Predicting Access to Persistent Objects through Static Code Analysis ⋆

Rizkallah Touma[1], Anna Queralt[1], Toni Cortes[1,2], and María S. Pérez[3]

[1] Barcelona Supercomputing Center (BSC)
[2] Universitat Politècnica de Catalunya (UPC)
[3] Ontology Engineering Group (OEG), Universidad Politécnica de Madrid
{rizk.touma, anna.queralt, toni.cortes}@bsc.es, mperez@fi.upm.es

Abstract. In this paper, we present a fully-automatic, high-accuracy approach to predict access to persistent objects through static code analysis of object-oriented applications. The most widely-used previous technique uses a simple heuristic to make the predictions while approaches that offer higher accuracy are based on monitoring application execution. These approaches add a non-negligible overhead to the application's execution time and/or consume a considerable amount of memory. By contrast, we demonstrate in our experimental study that our proposed approach offers better accuracy than the most common technique used to predict access to persistent objects, and makes the predictions farther in advance, without performing any analysis during application execution.

1 Introduction

Persistent Object Stores (POSs), such as object-oriented databases and object-relational mapping systems (e.g. Hibernate, DataNucleus, dataClay [11]), are storage systems that expose persistent data in the form of objects and relations between these objects. This structure is rich in semantics ideal for predicting access to persistent data [8] and has invited a significant amount of research due to the importance of these predictions in areas such as prefetching, cache replacement and dynamic data placement.

In this paper, we present a fully-automatic approach that predicts access to persistent objects through static code analysis of object-oriented applications. Our approach takes advantage of the symmetry between application objects and POS objects to perform the prediction process before the application is executed and does not cause any overhead. In our experimental study, we demonstrate the viability of the proposed approach by answering the following research questions:

- **RQ1:** What is the accuracy of the proposed approach?
- **RQ2:** How much in advance can the approach make the predictions?

We also compare our approach with the *Referenced-Objects Predictor* (see Section 2) and the experimental results show that our approach offers better accuracy in all of the studied benchmarks, with reductions in false positives of as much as 30% in some cases. Moreover, our approach predicts accesses farther in advance giving additional time for the predictions to be utilized.

2   Related Work

The simplest technique to predict access to persistent objects is the *Referenced-Objects Predictor* (ROP), which is based on the following heuristic: each time an object is accessed, all the objects referenced from it are likely to be accessed as well [8]. In spite of its simplicity, this predictor is widely used in commercial POSs because it does not involve a complex and costly prediction process.

More complex approaches have been based on analysis done during application execution using various techniques such as Markov-Chains [9], traversal profiling [5,6] and the Lempel-Ziv compression algorithm [2]. The approach presented in [4] introduces type-level prediction based on the argument that patterns do not necessarily exist between individual objects but rather between object types. Type-level access prediction can capture patterns even when different objects of the same type are accessed and does not store information for each individual object, which reduces the amount of used memory. In general, the main drawbacks of these approaches are the overhead they add to application execution time and the fact that they are based on a most-common case scenario which might lead to erroneous predictions in some cases.

The approach brought forward in this paper combines the idea of application type graphs, presented in [6], with type-level prediction. The work in [6] proceeds by creating an object graph and generating object-level access hints based on profiling done during application execution. On the other hand, our approach generates type-level access hints based on static code analysis, thus benefiting from the advantages of type-level prediction while avoiding the issues stemming from performing the process during application execution.

Previous approaches that use static analysis to predict access to persistent data have targeted specific types of data structures such as linked data structures [1, 3, 7], recursive data structures [10, 13] or matrices [12]. To the best of our knowledge, our work is the first that predicts access to persistent objects of any type prior to application execution.

