

Interprocedural Query Extraction for Transparent Persistence

Ben Wiedermann, Ali Ibrahim, and William R. Cook

Department of Computer Sciences, The University of Texas at Austin

{ben,aibrahim,wcook}@cs.utexas.edu

Abstract

Transparent persistence promises to integrate programming languages and databases by allowing procedural programs to access persistent data with the same ease as non-persistent data. In this work we demonstrate the practical feasibility of a technique for extracting queries from object-oriented programs that use transparent persistence to access data. A program analysis derives query structure and conditions across methods that access persistent data. The system combines static analysis and runtime query composition to handle procedures that return persistent values. Our prototype Java compiler implements the analysis, and handles recursion and parameterized queries. We evaluate the effectiveness of query extraction on the OO7 and TORPEDO benchmarks. This work is focused on programs written in the current version of Java, without languages changes. However, the techniques developed here may also be of value in conjunction with object oriented languages extended with high-level query syntax.

1. Introduction

Integrating programming languages and databases is an important problem with significant practical and theoretical interest. Integration is difficult because procedural languages and database query languages are based on different semantic foundations and optimization strategies [22]. From a programming language viewpoint, databases manage *persistent* data, which has a lifetime longer than the execution of an individual program [36, 1, 27]. Ideally a unified programming model, *transparent persistence*, should be applicable to both persistent and non-persistent data.

One of the key integration issues is the treatment of queries. Queries are not fundamentally necessary, given an object-oriented view of persistent data in which a program can traverse from one object to another. But there are at least two advantages to queries: they can provide higher-level constructs for programmers to access data, and they enable specialized query optimizations typically found in databases.

In this work, we develop a technique for extracting queries from programs that use traversals to access persistent data. The goal is to support query optimization. We

also discuss how query extraction can be combined with approaches that use higher-level queries. We previously presented a sound query extraction technique [33], but this work was limited to a kernel language without procedures and did not target a practical database platform. In this paper we implement query extraction for Java and evaluate its effectiveness on two benchmarks. The contributions of this work are:

- Define an interprocedural static analysis to extract queries from Java programs using transparent persistence. The analysis handles virtual method calls by introducing additional queries where necessary and composing analysis results at runtime.
- The analysis results are converted to queries targeting the popular Hibernate persistence system [6].
- The analysis supports recursion by unfolding in stages of finite depth.
- We evaluate the system using the OO7 [5] and TORPEDO [23] benchmarks.

The current implementation demonstrates the feasibility of this approach, without making changes to the Java language. Some important features are left for future work. Currently only operations that read persistent data are supported, not updates to persistent values. Aggregation operations and sorting are also not considered. These features are easier to implement using high-level queries, as in Linq [4, 26], but would require changes to the Java language. It is important to stress that the techniques developed here can also be used in conjunction with queries, as illustrated in the following section. This work suggests that the best solution may be a combination of queries, to specify aggregation and sorting, and query extraction to specify prefetch and merge queries.

2. Problem

Transparent persistence can be added to most any language by extending the concepts of automatic memory management and garbage collection to the management of persistent data: by identifying a root object as *persistent*, any object or value reachable from the root is also persistent [2]. For example, the Java program in Figure 1 uses several procedures to operate on a collection of employee objects. The code

```

1 class Client { ...
2 void reportZip(DataAccess db, int zip) {
3     for (Employee e : db.getEmployees())
4         if (e.zip == zip)
5             printfOver(e, 65000);
6     }
7     void printfOver(Employee e, final int salaryLimit) {
8         if (e.salary > salaryLimit)
9             printEmployee(e);
10    }
11    void printEmployee(Employee e) {
12        print(e.name); print(": ");
13        print(e.manager.name);
14    }}
15 class DataAccess { ...
16     Collection <Employee> getEmployees() {
17         return root.getEmployees();
18     }}

```

Figure 1: Procedures and transparent persistence.

is typical of web-based applications in using a data access layer, represented by the `DataAccess` class, to load persistent data.

The `DataAccess` class has direct access to the `root` variable, which represents a persistent store of objects. The `reportZip` method calls the `getEmployees` method of the data access layer to load employees. It then iterates through the employees to find the employees in a given zip code; these employees are printed using the `printfOver` method. The `printfOver` method checks employee salaries before printing. Loading of the employee’s manager is *lazy*: each manager object is loaded when needed.

A key problem with this approach is that the entire database of employees must be loaded, even though only a few employees may be printed. The operation should use an index to find the desired employees, using standard database optimizations. But transparent persistence does not easily leverage the power of database query optimization. To solve this problem, many persistence models allow programmers to execute queries. For example, Figure 2 is a hand-optimized rewrite of Figure 1 that uses Hibernate, an object-relational mapping tool, and its query language HQL [17] to execute a query. The query returns only employees whose salary is over a salary parameter, and whose zip code is a given zip code. The prefetch clause `left join fetch e.manager` indicates that each employee’s manager should also be loaded. The `if` statements in Figure 1 are not needed in Figure 2 because the query’s where clause ensures the query only returns employees for which the tests are true.

Although the programs in Figure 1 and Figure 2 print the same results, they have different performance and software engineering characteristics. For large data sets, the Hibernate version will typically be orders of magnitude faster, be-

```

1 void reportZip(DataAccess db, int zip) {
2     Query q = db.createQuery( // create query
3         "from Employee e
4         left join fetch e.manager
5         where e.zip == :zip
6         and e.salary > :salaryLimit");
7     // set the parameters
8     q.setParameter("zip", zip, Hibernate.INTEGER);
9     q.setParameter("salaryLimit", 65000, Hibernate.INTEGER);
10    for (Employee e : q.list ()) // execute the query
11        printEmployee(e); // no test required
12 }

```

Figure 2: Query execution using Hibernate.

```

1 void reportZip(DataContext db, int zip) {
2     // preload specification
3     DataLoadOptions dlo = new DataLoadOptions();
4     dlo.LoadWith<Employee>(e => e.manager); //ERROR!
5     db.LoadOptions = dlo;
6     // query
7     int salaryLimit = 65000;
8     var employees = from e in db.Employee
9         where e.zip == zip && e.salary > salaryLimit
10        select e;
11    foreach (Employee e in employees)
12        printEmployee(e);
13 }

```

Figure 3: Query and load options in Linq.

cause it leverages the power of relational query optimization [7]. Despite its performance benefits, there are several well-known drawbacks to the Hibernate version: Query strings and parameters are not checked at compile time for syntax or type safety, and passing parameters is awkward.

These problems have been fixed by more recent query mechanisms, including Linq [4, 26] and Safe Query Objects [9]. Figure 3 gives one attempt to implement this program in C# with Linq. In this example, prefetch is specified by setting `LoadOptions` on the `DataContext` object that executes the query. The sample illustrates a `LoadWith<Employee>(f)` option, which specifies that the object `f(o)` should be loaded whenever an object `o` of a type `T` is loaded. Unfortunately, the code will generate a runtime error, because load options are not allowed to create cycles in the type graph; the example loads an employee (manager) with every employee.

Alternatively, Fig. 4 uses Linq to create an employee object that contains a manager record, where the manager is loaded via a join. In this case the select clause of the query calls an `Employee` constructor that takes two arguments, the name and the manager object.

A fundamental problem with queries is that the modularity of the original program in Figure 1 is compromised, because the query in the main function `reportZip` contains im-

```

1 void reportZip(DataContext db, int zip) {
2   int salaryLimit = 65000;
3   var employees = from e in db.Employee
4     where e.zip == zip && e.salary > salaryLimit
5     join m in db.Employee on e.managerID equals m.ID
6     select new Employee( e.name, m );
7   foreach (Employee e in employees)
8     printEmployee(e);
9 }

```

Figure 4: Creating results with Linq.

plementation details about the behavior of the printEmployee subroutine. The reportZip function would have to be rewritten if printEmployee were changed to also print the employee’s department:

```

void printEmployee(Employee e) {
  print(e.name); print(" ");
  print(e.department.name); print(", "); // Added
  print(e.manager.name);
}

```

The query also merges the conditions that were originally given separately in reportZip and printfOver. It may be possible to preserve the original modularity of the program by assembling the query from fragments. However, this effort would significantly complicate the design and introduce more potential for errors.

This paper presents *query extraction*, a technique that can be used to infer queries from procedural programs. The goal is to analyze the program in Figure 1 and derive the query that is used in Figure 2, while preserving the procedure calls and modularity of the original.

3. Intraprocedural Query Extraction

This section recasts our previous approach to intraprocedural query extraction [33] as an attribute grammar. The attribute grammar describes our Java compiler closely and generalizes easily to the interprocedural case.

