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 programs to access persistent data with the same ease as non-persistent data. In this work we demonstrate the feasibility of optimizing transparently persistent programs by extracting queries to efficiently prefetch required data. A static analysis derives query structure and conditions across methods that access persistent data. Using the static analysis, our system transforms the program to execute explicit queries. The transformed program composes queries across methods to handle method calls that return persistent data. We extend an existing Java compiler to implement the static analysis and program transformation, handling 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.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers,Optimization; H.2.3 [*Database Management*]: Languages

General Terms Languages, Performance

Keywords Programming Languages, Databases, Static Analysis, Object-Relational Mapping, Attribute Grammars

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 [21]. From a programming language viewpoint, databases manage *persistent* data, which has a lifetime longer than the execution of an individual program [32, 1, 25]. 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 [29], 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:

- An interprocedural static analysis to extract queries from Java programs that use transparent persistence. The analysis handles virtual method calls by introducing additional queries where necessary and composing analysis results at runtime.
- A Java-based implementation that converts analysis results to queries that target the popular Hibernate persistence system [6].
- A practical approach to recursive data traversals that unfolds the recursion in stages of finite depth.
- An evaluation of the system using the TORPEDO [22] and OO7 [5] 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. Cur-

* This work was supported by the National Science Foundation under Grant No. 0448128.

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.

```
class Client { ...
  void reportZip(DataAccess db, int zip) {
    for (Employee e : db.getEmployees())
      if (e.zip == zip)
        printIfOver(e, 65000);
  }
  void printIfOver(Employee e, final int salaryLimit) {
    if (e.salary > salaryLimit)
      printEmployee(e);
  }
  void printEmployee(Employee e) {
      print(e.name); print(": ");
      print(e.manager.name);
}}
class DataAccess { ...
  Collection<Employee> getEmployees() {
    return root.getEmployees();
}}
```

Figure 1: Procedures and transparent persistence.

rently 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, 24], 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 prefetching and merging.

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 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 printIfOver method. The printIfOver 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

```
void reportZip(DataAccess db, int zip) {
  Query q = db.createQuery( // create query
    "from Employee e
     left join fetch e.manager
     where e.zip == :zip
       and e.salary > :salaryLimit");
  // set the parameters
  q.setParameter("zip", zip, Hibernate.INTEGER);
  q.setParameter("salaryLimit", 65000, Hibernate.INTEGER);
  for (Employee e : q.list()) // execute the query
      printEmployee(e);              // no test required
}
```

Figure 2: Query execution using Hibernate.

```
void reportZip(DataContext db, int zip) {
  // preload specification
  DataLoadOptions dlo = new DataLoadOptions();
  dlo.LoadWith<Employee>(e => e.manager); // ERROR!
  db.LoadOptions = dlo;
  // query
  int salaryLimit = 65000;
  var employees = from e in db.Employee
    where e.zip == zip && e.salary > salaryLimit
    select e;
  foreach (Employee e in employees)
      printEmployee(e);
}
```

Figure 3: Query and load options in Linq.

