Copyright
by
Benjamin Alan Wiedermann
2009

The Dissertation Committee for Benjamin Alan Wiedermann
certifies that this is the approved version of the following dissertation:

Integrating Programming Languages and Databases via Program Analysis and Language Design

Committee:

William R. Cook, Supervisor

Don Batory

Stephen M. Blackburn

Calvin Lin

Kathryn S. McKinley

Integrating Programming Languages and Databases via Program Analysis and Language Design

by

Benjamin Alan Wiedermann, B.A.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2009

Dedicated to the memory of my grandmother, Bobbie McClusky.

Acknowledgments

A nearly uncountable number of people have supported and guided me during my graduate career. This dissertation is the culmination of their support, these words mere hints at my gratitude. For their academic support and guidance, I am grateful to ...

... my advisor William Cook. William always seemed to know what would help me best. He was exacting but patient, enthusiastic and pragmatic. He gave me an excellent pair of training wheels, but he refused to let me keep them. For that, I am grateful.

... my academic brother Ali Ibrahim, who was my ally against impedance mismatch. The research in this dissertation benefitted enormously from his probing questions and smashingly good code.

... my former advisor Kathryn McKinley. Kathryn continued to support and advise me even after I "broke up with her". I am grateful to add my name to the long list of researchers who have benefitted from Kathryn's high standards, generous encouragement and selfless tutelage.

... my committee members William, Kathryn, Don Batory, Steve Blackburn and Calvin Lin. They each challenged me to seek clarity, to be precise and to provide context for my research.

... my graduate student colleagues and friends. Mike Bond and Milind Kulkarni spent thousands of meals, coffee breaks and social outings with me

and never once asked me to be quiet. They are heroes. I am grateful to have collaborated with Ben Hardekopf, Matt Taylor and Byeong-Cheol Lee. I am indebted to so many other people that I do not know how to list them. Please ask me, and I will gladly tell you all their names.

. . . the support staff at UT. Gloria Ramirez, Gem Naivar, Lydia Griffith, Katherine Utz and `gripe` made it easy to work at UT.

For their love and support, I am grateful to . . .

. . . my parents Janice Schonwetter, Mark Schonwetter, Bud Wiedermann and Pam Parker. From them, I learned to love learning. Everything else has followed.

. . . my extended family. The Parker-Wiedermanns, Bergmans, McCluskys, Tuckers and Schonwetters always welcomed me with loving arms, inviting homes and tons of smiles. They rarely asked me to fix their computers in return.

. . . my friends. Aaron Sporn, Andrea Moore and Tracey Fisher in particular have provided much-needed encouragement, escapes and laughs. I have been blessed to know many close friends. My generic, public gratitude belies the deep debt I owe them.

. . . No One Way Arts: Jeremy Gates, Andrea Moore, Jeni Rinner and Dan Stowell. They punched me in the art, just in time.

. . . my brother Mike Schonwetter. Candy and inspiration, still.

Integrating Programming Languages and Databases via Program Analysis and Language Design

Publication No. ____________

Benjamin Alan Wiedermann, Ph.D.
The University of Texas at Austin, 2009

Supervisor: William R. Cook

Researchers and practitioners alike have long sought to integrate programming languages and databases. Today's integration solutions focus on the data-types of the two domains, but today's programs lack *transparency*. A transparently persistent program operates over all objects in a uniform manner, regardless of whether those objects reside in memory or in a database. Transparency increases modularity and lowers the barrier of adoption in industry. Unfortunately, fully transparent programs perform so poorly that no one writes them. The goal of this dissertation is to increase the performance of these programs to make transparent persistence a viable programming paradigm.

This dissertation contributes two novel techniques that integrate programming languages and databases. Our first contribution—called *query extraction*—is based purely on program analysis. Query extraction analyzes

a transparent, object-oriented program that retrieves and filters collections of objects. Some of these objects may be persistent, in which case the program contains implicit queries of persistent data. Our interprocedural program analysis extracts these queries from the program, translates them to explicit queries, and transforms the transparent program into an equivalent one that contains the explicit queries. Query extraction enables programmers to write programs in a familiar, modular style and to rely on the compiler to transform their program into one that performs well.

Our second contribution—called *RBI-DB*$^{+}$—is an extension of a new programming language construct called a *batch block*. A batch block provides a syntactic barrier around transparent code. It also provides a latency guarantee: If the batch block compiles, then the code that appears in it requires only one client-server communication trip. Researchers previously have proposed batch blocks for databases. However, batch blocks cannot be modularized or composed, and database batch blocks do not permit programmers to modify persistent data. We extend database batch blocks to address these concerns and formalize the results.

Today's technologies integrate the data-types of programming languages and databases, but they discourage programmers from using procedural abstraction. Our contributions restore procedural abstraction's use in enterprise applications, without sacrificing performance. We argue that industry should combine our contributions with data-type integration. The result would be a robust, practical integration of programming languages and databases.

Table of Contents

Acknowledgments **v**

Abstract **vii**

List of Tables **xii**

List of Figures **xiii**

Chapter 1. Introduction **1**

Chapter 2. Background **10**

2.1 Enterprise Application Operations and Desiderata 10
2.2 Explicit, String-Based Queries 12
2.3 Explicit, First-Class Queries 13
2.3.1 Metaprogramming-Based Queries 14
2.3.2 Comprehension-Based Queries 14
2.3.3 Syntax-Based Queries 15
2.4 Orthogonal Persistence 16
2.4.1 The Principles of Orthogonal Persistence 16
2.5 Object Relational Mapping 19
2.6 Transparent Persistence 21

Chapter 3. Query Extraction:
Transparent Persistence from Static Analysis **22**

3.1 Introduction 22
3.2 Transparent Persistence vs. Explicit Queries 23
3.3 Overview of Query Extraction 28
3.4 Intraprocedural Query Extraction 31
3.4.1 Abstract Values 32

3.4.2 Intraprocedural Path Analysis 34
3.5 Interprocedural Query Extraction 43
3.5.1 Abstract Values 45
3.5.2 Interprocedural Path Analysis 46
3.5.3 Dynamic Query Composition 50
3.5.4 Query Extraction and Object-Oriented Programming 52
3.5.5 Recursion 53
3.5.6 Soundness 53
3.6 Implementation 54
3.6.1 JastAdd 55
3.6.2 Code transformation 55
3.6.3 Query Translation 56
3.7 Evaluation 59
3.7.1 TORPEDO 61
3.7.2 OO7 65
3.8 Related Work 73
3.9 Conclusions and Future Work 79

Chapter 4. RBI-DB$^+$: Transparent Persistence from Programming Language Design 82

4.1 Introduction 82
4.2 Remote Batch Invocation (RBI) 84
4.3 A Formal Description of RBI 90
4.3.1 An RBI Kernel Language 91
4.3.2 The Remote Intermediate Language (RIL) 91
4.3.3 Program Analysis 92
4.3.4 Reforestation 99
4.3.5 Program Transformation 102
4.4 RBI for Databases (RBI-DB) 105
4.5 Shortcomings of RBI-DB 105
4.6 Batch Methods 106
4.7 An Extended Kernel Language 110

4.8 Program Analysis 112
4.8.1 Location Analysis 114
4.8.2 Dependence Analysis 115
4.8.3 Validity Analysis 116
4.9 Program Transformation 118
4.10 RBI-DB$^+$'s Database Client 118
4.11 Extending RBI-DB$^+$ 123
4.12 Prototype Implementation 126
4.13 Related Work 131
4.14 Conclusions and Future Work 136

Chapter 5. Conclusions and Future Work 138

Appendices 140

Appendix A. Soundness Result for a Subset of Query Extraction 141

A.1 Values and Concrete Semantics 141
A.2 Conditional Paths, Abstract Values, and Abstract Semantics 144
A.3 Proof of Soundness 148

Bibliography 154

Vita 171

List of Tables

3.1 Number of (possibly recursive) methods/callsites in TORPEDO and OO7 across which query extraction statically extracts a query. 63

List of Figures

1.1 A program that operates over transient data. 2

1.2 A program equivalent to Figure 1.1 that operates over persistent data. 3

1.3 A program equivalent to Figure 1.1 that operates over persistent data and uses first-class queries with object-relational mapping. 6

3.1 Procedures and transparent persistence. 24

3.2 Query execution using Hibernate. 25

3.3 Query and load options in LINQ. 26

3.4 Creating results with LINQ. 27

3.5 Abstract values. 32

3.6 Synthesized attributes for intraprocedural traversal summaries. 34

3.7 Inherited attributes for intraprocedural traversal summaries. . 36

3.8 Traversal summary computation. 39

3.9 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. 41

3.10 Method `salaryInfo` can pre-load data for `printSalariesAbove` and `printSalariesBelow`. 48

3.11 Traversal passes through `getEmpToNotify` back to `employeeInfo`. 49

3.12 Callee can pre-load caller's data. 51

3.13 An example of a method which accesses persistent data. . . . 56

3.14 Code transformation for the method in Figure 3.13. 57

3.15 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. 62
3.16 Number of queries executed, objects loaded, and execution time per OO7 query use case. 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 filter that is not a query condition. 68
3.17 Number of queries executed, objects loaded, and execution time per OO7 traversal use case. 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. 69
3.18 Query extraction performance depends upon the structure of traversed data and upon unfolding depth. 72

4.1 An example batch block in a Java-like language. The block mixes local and remote code. The RBI compiler is responsible for partitioning the batch block into separate remote and local computations. 87
4.2 Remote and local partitions for the example in Figure 4.1. The boxes indicate the values that must be transferred from the server for use in the client. 88
4.3 Abstract syntax of RBI's kernel language. 90
4.4 Abstract syntax of RBI's Remote Intermediate Language (RIL). The boxes indicate additions to the kernel language of Figure 4.3. 92
4.5 Inherited attributes and semantic functions for the RBI location analysis. 93
4.6 Location analysis for RBI batch blocks. 95
4.7 Inherited attributes for the RBI validity analysis. 97
4.8 Validity analysis for RBI batch blocks. 98
4.9 Program transformation that labels remote expressions. The transformation is top-down, so it labels the largest remote subexpressions. 101
4.10 Partitioning and reforesting a batch block. 104

4.11 Batches cannot be composed. Methods salaryTest and departmentTest each require $n + 1$ round trips, where n is the number of elements in the remote collection. To achieve one round trip per method, the programmer would need to inline method process manually. 107
4.12 Virtual method dispatch and batching. The compiler can make only a lexical guarantee about batch blocks, because it cannot determine the run-time communication properties of an invoked method. 108
4.13 Abstract syntax of RBI-DB$^+$'s kernel language. This language is a superset of RBI's kernel language. The boxes indicate the language extensions. 111
4.14 Inherited attribute and semantic functions for the RBI-DB$^+$ location analysis. 112
4.15 Location analysis for RBI-DB$^+$ batch blocks. 113
4.16 Abstract syntax of an HQL-like query language. 119
4.17 Inherited attribute and semantic functions for the RBI-DB$^+$ HQL generation. 120
4.18 Projections for RBI-DB$^+$. 121
4.19 Operations for RBI-DB$^+$. 122
4.20 RBI-DB$^+$'s conservative analysis prevents the compiler from partitioning this example. The programmer should communicate that the print and getValue methods do not modify their receivers. 124
4.21 Example Java methods that contain batch blocks. 126
4.22 The reforested version of method capSalary in Figure 4.21. . 127
4.23 Expression that creates an RIL syntax tree for the remote computation of the example in Figure 4.21. 129
4.24 The remote and local computations for batch method test from Figure 4.21 130

A.1 Syntax of a persistent data kernel language. 142
A.2 Typical operational semantics, extended to collect used-sets. . 143
A.3 Abstract interpretation with paths and conditions. 145
A.4 Conditional record loading. 148

Chapter 1

Introduction

All our major financial, health, and government enterprises rely on systems that manipulate large datasets. These *enterprise applications* are notoriously difficult to design, debug, and maintain [Fow03]. Programmers would prefer to use all the modular abstractions afforded them by today's programming languages. Unfortunately, performance concerns force programmers to design enterprise applications that are inflexible and tightly coupled [Bla05]. Neither the academic research community nor industry has been able to solve this problem, despite three decades of effort.

The problem has no easy solution, no silver bullet. It seems likely that we will have to combine several practical and theoretical ideas to make progress. This dissertation contributes a new approach, namely that program analysis and programming language design techniques can help programmers develop enterprise applications that are both modular *and* efficient.

Our approach is motivated by an old idea: Programmers should not have distinguish between *persistent* and *transient* data [AM95]. Transient data exists only in memory and does not outlast program execution. Persistent data outlasts program execution and must be maintained in a persistent store.

```
1 void report(Collection<Employee> employees, int limit) {
2   for (Employee e : employees)
3     if (e.getSalary() > limit)
4       printEmployee(e);
5 }
6
7 void printEmployee(Employee e) {
8   print(e.getName() + ": " + e.getDepartment().getName());
9 }
```

Figure 1.1: A program that operates over transient data.

Today's programmers write programs over persistent data differently than they do over transient data. Figures 1.1 and 1.2 contain examples in a Java-like language. Both programs iterate over a collection of objects that correspond to a set of employees. If an employee's salary is greater than a given limit, the program prints that employee's name and department.

The example in Figure 1.1 operates over transient data. It contains two methods: `report` and `printEmployee`. The former calls the latter to display information about each employee. Because it uses a separate method to display information, this program is easy to modify. If the programmer wanted to print an employee's salary instead of the employee's department, she would only need to modify `printEmployee`.

The example in Figure 1.2 operates over persistent data. Method `report` uses a Call Level Interface (CLI) to retrieve its data from a database. Lines 3–6 create a parameterized query string that describes the needed data. Line 9 sets the query parameter's value. Line 12 executes the query against

```
1  void report(Connection conn, int limit) {
2    // create the query
3    PreparedStatement query = conn.prepareStatement(
4      "select Employee.name, Department.name
5       from Employees as e, Departments as d
6     where e.salary > ?");
7
8    // set the parameters
9    stmt.setInt(1, limit);
10
11   // execute the query
12   ResultSet rs = query.executeQuery();
13   while (rs.next())
14       printEmployee(rs.getString(1), rs.getString(2));
15 }
16
17 void printEmployee(String empName, String deptName) {
18   print(empName + ": " + deptName);
19 }
```

Figure 1.2: A program equivalent to Figure 1.1 that operates over persistent data.

the database and retrieves the results. Lines 13–15 iterate over the results to display them.

When a programmer writes a program like the one in Figure 1.2, she throws out all her best programming practices. Because the programmer must declare the program's data needs up-front, the program loses modularity. If the programmer wanted to modify this program to print an employee's salary, she would need to make three changes: (1) the signature and body of `printEmployee` must take and print an integer, (2) the call to `printEmployee` in line 14 must retrieve and pass an integer, and (3) the query in lines 4–6 should remove the unnecessary request for department and include a request for salary.

The program in Figure 1.2 thwarts the compiler's attempts to provide static type-safety guarantees. Because the programmer embeds the program's data needs in a string, the compiler cannot verify that the search or the program's use of the results are error-free. These errors are delayed until runtime, which makes it more difficult for the programmer to identify and fix all the program's errors.

Programmers forsake type safety and reduce the modularity of their programs for one reason: performance. Network latency so dominates program performance that a programmer must reorganize code manually to communicate with the database as little as possible. The database stores so much data that a programmer must provide it with as much information as possible, to enable the database to organize and execute an efficient search.

The tradeoff between performance and modular, type-safe enterprise applications has bedeviled us for decades [Atk78, Mai87, Bla05, CI07]. In 1978, Malcolm Atkinson wrote:

> Present database and programming language systems do not permit programming methodologies which are deemed important in producing reliable and well-engineered software [Atk78].

This disconnect is called *impedance mismatch* [Mai87]. During the 1980s and 1990s, Atkinson and others tried to overcome impedance mismatch by designing and implementing *orthogonally persistent* systems. An orthogonally persistent system is a monolithic combination of programming language and persistence technologies which treats persistent and transient data in a uniform manner.

Orthogonally persistent systems did not catch on in industry, and orthogonal persistence today is a dormant line of research. Its contributions, however, resound throughout today's technologies in the form of *object-relational mapping* (ORM) and *first-class queries*. These technologies integrate programming language and database data types, and they enable programmers to write type-safe queries. Microsoft's C# has adopted these technologies, which indicates that they may become dominant programming techniques in the near future [BH07].

Unfortunately, object-relational mapping and first-class queries do not overcome the impedance mismatch problem. In particular, these technologies

```
1  void reportZip(DataContext db, int limit) {
2    // preload specification
3    DataLoadOptions dlo = new DataLoadOptions();
4    dlo.LoadWith<Employee>(e => e.department);
5    db.LoadOptions = dlo;
6
7    // create the query
8    var employees = from e in db.Employee
9                    where e.salary > limit
10                   select e;
11
12   // execute the query
13   foreach (Employee e in employees)
14       printEmployee(e);
15 }
16
17 void printEmployee(Employee e) {
18   print(e.getName() + ": " + e.getDepartment().getName());
19 }
```

Figure 1.3: A program equivalent to Figure 1.1 that operates over persistent data and uses first-class queries with object-relational mapping.

still force programmers to sacrifice modularity for performance. Figure 1.3 contains an example. Lines 8–10 contain an explicit, first-class query. The query's syntax is part of the programming language, and the compiler can use object-relational mapping information to type-check the program statically.

The preload specification in lines 3–5 increases the program's performance at the expense of its modularity. The preload specification is an explicit prefetch of data related to the query. Without this specification, the program could require $n+1$ database communications, where n is the number of employees. With the preload specification, the program requires only one database communication. Because the program contains these explicit constructs, it also suffers the same lack of modularity as the program in Figure 1.2.

Atkinson's 1978 criticism rings true in 2009. The problem is that we have adopted only one kind of integration from orthogonal persistence, namely data-type integration. We have not adopted the idea that programs should operate over persistent and transient data in a uniform manner.

When David Maier first used the term impedance mismatch to describe the integration problem, he said:

> Whatever the database programming model, it must allow complex, data-intensive operations to be **picked out of programs** for execution by the storage manager, rather than forcing a record or object-at-a-time interface [Mai87]. (emphasis added)

Today's technologies require the programmer to pick data-intensive op-

erations out by hand and to restructure a program around these operations. This solution reduces modularity, even in small programs. If we are to overcome impedance mismatch, we must design a solution that permits programmers to write modular code. The challenge in crafting such a solution is to ensure that modularity does not inhibit performance. These goals motivate our contributions.

The first contribution is called *query extraction*. Query extraction is an interprocedural static analysis and program transformation that takes as input a program like the one in Figure 1.1 and produces as output a program like the one in Figure 1.2. With query extraction, the programmer can rely on the compiler to pick out the database operations automatically. Modularity need not inhibit performance.

Chapter 3 describes query extraction in detail. We formalize the interprocedural static analysis. The analysis handles virtual method calls by introducing additional queries where necessary and generating code to compose analysis results at runtime. We describe our Java-based implementation that converts analysis results to queries that target the popular Hibernate persistence system [Cen04]. Our implementation handles recursive data traversals by generating code to unfold recursion at runtime. We evaluate our implementation using the TORPEDO [Mar05] and OO7 [CDKN94] benchmarks.

The second contribution is called *RBI-DB$^+$*. RBI-DB$^+$ extends the ideas of query extraction to a more general, client-server paradigm. With RBI-DB$^+$, the programmer intermingles client and server code. The compiler

picks out the server code, generates a program that sends the server code away for execution, and retrieves the results for use in the client code. The client-server computation is bounded by a programmer-defined *batch block*—a programming construct that makes latency explicit.