The persistent data store is organized as a collection of linked records. The structure of the store can be defined by an Entity Relationship (ER) model [8] or UML class diagram [29]. A persistent record is a labeled product whose *fields* contain either basic values or references to other records. A reference/relationship can be either single-valued or multi-valued. Given an object, a *traversal* is a series of field accesses that loads one or more related objects. The special variable **root** represents the unique root of the database, from which all traversals start.

Any given operation in a program typically operates on a small *subset* of the complete database. A query can be viewed as a function that selects a subset of the database for use in a program. As long as the subset contains all the data actually needed by a particular operation, the subset can be used in place of the full database without affecting the

behavior of the program. Query extraction is *sound* if the queries derived from a program load all the data necessary to perform each operation. A query is *precise* if it loads no more data than is needed by the program.

In practice, query extraction does not need to be either sound or precise in order to be effective. This may seem disturbing, but unsoundness simply means that some objects are not loaded by the query and are instead loaded lazily as described in Section 2.

3.1 Paths as Abstract Values

The key idea behind query extraction is to perform a static analysis in which program values are replaced by *paths*. A path is an abstraction of a database value that represents the fields that were traversed to reach the value.

A path p is a three-tuple (\bar{f}, c, d) consisting of a sequence of field names \bar{f} that identify the steps taken to traverse a sequence of objects, a condition c under which the traversal is performed, and information d about the traversal’s data dependences. The condition and dependence information in a path may be omitted when they are not significant or may be easily determined from context.

Paths abstract over collections, so that all the objects in a collection have the same path. A special field name ι stands for an arbitrary element of a collection. For example, if **root** contains a collection employees, then the collection’s path is employees; the path to an arbitrary employee is employees. ι ; and the path to an employee’s salary is employees. ι .salary.

A path’s *condition* specifies the conditions under which the data described by the path are accessed. For example, the program in Figure 1 uses the expressions e.name and e.manager.name only under the enclosing **if** statement’s condition e.salary > salaryLimit. A query condition is a syntactic expression containing operators for numerical and logical operations over primitive values and paths. A condition in a program is a *query condition* if it can be included in a database query, as described in Section 3.2.

The *data dependence* in a path specifies whether the path’s values affect the data produced by the program, or if they affect only control flow. Data dependence determines whether a path forces loading of objects in a collection. For example, Figure 1 uses the zip code and salary fields only in conditional expressions. Even though these paths are used unconditionally, they do not force loading of the entire collection of employees because they are only used to determine control flow of the program.

3.2 Intraprocedural Path Analysis

The intraprocedural path analysis computes a *traversal summary*, or set of paths, for each statement and expression in a method. The analysis is specified as an attribute grammar [20] over Java abstract syntax. The synthesized attributes used in the analysis are defined in Figure 5. The

\cdot	Abstract Value: $AV(\cdot)$	Paths: $P(\cdot)$	Outgoing Store: $OS(\cdot)$	Incoming Store: IS
root	$(\{\epsilon\}, \mathbf{true}, \mathbf{false})$	$AV(\cdot)$	$IS(\cdot)$	
$e.f$	$AV(e).f$	$AV(\cdot)$	$IS(\cdot)$	
v	$IS(\cdot)[v]$	\emptyset	$IS(\cdot)$	
$e1 \text{ op } e2$	$AV(e1) \widehat{\text{op}} AV(e2)$	$TS(e1) \cup TS(e2)$	$IS(\cdot)$	
$v=e$	\perp	$TS(e)$	$[v \mapsto AV(v) \cup AV(e)]IS(\cdot)$	
if $e \text{ s1 else } s2$	\perp	$TS(e) \cup TS(s1) \cup TS(s2)$	$OS(s1) \cup OS(s2)$	
for $(v : e) s$	\perp	$TS(e) \cup AV(e).l \cup TS(s)$	$OS(s)/v$	$IS(s) \leftarrow [v \mapsto AV(e).l]IS(\cdot) \cup OS(s)$
<i>otherwise</i> $\llbracket e \rrbracket$	\perp	$TS(e)$	$IS(\cdot)$	

(a) Computing paths and values.

Query Condition: C	if $e \text{ s1 else } s2$	$C(s1) \leftarrow C(\cdot) \wedge AV(e)$ $C(s2) \leftarrow C(\cdot) \wedge \widehat{\text{not}}(AV(e))$ if e is a valid query condition
Data Dependence: D	<i>effective</i> $\llbracket e \rrbracket$	$D(e) \leftarrow \mathbf{true}$
Iterator Context: IT	for $(v : e) s$	$IT(s) \leftarrow IT(\cdot) + AV(e).l$ if $AV(e).l$ extends $IT(\cdot)$

(b) Auxiliary Definitions

\cdot	Traversal Summary: $TS(\cdot)$
v	$\begin{cases} (AV(\cdot), C(\cdot), \mathbf{true}) & \text{if } D(\cdot) \\ (P(\cdot), C(\cdot), \mathbf{false}) & \text{otherwise} \end{cases}$
$v=e$	$TS(e) \cup \text{iter}(\cdot)$
\cdot	$(P(\cdot), C(\cdot), D(\cdot)) \cup \begin{cases} \text{iter}(\cdot) & \text{if } D(\cdot) \\ \emptyset & \text{otherwise} \end{cases}$
where $\text{iter}(\cdot) = (\{\text{last}(IT(\cdot))\}, C(\cdot), \mathbf{true})$	

(c) Computing traversal summaries.

Figure 5: Attribute grammar that computes traversals for a subset of Java syntax.

domains for the attributes are:

$$\begin{aligned}
AV &: p + (AV \widehat{\text{op}} AV) && \text{abstract values: paths + ops} \\
P, TS &: \{p\} && \text{sets of paths} \\
IS, OS &: v \mapsto AV && \text{input and output stores} \\
D &: \text{boolean} && \text{data dependence flag}
\end{aligned}$$

Figure 5a defines attribute values for each kind of Java expression, represented by \cdot . The abstract value of the special variable **root** is a path value with no field traversals and the default condition and dependence: $(\epsilon, \mathbf{true}, \mathbf{false})$. A field traversal $e.f$ concatenates field names to previously computed paths, according to this definition:

$$AV(e).f = \{(\bar{f}.f, c, d) \mid (\bar{f}, c, d) \in AV(e)\}$$

A variable's abstract value is the value stored in attribute IS . If IS contains no binding for a variable, then the variable's abstract value is undefined (\perp). Binary operators are interpreted as abstract operators over abstract values; these are also used to represent query conditions.

The traversal path attribute P collects all the traversal summaries of an element's sub-expressions.

The *out-store* attribute $OS(\cdot)$ describes an element's affect on the *store*, or mapping from variables to the set of abstract values each variable may represent. The out store is typically a function of the inherited *in-store* attribute $IS(\cdot)$. The way in which stores flow between elements and their constituents is standard, although a few cases deserve mention.

Assignments affect the out-store attribute. The result maps the variable to the right-hand-side's abstract value. If the in-store already contains a value for a given variable, then the out-store contains the union of the old and new values.

A **for** statement creates a special path $AV(e).l$ that represents an arbitrary element of the collection value e . The attributes of the **for** statement's body are computed using a store that maps loop variable v to the special path. The body's in-store value depends on its out-store value, which forms a circular dependency. The final value for this attribute is the result of a fixed-point computation.

All other Java statements are given a default interpretation, by the *otherwise* case: the paths are unioned and the store is unchanged.

Query Conditions Figure 5b defines auxiliary attributes for *query conditions* C . The query condition attribute C collects conditions under which an expression or statement is executed. The attribute is inherited by all sub-expressions. A condition in an **if** statement can be a query condition if it satisfies three requirements: it contains only *portable operators*, it is *pointwise*, and it involves only *nested iterations*.

An operation is *portable* if it can be performed either in the database or in the program. For example, checking the existence of a file is not a portable operation. It is also essential that the operations have the same semantics in the database as in the program. This requires some translation, for example, to provide consistent handling of null values

in Java and SQL. Restricting query conditions to contain portable operations has little effect on programmers, because they would only expect portable operations to be included in a query.

A condition is *pointwise* if it can be evaluated on each item of a collection independently of all other items in the collection. The restrictions on query conditions involve checking for loop carried dependences, which are identified by a well-known static analysis. Programs very frequently contain loop carried dependencies, since they are created by any aggregation operation, including computing the sum or maximum of a collection. However, it is much less common that a variable involved in a loop carried dependence will be used in a filter condition. As an example, consider a program that prints only the values that form an increasing sequence from a collection:

```
int base = 0;
for (Data x in db.getItems()) {
    if (x.value >= base) {
        print(x.name);
        base = x.value;
    }
}
```

If two different collections are involved, then they must be *nested* so that the inner collection is a field of an item taken from the outer collection. Iteration of nested collections is very common: iterating over all purchase orders, then iterating over each item in the purchase order, is nested because the items are part of the purchase order. Non-nested iterations do sometimes arise; they correspond to ad-hoc joins that find correlations between collections that have no explicit relationship between them.