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 [15] 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, because 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.

```
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.

These problems have been fixed by more recent query mechanisms, including Linq [4, 24] and Safe Query Objects [8]. 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, Figure 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 cause 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 implementation 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 printIfOver. 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. Overview of Query Extraction

Query extraction infers a description of the subset of database values that a transparently persistent program needs in order to execute. The technique is a source-to-source transformation that takes as input an object-oriented program written in a transparent style and produces an equivalent program that contains explicit queries. Query extraction proceeds in two stages. First a path-based analysis computes an over-approximation of the database records required by each method in the program. Then the original program is transformed so that each method executes an explicit query. The explicit query pre-loads the database records specified by the analysis.

In this section, we describe the kinds of programs query extraction can handle. We then briefly outline the analysis and transformation phases, which are discussed in more detail in Sections 4–6.

Data Model Query extraction models the program's persistent data store as a rooted, directed graph of database records. 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. Our implementation relies on Hibernate to provide a description of the persistent data schema and to load database values into memory.

Program Model Query extraction assumes the program accesses persistent data transparently. The technique requires no change to the language, nor does it require the programmer to write annotations. The analysis identifies persistent data via a transitive closure of traversals from **root**.

Our prototype implementation operates on a subset of Java. It does not handle features like dynamic class loading and reflection. Furthermore, query extraction is defined for read-only operations on persistent data. Although our implementation is for Java, our technique is applicable to any object-oriented programming language.

Path-based Analysis The analysis phase of query extraction models program values as *paths*. A path describes a set of database records and consists of three components:

1. The sequence of field names that the program traverses to reach the records,
2. The condition(s) under which the program accesses the records,
3. A data dependence flag that indicates whether the program's result depends on the value of the database records.

$$
\begin{array}{rcl}
f \in \mathit{FieldName} & & v \in \mathit{VarName} \\
\mathsf{l} \in \mathit{Literal} & & \mathsf{op} \in \mathit{Op} \\
p \in \mathit{Path} & : & \overrightarrow{\mathit{FieldName}} \times \mathit{AbsOp} \times \mathit{Dependence} \\
o \in \mathit{AbsOp} & : & \mathit{Op} \times \overrightarrow{\mathit{AbsValue}} \\
d \in \mathit{Dependence} & : & \{\textsf{true} \mid \textsf{false}\} \\
av \in \mathit{AbsValue} & : & \bot + \overline{\mathit{Literal}} + \overline{\mathit{AbsOp}} + \overline{\mathit{Path}} \\
s \in \mathit{Store} & : & \mathit{VarName} \mapsto \mathit{AbsValue}
\end{array}
$$

Figure 5: Abstract values.

Section 4 describes an intraprocedural path-based analysis. Section 5 describes how the results of the intraprocedural analysis may be composed to compute a whole-program analysis.

Program Transformation The program transformation phase of query extraction first generates explicit queries based on the analysis results. The phase then outputs a new program that uses explicit queries to pre-load the necessary database values. The new program operates over these preloaded data values.

Soundness and Precision Query extraction is *sound* if the queries for each method in a program load all the data necessary to perform the method's operations. A query can be viewed as describing a subset of the database. For each method, the analysis is sound if the subset of the database specified by the explicit query can be substituted for the entire database without changing the behavior of the method.

Query extraction is *precise* if it loads no more data than is needed by each method in the program. Exact precision is undecidable in general. Our analysis tries to be as precise as possible, and our implementation contains some optimizations that improve precision.

4. Intraprocedural Query Extraction

Our previous approach to intraprocedural query extraction was expressed as an abstract interpretation for a language with no procedures [29]. This section recasts our previous approach as an attribute grammar, which more closely follows our implementation technique. We first define query extraction for a subset of Java that contains no methods. Section 5 extends query extraction to the interprocedural case.

4.1 Abstract Values

The path-based analysis approximates real-world values with abstract values. Figure 5 formally defines these abstractions. The domains $\mathit{FieldName}$ and $\mathit{VarName}$ describe the fields and variables that appear in a program, respectively. The domains $\mathit{Literal}$ and Op contain literal values (e.g., `1` or `"a"`) and operations (e.g., $==$ or $+$) respectively. These domains are syntactic, meaning their elements are described by the syntax of the analyzed programming language.

The most important abstract value is a path, which describes database values. A path p is represented as a three-tuple $(\overrightarrow{f}, c, d)$, consisting of an ordered sequence of field names $\overrightarrow{f}$ that identifies the fields traversed to reach a set of database values, a condition c under which the traversal is performed, and information d about the traversal's data dependences. We sometimes elide a path's condition and dependence information when that information is not significant or may be easily determined from context.

Paths abstract over collections, so that all the objects in a collection share 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 path for that collection is employees; the path for any employee in that collection is employees.ι; and the path for any employee's salary in that collection is employees.ι.salary.

A path's *condition* describes the circumstances under which a program accesses data values. The condition is taken from the domain AbsOp, whose elements are syntactic expressions that contain operators for numerical and logical operations over one or more abstract values.

Each path's condition is derived from one or more conditional expressions that appear in the program. 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 conditional expression that appears in a program is a *query condition* if it can be included in a database query, as described in Section 4.3. If a conditional expression does not satisfy the requirements for being a query condition, then the expression will not be associated with any path.