Chapter 4 describes RBI-DB$^+$ in detail. It begins with an overview of RBI—a programming abstraction in which the programmer delimits the scope of generic client-server interactions in exchange for a compiler guarantee about a program's latency behavior [TCJ09, IJCT09, IJI$^+$09, CTIW09, Ibr09]. Our contribution is to extend RBI in several ways. We provide the first formal description of its semantics. We extend RBI with batch methods, which programmers use to create batches that cross method boundaries. We present a new program analysis for RBI, based on dependences. The analysis clarifies RBI's semantics and makes it easier for future researchers to extend RBI. We implement RBI-DB$^+$ by extending a Java compiler. The implementation includes a program transformation that uses runtime inlining to accommodate virtual method calls, and it optimizes modifications of persistent data.

Query extraction and RBI-DB$^+$ contribute novel solutions for impedance mismatch. Our contributions are general apply to any task that intermingles computations from two domains. Our focus, however, is integrating object-oriented programming languages and relational databases. Industry has adopted previous research that integrates the data-types of these two domains. Chapter 5 describes how industry can achieve a more robust integration by combining today's technology with the techniques in this dissertation.

Chapter 2

Background

This chapter provides context for our work. We describe the kinds of database operations performed by enterprise applications. We also list some desirable properties of enterprise applications. We place our contributions in context by examining the history of programming language and database integration research. For each line of research, we evaluate the database operations it enables and the properties its programs exhibit.

2.1 Enterprise Application Operations and Desiderata

Enterprise applications perform *retrievals*, *filters*, *aggregations*, and *modifications* of persistent collections. Retrievals consist of selecting database records, combining or joining records, and projecting elements of the records for consumption by the enterprise application. A filter restricts a retrieval to those records which satisfy a specified condition. An aggregation is any operation which summarizes or orders a collection of records. Aggregations include common SQL aggregations like `sum`, as well as existential queries and grouping and sorting operations. An enterprise application may modify persistent data by adding or deleting records, or by changing the values of a record's elements.

Modifications can be made one-by-one or in bulk. We do not include modifications to the persistent data's *schema*, or description, as part of the enterprise application's responsibility.

Enterprise applications ideally should exhibit several desirable properties. Enterprise applications should be *safe*. One kind of safety is type safety, which means that programmers receive type errors at compile-time rather than run-time. Safety also means security. Programmers should not write programs that can be exploited easily by adversarial agents. Enterprise applications should be *modular*, so that programmers may employ good programming techniques to create maintainable and extensible programs. Programmers should be able to parameterize and compose operations. Enterprise applications should be *concise*, meaning that they should perform as few expensive network communications as possible. In general, it is important that both the general-purpose operations and the database operations be amenable to optimization.

Previous integration solutions were either garrulous instead of concise, or they required industry to adopt significant new technologies. Today's solutions all lack modularity. Some lack safety, while others hinder the optimization of certain database operations. In short, today's solutions are incomplete.

We now examine several lines of integration research in detail and discuss their support for database operations and viability as software systems.

2.2 Explicit, String-Based Queries

Programmers historically have specified queries explicitly, by embedding in their programs strings that contain a declarative description of data operations. The predominant mechanism for doing so—and the one which still prevails—is the *call-level interface (CLI)* [ISO03]. The CLI is a library-level integration solution. The programmer invokes a library function and passes a string that contains syntax for a given query language. The called function executes the query string against the database and returns the result. CLI enables the programmer to specify any operation supported by a given query language.

CLI is a concise solution, but it is neither modular nor safe. CLI is concise, because the programmer communicates an operation using one round-trip communication. Because this communication contains a complete description of the operation, the Database Management System (DBMS) can perform an efficient data search [Cha98].

CLI is not modular, because it requires the programmer to separate data retrieval from data use. The resulting program is more difficult to modify than a program that does not distinguish data retrieval from it use [Bla05]. The program also lacks modularity because some of its functionality is contained in strings, which complicates manual and automatic refactoring [TTS+08].

CLI is not safe, because the embedded SQL strings are not subject to static type checking. If a SQL string contains a type error, that error will

manifest itself at runtime. CLI does provide *parameterized queries*, which are a way to parameterize database operations. Parameterizing queries based on runtime values is cumbersome at best and a security risk at worse.

Researchers have increased the safety and modularity of CLI programs using static analysis. [GSD04, WGSD07, TTS+08]. This line of research provides static analyses that type-check embedded SQL strings. Tatlock et al. further provide a way to enable refactoring of some embedded SQL strings. These solutions provide more safety and modularity for legacy code. They do not enable the programmer to write modular enterprise applications from scratch.

CLI's lack of type safety inspired a new line of research. This research endeavors to provide safer queries by making them first-class members of a programming language.

2.3 Explicit, First-Class Queries

Future enterprise applications probably will be written using languages such as C# that provide queries as first-class members of the language [BH07]. First-class queries are full-fledged programming constructs, akin to classes or `for` loops. With first-class queries, programmers write safe, concise programs for a variety of database operations. However, because the programmer must separate data retrieval from use, this approach inhibits modularity. First-class queries appear in programming languages in one of three forms: *metaprogramming-based*, *comprehension-based*, and *syntax-based*.

2.3.1 Metaprogramming-Based Queries

The metaprogramming-based approach adds first-class queries to an existing language via the language's extensible and reflective mechanisms. *Safe Query Objects* provide a collection of Java classes that implement type-safe retrieval, filtering, and aggregation of persistent data [CR05]. Safe Query Objects use compile-time reflection to translate a programmer-specified query object into a database query operation. AraRat provides similar capabilities for C++, using template rewriting [GL07].

This line of research is attractive, because it enables programmers to write safe enterprise applications without adopting a new language. However, because the programmer must write explicit queries, the resulting programs lack some modularity.

2.3.2 Comprehension-Based Queries

The comprehension-based approach began with the premise that comprehension notation is an appropriate way to express queries [Tri92, BLS+94]. This premise gave rise to languages like Staple [McN86], Haskell/DB [LM99], and Kleisli [Won00]. All are functional languages that use a form of list or set comprehensions to specify database operations. These languages support a varying degree of database operations. For example, neither Haskell/DB nor Kleisli support data modifications. Haskell/DB and Kleisli translate comprehensions to an equivalent form in a query language, which enables programmers to write safe, concise programs.

Kleisli's query compilation can be seen in two other languages—Links [CLWY06] and Hop [SGL06]. These high-level programming languages are used to describe web applications, which often include database operations. The language environments also partition the application into client- and server-specific computations. In this case, the client is a web-browser.

All comprehension-based approaches enable comprehensions—which may be used on persistent as well as transient data—to be translated into database operations, when appropriate. Researchers have also applied this technique for use in querying in-memory objects [WPN06, WPN08].

2.3.3 Syntax-Based Queries

First-class queries may also be added to an existing language by extending the language's syntax and semantics to subsume a database query language like SQL. SQLJ is a version of Java wherein a programmer embeds SQL statements in a Java program [ANS99]. SQLJ programmers benefit from static typechecking, so their programs are safer. Because SQLJ programs contain explicit queries, they are concise but lack modularity.

Microsoft recently extended C# to include **L**anguage **IN**tegrated **Q**ueries, or LINQ [BH07]. LINQ enables C# programmers to write first-class queries and to iterate over the results. The query syntax provides a unified mechanism for high-level querying of database records, as well as XML data and in-memory objects. LINQ is the culmination of a line of research that seeks to integrate an object-oriented programming language with relational

concepts [Bie03, BMS05, MEW08]. LINQ programmers can express the full range of database operations, and LINQ programs are safe and concise. However, LINQ programs lack some modularity, because programmers must write explicit queries.

From the programmer's view, first-class queries provide a more complete integration solution than Wasserman et al.'s static analysis of query strings. However they require the programmer to write explicit queries, which reduces program modularity. By contrast, *implicit queries* encourage modular programs. Implicit queries are a by-product of orthogonally persistent systems.

2.4 Orthogonal Persistence

Orthogonal persistence is a language design philosophy which states that persistence should be transparent and independent of data type [AM95]. This philosophy was developed over two decades, culminating in the design and implementation of several orthogonally persistent languages and systems.

2.4.1 The Principles of Orthogonal Persistence

Orthogonally persistent systems exhibit three properties: *persistence independence*, *data type orthogonality*, and *persistence identification* [AM95].

Persistence Independence Programs should manipulate data in a manner that is independent of that data's persistence. The programmer need not

write code to transfer data to and from the persistent store. Instead, the runtime system automates data transfer. *Efficient* automatic data transfer is the responsibility of the system designers [AM95]. Some researchers have referred to this property as *transparency* [MH96]. It is arguably the most important property of an orthogonally persistent system [AM95].

Most orthogonally persistent systems implement persistence independence via *object faulting* [HM90]. An object-faulting mechanism retrieves and transforms persistent data on demand, making the data available in memory. This operation can incur great overhead because it requires *residency checks*—runtime tests to determine if persistent data has been loaded into memory—and because it performs many round-trip communications [Hos96]. This *record-at-a-time* retrieval behavior also limits performance because it inhibits query optimization [Mai87]. Most of the effort to optimize orthogonal persistence has focused on making object faulting efficient, a topic covered in-depth in Section 3.8.

Data Type Orthogonality Any instance of any type may persist for any length of time. In other words, data type and data lifetime are completely decoupled. Some researchers have referred to this property as simply *orthogonality* [MH96].

Orthogonality may require the runtime to translate persistent store values to programming language values during object faulting. The task of object faulting thus can be made easier by using an object-oriented persistent

store, such as an *object-oriented database management system (OODBMS)*. An OODBMS provides storage, recovery, concurrency, and query capabilities for object-oriented data. Because an OODBMS stores objects, it also models object-oriented concepts like encapsulation, types/classes, subtyping/inheritance, and overloading [ABD+89, Dem92]. The OODBMS model has been standardized [BEJ+00], and object-oriented databases occupy a niche role in modern industrial systems [CD96, Lea00, Wei07].

Most orthogonally persistent systems compromise data type orthogonality by restricting which kinds of values can be persistent [AM95]. Concurrency constructs represent a particular problem for data type orthogonality. [BZ99, MBMZ01, Atk01].

Persistence Identification A system determines data persistence in a manner that is independent of programmer annotations, the type system, etc. Atkinson and Morrison offer *persistence by reachability* as the only completely orthogonal mechanism for persistence identification [AM95].

Persistence by reachability is analogous to garbage collection. In a garbage-collected language, a heap-allocated value persists roughly as long as that value can be reached from a set of defined *roots*—usually global, register, and stack values [JL96]. For persistence reachability, the root set can be a special variable that represents the persistent store. One implementation of persistent Java defined the roots to be the set of class variables [MBMZ01]. Most modern systems employ type-based identification, so they do not exhibit

this property.

The principles of pure orthogonal persistence may have been too ambitious for practical systems. Most work on pure orthogonal persistence stopped around 2000, perhaps because of lack of funding and industrial disinterest. The modern-day descendant of orthogonally persistence is object relational mapping.

2.5 Object Relational Mapping

Object Relational Mapping (ORM) is a technique for describing an isomorphism between an object-oriented class hierarchy and a relational schema. ORM imposes an object graph over a database's relational tables. The graph can be described by a UML class diagram [Sof00], or equivalently, an Entity Relationship (ER) model [Che76].

ORM was developed in industry and enjoys wide-spread use in systems that combine an object-oriented, general-purpose programming language with a relational database management system [CD96, New06]. Industrial ORM tools—like TopLink [DSP03], Hibernate [Hib05], and EJB [MH98]—are the modern-day ancestors of Orthogonal Persistence [CD96].

The three principles of Orthogonal Persistence manifest themselves in ORM, albeit in more practical forms. To illustrate this point, we discuss the three principles in the context of Hibernate [Hib05]. Hibernate provides the most support for persistence identification. The identification is type-

based, rather than reachability-based. The Hibernate programmer identifies persistent data by writing a *mapping file* which describes the isomorphism between Java classes and their corresponding relational tables.

Hibernate does not place many restrictions on the kinds of data that may be persistent, so it provides a good deal of data-type orthogonality. In practice, Hibernate users persist Plain Old Java Objects (POJOs) and do not persist more complicated constructs likes threads [Hib05].

Hibernate provides two forms of persistence independence: data transfer and transparency. Data transfer is semi-independent from programs, because Hibernate programmers use a configuration file to specify how to connect to a database. The Hibernate system acts as middleware and marshals data between the database tables and application objects. Hibernate provides transparency via an object faulting mechanism.

Because ORM tools supply object faulting, they have the potential to achieve more transparency than other widely used industrial integration techniques like first-class queries. In practice, however, object faulting performs so poorly that ORM tools must provide an explicit query language [New06, CI07]. The explicit query language acts as an escape valve that enables programmers to write concise programs. Hibernate provides a string-based explicit query language called the Hibernate Query Language (HQL). It also provides first-class query constructs akin to the metaprogramming-based technologies described in Section 2.3.1.

Object Relational Mapping tools seem like a promising integration technique. They support all forms of database operations and ORM-based systems can exhibit a degree of safety. In reality, programmers who use ORM tools experience a worst-of-both-worlds effect [New06]. An ORM system can be either modular or concise, but not both.

Nevertheless, we believe that ORM tools have an important role to play in future integrated systems. ORM is best used for persistence identification. The performance boosts they provide due to caching and customizable loading strategies is also beneficial. We believe better integration techniques will result from placing a layer of abstraction over ORM. Indeed, all the first-class query techniques from Section 2.3 rely on some form of ORM to identify persistent data. What these systems lack—the void we propose to address in this dissertation—is transparency.

2.6 Transparent Persistence

A program exhibits *transparent persistence* if it operates uniformly over persistent and transient data. We use this term to distinguish orthogonally persistent programs from the programs to which our techniques apply. A transparently persistent program may use any technique to identify persistent data, and it need not exhibit orthogonality. All programs that do not use persistent data vacuously exhibit transparent persistence. The next chapter describes how query extraction permits a transparent programming style without sacrificing program performance.

Chapter 3

Query Extraction: Transparent Persistence from Static Analysis[1]

3.1 Introduction

In this chapter, we develop a technique for extracting queries from programs that use traversals to access persistent data. These programs are safe and modular, but not concise. We show how the compiler can improve their performance by applying program analysis and transformation. This technique is called *query extraction*. Query extraction permits programmers to write modular enterprise applications, without sacrificing performance. The contributions of this chapter 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 [Cen04].

[1] This chapter is published research [WIC08].

- A practical approach to recursive data traversals that unfolds the recursion in stages of finite depth.

- An evaluation of the system using the TORPEDO [Mar05] and OO7 [CDKN94] benchmarks.

The current implementation demonstrates the feasibility of this approach, without requiring changes to the Java language. Currently only operations that retrieve persistent data are supported. Query extraction does not support modifications to persistent values, nor does it support aggregation. Optimizing modifications requires knowledge of a query's scope. We discuss this issue in Section 3.9, and we contribute a solution that addresses this issue in Chapter 4. Aggregation is best expressed using first-class queries, as in LINQ [BH07]. It is important to stress that the techniques developed here can also be used in conjunction with queries. The best solution may be a combination of queries to specify aggregation, and query extraction to specify retrieval. We expand on this idea in Chapter 5.

3.2 Transparent Persistence vs. Explicit Queries

In Chapter 1, we described how programs operate differently over persistent and transient data by comparing a transparently persistent program to one that uses explicit queries. In this section, we expand on that example to motivate our work on query extraction.

The Java program in Figure 3.1 is a transparently persistent program

```
1 class Client { ...
2   void reportZip(DataAccess db, int zip) {
3     for (Employee e : db.getEmployees())
4       if (e.zip == zip)
5         printIfOver(e, 65000);
6   }
7   void printIfOver(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 3.1: Procedures and transparent persistence.

that 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.

```
1  void reportZip(DataAccess db, int zip) {
2    Query q = db.createQuery( // create the 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 3.2: Query execution using Hibernate.

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 3.2 is a hand-optimized rewrite of Figure 3.1 that uses Hibernate, an object-relational mapping tool, and its query language HQL [Hib05] 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 3.1 are not needed in Figure 3.2 because the query's `where` clause ensures the query only returns employees for which the tests are true.

```
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 3.3: Query and load options in LINQ.

Although the programs in Figure 3.1 and Figure 3.2 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 [Cha98]. Despite its performance benefits, there are several well-known drawbacks to the Hibernate version: The compiler does not check the query strings and parameters for syntax or type safety, and passing parameters is awkward.

These problems have been fixed by more recent query mechanisms, including LINQ [BH07] and Safe Query Objects [CR05]. Figure 3.3 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,

```
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 3.4: Creating results with LINQ.

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 3.4 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 3.1 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 chapter presents *query extraction*, a technique that can be used to infer queries from procedural programs. The goal is to analyze the program in Figure 3.1 and derive the query that is used in Figure 3.2, while preserving the procedure calls and modularity of the original.

3.3 Overview of Query Extraction

Query extraction infers a description of the superset 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 3.4–3.6.

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.

Section 3.4 describes an intraprocedural path-based analysis. Section 3.5 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 pre-loaded 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.

3.4 Intraprocedural Query Extraction

We now formally define query extraction using an attribute grammar. An attribute grammar is a way to specify the semantics of a context-free language [Knu68]. 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 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. Inherited attributes express nested program properties in a natural way. For example, an inherited attribute can associate an `if` statement's condition with that statement's branches. Our analysis requires exactly this kind information, which is why we choose to describe it using an attribute grammar.

Attribute values may induce a circular dependence. A fixed-point algorithm computes the value for each attribute [MH07].

We first define query extraction for a subset of Java that contains no

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

Figure 3.5: Abstract values.

methods. Section 3.5 extends query extraction to the interprocedural case.

3.4.1 Abstract Values

The path-based analysis approximates real-world values with abstract values. Figure 3.5 formally defines these abstractions. The domains *FieldName* and *VarName* describe the fields and variables that appear in a program, respectively. The domains *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 3.1 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 3.4.2. 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,

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

Figure 3.6: Synthesized attributes for intraprocedural traversal summaries.

Figure 3.1 uses the `zip` 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 $\perp$ (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.

3.4.2 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. The compiler may compose a method's traversal summary with those of other

methods to compute a whole-program analysis.

Figures 3.6, 3.7, and 3.8 define 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 3.6. 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 $\perp$, 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: $(\epsilon, \texttt{true}, \texttt{false})$. A field traversal `e.f` concatenates field names to previously computed paths, according to this definition:

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

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.

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

Figure 3.7: Inherited attributes for intraprocedural traversal summaries.

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 $\mathsf{AV}(\mathtt{e}).\iota$ 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 3.7).

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 is 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 requirement means that the compiler must perform 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 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 [Ban88]. Programs 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 3.7—collects conditions under which an expression or statement is executed. The attribute is inherited by all sub-expressions.

$\cdot$	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	$\mathsf{TS}(\mathrm{e}) \cup iter(\cdot)$
$other[\![\cdot]\!]$	$\mathsf{P}(\cdot) \cup \begin{cases} iter(\cdot) & \text{if } \mathsf{D}(\cdot) \\ \emptyset & \text{if } \neg\mathsf{D}(\cdot) \end{cases}$
where $iter(\cdot) = (\{last(\mathsf{IT}(\cdot))\}, \mathsf{C}(\cdot), \textbf{true})$	

Figure 3.8: Traversal summary computation.