3   Running Example

Figure 1 shows the partial implementation of a bank management system, all classes represent persistent types except the *BankManagement* class. The method *setAllTransCustomers()* updates the customers of all the transactions to a new customer restricting such updates to customers of the same company. In order

```
 1   public class BankManagement {          23        }
 2    private List<Transaction> trans;      24      return acc;
 3    private Customer manager;             25    }
 4                                          26  }
 5    public void setAllTransCustomers()    27
         {                                  28  public class Account {
 6      for (Transaction t : trans) {      29   private Customer cust;
 7        t.getAccount()                    30  }
           .setCustomer(manager);          31
 8      }                                   32   public void setCustomer(Customer
 9    }                                          newCust) {
10   }                                      33     if (cust.comp == newCust.comp) {
11                                          34       cust = newCust;
12   /* Persistent Classes */              35     }
13   public class Transaction {            36    }
14    private Account acc;                  37  }
15    private Employee emp;                 38
16    private TransactionType type;         39  public class Customer {
17                                          40   public Company comp;
18    public Account getAccount() {         41  }
19      if (type.typeID == 1) {            42
20        emp.doSomething();                43  public class Employee {
21      } else {                            44    public Department dept;
22        emp.dept.doSomethingElse();       45  }
```

Fig. 1. Example Object-Oriented Application Code

to do so, it retrieves and iterates through all the *Transaction* objects and then navigates to the referenced *Account* and *Customer* until reaching the *Company* of each transaction and compares it with the new customer's company.

For this example, ROP would predict that each time a *Transaction* object is accessed, the referenced *Transaction Type*, *Account* and *Employee* objects will be accessed as well. However, the method *setAllTransCustomers()* does not access the predicted *Transaction Type* and *Employee* objects but needs the *Customer* and *Company* objects which are not predicted.

On the other hand, using static code analysis we can see that when *setAllTransCustomers()* is executed it accesses: (1) the object *BankManagement. manager*, (2) all the *Transaction* objects, and (3) the *Account*, *Customer* and *Company* objects of each transaction by calling *getAccount()* and *setCustomer()*. We can also see that *getAccount()* might access the *Department* of the *Employee* of a *Transaction*, depending on which branch of the conditional statement starting on line 19 is executed. Using this information, we can automatically generate method-specific access hints that predict which objects are going to be accessed.

4 Proposed Approach

Assuming we have an object-oriented application that uses a POS, we define T as the set of types of the application and $PT \subseteq T$ as its subset of persistent types. Furthermore, $\forall t \in T$ we define (1) F_t: the set of persistent member fields of t such that $\forall f \in F_t : type(f) \in PT$, (2) M_t: the set of member methods of t.
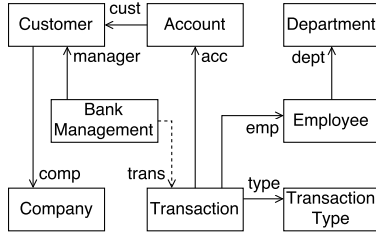
Fig. 2. Type graph G_T of the application from Fig. 1. Solid lines represent single associations and dashed lines represent collection associations.
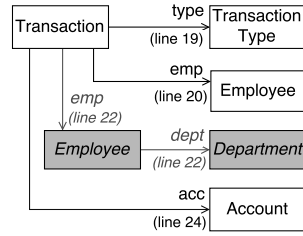
Fig. 3. Method type graph G_m of the method *getAccount()* from Fig. 1. Navigations highlighted in gray are branch-dependent.

4.1 Type Graphs

Application Type Graph The type graph of an application, as defined in [6], is a directed graph $G_T = (T, A)$ where:

- T is the set of types defined by the application.
- A is a function $T \times F \to PT \times \{single, collection\}$ representing a set of associations between types. Given types t and t' and field f, if $A(t, f) \to (t', c)$ then there is an association from t to t' represented by $f \in F_t$ where $type(f) = t'$ with cardinality c indicating whether the association is *single* or *collection*.

  Example. Figure 2 shows the type graph of the application from Fig. 1. Some of the associations of this type graph are: (1) A(Bank Management, trans) $\mapsto$ (Transaction, collection), (2) A(Transaction, acc) $\mapsto$ (Account, single).

Method Type Graph We construct the type graph G_m of a method $m \in M_t$ from the associations that are navigated by the method's instructions. A navigation of an association $t \to^f t'$ is triggered when an instruction accesses a field f in an object of type t (navigation source) to navigate to an object of type t' (navigation target) such that $A(t, f) \to (t', c)$. A navigation of a collection association has multiple target objects corresponding to the collection's elements.