A path's *data dependence* is a boolean flag that specifies whether the path's values affect the data produced by the program. 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.

An abstract value av can be the bottom element $\bot$ (which represents unknown information), a set of literals, a set of abstract operations, or a set of paths. A Store maps a variable name to an abstract value.

4.2 Attribute Grammars

An attribute grammar is a way to specify the semantics of a context-free language [18]. An *attribute* is a semantic function that is associated with a given node of a program's Abstract Syntax Tree (AST). For a simple calculator language, a value attribute would return the appropriate integer value

$\cdot$	Abstract Value $\mathsf{AV}(\cdot) : AbsValue$	Paths $\mathsf{P}(\cdot) : \overline{Path}$	Output Store $\mathsf{OS}(\cdot) : Store$
null	$\bot$	$\emptyset$	$\mathsf{IS}(\cdot)$
l	$\{\mathsf{l}\}$	$\emptyset$	
root	$\{(\epsilon, \textbf{true}, \textbf{false})\}$	$\mathsf{AV}(\cdot)$	
e.f	$\mathsf{AV}(\mathsf{e}).\mathsf{f}$	$\mathsf{AV}(\cdot)$	
e1 op e2	$\mathsf{AV}(\mathsf{e1})$ op $\mathsf{AV}(\mathsf{e2})$	$\mathsf{TS}(\mathsf{e1}) \cup \mathsf{TS}(\mathsf{e2})$	
v	$\mathsf{IS}(\cdot)[\mathsf{v}]$	$\emptyset$	
v=e	$\bot$	$\mathsf{TS}(\mathsf{e})$	$[\mathsf{v} \mapsto \mathsf{AV}(\mathsf{v}) \cup \mathsf{AV}(\mathsf{e})]\mathsf{IS}(\cdot)$
if e s1 **else** s2		$\mathsf{TS}(\mathsf{e}) \cup \mathsf{TS}(\mathsf{s1}) \cup \mathsf{TS}(\mathsf{s2})$	$\mathsf{OS}(\mathsf{s1}) \cup \mathsf{OS}(\mathsf{s2})$
for (v : e) s		$\mathsf{TS}(\mathsf{e}) \cup \mathsf{AV}(\mathsf{e}).\iota \cup \mathsf{TS}(\mathsf{s})$	$\mathsf{OS}(\mathsf{s})/\mathsf{v}$
s1;s2		$\mathsf{TS}(\mathsf{s1}) \cup \mathsf{TS}(\mathsf{s2})$	$\mathsf{OS}(\mathsf{s2})$
other$[\![\mathsf{e}]\!]$		$\mathsf{TS}(\mathsf{e})$	$\mathsf{IS}(\cdot)$

(a) Synthesized attributes.

Inherited Attribute	$\cdot$	Inherited Attribute Values for Descendants
Input Store $\mathsf{IS} : Store$	**for** (v : e) s	$\mathsf{IS}(\mathsf{s}) \leftarrow [\mathsf{v} \mapsto \mathsf{AV}(\mathsf{e}).\iota]\mathsf{IS}(\cdot) \cup \mathsf{OS}(\mathsf{s})$
	s1;s2	$\mathsf{IS}(\mathsf{s2}) \leftarrow \mathsf{OS}(\mathsf{s1})$
Query Condition $\mathsf{C} : AbsOp$	**if** e s1 **else** s2	$\mathsf{C}(\mathsf{s1}) \leftarrow \mathsf{C}(\cdot) \wedge \mathsf{AV}(e)$ $\mathsf{C}(\mathsf{s2}) \leftarrow \mathsf{C}(\cdot) \wedge \mathsf{not}(\mathsf{AV}(e))$ if e is a valid query condition
Data Dependence $\mathsf{D} : Dependence$	*effectful*$[\![\mathsf{e}]\!]$	$\mathsf{D}(\mathsf{e}) \leftarrow \textbf{true}$
Iterator Context $\mathsf{IT} : \overrightarrow{Path}$	**for** (v : e) s	$\mathsf{IT}(\mathsf{s}) \leftarrow \mathsf{IT}(\cdot) + \mathsf{AV}(e).\iota$ if $\mathsf{AV}(e).\iota$ extends $\mathsf{IT}(\cdot)$

(b) Inherited attributes.

$\cdot$	Traversal Summary $\mathsf{TS}(\cdot) : \overline{Path}$
v	$\begin{cases} (\mathsf{AV}(\cdot), \mathsf{C}(\cdot), \textbf{true}) & \text{if } \mathsf{D}(\cdot) \\ (\mathsf{P}(\cdot), \mathsf{C}(\cdot), \textbf{false}) & \text{if } \neg\mathsf{D}(\cdot) \end{cases}$
v=e	$\mathsf{TS}(\mathsf{e}) \cup iter(\cdot)$
other$[\![\cdot]\!]$	$\mathsf{P}(\cdot) \cup \begin{cases} iter(\cdot) & \text{if } \mathsf{D}(\cdot) \\ \emptyset & \text{if } \neg\mathsf{D}(\cdot) \end{cases}$
where $iter(\cdot) = (\{last(\mathsf{IT}(\cdot))\}, \mathsf{C}(\cdot), \textbf{true})$	

(c) Computing traversal summaries.

Figure 6: Attribute grammar that computes traversals for a subset of Java syntax.

for an AST node of type Int and would return the sum of the operands for an AST node of type Add.

Attributes are typically partitioned into two classes: *synthesized* and *inherited*. The value of a synthesized attribute may depend on the values of its node's descendants. The value of an inherited attribute may depend on the values of its node's ancestors. Attribute values may induce a circular dependence. A fixed-point algorithm computes the value for each attribute [20].

4.3 Intraprocedural Path Analysis

The intraprocedural path analysis computes a *traversal summary* for each statement and expression in a method. A traversal summary is a set of paths representing all the data needed to execute a statement or expression. A method's traversal summary may be composed with those of other methods to compute a whole-program analysis.

Figure 6 defines the path-based analysis as an attribute grammar over Java abstract syntax. The traversal summary of a syntactic element is defined by a synthesized attribute TS. Attribute TS is itself defined with the help of three other synthesized attributes: AV, P, and OS, whose definitions appear in Figure 6a. In this figure, the $\cdot$ symbol represents a Java expression or statement. Attribute P is a synthesized attribute that collects all the traversal summaries of an element's sub-expressions.

The attribute AV represents the abstract value of a given expression or statement. The abstract value of **null** is $\bot$, and the abstract value of a literal is the set containing that literal.

The abstract value of the special variable **root** is a path with no field traversals and the default condition and dependence: $(\epsilon, \textbf{true}, \textbf{false})$. A field traversal e.f concatenates field names to previously computed paths, according to this definition:

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

Binary operations are interpreted as abstract operators over abstract values; these operations are also used to represent query conditions.

The synthesized *output store* attribute OS describes an element's effect on the store. The output store is typically a function of the inherited *input store* attribute IS. The way in which stores flow between elements and their constituents is standard, although a few cases deserve mention.

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

A **for** statement creates a special path $\mathsf{AV}(\mathsf{e}).\iota$ 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 input store value depends on its output store value, which forms a circular dependency (Figure 6b).

A variable's abstract value is the value contained in the input store. If the input store contains no binding for a variable, then the variable's abstract value is undefined ($\perp$).

All other Java statements (e.g., exception-handling blocks, **while** loops, etc.) are given a default interpretation, by the *other* case: the paths are unioned and the store is unchanged.

Query Conditions Under certain circumstances, a conditional expression that appears in a program may be shipped to the database as a *query condition*. Query conditions filter the data loaded by a program. A condition in an **if** statement can be a query condition only if it satisfies three requirements: 1) it contains only *portable operators*, 2) it is *pointwise*, and 3) if the condition appears in a nested loop, the loop's collections must participate in a *master-detail* relationship.

An operation is *portable* if it can be performed both in the database and 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 expect only 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 dependences, 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;
}}
```