Data Dependence and Iterators Figure 3.7 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 $effectful[\![\mathtt{e}]\!]$ 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 3.8 defines the traversal summary attribute TS. It combines the path, condition, and data dependence attributes into a traver-

sal 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 3.9 illustrates an application of the path-based analysis. The analyzed program is a one-method version of the example from Figure 3.1. 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,

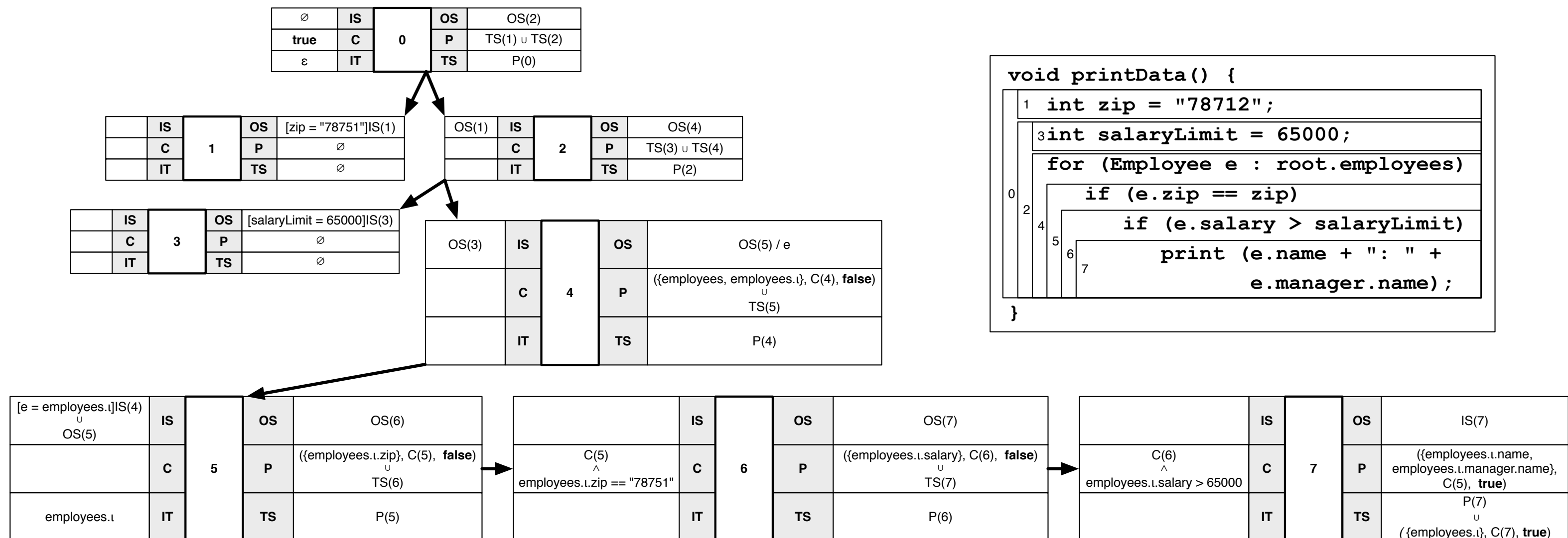


Figure 3.9: An illustrated example of the intraprocedural, path-based analysis. Each example statement is numbered, and each number corresponds to a node in the AST. Each node is a table whose middle column is the statement number and whose left and right columns contain values for synthesized and inherited 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.

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**

3.5 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 [IKY+00], and closely related class hierarchy analysis [DGC95], 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 ver-

sion of the caller for each method that can be called [AU77]. 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 [IKY+00]. We believe that these techniques could work with query extraction, but we have not yet tried to integrate them.

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

We extend the definition of paths from Section 3.4.1 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 3.1 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 3.1 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.

3.5.2 Interprocedural Path Analysis

Interprocedural query extraction extends the language and attribute grammar of Section 3.4 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:

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

Callsites A callsite's traversal summary and abstract value depend on whether the analysis can predict the called 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 value is a new path that is rooted at the callsite's label:

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

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

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 3.10 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:

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

```
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 3.11: Traversal passes through `getEmpToNotify` back to `employeeInfo`.

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.

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

If method g is recursive, then this analysis diverges. We discuss how to ensure termination in Section 3.5.5.

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 3.11 contains an example of pass-through traversals. Method `getEmpToNotify`'s return value is the result of a traversal from its first pa-

rameter `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:

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

3.5.3 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

```
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 3.12: Callee can pre-load caller's data.

example, assume the analysis determines that method `getHRDepartment` in Figure 3.12 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:

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

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.

3.5.4 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 models `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 3.4.2.

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 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.

3.5.5 Recursion

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

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

Our current implementation uses a path representation that is more expressive than our formal definition. It generates the path

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

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`.

3.5.6 Soundness

Appendix A proves the soundness of our analysis for a small kernel language without methods. The full analysis in this chapter 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 does not handle reflection, class loading and dynamic class generation, which break devirtualization's closed-world assumption. 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.

3.6 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 [HE07]. Query extraction is implemented as a source to source transformation that rewrites the program to include queries. The transformed program executes queries in the Hibernate Query Language (HQL). 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 compiles and runs on a standard Java virtual machine.

3.6.1 JastAdd

JastAdd compiles circular reference attribute grammars into compilers. JastAdd includes a Java 1.5 compiler implementation, which we extended to perform query extraction. JastAdd provides as part of the Java specification a control flow analysis that 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.

3.6.2 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.

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

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 programmer writes the code in Figure 3.13, and our compiler transforms it to the code in Figure 3.14, which is simplified to omit type packages.

Callsites inside the method (e.g., lines 27–28 in in Figure 3.14) are transformed to use the version of the method that accepts additional traversal information.

3.6.3 Query Translation

Our prototype compiler targets the Hibernate Query Language (HQL), which the Hibernate library automatically translates to SQL. The compiler translates a traversal summary into as few queries as possible given the constraints of HQL. If a query condition contains a traversal from an object which

```
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 3.14: Code transformation for the method in Figure 3.13.

may be null, then the transformed program may eliminate a `NullPointerException` that would have occurred in the original program. The compiler can address this semantic difference 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 compiler supports recursion by unfolding recursive paths a finite number of times. It compiler statically generates one query for the unfolded traversal summary. The compiler also generates code to detect when program execution traverses beyond the objects already loaded. Upon this event, the program dynamically executes additional queries using the same unfolded traversal summary. The number of unfoldings is a user-supplied parameter `nUnfold` to query extraction that tunes how the compiler generates recursive queries. Concretely, if a program execution recursively traverses data organized as a binary tree of depth n and `nUnfold` $= m$, then the program's first query will retrieve the top m levels of the tree. When the program reaches one of 2^m nodes at depth m, it executes another query that retrieves m levels of the subtree rooted at that node. This process continues until the program finishes its traversal. The current implementation allows recursive paths only if they are generated by method recursion, because the implementation performs queries only at method boundaries. Further engineering is required to generate code that takes advantage of the full generality of recursive paths supported by the analysis.

3.7 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 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—indicate the quality of our prototype implementation. We present our implementation's performance for these metrics and conclude that our prototype performs well in most 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 non-deterministic, 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 to gather samples[GBE07, HGBM08]. 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 ex-

plicit queries. We examined existing database benchmarks and located two that serve our purposes. The TORPEDO benchmark measures the number of queries executed by object-relational mappers for Java [Mar05]. The OO7 benchmark measures the performance of object-oriented database management systems [CDKN94]. We had to modify both benchmarks, so that we could use them to evaluate our analysis.

3.7.1 TORPEDO

The TORPEDO 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. 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

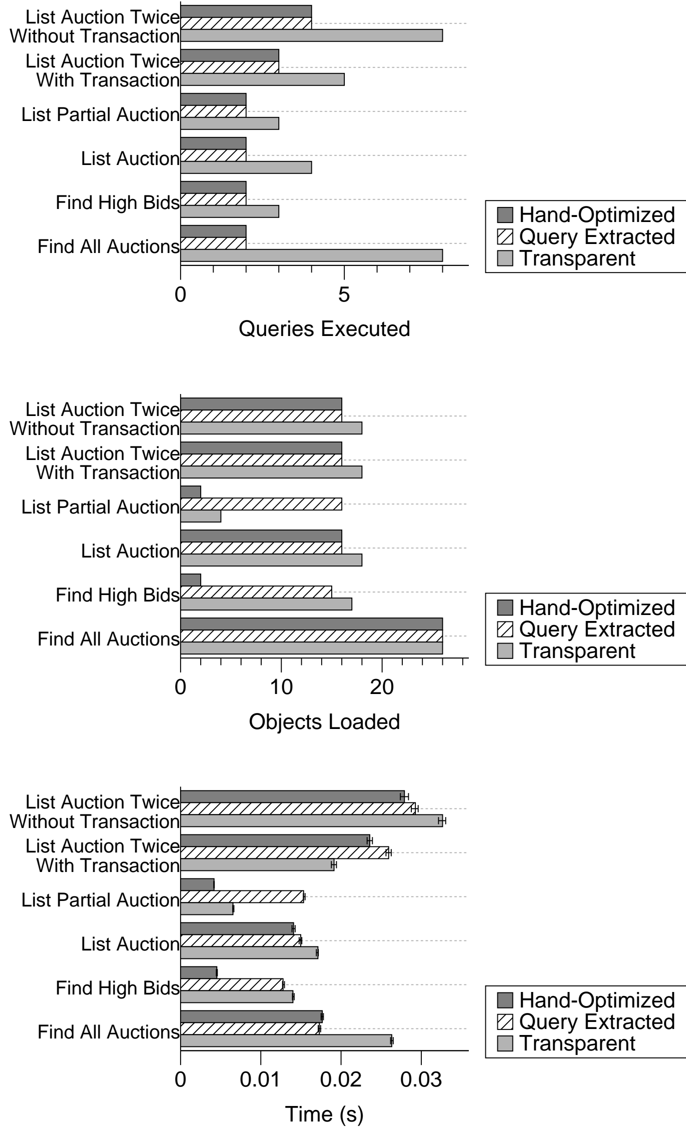


Figure 3.15: Number of queries executed, number of objects loaded, and execution time for the TORPEDO benchmark. Query extraction locates and executes the same number of queries as the hand-optimized version in all use cases, loads fewer objects than the transparent version in all but one use case, and outperforms the transparent version in many cases.

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

Table 3.1: Number of (possibly recursive) methods/callsites in TORPEDO and OO7 across which query extraction statically extracts a query.

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 3.1 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 3.15 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. 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 auctions 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.

3.7.2 OO7

The OO7 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 3.1 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 3.6.3. We first performed query extraction with an unfolding depth of one. Figures 3.16 contains the results for the query use cases, and Figure 3.17 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

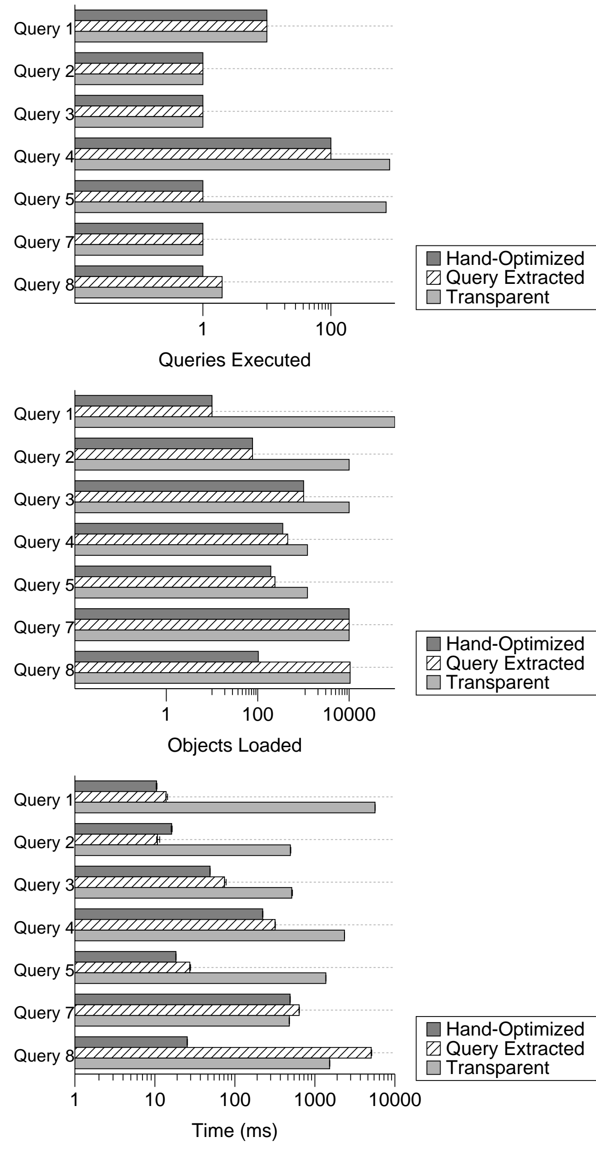


Figure 3.16: Number of queries executed, objects loaded, and execution time per OO7 query use case. In general, query extraction performs comparably to hand-optimized code and favorably to transparent persistence. Query extraction does not perform well for Query 8, because that use case contains a filter that is not a query condition.

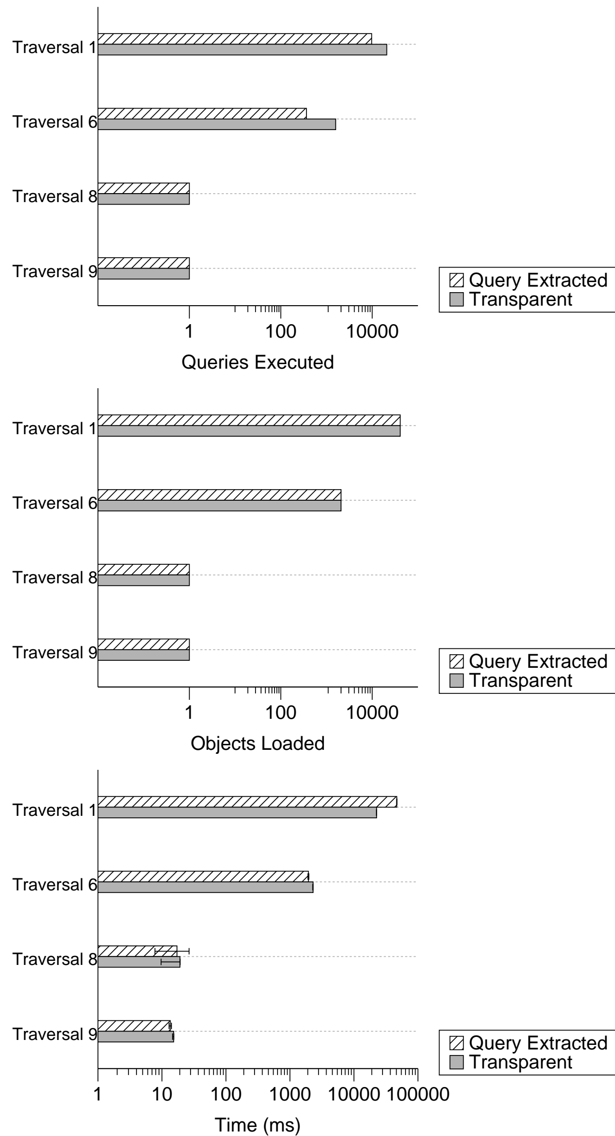


Figure 3.17: Number of queries executed, objects loaded, and execution time per OO7 traversal use case. Query extraction performs better than transparent persistence for Traversal 6 and worse than transparent persistence for Traversal 1. This difference occurs because Traversal 1 traverses a highly connected graph, and an HQL query cannot efficiently retrieve such a structure.

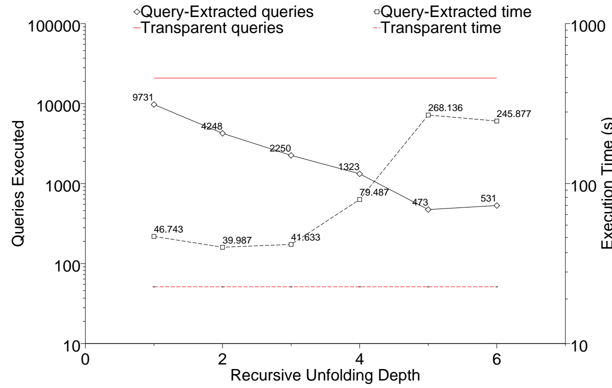
an ad-hoc join of two collections. Query extraction does not optimize these kinds of traversals. The hand-optimized version loads all the required objects with a single query. The query-extracted version performs one query for each collection, and loads too many objects because the use case's condition violates the *master-detail* restriction discussed in Section 3.4.2. 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.

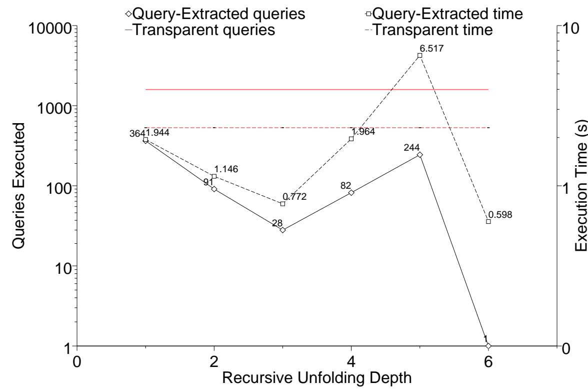
For all use cases except Query 8, the query-extracted version takes time comparable to the other two versions. The query-extracted version of Query 8 takes three orders of magnitude longer to execute than the hand-optimized version, because the query contains a filter that is not a query condition. Specifically, the query performs a cross-join on two collections that do not participate in a master-detail relation.

Performance for Traversal Use Cases The evaluation for these uses cases compares query-extracted performance 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 3.18 illustrates this difference in more detail by varying the unfolding depth for these two use cases. Figure 3.18a 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



(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 3.18: Query extraction performance depends upon the structure of traversed data and upon unfolding depth.

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 3.18b 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 3.6.3 is suboptimal for unfolding depths which are not a factor of six. In general, the optimal unfolding depth depends on the data characteristics.

3.8 Related Work

We discussed many alternative integration solutions in Chapter 2. These solutions include compiler analysis of embedded query strings [GSD04, WGSD07, TTS$^+$08] and language constructs like LINQ [BH07] that provide first-class, explicit database query constructs. LINQ and JQL [WPN06, WPN08] also provide first-class explicit query constructs for in-memory objects.

These solutions differ from ours 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 con-

version 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.

MIDAS transforms a Cobol program that uses the network model to access database values into one that uses declarative, relational queries [CF03]. The network model preceded the relational calculus. It resembles transparent persistence in that programmers traverse relationships rather than declare explicit queries. Like query extraction, MIDAS abstracts a program's implicit data traversals then transforms the program to execute explicit queries. MIDAS detects some common forms of aggregations; whereas query extraction does not. The goal of MIDAS is to transform legacy Cobol code so that is uses contemporary technology and takes advantage of that technology's per-

formance characteristics. Query extraction leverages the same technology's performance, but its goal is to allow programmers to take advantage of procedural abstraction. The two techniques' design and implementation also differ. In particular, MIDAS is a completely static approach; whereas query extraction dynamically composes queries across virtual method calls.

Queryll uses bytecode rewriting to translate Java code to SQL [IZ06]. Queryll's design goals differ from ours in that we aim to provide maximal transparency. Queryll, on the other hand, trades some transparency for ease of analysis and perhaps more predictable program behavior. Queryll programmers specify database operations by writing Java loops over a custom collection type. This type identifies persistent values. The Queryll programmer iterates over persistent collections, adds values to a local collection, then iterates over the local collection to perform computations on persistent data. Queryll supports roughly the same database operations as query extraction. It does not optimize queries that cross method boundaries, nor does it support nested queries.

The orthogonal persistence research community introduced many optimizations whose goals were similar to ours. The biggest performance hit for orthogonal persistence—and the best target for optimization—is its object-faulting mechanism. Hosking and Moss identified two related techniques to reduce the overhead of object faulting: prefetching *co-resident* objects and removing redundant residency checks [HM90]. Object a is co-resident with object b if loading a always requires that b eventually be loaded. The runtime

can load co-resident objects at the same time to avoid multiple round-trip communications with the persistent store. Moss and Hosking proposed that programmers use annotations to provide co-residency information about data types, formal parameters, and return values [MH94]. They suggest a static analysis may be able to infer this information, but do not seem to have pursued that suggestion [HM91]. Query extraction confirms this suggestion as a useful pursuit.

Hosking et al. provide a static analysis that performs partial redundancy elimination (PRE) of residency checks [HNCB99]. Brahnmath et al. proposed a similar analysis for hoisting residency checks that result from array accesses [BNHC99]. Hoisting can move a residency check to an earlier point in an execution path. If the moved barrier causes an object fault, then hoisting performs a primitive kind of prefetching. This prefetching, however, is merely a side effect of read-barrier elimination. The optimization does not characterize a program's bulk data accesses, nor the conditions under which the accesses occur. Query extraction is a more high-level optimization that directly addresses the record-at-a-time loading problem.

Hosking et al.'s analysis also performs PRE of *write-barriers*—runtime checks that determine if persistent data has been modified. Query extraction does not handle modifications of persistent data.

The compiler for E, persistent version of C++, implements a path-based static analysis that attempts to optimize the loading of persistent data [Ric89]. When an E program loads a persistent value, that value is *pinned* in

memory until the program indicates the value is no longer needed. A pinned value is not evicted from memory; thus, pinning can improve the performance of programs with good temporal locality. However, pinning can be an expensive operation. E's *Compiled Item Faulting (CIF)* is an analysis that examines a program's use of persistent data to locate and remove redundant pin/unpin operations. It also lifts partially redundant pins. CIF and Hosking et al.'s analyses are similar. CIF is more similar to query extraction than is Hosking et al.'s, because CIF is a path-based analysis of persistent data use. Query extraction differs from CIF in that it includes information about the conditions under which data access occurs. Whereas CIF is specific to E and its use of pinning, query extraction applies to a variety of persistent systems, because it is a source-to-source translation that generates explicit queries.

The DBPL language [SM94] and it successor Tycoon [MSS95] explored optimization of search and bulk operations within the framework of orthogonal persistence. Tycoon proposed integrating compiler optimization and database query optimization [GM00]. Queries that cross modular boundaries were optimized at runtime by dynamic compilation [SMV93]. 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 runtime *traversal context*—access patterns, including paths—to dynamically predict database loads and prefetch the predicted values [LAC+96, BPS99, HMW03, IC06]. Because our work generates queries which could be

used in object persistence architectures, the two techniques could be combined to achieve further performance benefits.

Vitenberg et al. describe a traversal-based analysis for predicting the persistent values a program may need [VKS04]. Their approach supports runtime improvement of transaction lock scheduling. Kvilekval and Singh use shape analysis to dynamically prefetch remote data for mobile clients [KS04]. Their work reduces the effect of disconnects in mobile computing environments. Ours is a mostly static approach that supports program transformation to bulk-load persistent data. Our analysis also differs in that it identifies traversal conditions.

Query extraction has some superficial commonalities with research lines that optimize programs by prefetching in-memory data. A summary of this research is out of the scope of this dissertation. We refer the reader to Vanderwiel and Lilja's survey paper [VL00] and to specific techniques for detecting which data should be prefetched, including techniques based on properties of loop induction variables [CKP91], numerical properties of array accesses [MLG92], and structural properties of the program data [LM96]. Regardless of its techniques, software memory prefetching speeds up programs by analyzing its loop computations and prefetching data that the program will require. Query extraction is similar in that it analyzes a program's persistent data needs and prefetches that data.

The difference in scales between in-memory and persistent data prefetching affects their techniques, as well as their measures of success [Ibr09]. Net-

work latency is orders of magnitudes larger than memory latency. An object relational cache in virtual memory is larger than an on-chip memory cache. These differences mean that successful persistent data prefetching schemes can be less precise than their in-memory counterparts.

3.9 Conclusions and Future Work

This chapter presented an automatic technique for extracting queries from object-oriented programs that use transparent persistence. Transparent persistence alone performs so poorly that it is not a viable way to write programs. Query extraction recognizes and extracts the implicit queries in a transparent program, and it transforms the transparent program into a program that executes the implicit queries. The transformed program behaves comparably to an equivalent one written by hand. Programmers need not relinquish procedural abstraction to write efficient programs that retrieve data from a database. We contributed a prototype Java compiler with query extraction that includes 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 automatic query extraction in a practical setting.

Query extraction succeeds because it detects filters on persistent data and because it propagates query information across procedure boundaries. The collection of query condition requirements is the key concept we used to design a pragmatic solution. Query extraction detects many kinds of filters,

which improves program performance because the server executes the filters and reduces the dataset size.

Query extraction propagates query information across procedure boundaries in two ways. 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. Our solution handles procedure parameters by devirtualization and query separation and handles procedure results by a novel combination of static analysis and dynamic query composition. Although a caller preloads data for its callee only if devirtualization succeeds, the dynamic composition always allows a callee to preload data for its caller.

Query extraction offers some opportunities for improvement. It currently does not optimize modifications of persistent data. Such optimization requires information about the scope of the update operation. Without this information, the compiler would need to generate a query for each update operation, to ensure data consistency. Data modifications would suffer from the same record-at-a-time performance that plagues transparent persistence.

Information about query scope can come from one of two sources. Either the compiler can infer it, or the programmer can provide it. We leave open the question of whether the compiler can infer query scope. In the next chapter, we pursue a way for the programmer to provide query scope. Specifically, we discuss and modify Remote Batch Invocation (RBI)—a programming abstraction in which the programmer delimits the scope of generic client-server interactions in exchange for a compiler guarantee about a program's latency

behavior. The contributions we present in the next chapter render RBI a suitable technology by which programmers can express transparent retrievals and modifications of persistent data.

Chapter 4

RBI-DB$^+$: Transparent Persistence from Programming Language Design

4.1 Introduction

In this chapter, we formalize and extend Remote Batch Invocation (RBI) so that programmers can use it to write transparent enterprise applications. RBI generalizes query extraction to general-purpose, client-server computing. The RBI programmer intermingles client and server computations, and the RBI compiler picks out server computations to be executed remotely in bulk. The programmer specifies client-server computing inside a programming construct called a *batch block*. A batch block delimits the scope of remote computation and makes latency explicit.

Remote Batch Invocation is a new concept, and it is under active development [TCJ09, IJCT09, IJI$^+$09, CTIW09, Ibr09]. Because it is new, it lacks some of properties that make it suitable for transparent enterprise applications. Programmers cannot write batches that cross method boundaries, which limits program modularity. The RBI implementation conflates two notions: distinguishing client from server code and verifying that a batch block

obeys latency constraints. Conflating these two notions makes the implementation difficult to understand and extend. The RBI database implementation does not recognize or execute database modifications. RBI also lacks a formal description. This chapter addresses these shortcomings through the following contributions:

- A formal description of Remote Batch Invocation.
- Batch methods, which programmers use to create batches that cross method boundaries.
- A new program analysis for RBI, based on dependences. The new analysis disambiguates location identification from batch block verification and makes the RBI implementation more extensible.
- A program transformation that uses runtime inlining to accommodate virtual batch methods.
- An extension to RBI for databases that recognizes and executes modifications to persistent data.
- A prototype implementation of our work, which we call RBI-DB$^+$.

We begin this chapter with an informal description of Remote Batch Invocation. We then present a formal description and discuss some of RBI's shortcomings. The second half of this chapter addresses these shortcomings and describes our prototype implementation.

4.2 Remote Batch Invocation (RBI)

Remote Batch Invocation (RBI) is a programming abstraction that helps programmers manage the tradeoff between program transparency and program performance for a wide range of distributed computing paradigms, including Remote Procedure Call (RPC) [IJCT09], Web Services [IJI+09], and Database Management Systems (DBMS) [Ibr09]. Chapter 2 described the transparency/performance tradeoff in the context of object-oriented programs and relational databases. Remote Procedure Call and Web Services present the tradeoff in other contexts.

Remote Procedure Call [TA90], and its object-oriented counterparts [Obj97, Sun97, BK98], treat local- and remote-code invocation uniformly as a procedure call. RPC is highly transparent, because it hides the details of serializing a remote call's arguments and of transferring control between local and remote code. Ibrahim et al. note that RPC suffers from the same record-at-a-time problem as transparent persistence [IJCT09]. The program's latency properties are hidden, and every remote call requires one round-trip communication. Object-oriented programming exacerbates this problem, because it encourages shorter methods and more method calls.

A web service is a general term that applies to any remotely located behavior accessible via HTTP. A web service's implementation may rely on any number of technologies, including RPC. Ibrahim et al. [IJI+09] focus on document-oriented web services [CCMW01, PTDL07]. The authors of these services are plagued by design issues. If they design a service with a fine-

grained interface—which is consistent with the object-oriented programming style—then their distributed system will exhibit poor performance. The system's clients will have high latency overheads, because each use may require multiple web service calls. The web service, in turn, may be overrun by requests to perform many short-lived operations.

Ibrahim et al. note that many other researchers have attempted to overcome these limitations, but that no existing technique provides the following desirable properties:

Latency should be explicit. The programmer should be able to reason about a program's latency properties. When latency is implicit, the programmer lacks a cost model and cannot predict the program's run-time performance behavior.

The technology should enable *latency composition.* A technology exhibits latency composition if it enables *remotely composable* operations to be executed in a single round trip. Two operations are remotely composable if they can be executed remotely, together, and without any intervention from the client.

Performance concerns should not determine a service's interface. Performance concerns often force both RPC and web-service programmers to artificially coarsen a remote service's interface, to reduce the system's latency overhead [Fow03]. These techniques effectively hard-code clients' needs into

the system, making it difficult for the system to accommodate any changes. Service implementers should be free to define interfaces at a granularity that is independent of any single client's needs.

Ibrahim et al.'s Remote Batch Invocation is a new distributed programming abstraction that exhibits these three properties. RBI allows programmers to be explicit about latency, compose remote operations, and execute logically fine-grained operations efficiently. The resulting programs are perform comparably to those written in more modular style [IJCT09, IJI+09].

The key concept of RBI's programming model is a new programming-language construct called the *batch block*. A batch block makes latency explicit, because the RBI programming model guarantees that each lexical occurrence of a batch block corresponds to at most one round-trip communication between client and server. The batch block provides transparency, because it allows the programmer to specify remote operations in a manner similar to RPC. Batch blocks also transfer some of the burdens of distributed, client-server programming—such as constructing connections and serializing data—from the programmer to the compiler.

Figure 4.1 contains an example batch block. Variable `b` in line 1 indicates the root of remote computation. It's type, `BatchService`, is a *batch handler*. The batch handler manages the interaction between remote and local computations at runtime. The body of the batch block intermingles remote and local code. The programming model assumes that local operations cannot