Data Dependence and Iterators Figure 5b also defines auxiliary attributes data dependence D and the iterator context IT.

The data dependence attribute D flags expressions whose execution affects the program output. It is true for any statement or expression that can affect state that is outside the method, including assignment to object fields, and arguments to library methods (like `print` methods). Rather than list all contexts that can affect the store, they are summarized as *effective*[*e*] contexts containing an expression *e*. Data dependence defaults to **false**.

The iterator context attribute IT maintains a list of inner-iteration variable paths that extend outer-iteration paths. This attribute helps determine whether a conditional expression satisfies the *nested* restriction for query conditions. IT also helps the analysis keep track of which collections should be marked data dependent.

Traversal Summaries Figure 5c defines the traversal summary attribute TS. It combines the path, condition, and data dependence attributes into a traversal summary. Given a set of paths *S* and a condition *c* and data dependence *d*, the notation (S, c, d) represents a new set of paths whose conditions

and data dependence are replaced:

$$(S, c, d) = \{(\bar{f}, c, d) \mid (\bar{f}, -, -) \in S\}$$

The traversal summary rules also implement two special situations. The first case, for variables, ensures that using an expression has the same effect as using a variable that has been assigned the value of the expression. The second two cases, for assignment and for any other expression, deal with an interaction between data dependence and iteration variables. The intuition behind these definitions is that an data dependent statement inside a loop causes the program to have a data dependence on the iteration variable of the loop. For example, if the loop includes a statement `x=x+1` then the program has a data dependence on the *existence* of the items in the collection (which satisfy the condition C). The formal definition adds a data-dependent traversal path for the current iterator context whenever the analysis encounters an assignment or a data-dependent expression.

4. Interprocedural Query Extraction

Interprocedural analysis allows query extraction to propagate query information across procedure boundaries. While the basic framework for our interprocedural analysis is standard, several aspects of the problems are unique—which call for specialized solutions.

Query information can flow in two directions, corresponding to procedure parameters and return values. For procedure parameters, the caller should prefetch the data needed by the procedure. For persistent return values, the procedure should prefetch the data needed to support the traversals in the caller from the procedure result. Thus the query information moves in the opposite direction from the values.

For persistent return values, the caller can send a query fragment as an extra argument to the procedure, indicating what traversals will be performed on the procedure return value. This mechanism works even for virtual function calls. The query fragments are combined at runtime, leading to a novel combination of static query extraction and dynamic query composition for procedure return values.

Transmitting information from procedures to call sites is more difficult, especially in object-oriented programs that make extensive use of virtual method calls. There are at least four possible approaches to this situation: *devirtualization*, *specialization*, *over-approximation*, and *query separation*.

Devirtualization, and closely related class hierarchy analysis, are techniques to identify a specific method that will be called at a given virtual method call site. It works by examining a complete program to discover whether there is only one implementation for a given method signature.

Polyvariant specialization is a technique for compiling a separate version of the caller for each method that can be called. This technique is used in JIT compilers, where

it is very effective. The main drawback is the potential for significant increase in code size.

Over-approximation could be used to compute a query that approximates the queries from all matching method bodies. This technique works best if the queries from different virtual methods are similar. If not, extra data could be loaded that is not needed.

Query separation is a simple approach: in cases where devirtualization fails, simply require the procedure to execute a new query to load its own data.

Our current implementation uses devirtualization and query separation to handle call sites with persistent parameters.

The first three techniques all rely on the closed-world assumption: the results are valid only if the dynamic class hierarchy does not differ from the static class hierarchy. Java programs can violate this assumption by dynamic class loading and runtime code generation. Techniques have been developed to update static analysis results when the set of classes changes dynamically [19]. We believe that these techniques could work with query extraction, but we have not yet tried to integrate them.

4.1 Interprocedural Paths

Whereas intraprocedural analysis considers only paths that traverse from **root**, interprocedural analysis must consider paths traversed from method parameters and from method return values.

We extend the definition of paths from Section 3.1 to include a path **root**. A *rooted path* is a four-tuple (r, \bar{f}, c, d) where the root r can be the **root**, a persistent method parameter, or a callsite return.

Interprocedural query extraction composes traversal summaries to create summaries of data traversals that cross method boundaries. The composition operation for paths is:

$$(r, f, c, d) \circ (r', f', c', d') = (r, f.f', c \wedge c', d \vee d')$$

Composition is lifted to operate on sets of paths by composing all combinations of paths from each set.

4.2 Interprocedural Path Analysis

Interprocedural query extraction extends the attribute grammar of the previous section to handle method declarations M , callsites $g_l(a_1, \dots, a_n)$ and returns **return** e . The analysis assigns a unique label l to each callsite of a method g .

A **return** statement's attributes are computed from the attributes of the returned expression. The abstract value of a method $AV(M)$ is the union of all the abstract values for the method's **return** statements:

$$\begin{aligned} TS(\mathbf{return} \ e) &= TS(e) \\ AV(\mathbf{return} \ e) &= AV(e) \\ AV(M) &= \bigcup_{s=\mathbf{return} \ e \in M} AV(s) \end{aligned}$$

```

1 public void salaryInfo () {
2   for (Department d : root.departments) {
3     printSalariesAbove (d,65000);
4     printSalariesBelow (d,30000);
5   }}
6 public void printSalariesAbove (Department d,
7   final double amount) {
8   for (Employee e : d.employees) {
9     if (e.salary > amount)
10      System.out.println (e.name);
11  }}
12 public void printSalariesBelow (Department d,
13   final double amount) {
14   for (Employee e : d.employees) {
15     if (e.salary < amount)
16      System.out.println (e.name);
17  }}

```

Figure 6: Method `salaryInfo` can pre-load data for `printSalariesAbove` and `printSalariesBelow`.

4.3 Method Calls

A callsite's traversal summary and abstract value depend on whether the analysis can predict the called method's data traversals. When a callsite passes a persistent value to a method, the called method's traversals indicate the shape of data that the method expects. The caller should execute a query to load the data required by the callee. For example, the program in Figure 6 requires only those employees who make less than \$30,000 or more than \$65,000. To load just these records efficiently, the analysis must synthesize the conditions from the two `print` methods.

The first case is a call site that cannot be devirtualized, denoted $\tilde{g}_l(a_1, \dots, a_n)$. In this case, the called method must execute its own query by using its traversal summary, taking the actual method arguments as roots. The call arguments are included in the traversal summary of the caller:

$$TS(\tilde{g}_l(a_1, \dots, a_n)) = \bigcup_{i=1}^n TS(a_i)$$

The abstract value of the call site is a new path using the label as a root:

$$AV(\tilde{g}_l(\dots)) = \{(l, \epsilon, c, d)\}$$

In a static, final, or devirtualized method call, the method implementation that will be invoked by the call is known statically. If the called method takes persistent parameters, the caller preloads the data needed to support the callee's traversals from those parameters. We define a method's traversal summary to be a map from persistent roots to paths. The set of paths a method M traverses from a given parameter P_i is:

$$\mathcal{P}_i^M = TS(M)[P_i]$$

```

1 public void employeeInfo() {
2   for (Department d : root.departments) {
3     for (Employee emp : d.employees) {
4       Employee e = getEmpToNotify(emp,65000);
5       System.out.println (e.department.name);
6     }}
7   public Employee getEmpToNotify(Employee e,
8                                 final double amount) {
9     if (e.salary > amount)
10      return e;
11    else
12      return e.manager;
13  }

```

Figure 7: Traversal passes through getEmpToNotify back to employeeInfo.

If a method returns a path that traverses from a given parameter P_i , then that path is denoted:

$$\mathcal{R}_i^M = AV(M)[P_i]$$

The abstract value domain is also extended by defining $AV(\cdot)[R]$ to be the empty set if it contains only abstract operations. A method's *pass-through* paths are those paths in its return value that are traversals from a parameter value:

$$\mathcal{T}^M = \bigcup_{i=1}^n \mathcal{R}_i^M$$

where n is the number of parameters for M . In Figure 7, method getEmpToNotify's return value is the result of a traversal from its first parameter e . Method employeeInfo, which calls getEmpToNotify traverses the return value's department field. The calling method detects this pass-through path and preloads the employee's department.