The analysis will not attach the condition in the example's **if** statement to any paths, because the condition induces a loop-carried dependence.

A conditional expression that appears in nested loops is a query condition only if the collections over which the loops iterate participate in a *master-detail* relation. An inner-loop collection is a detail of an outer-loop collection if the inner collection is a traversal from the iteration variable of the outer loop. These master-detail loops are a common idiom. For example, a program might iterate over all purchase orders, then iterate over each item in the purchase order. Other kinds of nested loops do sometimes arise; they correspond to ad-hoc joins that find correlations between collections that have no explicit relationship between them.

The query condition attribute C—defined in Figure 6b—collects conditions under which an expression or statement is executed. The attribute is inherited by all sub-expressions.

Data Dependence and Iterators Figure 6b also defines auxiliary inherited attributes data dependence D and iterator context IT.

The data dependence attribute D flags expressions whose execution directly affects the program output. It is true for any statement or expression that can affect non-local state, 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 *effectful*$[\![\mathsf{e}]\!]$ 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 6c defines the traversal summary attribute TS. It combines the path, condition, and data dependence attributes into a traversal summary. Given a set of paths P, a condition c, and data dependence d, the notation (P, c, d) represents a new set of paths whose conditions and data dependence are replaced:

$$(P, c, d) = \{(\overrightarrow{f}, c, d) \mid (\overrightarrow{f}, \_, \_) \in P\}$$

The traversal summary computation depends on the kind of statement or expression being analyzed. For variables, the

computation ensures that using an expression has the same effect as using a variable that has been assigned the value of the expression. Any data-dependent expression generates an extra path that corresponds to the inner-most iteration. The intuition behind this definition is that data dependence 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.

Example Figure 7 illustrates an application of the path-based analysis. The analyzed program is a one-method version of the example from Figure 1. Each program statement is numbered, and each number corresponds to a node in the program's AST (nodes for expressions are elided). An AST node is represented as a table whose middle column is the statement number and whose left and right columns contain values for synthesized and inherited attributes, respectively. If a node contains no value for an inherited attribute, then that attribute's value is the same as the parent node's value.

The analysis begins with an empty input store, a **true** query condition, and an empty iterator context. Statements one and three are assignments, and their corresponding output stores contain the results of the assignment.

The **for** loop also modifies the store by assigning a path to the loop variable e. Note that this assignment appears in the input store for statement five. The **for** loop also sets the iterator context value for statement five, to indicate that the loop's body operates on elements of the employees collection.

Statements five and six are **if** statements that set the query condition for their corresponding children. Statement seven is effectful because it prints data, so its traversal summary includes the iterator context.

The traversal summary for the entire method consists of the following paths:

$\overrightarrow{f}$	c	d
employees employees.ι employees.ι.zip	**true**	**false**
employees.ι.salary	employees.ι.zip=="78751"	**false**
employees.ι.name employees.ι.manager.name employees.ι	employees.ι.zip=="78751" $\wedge$ employees.ι.salary $>$ 65000	**true**

5. 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 prefetches the data needed by the procedure. For persistent return values, the procedure prefetches 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.

Transmitting query information across procedure boundaries is difficult, especially in object-oriented programs that make extensive use of virtual method calls. There are at least five possible approaches to this situation: *devirtualization*, *specialization*, *over-approximation*, *query separation*, and *dynamic composition*.

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.

Dynamic composition of static analysis allows queries to be combined across virtual methods calls. It can be used for method arguments and method return values.

Our current implementation uses devirtualization and query separation to handle callsites with persistent parameters. Our implementation uses dynamic composition to handle virtual callsites that return persistent values.

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 [17]. We believe that these techniques could work with query extraction, but we have not yet tried to integrate them.

5.1 Abstract Values

To extend the analysis of the previous section to the interprocedural case, we must first extend the abstract values.

Rooted Paths Whereas intraprocedural analysis considers only paths that traverse from variable **root**, interprocedural

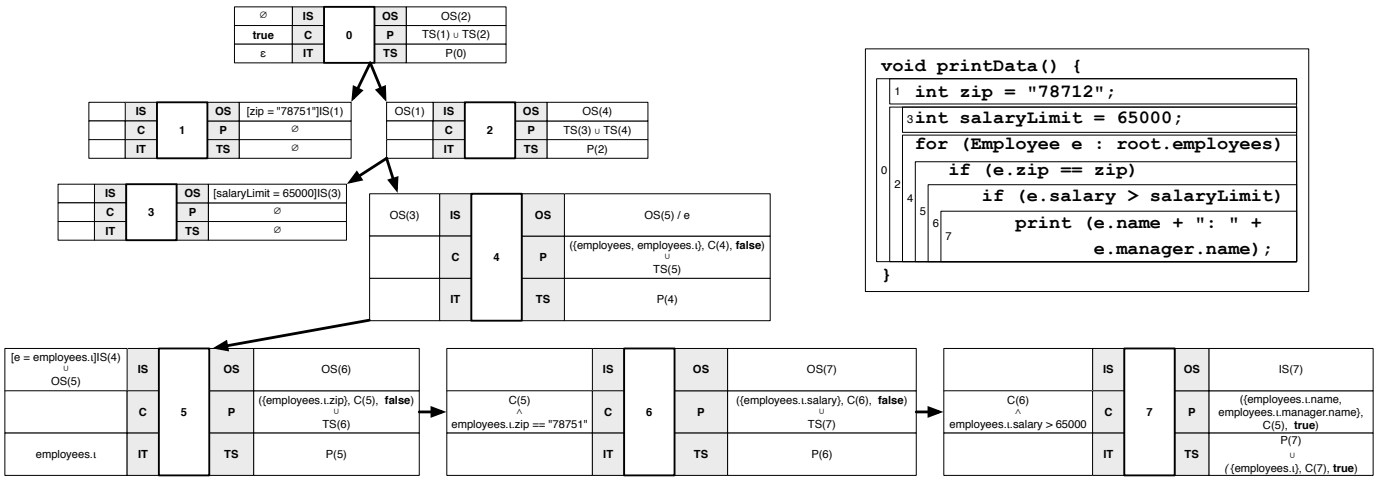


Figure 7: An illustrated example of the intraprocedural, path-based analysis. Each example statement is numbered, and each number corresponds to a node in the AST. Each node is a table whose middle column is the statement number and whose left and right columns contain values for synthesized and inherited attributes, respectively. If a node contains no value for an inherited attribute, then that attribute's value is the same as the parent node's value.

analysis must consider paths traversed from method parameters and from method return values.