```
1    batch (BatchService b : new BatchService(...)) {
2       for (Employee emp : b.getEmployees()) {
3          if (emp.getSalary() > 65000)
4             this.print(emp.getName());
5       }
6    }
```

Figure 4.1: An example batch block in a Java-like language. The block mixes local and remote code. The RBI compiler is responsible for partitioning the batch block into separate remote and local computations.

affect objects on the remote machine, and vice versa.

The RBI compiler partitions the block into remote and local code and creates data structures to pass results between the remote and local partitions. This behavior has two stages: program analysis and program transformation.

During the program analysis phase, the RBI compiler assigns a *location* to each part of the program. A location determines whether a computation should occur on the client or the server.

The compiler uses its location analysis to determine whether a batch block is *valid*. A valid batch block is one whose statements may be reordered so that all the remote computations occur before the local computations. The compiler guarantees that each valid batch block requires at most one round-trip communication to perform its task. This guarantee makes a program's latency behavior more explicit. The guarantee is lexical: a batch block may call other methods which themselves contain batch blocks. The programmer is not prevented from calling, for example, library code which may contain its own batch blocks. Batch blocks may not be lexically nested.

```
for (Employee emp : b.getEmployees()^A) {
    if (emp.getSalary() > 65000^B)
        emp.getName()^C;
}
```

(a) The remote partition.