  Example. Figure 3 shows G_m of method *getAccount()* from Fig. 1.

Augmented Method Type Graph We construct the augmented method type graph AG_m of a method $m \in M_t$ by adding association navigations that are caused by the invocation of another method $m' \in M_{t'}$ to G_m as follows:

- The type graph of the invoked method $G_{m'}$ is added to G_m through the navigation $t \to^f t'$ that caused the invocation of m'.
- The association navigations that are triggered by passing a persistent object as a parameter to m' are added directly to G_m.

  Example. Figure 4 shows the augmented method type graph AG_m of method *setAllTransCustomers()*. Note that the navigations $BankManagement \to^{manager} Customer \to^{comp} Company$ are triggered by passing the persistent object *Bank Management.manager* as a parameter to the method *setCustomer(newCust)*.
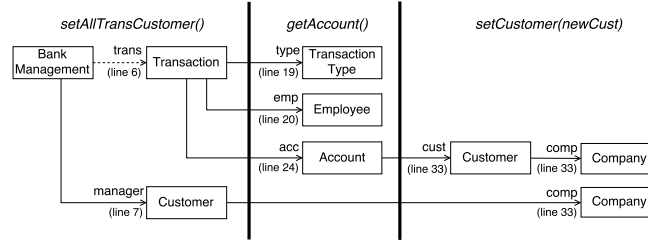
Fig. 4. Augmented method type graph AG_m of *setAllTransCustomers()* from Fig. 1.

4.2 Access Hints

We traverse a method's augmented graph and generate its set of access hints as:
$$AH_m = \left\{ ah \mid ah = f_1.f_2.\ldots.f_n \ \ where \ t_i \rightarrow^{f_i} t_{i+1} \in AG_M : 1 \le i < n \right\}$$
Each access hint $ah \in AH_m$ corresponds to a sequence of association navigations in AG_m and indicates that the navigations' target object(s) is/are accessed.

Example. The augmented method type graph AG_m of Fig. 4 results in the following set of access hints for method *setAllTransCustomers()* (hints starting with the collection *trans* predict that *all* its elements will be accessed):
$$AH_m = \{trans.type, \ trans.emp, \ trans.acc.cust.comp, \ manager.comp\}$$

4.3 Nondeterministic Application Behavior

Branch-Dependent Navigations They are navigations that might not be triggered depending on a method's branching behavior and which of its branches are executed. Including branch-dependent navigations in G_m might result in false positives if the branch from which the navigation is triggered is not executed while excluding them might result in a miss if the branch is indeed executed (both strategies are evaluated in Section 5). We divide them in two types:
 – Navigations *not* triggered inside all the branches of a conditional statement.
 – Navigations of collection associations *not* triggered in all the iterations of a loop statement due to branching instructions (*continue, break, return*).

Example. In Fig. 3, the navigations $Transaction \rightarrow^{emp} Employee \rightarrow^{dept} Department$, highlighted in gray, are branch dependent (they are only triggered in one of the conditional statement's branches) while the navigation $Transaction \rightarrow^{emp} Employee$ is *not* (it is triggered inside both branches).