The argument summaries in a devirtualized callsite are composed with the traversal summary of the called method. If method f calls method g at devirtualized callsite l , the callsite's traversal summary consists of the traversals made by the arguments plus the traversals performed within g .

$$TS(g_l(a_1, \dots, a_n)) = \bigcup_{i=1}^n \left\{ \begin{array}{l} TS(a_i) \cup \\ (AV(a_i) \circ \mathcal{P}_i^g) \end{array} \right\}$$

If method g is recursive, then this analysis diverges. We discuss how to ensure termination in Section 4.5.

The abstract value of a devirtualized method is the composition of any argument paths with any corresponding paths in the called method's abstract value:

$$AV(g_l(a_1, \dots, a_n)) = \left(\bigcup_{i=1}^n (AV(a_i) \circ \mathcal{R}_i^g) \right) \cup (AV(g) - \mathcal{T}^g)$$

Method Return Values For method return values, query information flows from the caller to the callee. In using

```

1 public void hrDeptInfo() {
2   Department d = getHRDepartment();
3   for (Employee e : d.employees) {
4     System.out.println (e.manager.name);
5   }}
6   public Department getHRDepartment() {
7     for (Department d : root.departments) {
8       if (d.id == 1)
9         return d;
10    }}

```

Figure 8: Callee can pre-load caller's data.

a persistent value returned by a procedure, the caller may perform additional traversals to access related information. Ideally the callee would load subsequent data needed by the caller.

For example, method hrDeptInfo in Figure 8 calls getHRDepartment, which returns a Department. Method hrDeptInfo subsequently traverses this value's employees field and each subsequent employee's manager field. Ideally getHRDepartment should fetch this data for hrDeptInfo.

To support this behavior, every method that returns a persistent value is changed to take an additional argument S^f that contains the caller's traversals from the callee's return value. The caller also passes a flag devirtualized that indicates whether the callsite was devirtualized, in which case the callee need not load pass-through paths. The callee uses this information to generate a dynamic query TS' based on its own static traversal summary:

$$TS'(g) = TS(g) \cup \begin{cases} (AV(g) - \mathcal{T}^g) \circ S^f & \text{devirtualized} \\ AV(g) \circ S^f & \text{otherwise} \end{cases}$$

Recall that static query composition from caller to callee required the caller to bind values for any parameters that appeared in the callee's abstract value. Dynamic query composition from callee to caller requires the callee to bind values for any parameters that appear in the callee's return summary. The caller helps satisfy this requirement by providing the callee with the necessary values.

Conditions and Dependence Conditions present a problem for composing paths across procedure boundaries. If a callee's abstract value references one of its parameters, then the caller must replace the parameter with its corresponding argument's abstract value. If the callee's abstract value references a local variable, then the caller must conservatively replace that condition with **true**. If a callsite cannot be devirtualized, then its arguments are marked data-dependent.

Query Parameters A *query parameter* is a value that comes from the Java program and may be different for each execution of the query. For example, printfOver in Figure 1 includes a condition that depends on the salaryLimit method parameter. A condition may contain only *bindable*

variables. A variable is bindable if its last assignment occurs before the query which uses the variable executes. Our current implementation executes queries only at the beginning of a method's execution. Thus a variable must be a **final** parameter, or a variable that makes a traversal from a **final** parameter. This restriction still allows the condition in Figure 1 to be attached but disallows any conditions that depend on the results of, for example, a virtual method call invoked after executing the query.

4.4 Query Extraction and Object-Oriented Programming

Object-oriented programs exhibit several features that require a customized solution. An instance of a persistent record may use **this** to reference its fields. The analysis accommodates this behavior by modeling **this** as a path root, whenever **this** is a persistent record. If a program assigns a persistent value to an object's field, the analysis marks the assigned expression as dependent.

Java strings are instances, rather than primitive values. As such, string comparison in Java uses a string's equals method. Our analysis detects this comparison in **if** statements and converts the expression to a query condition, if the expression satisfies the restrictions from Section 3.2.

A common idiom in data-centric, object-oriented programs is to iterate through a collection and build a new collection by adding an element only if the element satisfies some condition. In general, our analysis does not track paths assigned to user-created data. However, for this idiom, the analysis computes the collection's abstract value as the union of the abstract values added to the collection. This approach maintains soundness, under the assumption that the add method makes no traversals of its own.

4.5 Recursion

The analysis may generate infinite-size values, by examining recursive methods or by examining recursive paths over which the program iterates. For example:

```
int totalManagerSalaries (Employee e) {
  if (e != null) {
    return e.salary + totalManagerSalaries(manager);
  } else {
    return 0;
  }
}
```

In our previous work, the analysis detected recursive field traversal and widened the path immediately to \top . Thus the analysis was uninformative in the presence of recursive traversals. Our current implementation uses a path representation that is more expressive and generates the path

$$e.manager^+.salary$$

Our current implementation also widens abstract values. The join of two values is \top if one of the values contains the

other. If the analysis widens a query condition to \top , then that condition becomes **true**.

4.6 Soundness

We previously proved the soundness of our analysis for a small kernel language. The analysis in this paper is not sound, because persistent values may escape through object fields; however, the program will fallback to the lazy loading provided by the persistence architecture. There is an important caveat; the persistent architecture does not know about filtered collections. If the analysis allows a filtered collection to escape, then such a collection may be used by other parts of the program that expect a differently filtered collection. A conservative solution is to remove conditions on a path which abstract collection values if the path can escape. A more sophisticated analysis could keep track when a collection reference escapes the scope of a method and reload it as an unfiltered collection. The current analysis also does not handle reflection. In particular, class loading and dynamic class generation break the devirtualization's closed-world assumption. Finally, our implementation like most persistent architectures preserves object reference identity within a single query, but does not guarantee reference identity across the entire program lifetime.

5. Implementation

We implemented query extraction using JastAdd, a Java-based compiler system that enables program analyses to be written in a modular, declarative fashion [16]. Query extraction is implemented as a source to source transformation which rewrites the program to include queries. The transformed program executes queries in HQL (Hibernate Query Language). The input to the system is a Java program and a Hibernate configuration file that identifies persistent classes and their mapping to database tables. The output is a Java source program which can be compiled and run on a standard Java virtual machine.

5.1 JastAdd

JastAdd includes a Java 1.5 compiler compiler, which we extended to perform query extraction. JastAdd provides as part of the Java compiler a control flow analysis which handles all Java control flow constructs including exceptions. Our analysis takes advantage of this control flow analysis to connect the input and output store attributes. JastAdd also includes an experimental devirtualization analysis which makes the closed-world assumption.

5.2 Code transformation

A *persistent method* is a method that accesses the special variable **root**, takes persistent parameters, or returns a persistent value. For each persistent method m , the analysis provides two values.

```

1 public Bid highBid(double threshold) {
2     AuctionService as = new AuctionService1();
3     for (Bid b : root.bids) {
4         if (b.amount > threshold) {
5             as.printBid(b);
6             System.out.println("Bid of " + b.amount);
7             return b;
8         }}
9     return null;
10 }

```

Figure 9: An example of a method which accesses persistent data.

1. A traversal summary for `root` and for any devirtualized callsites.
2. A traversal summary for the method return value.

The two traversal summaries are encoded into helper methods named `m_AV` and `m_RAV`, for “abstract values” and “result abstract value” respectively.

Persistent methods are augmented with three extra arguments: `callerPaths`, `callerParams`, and `loadParams`. The `callerPaths` parameter is a traversal summary for paths rooted at the return value of the method. This summary is composed with the traversal summary of the method at runtime. The `callerParams` parameter provides values for any parameters mentioned in `callerPaths`. The `loadParams` parameter specifies that the caller could not devirtualize the call to this method. In this case, the method will have to execute queries for any persistent parameters. For example the code in Figure 9 is transformed to the code in Figure 10 which is simplified to omit type packages.

Callsites inside the method (such as `printBid`) callsite in Figure 10 are transformed to use the version of the method which accepts additional traversal information.