```
for (A) {
    if (B)
        this.print(C);
}
```

(b) The local partition.

Figure 4.2: Remote and local partitions for the example in Figure 4.1. The boxes indicate the values that must be transferred from the server for use in the client.

The RBI programming model also stipulates that a valid batch block's local partition may send only primitive values and arrays of primitive values to the remote partition, and vice versa. RBI neither assumes nor requires that the local and remote services represent objects in the same way.

If the RBI compiler determines that a batch block is invalid, it issues an error. Otherwise, the compiler *partitions* the block into contiguous remote and local computations, and it creates data structures that serve as intermediaries between the two partitions.

Figure 4.2 shows the remote and local partitions of the batch block from Figure 4.1. The boxes indicate remote values that must be transferred from the server to the local computation. The RBI compiler automatically generates the necessary intermediate data structures. This process is called *reforestation* [Ibr09], which is analogous to the functional programming optimization known as *deforestation* [Wad]. Whereas deforestation fuses computation for the purpose of removing programmer-created intermediate data structures; reforestation introduces intermediate data structures for the purpose of com-

municating between compiler-created partitions.

RBI always executes remote operations before local operations. Because the programmer intermingles remote and local code, RBI may ultimately reorder the code in a batch block which can lead to unexpected behavior. However, RBI does not change the relative order of operations within a given partition.

The batch handler executes the remote computation and marshals the results into an appropriate intermediate data structure. The batch handler executes at runtime. It takes as input a description of the remote computation and produces as output the remote computation's results. RBI defines a generic description of remote computation called the Remote Intermediate Language (RIL). The RIL is a form of mobile code, which is evaluated remotely [SG90].

Each form of distributed computing requires its own batch handler. For example, a document-oriented web service might require a batch handler that translates RIL to SOAP [IJI+09]; whereas, a relational database application might require a batch handler that translates RIL to SQL [Ibr09].

To summarize, the components of the partitioned and reforested program work in concert as follows:

$$\text{let } (rs = h(r)) \text{ in } l(rs)$$

where h represents the batch handler, r represents the compiler-generated RIL description of remote computation, l represents the compiler-generated local

$$
\begin{array}{rcl}
 & & l \in \mathit{ContainerLabel} \times \mathit{ElementLabel} \\
 & & v \in \mathit{VariableName} \qquad p \in \mathit{PrimitiveValue} \\
 & & pt \in \mathit{PrimitiveType} \qquad c \in \mathit{ClassName} \\
 & & m \in \mathit{MethodName} \qquad hc \in \mathit{HandlerClassName} \\
t \in \mathit{Type} & ::= & pt \mid c \mid hc \\
e \in \mathit{Expression} & ::= & v \mid \textsf{this} \mid \textsf{op}_n(e_1, \ldots, e_n) \mid e^l \\
 & \mid & e.m(e_1, \ldots, e_n) \mid \textsf{new}\ c(e_1, \ldots, e_n) \\
\textsf{op}_0 \in \mathit{Constant} & ::= & p \\
\textsf{op}_1 & ::= & \neg \mid - \\
\textsf{op}_2 & ::= & \wedge \mid \vee \mid > \mid < \mid = \mid * \mid / \mid + \mid - \\
s \in \mathit{Statement} & ::= & e \mid \textsf{skip} \mid s; s \mid \textsf{return}\ e \\
 & \mid & \textsf{final}\ t\ v\ := \ e \mid t\ v\ := \ e \mid v\ := \ e \\
 & \mid & \textsf{if}\ e\ \textsf{then}\ s\ \textsf{else}\ s \mid \textsf{for}\ (t\ v\ := \ e)\ s \\
 & \mid & \textsf{batch}\ (t\ v\ := \ e)\ s
\end{array}
$$

Figure 4.3: Abstract syntax of RBI's kernel language.

computation, and rs represents the intermediate data structure that contains the results of remote computation.

This semi-formal description of RBI gives an intuition of the source-to-source translation performed by the RBI compiler. We now proceed with a more formal description of RBI.

4.3 A Formal Description of RBI

In prior work, RBI has been described in various levels of detail, and all the descriptions are primarily informal. This section contains the first fully formal presentation of RBI. We base our formal description on the mostly informal one that appears Ibrahim's dissertation and on his prototype imple-

mentation [Ibr09].

4.3.1 An RBI Kernel Language

We formalize RBI using a kernel language that is a subset of Java. The language's, abstract syntax appears in Figure 4.3. The languages permits loops, conditionals, method calls, and operations over sub-expressions. The **batch** statement consists of the batch root variable declaration and a body. The language contains labels and labeled expressions. Labels are used exclusively by the compiler when it partitions a program; programmers do not write labels. Types in the language are partitioned into primitive types, *handler classes*, and all other classes. The handler classes correspond to batch handlers and provide the ability to ship a computation to a remote server.

4.3.2 The Remote Intermediate Language (RIL)

The Remote Intermediate Language (RIL) is a generic grammar for describing remote computation. The RIL largely determines which operations may be executed remotely. Figure 4.4 contains the abstract RIL syntax. It is a subset of the RBI kernel language, with the exception that labeled expressions appear only in statements and not as sub-expressions. RIL permits conditionals, loops, method calls, and operations. RIL also permits binding a value to a variable, but it does not permit re-binding a value to any variable. In this way, the **final** statement acts more like a **let** statement, and RIL does not define a traditional assignment statement. RIL's lack of an assignment operator

$$
\begin{array}{rcl}
 & & l \in \mathit{ContainerLabel} \times \mathit{ElementLabel} \\
 & & v \in \mathit{VariableName} \qquad p \in \mathit{PrimitiveValue} \\
 & & pt \in \mathit{PrimitiveType} \qquad c \in \mathit{ClassName} \\
 & & m \in \mathit{MethodName} \\
t \in \mathit{Type} & ::= & pt \mid c \\
e \in \mathit{Expression} & ::= & v \mid \mathsf{op}_n(e_1, \ldots, e_n) \mid e.m(e_1, \ldots, e_n) \\
\mathsf{op}_0 \in \mathit{Constant} & ::= & p \\
\mathsf{op}_1 & ::= & \neg \mid - \\
\mathsf{op}_2 & ::= & \wedge \mid \vee \mid > \mid < \mid = \mid * \mid / \mid + \mid - \\
s \in \mathit{Statement} & ::= & e\boxed{l} \mid \mathsf{skip} \mid s; s \\
 & \mid & \mathsf{final}\ t\ v\ := e\boxed{l} \\
 & \mid & \mathsf{if}\ e\boxed{l}\ \mathsf{then}\ s\ \mathsf{else}\ s \mid \mathsf{for}\ (t\ v\ := e\boxed{l})\ s
\end{array}
$$

Figure 4.4: Abstract syntax of RBI's Remote Intermediate Language (RIL). The boxes indicate additions to the kernel language of Figure 4.3.

prevents RBI from expressing common operations like remote aggregation.

The RBI compiler translates a batch block's remote computations into an RIL program, which the batch client sends to the remote server. Before the compiler can generate RIL, it must first analyze each batch block to distinguish local from remote computations and valid from invalid batch blocks.

4.3.3 Program Analysis

The compiler analyzes each program and assigns a *location* to each element of a program's Abstract Syntax Tree (AST). A location describes whether an AST node should be executed locally or remotely. The analysis also determines whether each batch block is *valid*. If a batch block is not valid,

Inherited Attribute	$\cdot$	Inherited Attribute Values for Descendants
Inside Batch? $\mathsf{IB}: \mathit{Boolean}$	**batch** $(t\ v\ := \ e)\ s$	$\mathsf{IB}(s) \leftarrow \text{true}$
Batch Root $\mathsf{BR}: \mathit{VariableName}$	**batch** $(t\ v\ := \ e)\ s$	$\mathsf{BR}(s) \leftarrow v$
Loop Variables $\mathsf{LV}: \overline{\mathit{VariableName}}$	**for** $(t\ v\ := \ e)\ s$	$\mathsf{LV}(t\ v\ := \ e) \leftarrow \mathsf{LV}(\cdot) \cup \{v\}$ $\mathsf{LV}(s) \leftarrow \mathsf{LV}(\cdot) \cup \{v\}$

$$
\begin{aligned}
\mathit{Decl}(v) : \mathit{Variable} \rightarrow \mathit{Statement}_{\perp} &= \begin{cases} \textbf{final}\ t\ v\ := \ e & \text{if } \exists\ \textbf{final}\ t\ v\ := \ e \\ t\ v\ := \ e & \text{if } \exists\ t\ v\ := \ e \\ \perp & \text{otherwise} \end{cases} \\
\mathit{LocalAssigns} : \overline{\mathit{VariableName}} &= \{v \mid \mathsf{IB}(v\ := \ e) \wedge \neg\mathsf{IB}(\mathit{Decl}(v))\}
\end{aligned}
$$

Figure 4.5: Inherited attributes and semantic functions for the RBI location analysis.

the compiler issues an error and terminates. If the batch block is valid, the compiler partitions the block.

Figure 4.5 defines inherited attributes and semantic functions required by the location and validity analyses. The IB attribute determines whether an AST node is contained by a batch statement. This attribute is true for all batch-block statements and false for all other AST nodes.

The BR attribute provides the name of a batch block's root variable. The LV attribute similarly provides the names of all enclosing loop variables. The *Decl* semantic function associates each program variable with its declaration. The set *LocalAssigns* contains the names of all variables which are declared outside a batch, but whose values are re-defined inside a batch. The location analysis uses these attributes and semantic functions to determine a variable's location.

RBI's location analysis assigns to every AST node one of the following locations: pre-local, remote, or post-local.[1] Conceptually, pre-local expressions may be used as input to a remote operation; whereas post-local expressions may be used only in conjunction with the outputs of a remote operation.

We summarize RBI's location analysis in Figure 4.6. The first row of this figure assigns a pre-local location to all nodes not contained by a batch block. The remaining rows apply to expressions and statements contained in

[1] Ibrahim et al. uses the term *static local* instead of pre-local and *non-static local* instead of post-local [IJCT09]. We introduce new terms, because they are more descriptive when considered in the context of the subsequent sections.

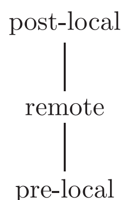
$\cdot$	Location $\mathsf{L}(\cdot) : \mathit{Location}$
$n \in \mathit{RBI}$	pre-local if $\neg\mathsf{IB}(\cdot)$
v	$\begin{cases} \text{post-local} & \text{if } \cdot \in \mathit{LocalAssigns}(\cdot) \\ \mathsf{L}(\mathit{Decl}(\cdot)) & \text{otherwise} \end{cases}$
this	post-local
$\mathbf{op}_n(e_1, \ldots, e_n)$	$\bigsqcup_{i=1}^{n} \mathsf{L}(e_i)$
$e_0.m(e_1, \ldots, e_n)$	$\mathsf{L}(e_0)$
new $c(e_1, \ldots, e_n)$	post-local
p	pre-local
skip	pre-local
$s_1; s_2$	$\mathsf{L}(s_1) \sqcup \mathsf{L}(s_2)$
return e	post-local
final $t\ v\ := \ e$	$\mathsf{L}(e)$
$t\ v\ := \ e$	$\begin{cases} \text{remote} & \text{if } v = \mathsf{BR}(\cdot) \\ \mathsf{L}(e) & \text{if } v \in \mathsf{LV}(v) \\ \text{post-local} & \text{otherwise} \end{cases}$
$v\ := \ e$	post-local
if e **then** s_1 **else** s_2	$\begin{cases} \text{remote} & \text{if } \mathsf{L}(e) = \text{remote} \\ \mathsf{L}(e) \sqcup \mathsf{L}(s_1) \sqcup \mathsf{L}(s_2) & \text{otherwise} \end{cases}$
for $(t\ v\ := \ e)\ s$	$\begin{cases} \text{remote} & \text{if } \mathsf{L}(e) = \text{remote} \\ \mathsf{L}(e) \sqcup \mathsf{L}(s) & \text{otherwise} \end{cases}$

Figure 4.6: Location analysis for RBI batch blocks.

a batch block.

The location analysis relies on transitive reachability to identify remote locations. A variable's declaration determines its location. The batch variable has a remote location. The location of a **final** variable declaration depends on the location of that variable's value, as does a loop variable declaration. If a variable is declared outside a batch block and not reassigned inside the batch block, the analysis assigns the pre-local location to that variable. Its value may be used as input to a remote location. All other declarations receive a post-local location.

Primitive values receive a pre-local location, because these values may be sent to the remote server. Class instantiations and **this** receive a post-local location, because these operations may be performed only locally. A method call's location is the location of its receiver. An operation's location is the join of its sub-expressions according to the lattice:



The **skip** statement can be executed either remotely or locally, so the analysis assigns it a pre-local location. The **return** and assignment statements may only execute locally, so they receive a post-local location. The location of **if** and **for** statements depend on the locations of their conditional and collection expressions, respectively.

Inherited Attribute	$\cdot$	Inherited Attribute Values for Descendants
Inside Local?	**for** $(t\ v\ :=\ e)\ s$	$\mathsf{IL}(s) \leftarrow \mathsf{L}(\cdot) \neq \text{remote}$
IL: *Boolean*	**if** e **then** s_1 **else** s_2	$\mathsf{IL}(s_1) \leftarrow \mathsf{L}(\cdot) \neq \text{remote}$ $\mathsf{IL}(s_2) \leftarrow \mathsf{L}(\cdot) \neq \text{remote}$

Figure 4.7: Inherited attributes for the RBI validity analysis.

Once the analysis has assigned a location to each AST node, the RBI compiler can distinguish valid from invalid batches. A batch is invalid for one or both of the following reasons: (1) The batch contains a remote computation that the RIL does not support (e.g., assignment), or (2) the batch contains a remote expression that depends on a post-local expression. The latter condition is an indirect way of saying that the batch would require more than one round-trip to execute.

Figure 4.7 contains an inherited attribute for the validity analysis. The IL attribute determines whether a given AST node is contained in a *local control structure*. A local control structure is an **if** or **for** statement whose conditional or collection expressions, respectively, are local computations. The validity analysis uses the value of this attribute to help determine whether the body of a control structure is valid.

Figure 4.8 defines a synthesized attribute called VB, which determines whether an AST node is part of a valid batch block. An entire batch statement is valid if it is not nested within another batch, if the root variable corresponds

$\cdot$	Valid Batch? $\mathsf{VB}(\cdot) : \mathit{Boolean}$
v	$\mathsf{IL}(\cdot) \Rightarrow \mathsf{L}(\cdot) \neq \text{remote}$
this	true
$\textbf{op}_n(e_1, \ldots, e_n)$	$\bigwedge_{i=1}^{n} \mathsf{VB}(e_i)$
$e_0.m(e_1, \ldots, e_n)$	$\mathsf{L}(\cdot) = \text{remote} \Rightarrow \bigsqcup_{i=0}^{n} \mathsf{L}(e_i) \sqsubseteq \text{remote}$ $\wedge$ $\bigwedge_{i=0}^{n} \mathsf{VB}(e_i)$
new $c(e_1, \ldots, e_n)$	$\bigsqcup_{i=1}^{n} \mathsf{L}(e_i) \sqsubseteq \text{post-local}$ $\wedge$ $\bigwedge_{i=1}^{n} \mathsf{VB}(e_i)$
p	true
skip	true
$s_1; s_2$	$\mathsf{VB}(s_1) \wedge \mathsf{VB}(s_2)$
return e	$\mathsf{VB}(e)$
final $t\ v\ := \ e$	$\mathsf{VB}(e)$
$t\ v\ := \ e$	$\mathsf{L}(v) \neq \text{remote}$
$v\ := \ e$	$\mathsf{L}(v) \neq \text{remote}$
if e **then** s_1 **else** s_2	$\mathsf{VB}(e) \wedge \mathsf{VB}(s_1) \wedge \mathsf{VB}(s_2)$
for $(t\ v\ := \ e)\ s$	$\mathsf{VB}(e) \wedge \mathsf{VB}(s)$
batch $(t\ v\ := \ e)\ s$	$\neg\mathsf{IB}(\cdot) \wedge t \in \mathit{HandlerClassName} \wedge \mathsf{VB}(s)$

Figure 4.8: Validity analysis for RBI batch blocks.

to a batch handler, and if the batch's body is valid.

If a variable appears inside a local control structure, then that variable's location must not be remote. This restriction prevents local **if** and **for** statements from containing remote computations. The compiler must prevent such statements, because they correspond to a remote value that relies on a post-local value. For the same reason, the arguments of a remote method call cannot have location post-local.

The Remote Intermediate Language does not support assignment, so any non-**final** declaration or assignment cannot have a remote location. All other statements and expressions are valid if their sub-components are valid.

After the analysis determines that a batch block is valid, the compiler partitions the block into remote and post-local computations and reforests the two partitions.

4.3.4 Reforestation

Reforestation creates an intermediate data structure that holds the output from the remote computation and serves as input to the post-local computation. The RBI programming model permits only (arrays of) primitive values to pass between remote and local code. For example, the remote computation in Figure 4.2a operates over employees, but the local computations in Figure 4.2b operates over a collection of booleans and strings. This restriction does not prevent RBI from transmitting complex data. Rather, the

batch handler must marshall data into the following form:

$$l \in \mathit{ContainerLabel} \times \mathit{ElementLabel}$$

$$p \in \mathit{PrimitiveValue}$$

$$rs \in \mathit{ResultSet} \quad ::= \quad l\ p \mid l\ rs \mid l\ \overline{rs}$$

A *result set* is a labeled record over primitive values. Records may be nested, which corresponds to a complex remote object structure. A label may refer to a series of records, which corresponds to a remote collection.

A remote value's label is divided into two components: a *container label* and an *element label*. A value's container label refers to the remote object that contains that value. A value's element label uniquely identifies the value from other values in its container. There are two kinds of container labels: *root* and *collection*. A root container label refers to the batch root object, and a collection container label refers to a remote collection.

In the example from Figure 4.10, we can assign the following labels to the three remote values: Let c_0 be the root container label and c_1 be a collection container label that corresponds to the remote employees collection. Value A's label is (c_0, e_0), value B's label is (c_1, e_1), and value C's label is (c_1, e_2).

RBI's program transformation creates labels for each remote value, as part of the reforestation process.

Inherited Attribute	$\cdot$	Inherited Attribute Values for Descendants
Container Label	**batch** $(t\ v\ :=\ e)\ s$	$\mathsf{CL}(t\ v\ :=\ e) \leftarrow \mathit{freshLabel}()$
$\mathsf{CL}: \mathit{Label}$	**for** $(t\ v\ :=\ e)\ s$	$\mathsf{CL}(t\ v\ :=\ e) \leftarrow \mathit{freshLabel}()$

(a) Container Label inherited attribute.

$\cdot$	Container Label $\mathsf{CL}(\cdot) : \mathit{Label}$
v	$\begin{cases} \mathsf{CL}(\mathit{Decl}(\cdot)) & \text{if } \mathsf{L}(\cdot) = \text{remote} \\ \epsilon & \text{otherwise} \end{cases}$
this	ϵ
op$_n(e_1, \ldots, e_n)$	ϵ
$e_0.m(e_1, \ldots, e_n)$	$\mathsf{CL}(e_0)$
new $c(e_1, \ldots, e_n)$	ϵ
p	ϵ

(b) Container Label synthesized attribute.

$$\frac{\mathsf{L}(e) = \text{remote} \qquad l_1 = \mathsf{CL}(e) \qquad l_2 = \mathit{freshLabel}()}{\mathit{label}(e) = e^{(l_1, l_2)}}$$

$$\frac{\mathsf{L}(e) \neq \text{remote} \qquad e_i \in e \qquad \mathit{label}(e_i) = e_i'}{\mathit{label}(e) = [e_i'/e_i]e} \qquad \frac{e_i \in s \qquad \mathit{label}(e_i) = e_i'}{\mathit{label}(s) = [e_i'/e_i]s}$$

(c) Labeling transformation.

Figure 4.9: Program transformation that labels remote expressions. The transformation is top-down, so it labels the largest remote sub-expressions.

4.3.5 Program Transformation

The RBI compiler transforms a batch block to native code in two stages. First, a labelling stage assigns a label to each remote computation. Then, a partitioning / reforestation stage generates code for the batch's remote and post-local computations, along with references to the reforested result set.

The labelling stage transforms an RBI program into an equivalent program whose remote computations have been assigned reforestation labels. Figure 4.9 defines the transformation. An inherited attribute `CL` defined in Figure 4.9a, provides a container label for the root variable and all loop variables. Every expression also has a container label, defined as a synthesized attribute in Figure 4.9b. A remote variable's container label is determined by its declaration; all non-remote variables have an empty container label. The receiver of a field traversal or a method call determines the container label for those expressions. All other expressions have empty container labels.

Figure 4.9c defines the program transformation that replaces all remote expressions with an equivalent, labeled expression. Function *label* replaces each remote subexpression with its labeled equivalent. The transformation works in a top-down fashion, so it does not replace any remote expression whose parent expression is remote. It does, however, replace any remote expression whose parent expression is not remote. A similar rule replaces any remote expression that appears in a statement.

The result of this transformation generates an equivalent program whose

remote subexpressions have been assigned reforestation labels. The partitioning stage takes as input a labeled program and produces as output the remote and post-local partitions.

Figure 4.10 defines how the partitioning stage uses a labeled RBI program to partition and reforest a batch block. The semantic function *partition* in Figure 4.10a creates a new `ResultSet` that contains the results of executing the batch's remote partition. The batch's post-local partition consumes these results.

The partition stage generates a batch block's remote partition according to the computation described in Figure 4.10b. The semantic function *RIL* is a terse description of a small-step operational semantics [FFKD87] that transforms a labeled RBI program to an RIL program. Labeled expressions are copied from the RBI program to the RIL program. Non-labeled expressions that contain no subexpressions are transformed to **skip**. Non-labeled expressions that contain subexpressions are transformed to a series of transformed statements. Remote statements are copied to the RIL output, with their subcomponents replaced by their RIL equivalents. A non-remote statement is transformed into a series of statements that correspond to that statement's remote sub-components.

The partitioning stage also creates the batch block's post-local partition. This transformation, described in Figure 4.10c, simply replaces remote expressions that appear in an RBI program with their reforested counterpart. A remote, final assignment turns into **skip**. A remote expression turns into a

$$\begin{array}{l} \mathit{partition}(\textsf{\textbf{batch}}\ (t\ v\ :=\ e)\ s) = \\ \qquad \textsf{ResultSet results} := v.\textsf{executeBatch}(r);\ l \\ \text{where } s' = \mathit{label}(s),\ r = \mathit{RIL}(s'),\ l = \mathit{post\text{-}local}(s') \end{array}$$

(a) Batch partitioning and reforestation.

$$\mathit{RIL}(e^l) = e^l \qquad \frac{e \in \{p, v, \textsf{\textbf{this}}\}}{\mathit{RIL}(e) = \textsf{\textbf{skip}}} \qquad \mathit{RIL}(\textsf{\textbf{skip}}) = \textsf{\textbf{skip}}$$

$$\frac{e_i \in e \qquad 1 \le i \le m \qquad \mathit{RIL}(e_i) = e'_i}{\mathit{RIL}(e) = e'_1; \cdots; e'_m} \qquad \frac{n_i \in s \qquad 1 \le i \le m \qquad n_i \in \mathit{Expression} + \mathit{Statement} \qquad \mathit{RIL}(n_i) = n'_i}{\mathit{RIL}(s) = \begin{cases} [n'_i/n_i]s & \text{if } \mathsf{L}(s) = \text{remote} \\ n'_1; \cdots; n'_m & \text{otherwise} \end{cases}}$$

(b) Transforming a labeled RBI program to an RIL program.

$$\frac{\mathsf{L}(v) = \text{remote}}{\mathit{post\text{-}local}(v) = \mathsf{CL}(v)} \qquad \frac{e \in \{p, v, \textsf{\textbf{this}}\} \wedge \mathsf{L}(v) \neq \text{remote}}{\mathit{post\text{-}local}(e) = e}$$

$$\frac{\mathit{post\text{-}local}(e_0) = e'_0}{\mathit{post\text{-}local}(e_0.m(e_1, \ldots, e_n)^{(c_l, c_e)}) = e'_0.\textsf{get}(c_e)}$$

$$\frac{e_i \in e \qquad \mathit{post\text{-}local}(e_i) = e'_i}{\mathit{post\text{-}local}(e) = [e'_i/e_i]e}$$

$$\mathit{post\text{-}local}(\textsf{\textbf{final}}\ t\ v\ :=\ e^l) = \textsf{\textbf{skip}}$$

$$\frac{\mathit{post\text{-}local}(e^l) = e' \qquad \mathit{post\text{-}local}(s) = s'}{\mathit{post\text{-}local}(\textsf{\textbf{for}}\ (t\ v\ :=\ e^l)\ s) = \textsf{\textbf{for}}\ (\textsf{ResultSet}\ v := e')\ s'}$$

$$\frac{n_i \in s \qquad n_i \in \mathit{Expression} + \mathit{Statement} \qquad \mathit{post\text{-}local}(n_i) = n'_i}{\mathit{post\text{-}local}(s) = [n'_i/n_i]s}$$

(c) Transforming a labeled RBI program to its post-local equivalent.

Figure 4.10: Partitioning and reforesting a batch block.

request for that expression's reforested value.

4.4 RBI for Databases (RBI-DB)

Ibrahim's dissertation contributes a prototype database client for batches called RBI-DB [Ibr09]. It provides limited support for read-only queries by transforming the RIL into HQL statements that express filtered selections and projections. Section 4.10 contributes a more complete client, called RBI-DB$^+$, that provides support for updating persistent data, as well as queries over nested collections and conditions.

4.5 Shortcomings of RBI-DB

RBI-DB is a promising foundation on which to build a technique for transparent persistence; however, we must find a way to overcome a few of its shortcomings, if it is to be an effective technique for transparent persistence.

The most significant shortcoming is that RBI lacks a mechanism for modularizing batches. A programmer cannot refactor a common piece of code into a reusable, parameterized sub-batch. RBI's analysis conflates location and dependence, tying the analysis to the analyzed language and making it difficult to modify or extend. The database client of RBI-DB supports fewer features than does Query Extraction, and—more importantly—it does not provide any support for modifying persistent data.

The remainder of this chapter contributes extensions to RBI-DB that

address these three areas: the lack of modularity for batch blocks, the inflexible analysis, and the limited feature set of the database client.

4.6 Batch Methods

Batch methods are an extension to RBI that enable programmers to modularize and compose batch operations. The example if Figure 4.11 motivates the need to compose batches. The figure defines three methods. Methods `salaryTest` and `departmentTest` each iterate over a remote collection of employees, and each iteration performs a test and possibly calls method `process`. This method takes an employee as input and prints that employee's name.

RBI does not compose batches across method boundaries, so the program in Figure 4.11 performs poorly. Methods `salaryTest` and `departmentTest` each require $n+1$ round trips, where n is the number of employees. To achieve a constant number of round trips, the programmer would have to inline method `process` manually, which reduces the program's modularity.

Batch methods enable the programmer to write modular batches, which the compiler can optimize to improve performance. A batch method is a method whose body is a batch block. The compiler inlines the batch method's body at each of its callsites. The inlining is subject to some constraints and requires the programmer to provide some information about the batch method's behavior. This information is required because virtual method dispatch complicates interprocedural program analysis.

```
1  void salaryTest() {
2     batch (BatchService b : new BatchService(...)) {
3        for (Employee emp : b.getEmployees()) {
4            if (emp.getSalary() > 65000)
5                this.process(emp);
6        }
7     }
8  }
9
10 void departmentTest() {
11    batch (BatchService b : new BatchService(...)) {
12       for (Employee emp : b.getEmployees()) {
13           if (emp.getDepartment().getID() == 1)
14               this.process(emp);
15       }
16    }
17 }
18
19 void process(Employee emp) {
20     batch (BatchService b : new BatchService(...)) {
21       this.print(emp.getName());
22     }
23 }
```

Figure 4.11: Batches cannot be composed. Methods `salaryTest` and `departmentTest` each require $n+1$ round trips, where n is the number of elements in the remote collection. To achieve one round trip per method, the programmer would need to inline method `process` manually.