We extend the definition of paths from Section 4.1 to include a path root. A *rooted path* is a four-tuple $(r, \overrightarrow{f}, c, d)$ where the root r can be **root**, a persistent method parameter, or a callsite return.

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

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

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

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, printIfOver in Figure 1 includes a condition that depends on the salaryLimit method parameter. We extend the $AbsOp$ domain to allow operations and conditions to refer to local variables (including parameters). 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 does not attach a condition that depends on the results of, for example, a virtual method call invoked after executing the query.

Conditions and Dependence Conditions present a problem for composing paths across procedure boundaries. If a callee path's condition references one of its parameters, then the caller must replace the parameter with its corresponding argument's abstract value. If the callee path's condition 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.

5.2 Interprocedural Path Analysis

Interprocedural query extraction extends the language and attribute grammar of Section 4 to include method declarations M, callsites $g_l(a_1, \ldots, a_n)$ and returns **return** e. The analysis assigns a unique label l to each callsite of a method g. The remainder of this section describes how the analysis computes values for declarations, call sites, and returns.

Return Statements and Method Declarations A **return** statement's attributes are computed from the attributes of the returned expression. A method M's traversal summary is the union of its statements' summaries. The method's abstract value is the union of all the abstract values for the method's **return** statements:

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

Callsites A callsite's traversal summary and abstract value depend on whether the analysis can predict the called

```
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       print (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       print (e.name);
17 }}
```

Figure 8: Method salaryInfo can pre-load data for printSalariesAbove and printSalariesBelow .

method's data traversals. If so, then the analysis can compose the caller's traversals with those of the callee's, in order to pre-load the callee's data.

If a callsite cannot be devirtualized, the called method must execute its own query by using its traversal summary and taking the actual method arguments as roots. We denote by $\tilde{g}_l(a_1, \ldots, a_n)$ a callsite that cannot be devirtualized. The callsite's traversal summary includes its arguments' summaries:

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

The callsite's abstract values is a new path that is rooted at the callsite's label:

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

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 pre-loads the data needed to support that method's traversals from those parameters. For example, the program in Figure 8 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.

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:

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

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

```
1  public void employeeInfo() {
2    for (Department d : root.departments) {
3      for (Employee emp : d.employees) {
4        Employee e = getEmpToNotify(emp,65000);
5        print (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 9: Traversal passes through getEmpToNotify back to employeeInfo.

by the arguments plus the traversals performed within g.

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

If method g is recursive, then this analysis diverges. We discuss how to ensure termination in Section 5.5.

A callee may return a persistent value that depends on its parameters, and the caller may traverse from the returned value. In this case, the caller's traversals *pass through* the callee. If the callee can be devirtualized, then the caller can pre-load its pass-through traversals.

Figure 9 contains an example of pass-through traversals. 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 analysis of method employeeInfo detects this pass-through path and generates a summary that includes the department field.

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

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

The abstract value domain is also extended by defining $\mathsf{AV}(\cdot)[R]$ 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.

The abstract value of a devirtualized callsite consists of the caller's pass-through paths, plus those values in the callee's abstract value not affected by pass-through traversals:

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

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

Figure 10: Callee can pre-load caller's data.

5.3 Dynamic Query Composition

Although a caller may not pre-load data for a virtual callsite, it is possible for the callee to dynamically pre-load some of its caller's data. For example, assume the analysis determines that method getHRDepartment in Figure 10 is virtual. Then the callsite at line two cannot pre-load its pass-through traversals.

Our program transformation modifies method calls and definitions so that the callee may preload pass-through traversals for the caller. Every method that returns a persistent value is statically 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 at runtime to generate a dynamic query TS' based on its own static traversal summary:

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

Recall that static query composition from caller to callee requires the caller to bind values for any parameters that appear in the callee's abstract value. Dynamic query composition from callee to caller similarly 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.

5.4 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 4.3.

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 two assumptions: 1) the add method makes no traversals of its own and 2) the program does not modify the user-created collection after reading it.

5.5 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

$$\mathsf{e.manager}^{+}.\mathsf{salary}$$

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**.

5.6 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 that represents 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.

6. Implementation

We implemented query extraction using JastAdd—an attribute-grammar-based compiler system for Java that enables program analyses to be written in a modular, declarative fashion [14]. 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, the mapping of persistent classes to database tables, and the location of the database. The output is a Java source program which can be compiled and run on a standard Java virtual machine.

6.1 JastAdd

JastAdd compiles circular reference attribute grammars into compilers. JastAdd includes a Java 1.5 compiler specification, which we extended to perform query extraction. JastAdd provides as part of the Java specification 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. The Java specification also includes an experimental devirtualization analysis.

6.2 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. A traversal summary for **root**, persistent parameters, and 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