Overridden Methods A method $m \in M_t$ might have overridden methods OM_m in the subtypes of its type ST_t. When an object is defined of type t but initialized to a subtype $t' \in ST_t$, the methods executed on the object are not known until runtime. Hence, using the access hints of m might lead to erroneous predictions. We propose to handle this case by adding one of the following sets of access hints to AH_m (both strategies are evaluated in Section 5):
 – $\bigcap_{m' \in OM_m} AH_{m'}$: *intersection* of access hints of overridden versions of m.
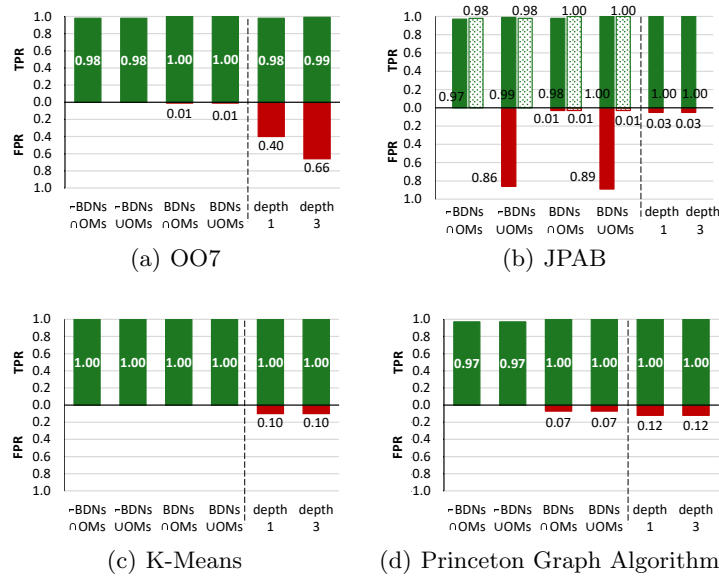 – $\bigcup_{m' \in OM_m} AH_{m'}$: *union* of access hints of overridden versions of m.

Fig. 5. True Positive Ratio (TPR) and False Positive Ratio (FPR) of our approach (left of the dashed line) compared with ROP (right of the dashed line). Columns represent:
- $\neg BDNs$ / $BDNs$: exclude / include branch-dependent navigations
- $\cup OMs$ / $\cap OMs$: intersection / union of overridden methods' access hints

5   Evaluation

We implemented a prototype of our approach in Java using **IBM Wala** and evaluated it on two benchmarks specifically designed for POSs and two benchmarks typically used for computation-intensive workloads:

- **OO7:** the *de facto* standard benchmark for POSs and OO databases.
- **JPAB:** measures the performance of ORMs compliant with Java Persistent API (JPA) using 4 types of workloads (persist, retrieve, query and update).
- **K-Means:** a clustering algorithm typically used as a big data benchmark.
- **Princeton Graph Algorithms (PGA):** a set of various graph algorithms with different types of graphs (undirected, directed, weighted).

We compared our approach with the *ROP* explained in Section 2 using the minimum possible depth of 1 as well as a depth of 3 to predict access to objects. In all the experiments, we used Hibernate 4.1.0 with PostgreSQL 9.3 as the persistent storage. In the following, we present our experimental results.

RQ1: What is the accuracy of the proposed approach?
We answered this question by testing the different strategies proposed in Section 4.3 to deal with branch-dependent navigations and overridden methods. Figure 5 shows the True Positive Ratio (correctly predicted objects / accessed objects) and False Positive Ratio (incorrectly predicted objects / total predicted objects)
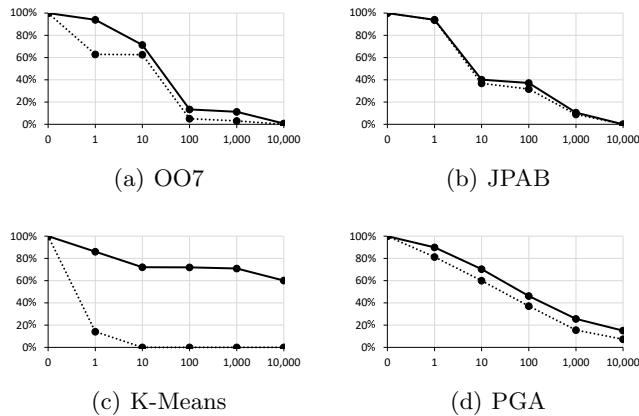
Fig. 6. The x-axis represents the number of persistent accesses between the prediction that a persistent object o will be accessed and the actual access to o. The y-axis represents the percentage of accesses that are predicted for each x-axis value. Solid lines represent our approach and dashed lines represent ROP.

of these strategies compared with *ROP*. Regardless of the used strategy, our approach results in fewer false positives in all of the studied benchmarks.