```
1  class Client {
2    void f() {
3       batch (BatchService b : new BatchService(...)) {
4          b.f(this.subBatch(b));
5       }
6    }
7    int subBatch(BatchService b) {
8       return b.g(this.localMethod(1));
9    }
10   int localMethod(int value) {
11     ...
12   }
13 }
14
15 class ClientExtension extends Client {
16    int subBatch(BatchService b) {
17       return this.localMethod(b.g(1));
18    }
19 }
```

Figure 4.12: Virtual method dispatch and batching. The compiler can make only a lexical guarantee about batch blocks, because it cannot determine the run-time communication properties of an invoked method.

Figure 4.12 demonstrates how virtual method dispatch interferes with batch composition. The `Client` class's `f` method contains a batch block. The last operation of this block invokes the remote operation `b.f`, which takes an `int` and returns an `int`. The remote operation depends on the value of local method `subBatch`. If we inline `subBatch`'s body at line 4, we obtain the expression

```
b.f(b.g(this.localMethod(1))
```

This expression requires only one round-trip to compute. If, however, we inline the overridden definition of `subBatch` from class `ClientExtension`, we obtain the expression

```
b.f(this.localMethod(b.g(1))
```

This expression requires two round-trips to compute. The compiler cannot determine statically which of the two `subBatch` definitions will execute at run-time, so it can make only a lexical guarantee about the batch block. The program will perform two round-trips at runtime, regardless of which `subBatch` version executes.

We desire a solution that permits the programmer to modularize batches and enables the compiler to compose over the modularization. Our solution is to define batch methods as a new family of methods. This family has three members: `batchL`, `batchR`, and `batchRL`. The letters `L` and `R` refer to local or

remote operations, and the order of the letters indicates the order in which a batch method performs the operations.

A programmer declares a batch method by modifying the method's signature with one of the three batch method family members. In Figure 4.12, class `Client`'s `subBatch` method can be turned into a batch method by modifying its signature to be

```
batchLR int subBatch(...)
```

The compiler can verify that a batch method's body satisfies its declaration. The overridden definition in class `ClientExtension` defines `batchRL` behavior. Because its definition does not match its superclass, the compiler issues an error.

A programmer may only invoke a batch method inside the body of a batch block. The batch method conceptually is a sub-batch of any batch blocks that call it. The compiler verifies that the batch method may be inlined.

4.7 An Extended Kernel Language

Figure 4.13 extends the RBI kernel language to support batch methods. In practice, a programmer would define a batch method by modifying the method's declaration. We avoid introducing method declarations to our kernel language, by modeling batch methods as annotated method calls. The empty annotation ϵ denotes a non-batch method call. The remaining annotations denote a batch method call of a particular kind. The kernel language also

$$
\begin{array}{rcl}
 & & l \in \mathit{ContainerLabel} \times \mathit{ElementLabel} \\
 & & v \in \mathit{VariableName} \qquad p \in \mathit{PrimitiveValue} \\
 & & pt \in \mathit{PrimitiveType} \qquad c \in \mathit{ClassName} \\
 & & m \in \mathit{MethodName} \qquad hc \in \mathit{HandlerClassName} \\
 & & \boxed{rc \in \mathit{RemoteClassName}} \\
t \in \mathit{Type} & ::= & pt \mid c \mid hc \mid \boxed{rc} \\
a \in \mathit{MethodCallAnnot} & ::= & \boxed{\epsilon \mid \textsf{R} \mid \textsf{L} \mid \textsf{RL}} \\
e \in \mathit{Expression} & ::= & v \mid \textsf{this} \mid \textsf{op}_n(e_1, \ldots, e_n) \mid e^l \\
 & \mid & e.m_{\boxed{a}}(e_1, \ldots, e_n) \mid \textsf{new}\ c(e_1, \ldots, e_n) \\
\textsf{op}_0 \in \mathit{Constant} & ::= & p \\
\textsf{op}_1 & ::= & \neg \mid - \\
\textsf{op}_2 & ::= & \wedge \mid \vee \mid > \mid < \mid = \mid * \mid / \mid + \mid - \\
s \in \mathit{Statement} & ::= & e \mid \textsf{skip} \mid s; s \mid \textsf{return}\ e \\
 & \mid & \textsf{final}\ t\ v\ := \ e \mid t\ v\ := \ e \mid v\ := \ e \\
 & \mid & \textsf{if}\ e\ \textsf{then}\ s\ \textsf{else}\ s \mid \textsf{for}\ (t\ v\ := \ e)\ s \\
 & \mid & \textsf{batch}\ (t\ v\ := \ e)\ s
\end{array}
$$

Figure 4.13: Abstract syntax of RBI-DB$^+$'s kernel language. This language is a superset of RBI's kernel language. The boxes indicate the language extensions.

Inherited Attribute	$\cdot$	Inherited Attribute Values for Descendants
Batch Root $\mathsf{BR}: \mathit{VariableName}$	**batch** $(t\ v\ :=\ e)\ s$	$\mathsf{BR}(s) \leftarrow v$
Loop Variables $\mathsf{LV}: \overline{\mathit{VariableName}}$	**for** $(t\ v\ :=\ e)\ s$	$\mathsf{LV}(t\ v\ :=\ e) \leftarrow \mathsf{LV}(\cdot) \cup \{v\}$ $\mathsf{LV}(s) \leftarrow \mathsf{LV}(\cdot) \cup \{v\}$

$$\mathit{Decl}(v) : \mathit{Variable} \rightarrow \mathit{Statement}_{\perp} \quad = \begin{cases} \textbf{final}\ t\ v\ :=\ e & \text{if } \exists\ \textbf{final}\ t\ v\ :=\ e \\ t\ v\ :=\ e & \text{if } \exists\ t\ v\ :=\ e \\ \perp & \text{otherwise} \end{cases}$$

$$\mathit{TypeLoc}(t) : \mathit{Type} \rightarrow \mathit{Location} \quad = \begin{cases} \text{mobile} & \text{if } t \in \mathit{PrimitiveType} \\ \text{remote} & \text{if } t \in \mathit{RemoteClassName} \\ & \ \ \|\ t \in \mathit{HandlerClassName} \\ \text{local} & \text{otherwise} \end{cases}$$

Figure 4.14: Inherited attribute and semantic functions for the RBI-DB$^+$ location analysis.

distinguishes remote classes, so that the program analysis may identify remote values more easily.

4.8 Program Analysis

RBI-DB$^+$ replaces the syntax-based analysis of RBI-DB with an analysis based on the dependences between locations. This simple analysis yields great benefit, largely due to the RBI programming model requirement that the remote and local memory spaces be distinct.

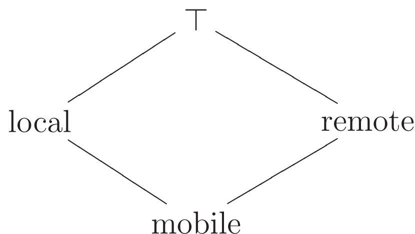
$\cdot$	Location $\mathsf{L}(\cdot) : \mathit{Location}$
v	$\mathsf{L}(\mathit{Decl}(\cdot))$
this	local
$\textbf{op}_n(e_1, \ldots, e_n)$	$\bigsqcup_{i=1}^{n} \mathsf{L}(e_i)$
$e_0.m_a(e_1, \ldots, e_n)$	$\begin{cases} \text{mobile} & \text{if } a \in \{\mathbf{R}, \mathbf{L}, \mathbf{RL}\} \\ \mathsf{L}(e_0) & \text{otherwise} \end{cases}$
new $c(e_1, \ldots, e_n)$	local
p	mobile
skip	mobile
$s_1; s_2$	$\mathsf{L}(s_1) \sqcup \mathsf{L}(s_2)$
return e	local
final $t\ v\ := \ e$	$\mathit{TypeLoc}(t)$
$t\ v\ := \ e$	$\begin{cases} \mathit{TypeLoc}(t) & \text{if } \mathsf{L}(v) \in \{\mathsf{BR}(\cdot)\} \cup \mathsf{LV}(\cdot) \\ \text{local} & \text{otherwise} \end{cases}$
$v\ := \ e$	local
if e **then** s_1 **else** s_2	$\begin{cases} \text{remote} & \text{if } \mathsf{L}(e) = \text{remote} \\ \mathsf{L}(e) \sqcup \mathsf{L}(s_1) \sqcup \mathsf{L}(s_2) & \text{otherwise} \end{cases}$
for $(t\ v\ := \ e)\ s$	$\begin{cases} \text{remote} & \text{if } \mathsf{L}(e) = \text{remote} \\ \mathsf{L}(e) \sqcup \mathsf{L}(s) & \text{otherwise} \end{cases}$

Figure 4.15: Location analysis for RBI-DB$^+$ batch blocks.

4.8.1 Location Analysis

The location analysis uses type information to partition a batch block's computations into their corresponding locations. Each type is classified into one of three locations: mobile, remote, or local. The mobile partition contains all primitive types. This partition is so-named because its values may pass between the remote and local partitions. The remote partition contains all types whose behaviors reside on the server. The programmer can distinguish these types via annotations, configurations, or additional language constructs. The local partition contains all remaining types.

Variables inherit the location of their declared type. An operation's location is the join of its sub-expressions, according to the following lattice:



A batch method call has mobile location. A non-batch method call's location is determined by its receiver. The locations of all other statements and expressions is similar to those defined for RBI in Figure 4.6.

One key difference between RBI-DB$^+$'s program analysis and the one we presented in Section 4.3.3 is that the latter conflated two concepts: location and dependence. The compiler requires both pieces of information to partition a batch block. RBI-DB$^+$ separates the location analysis from the dependence

analysis. In so doing, it achieves a more general way of describing the way data flows between locations.

4.8.2 Dependence Analysis

The dependence analysis computes a program's *Location Dependence Graph* (LDG), which tracks dependences between mobile, local, and remote computations. Conceptually, the LDG is a modified form of the Program Dependence Graph [FOW87], which combines data- and control-dependences in a single representation.

The dependence analysis computes an intraprocedural PDG and treats all calls for the form **this**.$m(\ldots)$ or $v.m(\ldots)$ as a flow dependence on subsequent uses of **this** or v. For most standard analyses, this assumption would not be conservative enough, because a method call may result in definitions of more than its receiver object. However, our location-based analysis can conservatively assume that every method call modifies values at that call's location, because the memory space of each location is distinct.

The Location Dependence Graph effectively replaces an AST node in the PDG with that node's location. It also summarizes the effects of batch method calls. The compiler constructs a program's Location Dependence Graph from its PDG according to the following algorithm:

1: Replace each PDG node n that is not a batch method call with the node $\mathsf{L}(n)_n$.
2: Replace each PDG node n that corresponds to a batch method call of kind **R** with remote_n.
3: Replace each PDG node n that corresponds to a batch method call of kind **L** with local_n.
4: For each PDG node n that corresponds to a batch method call of kind **RL**, create two new nodes remote_n and local_n. Add the edge $\text{remote}_n \rightarrow \text{local}_n$. Replace any edges of the form $n_i \rightarrow n$ with the edge $n_i \rightarrow \text{remote}_n$. Replace any edges of the form $n \rightarrow n_o$ with the edge $\text{local}_n \rightarrow n_0$.

The compiler uses the location analysis and a program's LDG to determine the validity of that program's batches.

4.8.3 Validity Analysis

A valid batch must satisfy the following five properties:

(I) It must not be nested within another batch.

(II) It must not pass remote values to local methods and vice versa.

(III) The compiler must be able to partition the batch block into remote and local computations.

(IV) The partitioned block must perform at most one round-trip.

(V) The partitioned block must be perform all its remote operations before all its local operations.

The compiler enforces property (I) by rejecting any program that contains nested batch blocks. The compiler enforces property (II) by rejecting any method call that does not satisfy the following condition:

$$a = \epsilon \Rightarrow \bigsqcup_{i=1}^{n} \mathit{TypeLoc}(\mathit{type}(e_i)) \sqsubseteq \mathsf{L}(e_0.m_a(e_1, \ldots, e_n))$$

where *type* gives an expression's type. The compiler enforces property (III) by rejecting any expression whose location is $\top$.

The remaining two properties are a function of the batch block's dependences. The compiler can enforce them by constructing a query over a reduced form of the LDG. The reduced form fuses composable operations, and the following algorithm constructs the reduced form from the LDG:

1: Collapse any strongly connected components in the LDG where each node in the component has the same location. Maintain reachability to and from the component, but remove any self-edges.
2: Collapse any edges between two nodes with the same location. Maintain reachability, but remove self-edges.
3: Collapse any edges between a mobile and a remote component, if the edge does not participate in a cycle. Maintain reachability, but remove self-edges.

The algorithm should be repeated until it reaches a fixpoint. Let $\mathcal{L}$ represent the resulting graph. Any path through $\mathcal{L}$ that contains more than one remote node requires multiple round trips, which violates property (IV). If any local node in $\mathcal{L}$ dominates a remote node, then the batch violates property (V).

4.9 Program Transformation

The RBI-DB$^+$ compiler extends the RBI-DB technique for code transformation from Section 4.3.5 to produce code for batch methods. The compiler statically transforms the body of each batch method m into a remote partition m_r and a local partition m_l.

The compiler also statically transforms each call to a batch method into two partitions. The remote partition of a batch method is an RIL method call to m_r. A flag indicates that m_r is a batch method call instead of a standard remote method call. The local partition is a call to m_l, taking as input the result set returned by m_r.

Virtual method dispatch requires sub-batch inlining to occur at runtime. The runtime handler makes a pre-processing pass of the RIL and inlines the result of calling m_r, for each batch method call that appears in the RIL.

With the modifications from the previous section, batches express database operations comparable to those handled by Query Extraction. We now extend the database client to permit modifications to persistent data.

4.10 RBI-DB$^+$'s Database Client

RBI-DB$^+$'s relational database client converts RIL to the Hibernate Query Language (HQL). The client is based on Ibrahim's prototype implementation [Ibr09]. Ibrahim's client supported a subset of RIL, which described filtered retrievals of persistent data. We extend that prototype to support

$$
\begin{array}{rcl}
 & f \in \mathit{PropertyName} & \quad p \in \mathit{PrimitiveValue} \\
n \in \mathit{NamedExpression} & ::= & e \ \textsf{\textbf{as}} \ f \\
e \in \mathit{Expression} & ::= & \overline{f} \mid \textsf{\textbf{op}}_n(e_1, \ldots, e_n) \mid \textsf{\textbf{case}} \ e \\
\textsf{\textbf{op}}_0 \in \mathit{Constant} & ::= & p \\
\textsf{\textbf{op}}_1 & ::= & \neg \mid - \\
\textsf{\textbf{op}}_2 & ::= & \wedge \mid \vee \mid > \mid < \mid = \mid * \mid / \mid + \mid - \\
t \in \mathit{Table} & ::= & \textsf{\textbf{from}} \ n \mid \textsf{\textbf{from}} \ n \ \textsf{\textbf{inner join}} \ n \\
w \in \mathit{Filter} & ::= & \textsf{\textbf{where}} \ e \mid \textsf{\textbf{where}} \ e \ \textsf{\textbf{in}} \ s \\
s \in \mathit{Statement} & ::= & \textsf{\textbf{select}} \ \overline{n} \ t \ w \\
 & \mid & \textsf{\textbf{update}} \ n \ \textsf{\textbf{set}} \ \overline{f} = e \ w \\
 & \mid & \textsf{\textbf{delete}} \ t \ w
\end{array}
$$

Figure 4.16: Abstract syntax of an HQL-like query language.

modifications of persistent data. We also add support for nested filters and nested iterations of structurally related data.

Figure 4.16 contains the abstract syntax for the subset of HQL generated by RBI-DB$^+$'s relational client. The client relies on naming conventions to determine property names. A programmer retrieves the value of a property by calling method `get`*Property*. A programmer updates the value of a property by calling method `set`*Property* and passing a single argument that contains the property's new value. RBI-DB$^+$ does not support creating new values.

We use an attribute grammar to define the translation from RIL to HQL. Figure 4.17 contains the inherited attributes and semantic functions required by the translation. Inherited attribute T associates an iteration with its corresponding table. The attribute distinguishes between nested and non-

Inherited Attribute	$\cdot$	Inherited Attribute Values for Descendants
Table $\mathsf{T}\colon Table$	$\textbf{for}\ (t\ v\ := \ e_0.\mathsf{get}\,Table()^l)\ s$	$\mathsf{T}(s) \leftarrow \begin{cases} \textbf{from}\ table\ \textbf{as}\ name(e^l) & \text{if } \mathsf{T}(\cdot) = \epsilon \\ \mathsf{T}(\cdot) + \textbf{inner join}\ \mathsf{T}(e^l)\ \textbf{as}\ name(e^l) & \text{otherwise} \end{cases}$
Condition $\mathsf{C}\colon Expression$	$\textbf{if}\ e^l\ \textbf{then}\ s_1\ \textbf{else}\ s_2$	$\mathsf{C}(s_1) \leftarrow \mathsf{C}(\cdot) \wedge \mathsf{P}(e)$ $\mathsf{C}(s_2) \leftarrow \mathsf{C}(\cdot) \wedge \neg\mathsf{P}(e)$

$$name(e^l) = l \qquad \qquad \frac{\exists\ \textbf{for}\ (t\ v\ :=\ e^l)\ s}{name(v) = l}$$

Figure 4.17: Inherited attribute and semantic functions for the RBI-DB$^+$ HQL generation.

$\cdot$	Projection $\mathsf{P}(\cdot) : \mathit{Expression}$
v	$\mathit{name}(\cdot)$
$\mathbf{op}_n(e_1, \ldots, e_n)$	$\mathbf{op}_n(\mathsf{P}(e_1), \ldots, \mathsf{P}(e_n))$
$e_0.\mathsf{get}\mathit{Property}()$	$\mathsf{P}(e_0).\mathit{property}$
p	p

Figure 4.18: Projections for RBI-DB$^+$.

nested iterations. Nested iterations correspond to an inner join. The T attribute relies on a semantic function *name*, which uses an RIL expression's label as an HQL alias.

Inherited attribute C associates a filter with traversals of persistent data. An **if** statement filters its branches. The filter is the conjunction of any outer filters with the result of the filtered expression's P attribute.

The P attribute—defined in Figure 4.18—converts an RIL expressions to its HQL equivalent. This attribute relies on the naming conventions to convert a method call to a property projection.

An RIL expression or statement can correspond to one of the following

$\cdot$	Operations $\mathsf{OP}(\cdot) : \overline{\mathit{OperationTuple}}$
$e_0.\mathsf{set}\mathit{Property}(e_1)^l$	$\{(\textbf{update } \mathsf{P}(e_0).\mathit{property} = \mathsf{P}(e_1), \mathsf{T}(\cdot), \mathsf{C}(\cdot), l)\}$
$e_0.\mathsf{delete}()^l$	$\{(\textbf{delete}, \mathsf{T}(\cdot), \mathsf{C}(\cdot), l)\}$
e^l	$\{(\textbf{select } \mathsf{P}(e), \mathsf{T}(\cdot), \mathsf{C}(\cdot), l)\}$
skip	$\emptyset$
$s_1; s_2$	$\mathsf{OP}(s_1) \cup \mathsf{OP}(s_2)$
final $t\ v\ :=\ e^l$	$\mathsf{OP}(e^l)$
if e^l **then** s_1 **else** s_2	$(\textbf{case } \mathsf{P}(e^l), \mathsf{T}(\cdot), \mathsf{C}(\cdot), l) \cup \mathsf{OP}(s_1) \cup \mathsf{OP}(s_2)$
for $(t\ v\ :=\ e^l)\ s$	$\mathsf{OP}(s)$

Figure 4.19: Operations for RBI-DB$^+$.

operations:

$$
\begin{array}{rcl}
\mathit{Operation} & ::= & \textbf{select } e \\
& | & \textbf{case } e \\
& | & \textbf{update } \overline{f} = e \\
& | & \textbf{delete}
\end{array}
$$

The **select** operation retrieves a value. The **case** operation retrieves the value **true** if the given remote expression is true and retrieves **false** otherwise. The **update** operation provides a new value for a given property, and the **delete** operation removes values. The client generates HQL for kind each operation relative to a database table and a filter. The client may also generate an alias for the operation. These four values—operation, database table, filter, and

alias—are collectively known as an *operation tuple*:

$$OperationTuple \in Operation \times Table \times Expression \times AliasName$$

RBI-DB$^+$'s database client computes operation tuples for a given RIL statement, then converts operation tuples to HQL statements. Figure 4.19 describes how the client computes operation tuples. Given an RIL statement s, the client computes its corresponding HQL statements as follows: It first partitions $\mathsf{OP}(s)$ into equivalence classes based on each tuple's operation, table, and filter. The **select** and **case** operations are considered equivalent for this partitioning. The client then converts each equivalence class into an HQL statement, according to the grammar in Figure 4.16.

4.11 Extending RBI-DB$^+$

We now describe extensions to RBI-DB$^+$ that increase its expressiveness. Validity property (V) prohibits a remote computation that depends on a local computation. If a programmer wants this behavior, he or she must manually hoist any pre-local computation outside the batch block and sometimes must manually reforest data between pre-local and remote computations. RBI-DB$^+$ can shoulder this burden, if we extend it.

Extending RBI-DB$^+$ to permit batch blocks that contain pre-local computations requires that we change three RBI-DB$^+$ components: the family of batch methods, the validity analysis, and the program transformation. We must extend the family of batch methods to include the members **LR** and