5.3 Query Translation

Our prototype compiler targets the Hibernate Query Language (HQL), which is automatically translated to SQL by the Hibernate library. A traversal summary is translated into as few queries as possible given the constraints of HQL. If a portable condition contains a traversal from an object which may be null, then the transformed program may eliminate a `NullPointerException` that would have occurred in the original program. This can be fixed by adding null checks to the HQL condition or more productively warning the user of this potential bug. Supporting recursive queries is challenging, because HQL/SQL do not support transitive closure. The implementation supports recursion by unfolding recursive paths a finite number of times. A query is generated for the unfolded traversal summary. If the program traverses beyond the objects already loaded, additional queries are executed using the same unfolded traversal summary. The number of unfoldings is a parameter `nUnfold` to query extraction al-

```

1 public Bid highBid( double threshold,
2     AbstractValueSet<Path> avs,
3     Map<String, Object> callerParams,
4     boolean loadParams)
5 {
6     // Prologue
7     AbstractValueSet<Path> returnAV = highBid_RAV();
8     AbstractValueSet<Path> ts = highBid_AV();
9     Map<String, Object> queryParamValues =
10     new HashMap<String, Object>();
11     queryParamValues.put("threshold",threshold);
12     ts = QueryExecutor.composeWithReturnAV(ts, returnAV, avs,
13     queryParamValues, callerParams, false);
14     Map<PathRoot, AbstractValueSet<Path>>
15     tsPartitioned = Path.mapRootToPaths(ts);
16     Root root = new edu.utexas.plq.Root();
17     Map<String, Object> methodParamMap =
18     new QueryExecutor().executeQueries(session,
19     root, ts, Root.persistentClasses(),
20     queryParamValues, returnAV, avs,
21     callerParams, loadParams);
22
23     // Original Code
24     AuctionService as = new AuctionService1();
25     for (Bid b : root.bids) {
26         if (b.amount > threshold) {
27             as.printBid(b, ts.get(new PathRoot(" callsite_0 ")),
28             queryParamValues,true);
29             return b;
30         }}
31     return null;
32 }

```

Figure 10: Code transformation for method in Figure 9.

lowing the user to tune how recursive queries are generated. Concretely, if a program recursively traverses data organized as a binary tree of depth n and $nUnfold = m$, then the first query will retrieve the top m levels of the tree. When the program reaches one of 2^m nodes at depth m , another query is executed which retrieves m levels of the subtree rooted at that node. This continues until the program finishes its traversal. In the current implementation, recursive paths are only allowed if they are generated by method recursion, because the implementation only performs queries at method boundaries. Further engineering is required to allow the full generality of recursive paths as supported by the analysis.

6. Evaluation

We evaluated query extraction’s potential by examining benchmarks that contain transparent code and hand-optimized queries. The results demonstrate that query extraction is a viable concept. The analysis extracts the same number of queries that appear in the hand-optimized version of *persistent programs*—programs that perform only transparent persistence and that contain no explicit queries. The pro-

Benchmark	Method Declarations				Callsites			
	Total	Persistent?	Preload?	Recursive?	Total	Persistent?	Preload?	Recursive?
TORPEDO	136	67	54	0	274	41	34	0
OO7	187	123	113	3	239	79	78	6

Table 1: Number of (possibly recursive) methods/callsites in TORPEDO and OO7 across which query extraction statically extracts a query.

gram generated by query extraction sometimes loads more objects from the database than an equivalent hand-optimized program, because query extraction must statically over-approximate a program’s data requirements. However the results demonstrate that the analysis is not overly conservative: The extracted program loads fewer objects than the transparent program in many cases, and the same number of objects as the equivalent hand-optimized program in some cases.

These two metrics—number of queries executed and number of objects loaded—are the most important indicators of query extraction’s scalability, because they characterize a program’s behavior with respect to persistence. Our prototype performs well for these metrics. Other metrics—such as total execution time and analysis time—are important indicators of query extraction’s usefulness as a real-world tool. We also present our implementation’s performance for these metrics, and conclude that our prototype performs well except in a few cases.

Benchmarks No standard suite of benchmarks exists for comparing transparent programs with equivalent, hand-optimized programs that contain explicit queries. We examined existing database benchmarks and located two that serve our purposes. The TORPEDO [23] benchmark measures the number of queries executed by object-relational mappers for Java. The OO7 benchmark [5] measures the performance of object-oriented database management systems. We had to modify both benchmarks, so that we could use them to evaluate our analysis. We explain the modifications in the subsequent sections.

Experimental Configuration Our experimental configuration consisted of a server to host the database and a client machine on which the benchmarks ran. The machines were located on the same local network, and ping reports an average roundtrip time of about .250 milliseconds. The server has a 2.4 GHz Intel Pentium 4 processor with an 8KB L1 cache, 512KB L2 cache, and 1GB RAM. The server’s operating system is based on the 32bit Linux 2.6.22 kernel, and the database is PostgreSQL version 8.2.6. The client has dual 3.0 GHz Pentium-D processors with a 16KB L1 cache, 1MB L2 cache, and 2GB RAM. The server’s operating system is based on the 32bit Linux 2.6.22 kernel. All the benchmarks

ran on Sun’s HotSpot JVM version 1.5.0, with a maximum heap size of 256MB and the ParallelOld garbage collector.

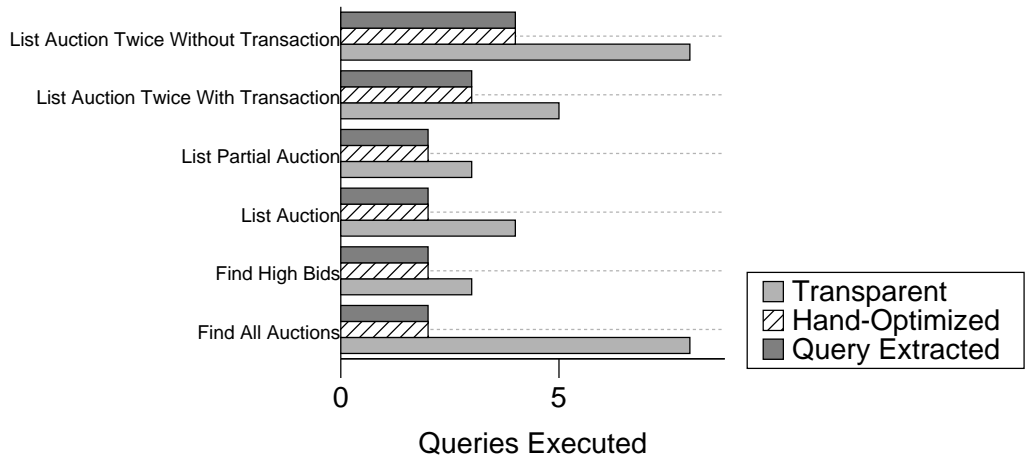
Measuring Execution Time Execution time is non-deterministic, due to the random behaviors of the operating system and JVM. To account for non-determinism, we gather a group of sample values and report the sample size, mean, and confidence interval for a 95% confidence level. We follow a multiple-iteration, multiple-JVM-invocation methodology [12, 14] to gather samples. We run several iterations of each benchmark within a single JVM invocation until the execution time reaches a steady state. Then we disable the JVM compiler, execute another iteration to clear the compilation queue, and compute the mean execution time of ten iterations. This value constitutes a sample execution time for a single benchmark invocation. We gather a group of samples by running multiple invocations.

6.1 TORPEDO

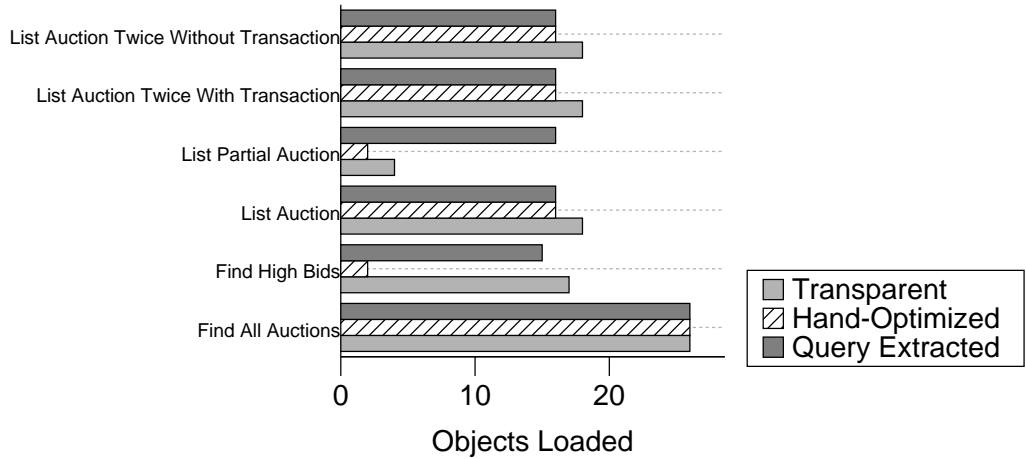
The TORPEDO [23] benchmark consists of a simple data model for an online auction service and 17 use cases which perform various operations on sample data. Six of these use cases perform read-only operations; the other 11 use cases modify the data. The application is separated into three layers: a data and persistence layer responsible for loading data from the database, a business logic layer responsible for implementing use cases, and a view layer responsible for presenting results.