```
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 11: An example of a method which accesses persistent data.

```
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 12: Code transformation for method in Figure 11.

code in Figure 11 is transformed to the code in Figure 12 which is simplified to omit type packages.

Callsites inside the method (e.g., lines 27–28 in in Figure 12) are transformed to use the version of the method that accepts additional traversal information.

6.3 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 query 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 allowing 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 supported by the analysis.

7. 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 program generated by query extraction sometimes loads more objects from the database than an equivalent hand-optimized program, because query extraction must statically overapproximate 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—indicate the quality of our prototype implementation. We present our implementation's performance for these metrics and conclude that our prototype performs well except in a few cases.

Experimental Configuration Our experimental configuration consists of a server that hosts the database and a client machine on which the benchmarks run. The machines are located on the same local network, and ping reports an average roundtrip time of about 250 microseconds. 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 client'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 nondeterministic, due to the random behaviors of the operating system and the 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 [11, 12] 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.

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 [22] 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.

7.1 TORPEDO

The TORPEDO [22] 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

	Method Declarations				Callsites			
Benchmark	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.

presenting results. The benchmark code is close to 900 lines, and the benchmark database contains 40 objects.

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 variable **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 interprocedural analysis. Table 1 lists how many methods and callsites the benchmark contains, how 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 about 33 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, because these metrics provide a more accurate picture of the behaviors of transparent, hand-optimized, and query-extracted programs.

Queries Executed Figure 13 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 include commits. Each use case requires a minimum of two queries because each use case executes a commit. For example, the hand-optimized and query-extracted TORPEDO versions execute two queries for the "Find All Auctions" use case: one query to load all the auc-

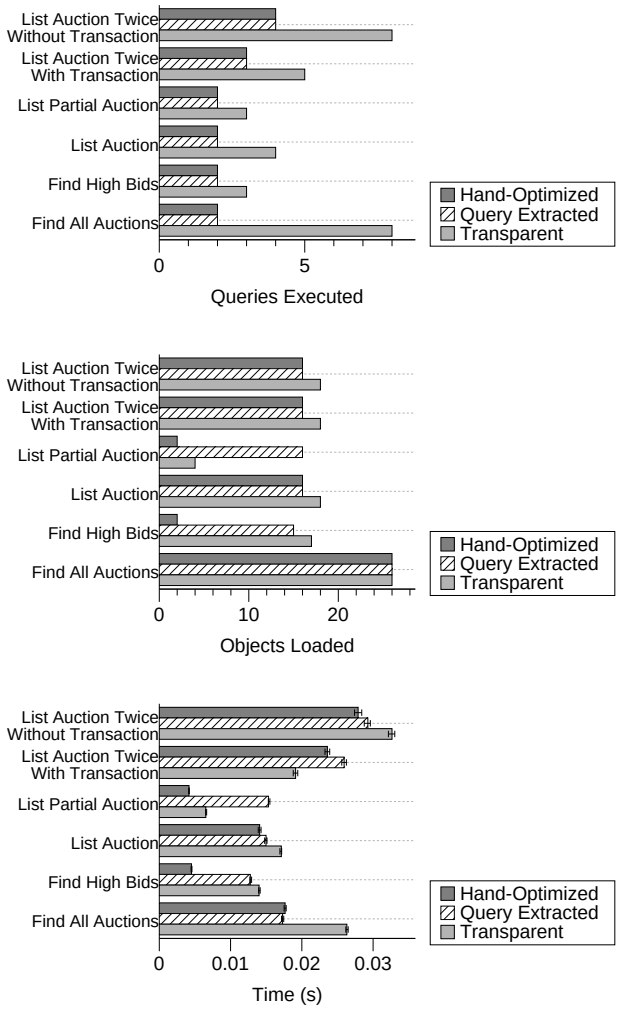


Figure 13: Number of queries executed, number of objects loaded, and execution time for the TORPEDO benchmark. Query extraction locates and executes the same number of queries as the hand-optimized version in all use cases, loads fewer objects than the transparent version in all but one use case, and outperforms the transparent version in many cases.

tions and related objects plus one commit. 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 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, and there is little difference in execution time among the three versions. The results show that our research-quality implementation is comparable to a hand-optimized program and outperforms the transparent version in all but two cases. The query-extracted version of "List Partial Auction" executes more complex queries (i.e., with more joins) than its transparent counterpart, so it takes about twice as much time to execute. The transparent version of "List Auction Twice With Transaction" takes less time than the other two versions because it takes advantage of caching. We found the overhead of run-time query composition to be negligible (around .004% of total execution time). We believe that more engineering effort would yield even better results.

7.2 OO7

The OO7 [5] benchmark 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 focuses on extensive traversals of hierarchical structures. However, the benchmark is widely used in the research community and presents some interesting challenges for query extraction.

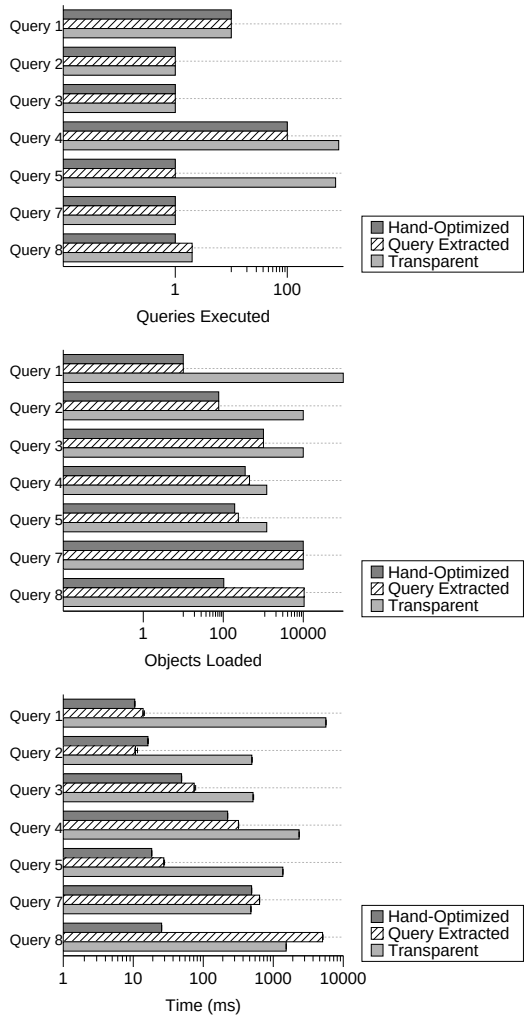