```
1    batch (BatchService b : new BatchService(...)) {
2        for (RemoteElement e : b.getRemoteCollection()) {
3            this.print(e.process(this.getValue()));
4        }
5    }
```

Figure 4.20: RBI-DB$^+$'s conservative analysis prevents the compiler from partitioning this example. The programmer should communicate that the `print` and `getValue` methods do not modify their receivers.

LRL. The program transformation must be modified to perform a pre-local analogue of the reforestation described in 4.10.

We modify the validity analysis by replacing property (V) with the following property:

(V′) The compiler must be able to partition the block into pre-local, remote, and post-local computations.

This property is satisfied if the LDG does not contain any remote node r and local nodes l_1 and l_2 such that (1) l_1 dominates r and l_2 and (2) l_2 post-dominates r. In other words, there cannot be two path from l_1 to l_2 such that one path contains r and one path does not.

Although property (V′) increases RBI-DB$^+$'s expressiveness, its analysis requires the compiler to reject many useful blocks. Figure 4.20 contains an example. RBI-DB$^+$'s dependence analysis does not examine the body of method calls but conservatively generates a data dependence from the method call **this**`.getValue()` to the use of **this** as a receiver of method `print`, and

vice-versa. The resulting LDG contains a cycle between the remote and local partitions, which violates property IV.

The programmer knows that methods `print` and `getValue` do not modify `this`, but the compiler does not. If the programmer could communicate this information to the compiler, the compiler could avoid generating the conservative dependence and subsequently accept the batch block as valid. The compiler would then reforest the single loop into three loops—one to collect the pre-local results of calling `getValue`, one to collect the remote results of calling `process` with pre-local inputs, and one to print the results of the remote values.

Annotations permit the programmer to communicate such information. There exist many kinds of annotations, lying on a spectrum between two extremes. On one extreme lie annotations that the compiler can verify. For example, the programmer could annotate the definition of method `getValue` as being *pure*, in the sense that it does not contain side effects. The compiler can verify this property, albeit conservatively.

On the other extreme lie annotations that the compiler cannot always verify. For example, the programmer may annotate the expression **this**`.getValue()` as one that does not contain side effects. This kind of annotation acts as a permission slip from the programmer to the compiler. The annotation permits the compiler to re-order the annotated expression as it sees fit.

The two extreme forms of annotations trade safety for expressiveness.

```
1 public void capSalary() {
2     batch<DBBatchClient> (DatabaseService d) {
3         for (Employee emp : d.getEmployees())
4             if (test(emp))
5                 emp.setSalary(75000);
6     }
7 }
8
9 public batchR boolean test(Employee emp) {
10     return emp.getSalary() > 65000;
11 }
```

Figure 4.21: Example Java methods that contain batch blocks.

We believe that more experience with batch programming is required before settling upon an appropriate solution.

4.12 Prototype Implementation

We implemented a prototype RBI-DB^{+} compiler by extending and modifying Ibrahim's compiler [Ibr09]. That compiler is itself an extension of the JastAddJ compiler [EH07]—a modular, attribute-grammar-based Java compiler. Our prototype compiler replaces the analysis in Ibrahim's compiler, includes some extensions for pure methods and pre-local computations, and extends the database client to perform modifications of persistent data.

Figure 4.21 contains an example program. Line 2 declares a batch block. We have modified the syntax so that the batch block is parameterized by the handler class, in this case `DBBatchClient`. Method `test` is a batch method. It's body does not declare a batch block, because the compiler treats

```
1  public void capSalary() {
2      try {
3        // Remote part
4        rbi.ast.Expression ast$ = see Figure 4.23;
5        rbi.runtime.BatchClient.BatchResults results$ =
6          new db.DBBatchClient().executeBatch(ast$);
7        rbi.runtime.CursorIterator emp$28692$Cursor =
8          results$.getCursors().get("emp$28692$Cursor");
9        rbi.runtime.Handle var$2 = results$.getHandles().get("var$2");
10
11       // Post-local part
12       while (emp$28692$Cursor.next()) {
13         if(test$doLocal(results$)){
14           var$2.get();
15         }
16       }
17     } catch(Exception e) { throw new RuntimeException(e); }
18 }
```

Figure 4.22: The reforested version of method `capSalary` in Figure 4.21.

the entire method body as a batch block.

Figure 4.22 contains the reforested version of method `capSalary`. Lines 3–9 contain the remote computation. Line 4 creates an object that represents an RIL abstract syntax tree. Line 5 executes the batch. This line causes the database client to translate the RIL to HQL, execute the query, and populate the result set. Lines 7–9 retrieve values from the result set, to use as input in the post-local computation.

The post-local computation iterates over the cursor that corresponds to the set of retrieved employees. For each value, it calls method `test$doLocal`, which corresponds to the batch method's local computation. If the test suc-

ceeds, then the post-local computation retrieves the value of `var$2`. This computation is unnecessary, because the remote computation has already done the work of updating the database. It may be possible for the compiler to detect this property and avoid generating post-local code in this case.

Figure 4.23 contains the abstract syntax tree for the RIL of method `capSalary`'s remote computation. We implemented several classes for the AST, all located in package `rbi.ast`. Line 3 begins the definition of `capSalary`'s main loop. The string at line 4 corresponds to the loop's container label. The string at line 5 identifies the loop variable's type. Lines 6–9 define the collection expression, and lines 10–23 define the loop body. Of particular interest is the batch method call, defined in lines 13–17. This method call returns an RIL sub-AST, which is used as the condition for the loop's `if` statement. The method takes two arguments: a string that corresponds to the sub-AST's label and list of argument expressions that act as input to the sub-batch.

The sub-batch's behavior is defined Figure 4.24, lines 1–16. Lines 3–6 map the first argument `emp` to a label. Lines 7–13 define the RIL abstract syntax tree that corresponds to the batch method's remote computation. Lines 14–15 use a `DynamicParameterVisitor` to iterate over the AST and replace all occurrences of the method arguments with their actual value, contained in `arg`.

Lines 18–22 of Figure 4.24 contain the local part of the batch method's computation. These lines retrieve from the results the value of label `var$3`, which corresponds to the salary test condition.

```
1  new rbi.ast.Sequence(
2    new rbi.ast.Expression[] {
3        new rbi.ast.Loop(
4            "emp$28692$Cursor",
5             "Employee",
6             new rbi.ast.MethodInvocation("var$0", true,
7                new rbi.ast.RootExpression(),
8                new rbi.ast.Expression[] {},
9                "getEmployees", "DatabaseService"),
10            new rbi.ast.Sequence(
11               new rbi.ast.Expression[] {
12                   new rbi.ast.IfStmt(
13                       test$getRBIExpression("var$1",
14                           new rbi.ast.Expression[] {
15                               new rbi.ast.Ref(
16                                   "emp$28692$Cursor",
17                                   "Employee", false) }),
18                       new rbi.ast.Sequence(
19                           new rbi.ast.Expression[] {
20                              new rbi.ast.MethodInvocation(
21                                   "var$2", true,
22                                    new rbi.ast.Ref(
23                                       "emp$28692$Cursor",
24                                       "Employee",
25                                       false),
26                                    new rbi.ast.Expression[] {
27                                       new rbi.ast.Constant(75000)
28                                    },
29                                   "setSalary",
30                                   "Employee") }),
31                       new rbi.ast.Sequence(
32                           new rbi.ast.Expression[] {}))
33               }))
34   });
```

Figure 4.23: Expression that creates an RIL syntax tree for the remote computation of the example in Figure 4.21.

```
1  public rbi.ast.Expression test$getRBIExpression(String location,
2          rbi.ast.Expression[] args) {
3      java.util.Map<String, rbi.ast.Expression> refMap =
4          new java.util.HashMap<String, rbi.ast.Expression>();
5      // emp->emp$61480
6      refMap.put("emp$61480", args[0]);
7      rbi.ast.Expression exp =
8          new rbi.ast.GreaterThanExpression("var$3", true,
9              new rbi.ast.MethodInvocation("var$4", true,
10                 new rbi.ast.Ref("emp$61480", "Employee", false),
11                 new rbi.ast.Expression[] {},
12                 "getSalary", "Employee"),
13             new rbi.ast.Constant(65000));
14     return exp.accept(
15         new rbi.runtime.DynamicParameterVisitor(), refMap);
16 }
17
18 public boolean test$doLocal(
19         rbi.runtime.BatchClient.BatchResults results$){
20     rbi.runtime.Handle var$3 = results$.getHandles().get("var$3");
21     return ((java.lang.Boolean)var$3.get());
22 }
```

Figure 4.24: The remote and local computations for batch method `test` from Figure 4.21 .

When a program calls method `salaryCap` in Figure 4.10, the database client executes the following two HQL queries:

```
update Employee as emp$28692$Cursor
set emp$28692$Cursor.salary = 75000
where (1 = 1) and (emp$28692$Cursor.salary > 65000)