The only exception is taking the union of overridden methods' access hints with *JPAB*, represented by the solid-colored set of columns in Figure 5(b), which results in a sharp increase in false positives. This is due to the implementation of *JPAB* which includes five different tests each with its independent persistent classes, all of which are subclasses of a common abstract class. Hence, taking the union of overridden methods' access hints results in predicting access to many objects unrelated to the test being executed. We reran the analysis excluding the methods of the common abstract class and their overridden versions. The results are shown by the dotted set of columns in Figure 5(b) and indicate that excluding this case, the behavior of *JPAB* is similar to that of the other benchmarks.

Based on the results of this experiment, we recommend excluding branch-dependent navigations when memory resources are scarce, since this strategy does not result in any false positives. By contrast, branch-dependent navigations should be included when we are willing to sacrifice some memory, which will be occupied with false positives, in return of a higher true positive ratio. Finally, we recommend to always take the intersection of overridden methods' access hints in order to avoid problems with special cases similar to *JPAB*.

RQ2: How much in advance can the approach make the predictions?
We measured how much in advance our approach can make the predictions and compared it with ROP by calculating the number of persistent accesses between the time that an object o is predicted to be accessed and the actual access to o. For example, Figure 6 shows that with *OO7*, 95% of predictions made by our approach are done at least 1 persistent access in advance and 70% of predictions

at least 10 persistent accesses in advance. The results shown in Fig. 6 indicate that in the case of *JPAB*, the improvement we obtain over ROP is very small because the benchmark's data model does not allow for predictions to be made far in advance. However, with the other benchmarks, most significantly with *K-Means*, our approach is able to predict accesses much farther in advance.

6   Conclusions

In this paper, we presented a novel approach to automatically predict access to persistent objects through static code analysis of object-oriented applications. The approach performs the analysis before application execution and hence does not add any overhead. The experimental results show that our approach achieves better accuracy than the most common prediction technique, the Referenced-Objects Predictor. Moreover, the true advantage of our approach comes from the fact that it can predict access to persistent objects farther in advance which indicates that the predictions can be exploited to apply smarter prefetching, cache replacement policies and/or dynamic data placement mechanisms.

References

1. B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Proceedings of PACT 2001*, pages 280–291, 2001.
2. K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. *SIGMOD Rec.*, 22(2):257–266, 1993.
3. E. H. Gornish, E. D. Granston, and A. V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proceedings of ICS 1990*, pages 354–368. ACM, 1990.
4. W. Han, K. Whang, and Y. Moon. A formal framework for prefetching based on the type-level access pattern in object-relational DBMSs. *IEEE Trans. Knowledge Data Eng.*, 17(10):1436–1448, 2005.
5. Z. He and A. Marquez. Path and cache conscious prefetching (PCCP). *The VLDB journal*, 16(2):235–249, 2007.
6. A. Ibrahim and W. Cook. Automatic prefetching by traversal profiling in object persistence architectures. In *Proceedings of ECOOP 2006*, pages 50–73. 2006.
7. M. Karlsson, F. Dahlgren, and P. Stenström. A prefetching technique for irregular accesses to linked data structures. In *Proceedings of HPCA*, pages 206–217. 2000.
8. N. Knafla. A prefetching technique for object-oriented databases. In *Advances in Databases*, volume 1271, pages 154–168. Springer-Verlag, 1997.
9. N. Knafla. Analysing object relationships to predict page access for prefetching. In *Proceedings of POS-8 and PJW-3*, pages 160–170. Morgan Kaufmann, 1999.
10. C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of ASPLOS VII*, pages 222–233. ACM, 1996.
11. J. Mart, A. Queralt, D. Gasull, et al. Dataclay: A distributed data store for effective inter-player data sharing. *Journal of Systems and Software*, 131:129 – 145, 2017.
12. T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of ASPLOS V*, pages 62–73. ACM, 1992.
13. A. Stoutchinin, J. N. Amaral, G. R. Gao, et al. Speculative prefetching of induction pointers. In *Proceedings of CC 2001*, pages 289–303. Springer-Verlag, 2001.