The OO7 specification defines three kinds of use cases: *queries*, *traversals*, and *structural modifications*. The queries perform read-only operations on the data. There are seven query use cases labeled Query 1 through Query 8 (Query 7 does not exist). The traversals scan the object graph and collect information. There are six traversal use cases, 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. Traversal 1, Traversal 2, Traversal 3, and Traversal 6 rely on recursion to scan the assembly hierarchy and part graphs. 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 about 41,000 objects. Our version of the OO7 code contains close to 1,300 lines of code.

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 query use cases contain hand-optimized, explicit queries. We created equivalent, transparent versions of these use cases. We then applied query extraction to the transparent use cases to generate versions that contain explicit queries. We compare the performance of all three versions. The traversal use cases are based on transparent persistence. We applied query extraction to these use cases. 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.

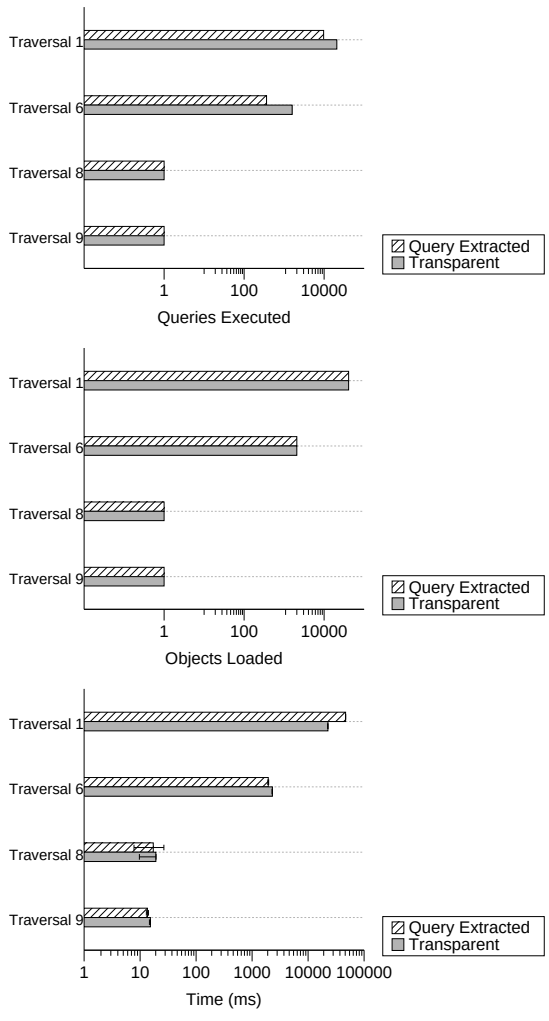
Table 1 lists the persistent characteristics of the methods in OO7. The query extraction analysis for OO7 took about 100 seconds.

OO7 does not specify which metrics to report, so we report number of queries executed, number of objects loaded, 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 6.3. We first performed query extraction with an unfolding depth of one. Figures 14a contains the results for the query use cases, and Figure 14b 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 the performance of hand-optimized, transparent, and query-extracted versions. The query-extracted version executes the same number of queries as the hand-optimized version for every use case except Query 8, which performs an ad-hoc join of two collections. Query extraction does not optimize these kinds of traversals. The hand-



(a) Performance for query use cases. In general, query extraction performs comparably to hand-optimized code and favorably to transparent persistence. Query extraction does not perform well for Query 8, because that use case contains a query condition that query extraction does not optimize.



(b) Performance for traversal use cases. Query extraction performs better than transparent persistence for Traversal 6 and worse than transparent persistence for Traversal 1. This difference occurs because Traversal 1 traverses a highly connected graph, and an HQL query cannot efficiently retrieve such a structure.

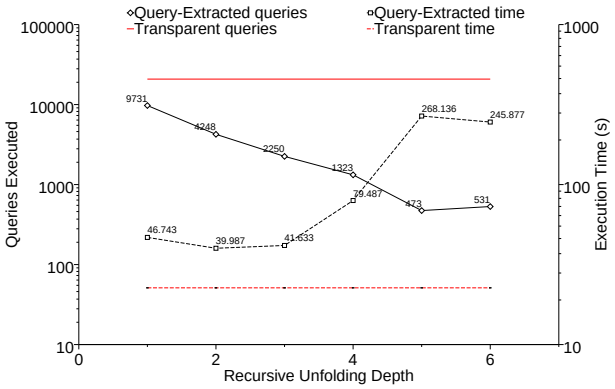
Figure 14: Queries executed, objects loaded, and execution time per OO7 query (a) and traversal (b) use case.

optimized version loads all the required objects with a single query. The query-extracted version performs one query for each collection, and loads too many objects because the use case's condition violates the *master-detail* restriction discussed in Section 4.3. The transparent version executes at least as many queries as the query-extracted version for all use cases.

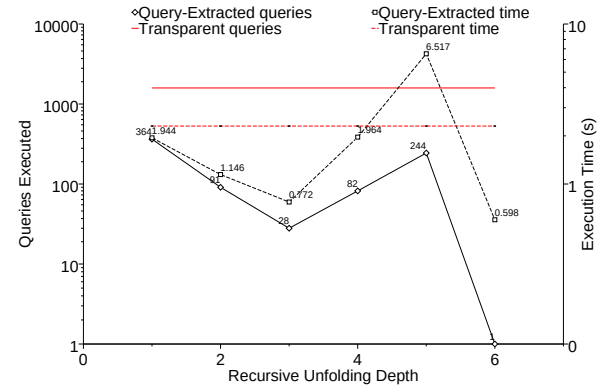
The query-extracted versions of Query 4 and Query 5 execute the same number of queries as the corresponding hand-optimized version, but load more objects. These extra objects are due to the fact that the transparent program from which the query-extracted version is generated traverses a relationship 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. The transparent version loads more objects than the other two versions for Query 4 and Query 5.

For all use cases except Query 8, the query-extracted version takes time comparable to the other two versions. 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 uses cases compares query-extracted performance



(a) Query extraction takes more time but executes fewer queries than transparent persistence for Traversal 1. The inverse relation between execution time and number of queries executed is due to the redundant data that a relational query must load to express traversal of a highly connected graph.



(b) Query extraction usually takes less time and executes fewer queries than transparent persistence for Traversal 6. This use case traverses a tree of height six.

Figure 15: Query extraction performance depends upon the structure of traversed data and upon unfolding depth.

against transparent persistence performance. Traversal 8 and Traversal 9 yield uninteresting comparisons for queries executed and objects loaded, because these use cases traverse a single object. Query extraction executes fewer queries than transparent persistence for Traversal 1 and Traversal 6. The transparent versions of these use cases load objects lazily, 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.

The execution time for both versions is comparable for Traversal 8 and Traversal 9. The transparent version performs better for Traversal 1; but the query-extracted version performs better for Traversal 6. Figure 15 illustrates this difference in more detail by varying the unfolding depth for these two use cases. Figure 15a 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 15b 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 five. This spike occurs because the data happens to have a complete tree structure of height six. Thus our implementation of recursion described in Section 6.3 is suboptimal for unfolding depths which are not a factor of six. In general, the optimal unfolding depth depends on the data characteristics.