We evaluated query extraction for TORPEDO by creating three versions of the benchmark. The *hand-optimized* version employs explicit queries to perform each use case. The *transparent version* uses simple queries to load the top-level object(s) required for each use case, then uses transparent persistence to load any subsequent objects. The *query-extracted* version is the result of applying our analysis to the transparent version, where each top-level query is replaced with the special keyword **root**. All three versions use Hibernate 3 to access the database. To simplify our testing we merged the view and business layers; the view layer did not contain any persistent traversals or types. We implemented only the six use cases that performed no data updates.

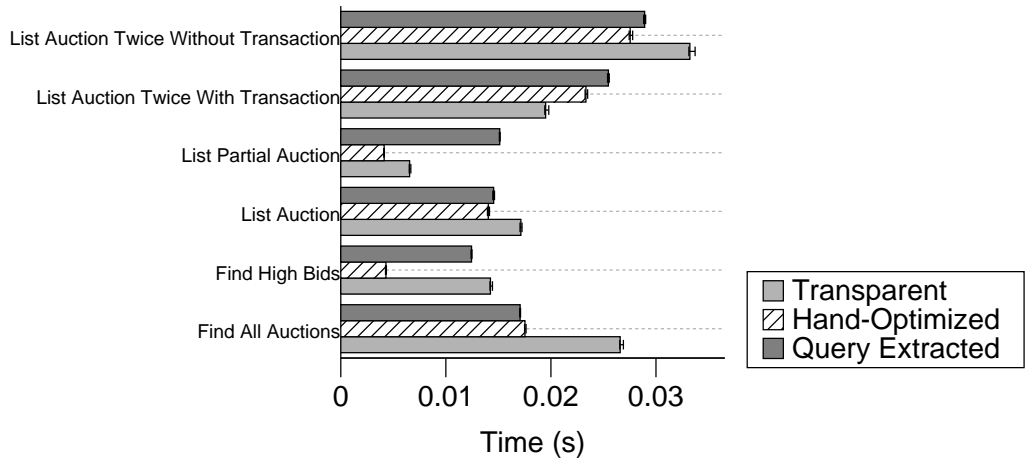
Although TORPEDO contains relatively simple traversals over a small database, the benchmark presents some difficulties for inter-procedural analysis. Table 1 lists how many methods and callsites the benchmark contains, how



(a) Queries executed per use case.



(b) Objects loaded per use case.



(c) Execution time per use case.

Figure 11: Number of queries executed and objects loaded for the TORPEDO benchmark. Query extraction locates and executes the same number of queries as the hand-optimized version in all use cases, loads fewer objects than the transparent version in all but one use case, and outperforms the transparent version in many cases.

many of those declare persistent parameters or return persistent value, how many of those for which query extraction can preload the method's data, and how many are recursive. TORPEDO has many persistent methods, but the application layers communicate through interfaces making devirtualization impossible in some cases. Every use case involves at least three methods in the different architectural layers and most involve many more. The query extraction analysis for TORPEDO took 32.94 seconds (± 0.34 seconds).

TORPEDO specifies only that researchers must report the number of queries executed. We report number of queries executed, number of objects loaded, and execution time for both benchmarks, because these metrics provide a more accurate picture of the behaviors of transparent, hand-optimized, and query-extracted programs.

Queries Executed Figure 11a shows that query extraction locates and executes the same number of queries as the hand-optimized TORPEDO. Both of these versions execute fewer queries than the transparent TORPEDO. The numbers for each use case includes commits. Each use case requires a minimum of two queries because each use case executes a commit. The hand-optimized and query-extracted TORPEDO execute one additional query to load all the auctions in the database. By contrast, the transparent TORPEDO requires eight queries to perform the same task: one to load all the auctions, three to load the items for each auction, three to load the collections of bids for each auction, and the commit.

Objects Loaded Figure 11b shows the total number of objects loaded by each TORPEDO version. In four of the six benchmarks the query-extracted version loads the same number of objects as the hand-optimized version, and both versions load at most as many objects as the transparent version. "Find High Bids" is naturally an aggregation task, because it searches for the maximum amount bid for a specified auction. The hand-optimized TORPEDO contains an aggregation query and loads the minimum number of objects. The query-extracted TORPEDO loads all the auctions' bids and computes the maximum in the client; however it still loads fewer objects than the transparent version, because the transparent version must first search for the specified auction.

The query-extracted TORPEDO loads many objects for "List Partial Auction", because the code for this use case invokes the same method as "List Auction", but passes a boolean flag that indicates the method should list only a portion of an auction. The analysis is context insensitive and cannot distinguish between the two cases, so it conservatively loads all the objects that may be required by the method.

Execution Time The TORPEDO database does not contain much data, so the use cases execute quickly. Figure 11c shows the use case execution times for each version of the benchmark. The confidence level is 95%, and the sample

size is 10 benchmark invocations. The results show that our research-quality implementation outperforms the transparent version in many cases. We found the overhead of runtime query composition to be negligible (around .004% of total execution time). We believe that more engineering effort would yield even better results.

6.2 OO7

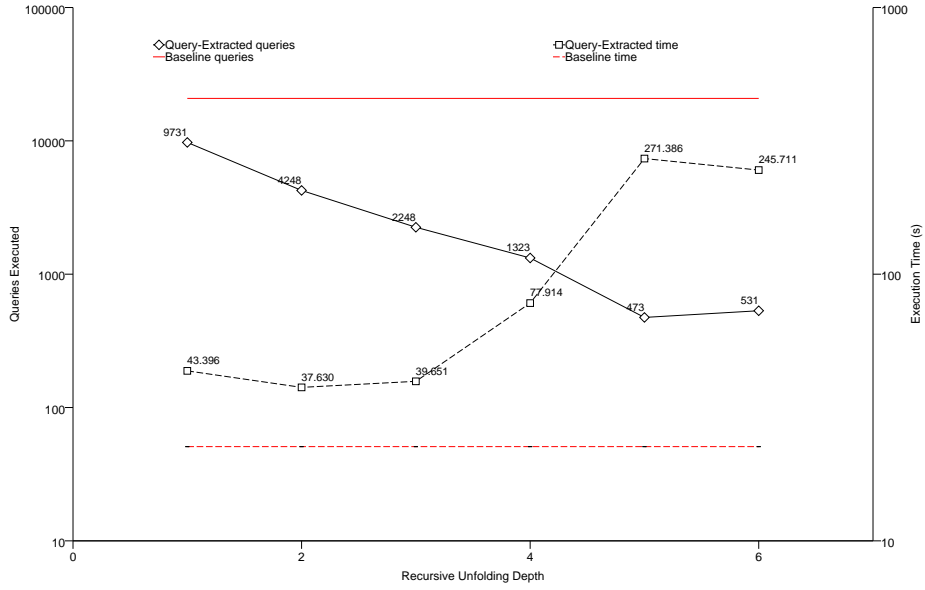
OO7 [5] is based on a CAD/CAM application that defines a composite structure by a highly recursive and interrelated graph of components and parts. The OO7 benchmark is not representative of the most common operations in typical transactional/enterprise applications, because OO7 focusses on extensive traversals of hierarchical structures. However, it is widely used and presents some interesting challenges for query extraction.

The OO7 specification defines three kinds of use cases: *traversals*, *queries*, and *structural modifications*. The traversals scan the object graph and collect information. There are six traversal use cases, confusingly labeled Traversal 1 through Traversal 9 (Traversal 4, Traversal 5, and Traversal 7 do not exist). Traversal 2 and Traversal 3 perform database updates; the remaining traversals perform read-only tasks. There are seven query use cases labeled Query 1 through Query 8 (Query 7 does not exist). All seven queries perform read-only operations. There are two structural modification use cases. One use cases inserts values into the database, and the other deletes values from the database. The specification describes three database sizes: small, medium, and large. Our evaluation is for the small database size, which contains around 41,000 objects.

