed in [3] for computing their conditional must-not-alias properties. Our abstraction extends this approach for the purpose of detecting hoistable data structures, by computing ICA-annotated points-to and dependence relationships. Object inlining [22,23] is a static technique that finds sets of objects that can be efficiently fused into larger objects, and fuses them. While both object inlining and our analysis aim to achieve better performance and need to find objects created in the same control flow region, our analysis targets a different class of performance problems. In addition, our analysis can assist a programmer to do manual tuning, a task that is difficult for object inlining to perform. Gheorghioiu *et al.* propose a static analysis [24] to identify *unitary* allocation sites whose instances are disjoint so that these instances can be preallocated and reused. While this is similar to the detection of disjoint data structures in our work, we can find more opportunities such as data structures with loop-invariant data content and shapes.

Work from [4] presents *recency abstraction*, a technique that distinguishes most-recently-allocated-object (MRAO) and non-MRAO for each allocation site in order to enable strong updates for a points-to analysis. While this is similar to our iteration abstraction that distinguishes objects created in the current iteration and previous iterations, our analysis uses such an abstraction for identifying loop-invariant data structures, instead of improving the precision of a points-to analysis.

There exists a body of reachability analyses that can discover shapes of data structures. Such algorithms range from flow-sensitive approximations of heap shape (e.g., [25,26,27]) to decision procedures (e.g., [28,29]). While our approach can be less precise than these algorithms (especially in handling recursive data structures), its precision may be sufficient to find hoistable data structures. In addition, our demand-driven analysis is more scalable and has been shown to run successfully on large-scale applications including `eclipse`.

Ownership types [30,31,32,33,34,35] provide a way of specifying object encapsulation and enabling local reasoning about program correctness in object-oriented programs. While ownership types may be sufficient to select loop data structures and check whether they are confined, these types cannot detect loop-invariant values, which are dependence-related properties.

7 Conclusions

This paper presents the first static technique that detects loop-invariant data structures. We focus on data models and look for logical data structures that can be hoisted. Instead of transforming the program and hoisting data structures automatically, we propose to measure the hoistability of a data structure: the dependence-based hoistability metric measures the amount of loop-invariant data in a data structure. We have implemented the analyses and presented an evaluation on a set of 19 Java benchmarks. Our experimental results demonstrate that the analysis can scale to large applications and the measurements can be useful in finding significant optimization opportunities.

Acknowledgements. We thank the anonymous reviewers for their valuable comments. This material is based upon work supported by the National Science Foundation under CAREER grant CCF-0546040, grant CCF-1017204, and by an IBM Software Quality Innovation Faculty Award. Guoqing Xu was supported in part by an IBM Ph.D. Fellowship Award.

References

1. Mitchell, N.: Personal communication (2009)
2. Aho, A., Lam, M., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (2006)
3. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: POPL, pp. 327–338 (2007)
4. Balakrishnan, G., Reps, T.: Recency-Abstraction for Heap-Allocated Storage. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 221–239. Springer, Heidelberg (2006)
5. Sridharan, M., Bodik, R.: Refinement-based context-sensitive points-to analysis for Java. In: PLDI, pp. 387–400 (2006)
6. Xu, G., Arnold, M., Mitchell, N., Rountev, A., Sevitsky, G.: Go with the flow: Profiling copies to find runtime bloat. In: PLDI, pp. 419–430 (2009)
7. Xu, G., Mitchell, N., Arnold, M., Rountev, A., Schonberg, E., Sevitsky, G.: Finding low-utility data structures. In: PLDI, pp. 174–186 (2010)

8. Vallée-Rai, R., Gagnon, E., Hendren, L., Lam, P., Pominville, P., Sundaresan, V.: Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In: Watt, D.A. (ed.) CC 2000. LNCS, vol. 1781, pp. 18–34. Springer, Heidelberg (2000)
9. Xu, G., Rountev, A.: Detecting inefficiently-used containers to avoid bloat. In: PLDI, pp. 160–173 (2010)
10. DaCapo bug repository: Bloat report, `http://sourceforge.net/tracker/?func=detail&aid=2975679&group_id=172498&atid=861957`
11. Ashes Suite Collection, `http://www.sable.mcgill.ca/software`
12. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler transformations for high-performance computing. ACM Computing Surveys 26(4), 345–420 (1994)
13. Wolfe, M.: High performance compilers for parallel computing. Addison-Wesley Publishing Companay (1996)
14. Allen, R., Kennedy, K.: Optimizing compilers for modern architectures: A dependence-based approach. Morgan Kaufmann Publishers Inc. (2001)
15. Mitchell, N., Sevitsky, G.: The causes of bloat, the limits of health. In: OOPSLA, pp. 245–260 (2007)
16. Dufour, B., Ryder, B.G., Sevitsky, G.: A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In: FSE, pp. 59–70 (2008)
17. Shankar, A., Arnold, M., Bodik, R.: JOLT: Lightweight dynamic analysis and removal of object churn. In: OOPSLA, pp. 127–142 (2008)
18. Shacham, O., Vechev, M., Yahav, E.: Chameleon: Adaptive selection of collections. In: PLDI, pp. 408–418 (2009)
19. Mitchell, N., Schonberg, E., Sevitsky, G.: Four trends leading to Java runtime bloat. IEEE Software 27(1), 56–63 (2010)
20. Xu, G., Mitchell, N., Arnold, M., Rountev, A., Sevitsky, G.: Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In: FoSER, pp. 421–426 (2010)
21. Bhattacharya, S., Nanda, M.G., Gopinath, K., Gupta, M.: Reuse, Recycle to De-bloat Software. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 408–432. Springer, Heidelberg (2011)
22. Dolby, J., Chien, A.: An automatic object inlining optimization and its evaluation. In: PLDI, pp. 345–357 (2000)
23. Lhoták, O., Hendren, L.: Run-time evaluation of opportunities for object inlining in Java. Concurr. Comput.: Pract. Exper. 17(5-6), 515–537 (2005)
24. Gheorghioiu, O., Salcianu, A., Rinard, M.: Interprocedural compatibility analysis for static object preallocation. In: POPL, pp. 273–284 (2003)
25. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. TOPLAS 24(3), 217–298 (1999)
26. Chang, B., Rival, X.: Relational inductive shape analysis. In: POPL, pp. 247–260 (2008)
27. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL, pp. 289–300 (2009)
28. Lev-Ami, T., Immerman, N., Reps, T., Sagiv, M., Srivastava, S., Yorsh, G.: Simulating Reachability Using First-Order Logic with Applications to Verification of Linked Data Structures. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 99–115. Springer, Heidelberg (2005)
29. McPeak, S., Necula, G.C.: Data Structure Specifications via Local Equality Axioms. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 476–490. Springer, Heidelberg (2005)
30. Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: OOPSLA, pp. 311–330 (2002)

31. Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating objects with confined types. TOPLAS 29(6), 32 (2007)
32. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: OOPSLA, pp. 292–310 (2002)
33. Boyapati, C., Liskov, B., Shrira, L.: Ownership types for object encapsulation. In: POPL, pp. 213–223 (2003)
34. Heine, D.L., Lam, M.S.: A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In: PLDI, pp. 168–181 (2003)
35. Banerjee, A., Naumann, D.A.: Ownership confinement ensures representation independence for object-oriented programs. J. ACM 52(6), 894–960 (2005)