8. 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 [31], Haskell/DB [19] and Links [9] 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 [8]. The Java Query Language (JQL) extends Java to support optimizable queries of in-memory objects [30].

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 Thiemann partition a sequential program executed at one location into semantically equivalent, independent, distributed processes [26]. Their approach provides software engineering benefits similar to ours, except for multi-tier applications.

The DBPL language [28] and it successor Tycoon [23] explored optimization of search and bulk operations within the framework of orthogonal persistence. Tycoon proposed integrating compiler optimization and database query optimization [10]. Queries that cross modular boundaries were optimized at runtime by dynamic compilation [27]. 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, 13, 16]. Because our work generates queries which could be used in object persistence architectures, the two techniques could be combined to achieve further performance benefits.

9. Conclusion

This paper presented 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 is preloaded by the caller; and conversely, the procedure preloads all the data needed by the call site from its return value. Procedure parameters are handled by devirtualization and query separation, while procedure results are handled by a novel combination of static analysis and dynamic query composition. While parameters are only preloaded if devirtualization succeeds, the dynamic composition always allows a callee to preload data for its caller. We presented a prototype Java compiler with query extraction including support for recursion query parameters, persistent parameters and return values. We evaluated the technique using the TORPEDO and OO7 benchmarks. This work demonstrates the feasibility of query extraction in a practical setting.

Acknowledgments

We gratefully acknowledge the anonymous OOPSLA 2007 and 2008 reviewers, PLDI 2008 reviewers, Ben Delaware, and Milind Kulkarni for their comments on the paper. We thank Kathryn McKinley, Mike Bond, and Jungwoo Ha for their advice on methodology.

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 cω. 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] 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.

[9] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects*, pages 266–296, Amsterdam, The Netherlands, November 2006.

[10] A. Gawecki 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.

[11] 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.

[12] J. Ha, M. Gustafsson, S. M. Blackburn, and K. S. McKinley. Microarchitectural characterization of production jvms and java workloads. Mar 2008.

[13] W.-S. Han, Y.-S. Moon, and K.-Y. Whang. PrefetchGuide: capturing navigational access patterns for prefetching in client/server object-oriented/object-relational dbmss. *Information Sciences*, 152(1):47–61, 2003.

[14] G. Hedin and T. Ekman. The JastAdd system – modular extensible compiler construction. *Science of Computer Programming*, 69:14–26, 2007.

[15] Hibernate reference documentation. `http://www.hibernate.org/hib_docs/v3/reference/en/html`, May 2005.

[16] 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.

[17] 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.

[18] D. E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.

[19] 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.

[20] E. Magnusson and G. Hedin. Circular reference attributed grammars — their evaluation and applications. *Sci. Comput. Program.*, 68(1):21–37, 2007.

[21] 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.

[22] 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.

[23] 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.

[24] Microsoft Corporation. The LINQ project. `msdn.microsoft.com/netframework/future/linq`.

[25] 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.

[26] 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.

[27] 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.

[28] J. W. Schmidt and F. Matthes. The DBPL project: advances in modular database programming. *Inf. Syst.*, 19(2):121–140, 1994.

[29] 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.

[30] 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.

[31] L. Wong. Kleisli, a functional query system. *J. Funct. Program.*, 10(1):19–56, 2000.

[32] M. Zand, V. Collins, and D. Caviness. A survey of current object-oriented databases. *SIGMIS Database*, 26(1):14–29, 1995.