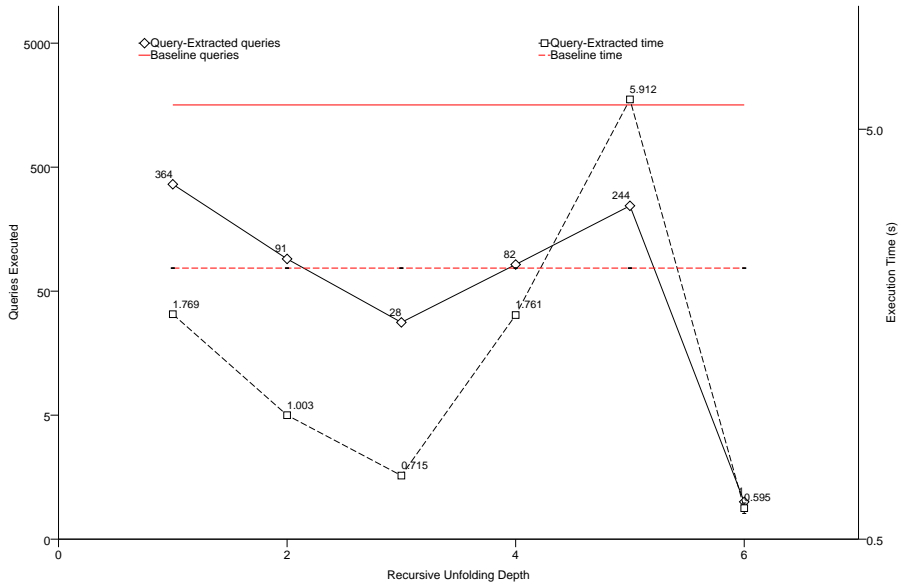
Our evaluation is based on a version of OO7 that uses Hibernate 3, which we had implemented for a previous research effort. Our OO7 version implements the 11 read-only use cases in the specification and omits the four use cases that perform updates. The traversal use cases are based on transparent persistence. We applied query extraction to these use cases to generate versions that contain explicit queries. We then compared the performance of these two versions. Neither the OO7 specification nor reference implementations provide a version of the traversals that contain hand-optimized queries, so our evaluation for these use cases compares query-extracted performance to transparent persistence performance. The query use cases contain hand-optimized, explicit queries. We compared these use cases version to versions generated by applying query extraction to equivalent, use cases. We present results for this comparison. Based on the results of Section 6.1, we expect that query-extracted performance would compare favorably to transparent performance for these use cases.

Table 1 lists the persistent characteristics of the methods in OO7. The query extraction analysis for OO7 took 98.39 seconds (± 3.60 seconds).

OO7 does not specify which metrics to report, so we report number of queries executed, number of objects loaded,



(a) Traversal 1 queries and time depend on unfolding depth.



(b) Traversal 6 queries and time depend on unfolding depth.

Figure 14: OO7 performance for original and query-extracted versions.

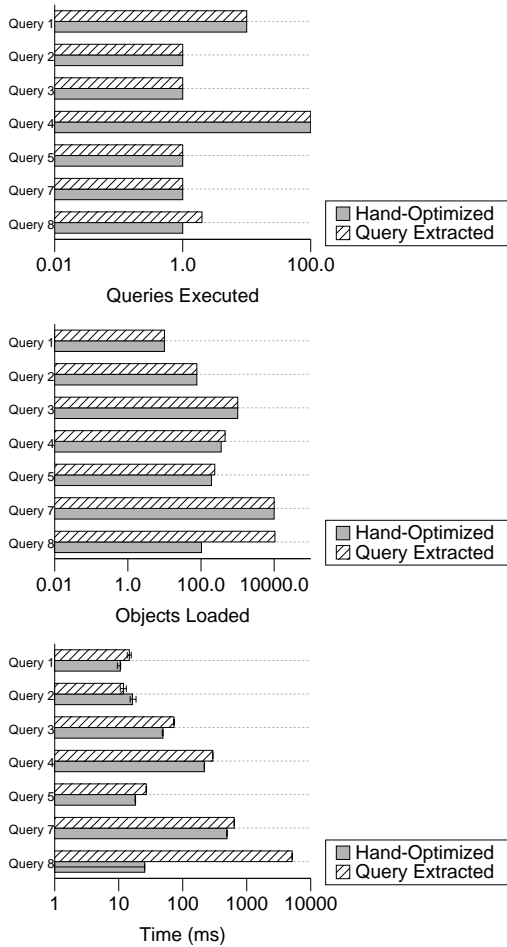


Figure 12: Queries executed, objects loaded, and execution time per Query use case.

and execution time. Because OO7 contains recursive traversals of an object graph, the performance of the query-extracted version for some use cases depends on how deep the analysis unfolds recursive traversals. Query extraction is parameterized by this depth, as described in Section 5.3. We first compared the performance of the transparent OO7 against a query-extracted OO7 with an unfolding depth of 1. Figures 6.2 contains the results for the Query use cases, and Figure 6.2 contain the results for the Traversal use cases. All execution time values are for a confidence level of 95% and a sample size of ten benchmark iterations.

Performance for Query Use Cases The evaluation for these uses cases compares query-extracted performance against hand-optimized performance. Query extraction performs as well as the hand-optimized version for the first three queries.

The query-extracted versions of Query 4 and Query 5 execute the same number of queries as the corresponding hand-optimized version, but load more objects. These extra

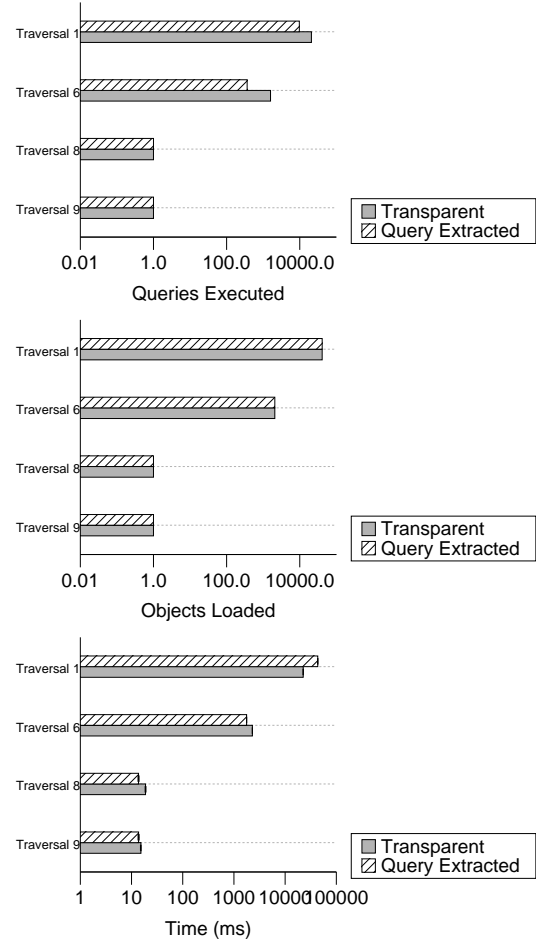


Figure 13: Queries executed, objects loaded, and execution time per Traversal use case.

objects are due to the fact that the transparent program from which the query-extracted version is generated traverses a relationship to evaluate a condition, so query extraction must load the traversed object to maintain the program’s semantics. The hand-optimized query references the same relationship in the query’s where clause, but does not need to load the objects referenced in the condition.

Query 8 performs an ad-hoc join of two collections. Query extraction does not optimize these kinds of traversals. The hand-optimized version loads all the required objects with a single query. The query-extracted version performs one query for each collection, but loads too many objects because the use case’s condition violates the *nested* restriction discussed in Section 3.2.

For all use cases except Query 8, the query-extracted version takes about the same time to execute as the hand-optimized version, or is a little slower. The query-extracted version of Query 8 takes three orders of magnitude longer to execute than the hand-optimized version, because query extraction does not optimize this query.

Performance for Traversal Use Cases The evaluation for these use cases compares query-extracted performance against transparent performance. Traversal 8 and Traversal 9 yield uninteresting comparisons for queries executed and objects loaded, because these use cases traverse a single object. The query-extracted version executes more quickly than its transparent counterpart for these two use cases. Query extraction executes fewer queries for Traversal 1 and Traversal 6. The transparent versions of these use cases employ transparent persistence to lazily load objects as they are traversed. The query-extracted version prefetches these objects and so uses fewer queries. The query-extracted versions of these two use cases always load the same number of objects as the transparent version, because the use cases traverse *all* the objects in the database along a certain path.

Figure 14a shows how the number of queries and the execution time vary with unfolding depth for Traversal 1. This use case traverses the entire object graph stored in the database—a behavior that cannot be easily expressed with a relational query language like HQL. Although the query-extracted version executes fewer queries than the transparent version, the query-extracted version takes much more time to execute. This overhead is due to the large amount of redundant data in each row that is required to represent an object graph in a relational table. As the number of queries decreases, the amount of redundant data increases, so there is an inverse relation between the number of queries and the execution time.

The results for Traversal 6 highlights query extraction’s advantages. This use case traverses a sub-tree of the object graph. Figure 14b shows how the number of queries and the execution time vary with unfolding depth for this use case. Note that the number of queries is a good indication of execution time for this use case. The query-extracted version always executes fewer queries than the transparent version, and takes less time than the transparent version except when the unfolding depth is 5. This spike occurs because the data happens to have a tree structure of height six. Thus unfolding depths that are not factors of six are suboptimal, for the reasons described in Section 5.3. In general, the optimal unfolding depth depends on the data characteristics.

7. Related Work

Interest in the problem of integrating programming languages and databases has enjoyed a recent resurgence. Most researchers have focused on providing queries as first-class members of a programming language. Kleisli [35], Haskell/DB [21] and Links [10] are functional programming environments that provide comprehension syntax as a means to specify database queries. LINQ extends C#’s syntax and type system to include similar features [4]. Safe queries provide typed, first-class queries for Java [9]. The Java Query Language (JQL) extends Java to support optimizable queries of in-memory objects [34].

These solutions all provide a language-based, safe alternative to embedded query strings. They differ from our solution in that they require programmers to learn a new syntax or API, and that the programmer must write explicit queries. Although these queries benefit from type-safety and automatic conversion to a database query language, programmers still bear the burden of declaring their needs for persistent data. This declaration introduces a subtle dependency in the program between the structure of the data declared in the query and the structure of data traversed by the program. Furthermore, explicit queries reduce the modularity of programs by concentrating queries in one program location, reducing opportunities to exploit redundant data traversals.

Our approach infers a programmer’s persistent data use by analyzing programs in an existing language. However, static analysis cannot in general detect common query idioms like aggregation and existence. New languages, type systems, and constructs also are better solutions for other artifacts of impedance mismatch like null values. It is an open problem to find the “sweet spot” between these two approaches, but we believe their combination provides the most promise for integrating programming-languages and databases.

Neubauer and Theimann partition a sequential program run at one location into semantically equivalent, independent, distributed processes [28]. Their approach provides software engineering benefits similar to ours, except for multi-tier applications.

The DBPL language [32] and its successor Tycoon [25] explored optimization of search and bulk operations within the framework of orthogonal persistence. Tycoon proposed integrating compiler optimization and database query optimization [11]. Queries that cross modular boundaries were optimized at runtime by dynamic compilation [31]. The languages included explicit syntax for writing queries or bulk operations on either persistent or non-persistent data.

Several researchers have extended object persistence architectures to leverage *traversal context*—access patterns, including paths—to dynamically predict database loads and prefetch the predicted values [3, 15, 18]. Because our work generates queries which could be used in object persistence architectures, the two techniques could be combined to achieve further performance benefits.

8. Conclusion

This paper presents an automatic technique for extracting queries from object-oriented programs that use transparent persistence. The work builds on our previous formal study for query extraction in a kernel language without procedures. The key problem for interprocedural analysis is propagating query information across procedure boundaries: persistent data needed for procedure parameters should be preloaded by the caller, and conversely, the procedure should preload all the data needed by the call site from its return value.

Procedure parameters are handled by devirtualization, but procedure results use a novel combination of static analysis and dynamic query composition. While parameters are only preloaded if devirtualization succeeds, the dynamic composition always allows a called method to preload data for its callee. We present a prototype Java compiler with query extraction including support for recursion query parameters, persistent parameters and return values. The system targets Hibernate Query Language; and discussed some issues in automatically generating queries with different query languages. We evaluated the technique using the TORPEDO and OO7 benchmarks. This work demonstrates the feasibility of query extraction in a practical setting.

References

- [1] M. Atkinson and R. Morrison. Special issue on persistent object systems. *VLDB Journal*, 4(3), 1995.
- [2] M. P. Atkinson and R. Morrison. Orthogonally persistent object systems. *VLDB Journal*, 4(3):319–401, 1995.
- [3] P. A. Bernstein, S. Pal, and D. Shutt. Context-based prefetch for implementing objects on relations. In *The VLDB Journal*, pages 327–338, 1999.
- [4] G. M. Bierman, E. Meijer, and W. Schulte. The essence of data access in $c\omega$. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, pages 287–311, 2005.
- [5] M. J. Carey, D. J. DeWitt, C. Kant, and J. F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 414–426. ACM Press, 1994.
- [6] D. Cengija. Hibernate your data. *onJava.com*, 2004.
- [7] S. Chaudhuri. An overview of query optimization in relational systems. In *Proc. of Symp. on Principles of Database System (PODS)*, pages 34–43, 1998.
- [8] P. P. Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.
- [9] W. R. Cook and S. Rai. Safe query objects: statically typed objects as remotely executable queries. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 97–106, New York, NY, USA, 2005. ACM Press.
- [10] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. (unpublished manuscript), 2007.
- [11] A. Gaweci and F. Matthes. Integrating query and program optimization using persistent CPS representations. In M. P. Atkinson and R. Welland, editors, *Fully Integrated Data Environments*, ESPRIT Basic Research Series, pages 496–501. Springer Verlag, 2000.
- [12] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 57–76, New York, NY, USA, 2007. ACM.
- [13] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proc. of the Intl. Conf. on Software Engineering (ICSE)*, pages 645–654, 2004.
- [14] J. Ha, M. Gustafsson, S. M. Blackburn, and K. S. McKinley. Microarchitectural characterization of production jvms and java workloads. Mar 2008.
- [15] W.-S. Han, Y.-S. Moon, and K.-Y. Whang. PrefetchGuide: capturing navigational access patterns for prefetching in client/server object-oriented/object-relational dbms. *Information Sciences*, 152(1):47–61, 2003.
- [16] G. Hedin and T. Ekman. The JastAdd system – modular extensible compiler construction. *Science of Computer Programming*, 69:14–26, 2007.
- [17] Hibernate reference documentation. http://www.hibernate.org/hib_docs/v3/reference/en/html, May 2005.
- [18] A. Ibrahim and W. Cook. Automatic prefetching by traversal profiling in object persistence architectures. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, 2006.
- [19] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A study of devirtualization techniques for a java just-in-time compiler. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 294–310, New York, NY, USA, 2000. ACM Press.
- [20] D. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, 1968.
- [21] D. Leijen and E. Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd conference on Domain-specific languages*, pages 109–122. ACM Press, 1999.
- [22] D. Maier. Representing database programs as objects. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Languages, Papers from DBPL-1*, pages 377–386. ACM Press / Addison-Wesley, 1987.
- [23] B. E. Martin. Uncovering database access optimizations in the middle tier with TORPEDO. In *Proceedings of the 21st International Conference on Data Engineering*, pages 916–926. IEEE Computer Society, 2005.
- [24] V. Matena and M. Hapner. Enterprise Java Beans Specification 1.0. Sun Microsystems, 1998.
- [25] F. Matthes, G. Schroder, and J. Schmidt. Tycoon: A scalable and interoperable persistent system environment. In M. Atkinson, editor, *Fully Integrated Data Environments*. Springer-Verlag, 1995.
- [26] Microsoft Corporation. The LINQ project. msdn.microsoft.com/netframework/future/linq.
- [27] R. Morrison, R. C. H. Connor, G. N. C. Kirby, D. S. Munro, M. P. Atkinson, Q. I. Cutts, A. L. Brown, and A. Dearle. The Napier88 persistent programming language and environment. In M. P. Atkinson and R. Welland, editors, *Fully Integrated Data Environments*, pages 98–154. Springer, 1999.

- [28] M. Neubauer and P. Thiemann. From sequential programs to multi-tier applications by program transformation. In *Proc. of the ACM Symp. on Principles of Programming Languages (POPL)*, pages 221–232, 2005.
- [29] Rational Software. Whitepaper on the UML and Data Modeling, 2000.
- [30] C. Russell. Java Data Objects (JDO) Specification JSR-12. Sun Microsystems, 2003.
- [31] J. Schmidt, F. Matthes, and P. Valduriez. Building persistent application systems in fully integrated data environments: Modularization, abstraction and interoperability. In *Proceedings of Euro-Arch'93 Congress*. Springer Verlag, Oct. 1993.
- [32] J. W. Schmidt and F. Matthes. The DBPL project: advances in modular database programming. *Inf. Syst.*, 19(2):121–140, 1994.
- [33] B. A. Wiedermann and W. R. Cook. Extracting queries by static analysis of transparent persistence. In *Proc. of the ACM Conf. on Principles of Programming Languages (POPL)*, pages 199–210, 2007.
- [34] D. Willis, D. J. Pearce, and J. Noble. Efficient object querying in Java. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, Nantes, France, 2006.
- [35] L. Wong. Kleisli, a functional query system. *J. Funct. Program.*, 10(1):19–56, 2000.
- [36] M. Zand, V. Collins, and D. Caviness. A survey of current object-oriented databases. *SIGMIS Database*, 26(1):14–29, 1995.