select emp$28692$Cursor.salary as var$4,
       case when (emp$28692$Cursor.salary > 65000)
                then true else false end as var$3
from Employee as emp$28692$Cursor
where (1 = 1) and (emp$28692$Cursor.salary > 65000)
```

The first finds every employee whose salary is greater than 65000 and sets a new salary of 75000. The second query retrieves the salary and the value of the conditional for each employee whose salary is greater than 65000. The post-local computation uses the results of this query. As we noted, these results are not strictly needed, because the program only performs updates and does nothing with the returned database values. Our prototype implementation performs the correct behavior, but it could be optimized to remove unnecessary work and simplify the queries.

4.13 Related Work

Remote Batch Invocation contributes a generic mechanism for partitioning and reforesting distributed programs that intermingle client and server computations. The published papers on RBI discuss its role in the context of distributed computing [TCJ09, IJCT09, IJI+09, Ibr09]. Our work extends Remote Batch Invocation in three ways: (1) we add batch methods, which permit

programmers to parameterize and compose batch blocks, (2) we modify RBI to rely on a more general, dependence-based analysis to partition and reforest batch blocks; and (3) we extend the database client to permit programmers to express a wider variety of database operations, including modifications. In this section, we discuss how these extensions and modifications relate to prior research.

Links is a functional web-programming language that partitions user code into client, server, and database computations [CLWY06]. Its design goals are not the same as RBI's. Links is intended to provide a single programming language that programmers can use to build interactive Web applications. As such, the Links language and compiler provides no sophisticated mechanisms for intermingling and composing client-server computations. Links permits programmers to intermingle client and server computation only at the function level. The Links compiler cannot compose abstractions across function boundaries. As a result, programmers must partition their programs into separate client and server computations, and they cannot compose database operations without sacrificing performance. Links programmers are able to express some database aggregation operations, but they are not able to express database modifications.

The *multi-tier calculus* is a way of reasoning about distributed computation [NT05]. Its goal is to decompose a single, sequential computation into many distributed computations. The distributed computations' aggregate semantics are equivalent to that of the original, sequential version. Although the

goals of the multi-tier calculus differ from those of RBI, its techniques are similar to ours. The calculus is defined formally. Its type-based location analysis and program transformation are similar to ours. Because its goal is to transform entire programs, it makes no guarantees about latency. A transformed program's distributed computations are free to communicate as many times as necessary for the program to complete; although the transformation does attempt to minimize the number of communications. Part of the analysis and transformation are therefore less complex than the ones presented here. For example, the multi-tier calculus does not need to consider data dependences among locations, nor does it model loops.

Thor is an object-oriented database system whose implementation includes batching optimizations to reduce the system's latency overhead [LAC+96]. Thor's *batched future* is a runtime mechanism that delays a program's retrieval of remote objects until the program requires their values [BL94]. The Thor runtime sends a list of the program's data traversals to the database, which is responsible for retrieving the program's data. *Batched control structures* extend this descriptive capability to include loops and conditions, but not method invocations [Zon95]. *Basic value promises* are analogous to RBI's mobile values, and they permit Thor to delay their retrieval until the local program needs their values [Zon95]. Thor's runtime optimizations do not partition and reforest intermingled code. The runtime triggers batch execution based on data types, rather than data dependences.

Yeung and Kelly describe a runtime mechanism similar to Thor's, but

which permits more intermingling of client and server computations [YK03]. Their mechanism profiles a running program and delays any RMI calls until the program requires their results. The mechanism distinguishes between remote and local computation within a single method. Whenever a local computation needs the result of a remote computation, the mechanism triggers a batch RMI call. Yeung and Kelly's work thus improves an RMI application's latency behavior. Their mechanism does not modify code, so it permits less efficient intermingling of client and server computations than does RBI-DB$^+$.

Several systems employ static—and sometimes dynamic—analyses to partition distributed computations. CAGES [Ham76] combines static analysis of inter-module communication combined with runtime information to help programmer partition an application between a host and satellite computers. ICOPS [MvD76] performs a similar task using dynamic profiling. JavaParty [PZ97] and J-Orchestra [TS09] use programmer-supplied type annotations to automatically partition an application's classes among distributed computation units.

Pangea has similar goals but accomplishes them by constructing a static object graph that summarizes the relationships and communication patterns among a program's objects. This structure captures information about immutability and privacy to determine objects which may be replicated at many sites and objects among which no runtime communication can occur. Pangea uses this information to assign objects to computation units in a way that attempts to minimize communication. Other research projects have

extended this idea [DFO06] or employed dynamic analysis towards similar ends [HS99, WF08].

This line of research allows for a more complex distribution pattern than our client-server one. Its goal is to automatically distribute parallel programs. Whereas these systems seek to create a distribution with desired communication properties, RBI essentially verifies that a program conforms to a particular distribution and a given communication property.

Swift employs techniques similar to RBI, but in the domain of secure web applications [CLM+07]. Swift programmers annotate their programs with flow policies. The Swift compiler uses a static analysis to partition the program into a new one that obeys the programmer's annotations and attempts to minimize client-server communication. Zdancewic et al. describe a similar approach to partition general-purpose programs [ZZNM02].

RPC chains are a form of continuation-passing-style programming in which programmers express RPC calls among multiple sites [SAKM09]. RPC chains are similar to batch methods in that the programmer writes modular systems without sacrificing performance. They differ from our work in that they forward computation to many remote locations. They do not permit programmers to intermingle remote and local computations without sacrificing performance.

The goals and techniques of high-performance computing are similar to those of RBI. Both lines of research employ static, dependence-based analysis

to improve the performance of distributed applications. Optimizing compilers for high-performance computing tend to focus on analyzing array accesses, to determine which array indices carry loop dependences. Good optimization of array-based loops requires sophisticated analyses to determine the relationships among these expressions [Ban88]. By contrast, RBI need only examine the relationships between locations, so it can achieve good results with more coarse-grained analyses.

4.14 Conclusions and Future Work

We have formalized RBI-DB$^+$, an extension to Remote Batch Invocation for databases. The extension and its prototype implementation have several desirable properties: (1) programmers can parameterize and compose batches, (2) programmers can express a variety of database operations, and (3) future researchers can extend RBI, thanks to its dependence-based analysis.

We have focused our efforts on placing RBI upon a solid foundation from which future researchers can extend it and on designing and implementing a database client for RBI that optimizes modifications of persistent data. We have not provided a full evaluation of this client's performance. Ibrahim et al. show that the performance of modular RBI programs is comparable to hand-written applications for RMI and web services clients [IJCT09, IJI$^+$09]. We believe that we can achieve the same results for our database client. In particular, we intend to evaluate RBI-DB$^+$against query extraction, pure transparent persistence, and explicit queries.

We do not know if our client-server dependence analysis will scale to multiple partitions. It would be an interesting pursuit to extend the dependence analysis to an arbitrary number of locations and compare the results against other general-purpose, distributed computing partitioning like JavaParty, J-Orchestra, and Zdancewic et al.'s work.

The requirement that batch blocks execute only one round trip communication may be too restrictive for some kinds of programs. Instead, programmers could parameterize batch blocks by a given number of round trip communications. This feature would make batch blocks more expressive and would fill in the spectrum between the multi-tier calculus, which permits an arbitrary number of round trips, and our approach. To effect this change, we would need to modify the syntax of batch blocks to take the number of round trips as a parameter. We would also need to modify the validity analysis to verify that the batch block conformed to its latency specification. Modifying the validity analysis is not difficult, because validity is expressed as a query over the location dependence graph.

Whereas query extraction is more transparent, RBI-DB$^+$ gives programmers a more predictable performance model and optimizes persistent data modifications. Neither approach detects or optimizes insertions or aggregations, which are optimized by less modular integration solutions like LINQ. We believe our approach is orthogonal to LINQ and that it is possible to design an integration solution that combines the best of both approaches. We outline this solution in the next chapter.

Chapter 5

Conclusions and Future Work

We have identified an unsolved problem for impedance mismatch: Today's programmers must abandon procedural abstraction to obtain good performance from programs that access databases. We have contributed two techniques that address this problem. Our techniques enable programmers to write modular, safe and concise programs that use databases.

Prior research on integrating programming languages and databases has focused on safe and concise programs, to the exclusion of modularity. These efforts have integrated the different data-types from the two domains. Industry has adopted these techniques, and today's programmers benefit from data-type integration.

It is our hope that industry eventually will adopt techniques that permit, rather than prohibit, efficient procedural abstraction. The techniques in this dissertation compliment contemporary data-type integration techniques, so the two can be combined. We believe the result will be a robust, practical solution to impedance mismatch.

Query extraction may be the easier technique to transfer, because it does not require any changes to languages like Java and C#. The prototype

implementation we presented in Chapter 3 can serve immediately as a preprocessor to Java programs. If virtual machine developers want to embed query extraction, then they will need to re-design the analysis to operate on bytecode, rather than the abstract syntax tree. This redesign accommodates dynamic class loading. It also accommodates the devirtualization assumptions described in Section 3.5. Alternatively, a virtual machine can implement polyvariant specialization to handle virtual method calls.

RBI-DB$^+$ is a more robust integration technique, but it requires changes to introduce batch blocks and methods to languages like Java and C#. RBI-DB$^+$ lacks the capabilities to perform two kinds of database operations: insertions and aggregations. LINQ has excellent support for aggregations, but does not support procedural abstraction for queries. We recommend adding batch blocks and methods to C#. C# programmers can then take advantage of RBI to express bulk data retrievals and modifications and take advantage of LINQ's existing aggregation capabilities. Java soon will have closures, which developers can leverage to express aggregations similar to LINQ's. If Java were to add batch blocks and methods, it could compete with LINQ's expressiveness.

Whatever changes industry adopts, we hope that they free tomorrow's programmers from worrying about performance, so that programmers may concentrate on designing, developing, debugging and maintaining some of society's most critical systems.

Appendices

Appendix A

Soundness Result for a Subset of Query Extraction [1]

This appendix contains a proof of soundness for intraprocedural query extraction on a subset of the Java language.

The language's abstract syntax is defined in Figure A.1. The traversal expression $e.f$ projects a field f of a record e. The value of $e.f$ can be a simple value, or references to one or more records. Persistent data is introduced through a special `root` variable that refers to a record representing persistent data. Primitive functions $\mathtt{op}_n$ have a specified number of arguments n. Infix notation is used where appropriate. The `for` command allows iteration over the elements of a collection. A simple static type system for records is assumed for this language [Pie02]; programs are assumed to be well typed.

A.1 Values and Concrete Semantics

A program operates over values in the domain:

$$v \in \mathit{Value} = \mathit{Basic} + \mathit{RecordID} + \overline{\mathit{RecordID}}$$

[1] This appendix is published research [WC07].

$$
\begin{array}{rcl}
 & l \in \textit{Variable} \qquad f \in \textit{Field} & \\
e \in \textit{Expression} & ::= & l \mid e.f \mid \mathsf{op}_n(e_1, \ldots, e_n) \\
\mathsf{op}_0 \in \textit{Constant} & ::= & \textbf{true} \mid \textbf{false} \mid \textit{number} \mid \textit{string} \\
\mathsf{op}_1 & ::= & \neg \mid \textbf{print} \\
\mathsf{op}_2 & ::= & \wedge \mid \vee \mid > \mid < \mid = \mid \geq \mid \leq \mid \neq \\
c \in \textit{Command} & ::= & \textbf{skip} \mid l := e \mid c; c \\
 & \mid & \textbf{if } e \textbf{ then } c \ [\textbf{else } c] \\
 & \mid & \textbf{for } l \textbf{ in } e \textbf{ do } c
\end{array}
$$

Figure A.1: Syntax of a persistent data kernel language.

where *Basic* is the domain of basic values (integers, Booleans, and strings), and *RecordID* is the domain of *record identifiers* that reference persistent database values. When a program traverses a record identifier, a runtime function *Load* :: *RecordID* $\times$ *Field* $\rightarrow$ *Value* retrieves the corresponding record's field value(s). A special record identifier r_0 corresponds to the persistent store's root, and the store's structure is the graph formed by the transitive closure of traversals from r_0.

Our analysis summarizes the set of persistent values a program uses. These values—which we refer to as the program's *used-set*—can then be loaded in bulk before the program needs them. We define the operational semantics of the base language in a typical way in Figure A.2, but extend the semantics to keep track of a computation's used-set.

The operational semantics has evaluation relations $\langle e, \sigma \rangle \rightarrow \langle v, \rho \rangle$ and $\langle c, \sigma \rangle \rightarrow \langle \sigma, \rho \rangle$ where ρ is the set of database values that were loaded during the entire computation. U-Traverse is the only rule that loads database

$$\langle l, \sigma \rangle \rightarrow \langle \sigma[l], \emptyset \rangle \qquad \text{(U-VAR)}$$

$$\langle \textbf{skip}, \sigma \rangle \rightarrow \langle \sigma, \emptyset \rangle \qquad \text{(U-SKIP)}$$

$$\frac{\langle e, \sigma \rangle \rightarrow \langle r, \rho_e \rangle \quad \rho_f = \mathit{Load}(r, f)}{\langle e.f, \sigma \rangle \rightarrow \langle \rho_f, \rho_e \cup \rho_f \rangle} \qquad \text{(U-TRAVERSE)}$$

$$\frac{\langle e_i, \sigma \rangle \rightarrow \langle v_i, \rho_i \rangle \quad \textbf{for } i \in \{1, \ldots, n\}}{\langle \mathsf{op}_n(e_1, \ldots, e_n), \sigma \rangle \rightarrow \langle f_{\mathsf{op}_n}(v_1, \ldots, v_n), \cup \rho_i \rangle} \qquad \text{(U-OP)}$$

$$\frac{\langle c_1, \sigma \rangle \rightarrow \langle \sigma', \rho_1 \rangle \qquad \langle c_2, \sigma' \rangle \rightarrow \langle \sigma'', \rho_2 \rangle}{\langle c_1;\, c_2, \sigma \rangle \rightarrow \langle \sigma'', \rho_1 \cup \rho_2 \rangle} \qquad \text{(U-SEQ)}$$

$$\frac{\langle e, \sigma \rangle \rightarrow \langle v, \rho_e \rangle}{\langle l{:=}e, \sigma \rangle \rightarrow \langle [l \mapsto v]\sigma, \rho_e \rangle} \qquad \text{(U-ASSIGN)}$$

$$\frac{\langle e, \sigma \rangle \rightarrow \langle \mathit{true}, \rho_e \rangle \qquad \langle c_1, \sigma \rangle \rightarrow \langle \sigma', \rho_c \rangle}{\langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \sigma \rangle \rightarrow \langle \sigma', \rho_e \cup \rho_c \rangle} \qquad \text{(U-IFT)}$$

$$\frac{\langle e, \sigma \rangle \rightarrow \langle \mathit{false}, \rho_e \rangle \qquad \langle c_2, \sigma \rangle \rightarrow \langle \sigma', \rho_c \rangle}{\langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \sigma \rangle \rightarrow \langle \sigma', \rho_e \cup \rho_c \rangle} \qquad \text{(U-IFF)}$$

$$\frac{\langle e, \sigma \rangle \rightarrow \langle \{r_1, \ldots, r_n\}, \rho_1 \rangle \quad \langle c, [l \mapsto r_i]\sigma_i \rangle \rightarrow \langle \sigma_{i+1}, \rho_{i+1} \rangle \quad \textbf{for } i \in \{1, \ldots, n\}}{\langle \textbf{for } l \textbf{ in } e \textbf{ do } c, \sigma_1 \rangle \rightarrow \langle \sigma_{n+1} \backslash l, \cup \rho_i \rangle} \qquad \text{(U-FOR)}$$

Figure A.2: Typical operational semantics, extended to collect used-sets.

values, so the rule adds the newly loaded values ρ_f to the set. All other rules collect the loaded values for any sub-computation, where $\bigcup \rho_i$ is shorthand for $\bigcup_{i=1}^{n} \rho_i$.

A.2 Conditional Paths, Abstract Values, and Abstract Semantics

A *conditional path* $p[k]$ represents a query of the database for values located at path p for which the condition k is true. The condition is expressed as an operation on abstract values, including other paths:

$$\begin{aligned} k \in \mathit{Condition} &::= \mathsf{op}_n^t(\hat{v}_1, \ldots, \hat{v}_n) \\ cp \in \mathit{CPath} &::= p[k] \ \mid \ p[k]^{\downarrow} \\ \hat{v} \in \widehat{\mathit{Value}} &::= \wp(\mathit{CPath}) + \wp(\mathit{Condition}) + \top \end{aligned}$$

A path marked $p[k]^{\downarrow}$ is involved in a data dependence. A non-conditional path p is lifted to a conditional path $p[\mathit{true}]$ signifying that the path is always traversed. The label t uniquely identifies an operator. Conditions are restricted to include only one occurrence of a labeled operator, allowing the domain *CPath* to be finite.

Figure A.3 computes the conditional paths a program traverses. The evaluation relation $k, I \vdash \langle e, \hat{\sigma}\rangle \hat{\rightharpoonup} \langle \hat{v}, \pi\rangle$ carries a context that consists of the condition k under which traversals may take place and the *collection traversal list* I. This list represents the nesting structure of collection traversals and is used to identify query conditions. The first element of the list is the set of

$$\frac{\begin{array}{c} e \text{ contains no loop-carried dependences} \\ k, I \vdash \langle e, \hat{\sigma}\rangle \mathrel{\hat{\rightarrow}} \langle \hat{v}, \pi_e\rangle \\ \mathit{Distinct}(\mathit{Paths}(\hat{v})) \qquad \mathit{Trim}(\mathit{Paths}(\hat{v})) \subseteq I \\ (k \wedge \hat{v}), I \vdash \langle c_1, \hat{\sigma}\rangle \mathrel{\hat{\rightarrow}} \langle \hat{\sigma}_1, \pi_1\rangle \\ (k \wedge \neg\hat{v}), I \vdash \langle c_2, \hat{\sigma}\rangle \mathrel{\hat{\rightarrow}} \langle \hat{\sigma}_2, \pi_2\rangle \\ \pi' = \pi_e \sqcup \pi_1 \sqcup \pi_2 \end{array}}{k, I \vdash \langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \hat{\sigma}\rangle \mathrel{\hat{\rightarrow}} \langle \hat{\sigma}_1 \sqcup \hat{\sigma}_2, \pi'\rangle} \quad (\text{K-IF}_1)$$

$$\frac{\begin{array}{c} k, I \vdash \langle e, \hat{\sigma}\rangle \mathrel{\hat{\rightarrow}} \langle \pi_e, \pi\rangle \\ \pi_f = \begin{cases} \top & f^t \in \pi_e \\ \{p.f^t[k] \mid p \in \pi_e\} & f^t \notin \pi_e \end{cases} \end{array}}{k, I \vdash \langle e.f^t, \hat{\sigma}\rangle \mathrel{\hat{\rightarrow}} \langle \pi_f, \pi \sqcup \pi_f\rangle} \quad (\text{K-TRAVERSE})$$

$$\frac{\begin{array}{c} k, I \vdash \langle e_i, \hat{\sigma}\rangle \mathrel{\hat{\rightarrow}} \langle \hat{v}_i, \pi_i\rangle \qquad \textbf{for } i \in \{1, \ldots, n\} \\ \hat{v} = \begin{cases} \top & \mathsf{op}_n^t \in \hat{v}_i \\ \mathsf{op}_n^t(\hat{v}_1, \ldots, \hat{v}_n) & \mathsf{op}_n^t \notin \hat{v}_i \end{cases} \end{array}}{k, I \vdash \langle \mathsf{op}_n^t(e_1, \ldots, e_n), \hat{\sigma}\rangle \mathrel{\hat{\rightarrow}} \langle \hat{v}, \bigsqcup \pi_i\rangle} \quad (\text{K-OP})$$

$$\frac{\begin{array}{c} \text{K-IF}_1 \text{ does not apply} \\ k, I \vdash \langle c_1, \hat{\sigma}\rangle \mathrel{\hat{\rightarrow}} \langle \hat{\sigma}_1, \pi_1\rangle \\ k, I \vdash \langle c_2, \hat{\sigma}\rangle \mathrel{\hat{\rightarrow}} \langle \hat{\sigma}_2, \pi_2\rangle \\ \pi' = \pi_e \sqcup \pi_1 \sqcup \pi_2 \end{array}}{k, I \vdash \langle \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \hat{\sigma}\rangle \mathrel{\hat{\rightarrow}} \langle \hat{\sigma}_1 \sqcup \hat{\sigma}_2, \pi'\rangle} \quad (\text{K-IF}_2)$$

$$\frac{\begin{array}{c} k, I \vdash \langle e, \hat{\sigma}\rangle \mathrel{\hat{\rightarrow}} \langle \pi_e, \pi\rangle \\ \pi_\iota = \{p.\iota^l \mid p \in \pi_e\} \\ I' = \mathit{Extend}(I, \pi_\iota) \\ (\hat{\sigma}', \pi') = \bigsqcup\{(\textbf{do}\langle k, I', c, l, \pi_\iota\rangle)^n(\hat{\sigma}, \emptyset) \mid n \in \mathbb{N}\} \end{array}}{k, I \vdash \langle \textbf{for } l \textbf{ in } e \textbf{ do } c, \hat{\sigma}\rangle \mathrel{\hat{\rightarrow}} \langle \hat{\sigma}' \backslash l, \pi \sqcup \pi_\iota \sqcup \pi'\rangle} \quad (\text{K-FOR})$$

$$\frac{k, I \vdash \langle e, \hat{\sigma}\rangle \mathrel{\hat{\rightarrow}} \langle \hat{v}, \pi\rangle}{k, I \vdash \langle l{:=}e, \hat{\sigma}\rangle \mathrel{\hat{\rightarrow}} \langle [l \mapsto \hat{v}] \sqcup \hat{\sigma}, \pi^{\downarrow} \sqcup I_{|I|}[k]^{\downarrow}\rangle} \quad (\text{K-ASSIGN})$$

$$\frac{k, I \vdash \langle c, [l \mapsto \pi_\iota] \sqcup \hat{\sigma}\rangle \mathrel{\hat{\rightarrow}} \langle \hat{\sigma}', \pi'\rangle}{\textbf{do}\langle k, I, c, l, \pi_\iota\rangle(\hat{\sigma}, \pi) = (\hat{\sigma}', \pi \sqcup \pi')} \quad (\text{K-DO})$$

$$\frac{\forall p_1, p_2 \in \pi : \mathit{Erase}(p_1) = \mathit{Erase}(p_2) \Rightarrow p_1 = p_2}{\mathit{Distinct}(\pi)}$$

$$\begin{aligned} \mathit{Erase}(\bar{f}_1.\iota^{l_1}.\cdots.\iota^{l_n}.\bar{f}_{n+1}) &= \bar{f}_1.\cdots.\bar{f}_{n+1} \\ \mathit{Trim}(\pi) &= \{p.\iota^l \mid p.\iota^l.\bar{f} \in \pi\} \\ \mathit{Paths}(\pi) &= \pi \\ \mathit{Paths}(\mathsf{op}_n(\hat{v}_1, \ldots, \hat{v}_n)) &= \mathit{Paths}(\hat{v}_1) \sqcup \cdots \sqcup \mathit{Paths}(\hat{v}_n) \end{aligned}$$

$$\mathit{Extend}(I, \pi_\iota) = \begin{cases} \pi_\iota & |I| = 0 \\ I,\ \pi_\iota & \mathit{Prefixes}(I_{|I|}, \pi_\iota) \\ I & \text{otherwise} \end{cases}$$

$$\frac{\forall p_2 \in \pi_2 : (\exists p_1 \in \pi_1, \exists p' \in \mathit{Field}^* : p_1.p' = p_2)}{\mathit{Prefixes}(\pi_1, \pi_2)}$$

Figure A.3: Abstract interpretation with paths and conditions.

paths for the outermost loop variable, and the last element is the set of paths for the innermost (current) loop variable. If list I has length $|I|$, then $I_{|I|}$ denotes the innermost loop variable paths. Initially k is set to *true*, and I is empty.

The abstract semantics has evaluation relations for expressions $\langle e, \hat{\sigma} \rangle \hat{\rightarrow} \langle \hat{v}, \pi \rangle$ and commands $\langle c, \hat{\sigma} \rangle \hat{\rightarrow} \langle \hat{\sigma}', \pi \rangle$, where $\hat{v}$ is an abstract value, $\hat{\sigma}$ maps variables to abstract values, and π is the set of paths traversed by a computation.

Rule K-IF$_1$ identifies query conditions. The analysis first determines that e contains no loop-carried dependences. This determination is the result of a standard analysis; for brevity, we omit its details.

The premise $\mathit{Distinct}(\mathit{Paths}(\hat{v}))$ ensures that all the paths in the condition's abstract value are *distinct*, in the sense that they do not traverse the same set of fields with different iteration field names.

The analysis also checks that all the paths in the expression are based on lexically enclosing iteration paths. The function *Trim* applied to a set of paths π returns all possible paths for the inner-most loop variable. Thus, the premise $\mathit{Trim}(\mathit{Paths}(\hat{v})) \subseteq I$ ensures that any iteration paths that appear in the condition's abstract value are based on lexically enclosing iteration paths.

The lexically enclosing iterations needed by K-IF$_1$ are created by K-FOR. The rule appends a new set of paths π_ι to the list of iteration paths I only if all paths in π_ι extend some path in $I_{|I|}$, the list's most recently added member. In this way, K-FOR maintains the constraint that I is a list

of lexically enclosing iteration paths. Rule K-FOR relies on rule K-DO to compute the fixed-point of the abstract computation on the loop body.

Query conditions are used in the true and false branches of the `if` command. Given the abstract value $\hat{v}$ of condition e, the true-branch body is evaluated under the condition $k \wedge \hat{v}$ and the false-branch body under the condition $k \wedge \neg\hat{v}$. When the program makes a traversal, rule K-TRAVERSE attaches the condition k to the path generated by the traversal.

If the condition does not satisfy the requirements of a query condition, rule K-IF$_2$ does not augment the paths with conditions.

Rule K-ASSIGN performs assignments but also marks all paths in the bound expression as having data dependences. $\pi^{\downarrow}$ means the marking of all paths in π with $^{\downarrow}$, and $I_{|I|}[k]^{\downarrow}$ means marking all paths in $I_{|I|}$ with condition k and $^{\downarrow}$. The loop variable paths themselves are data dependent, because the execution of any assignment can depend upon the *existence* of an element in an iteration, even if no fields of the loop variable are used. For example, in the program `for x in p do y := y + 1`, the variable `x` is never used, yet there is still a data dependence upon it because its elements must be enumerated.

Rule K-OP defines the semantics of operations on abstract values. The operands are evaluated and the operator is retained in the result. The rule also includes a widening clause to ensure convergence to a fixed-point. The rule is similar to the one for traversals: If the syntactic use of the operator $\texttt{op}_n$ already occurs in some $\hat{v}_i$, then the expression evaluates to $\top$.

$$
\begin{aligned}
CLoad_i(r, \epsilon[k], \phi) &= \begin{cases} \{r\} & true \sqsubseteq eval(k, \phi) \\ \emptyset & \text{otherwise} \end{cases} \\
CLoad_i(r, f.p[k], \phi) &= \bigcup_{r' \in Load(r,f)} CLoad_i(r', p[k], \phi) \\
CLoad_1(r, \iota^l.p[k], \phi) &= CLoad_1(r, p[k], [\iota^l \mapsto r]\phi) \\
CLoad_2(r, \iota^l.p[k], \phi) &= CLoad_2(\phi(\iota^l), p[k], \phi)
\end{aligned}
$$

$$
\begin{aligned}
eval(p[k], \phi) &= \bigsqcup CLoad_2(\mathsf{r}_0, p[k], \phi) \\
eval(\mathsf{op}_n(\hat{v}_1 \ldots \hat{v}_n), \phi) &= f'_{\mathsf{op}_n}(eval(\hat{v}_i, \phi), \ldots, eval(\hat{v}_n, \phi)) \\
eval(S, \phi) &= \bigsqcup_{\hat{v} \in S} eval(\hat{v}, \phi)
\end{aligned}
$$

Figure A.4: Conditional record loading.

A.3 Proof of Soundness

Our soundness proof shows that the abstract semantics load a superset of the data required by the concrete semantics. The concrete load operation for conditional paths is defined in Figure A.4. Function $CLoad_i$ loads all records reachable by a path p, provided the path's condition k may be *true*. A mapping ϕ binds an iterator field name to a specific record identifier, to be referenced in the evaluation of the path's condition. $CLoad_1$ creates iterator field name bindings, and $CLoad_2$ uses the bindings.

Function *eval* defines condition evaluation. Operator evaluation calls

$eval$ on the operands and applies f'_{op_n} to the results, where

$$f'_{\mathsf{op}_n}(\hat{v}_1, \ldots, \hat{v}_n) = \begin{cases} \top & \top \in \{\hat{v}_1, \ldots, \hat{v}_n\} \\ f_{\mathsf{op}_n}(\hat{v}_1, \ldots, \hat{v}_n) & \text{otherwise} \end{cases}$$

Note that because the underlying operators are monotonic, all functions f' are also monotonic.

Path evaluation calls $CLoad_2$ on the path, passing bindings ϕ for any iterator field names that appear in the path. Note that, because the evaluated path appears only in an operation expression, the result of path evaluation must be a set that contains a single basic value. The least upper bound operation ($\bigsqcup$) retrieves this value from the set. Evaluating a set of abstract values yields the least upper bound of evaluating each value in the set.

A set π of conditional paths safely approximates the persistent values a program loads if it describes a superset of those values:

$$\begin{aligned} \rho \; \mathcal{R} \; \pi \quad &\Leftrightarrow \quad \rho \subseteq \bigcup_{p[k] \in \pi} CLoad_1(\mathsf{r_0}, p[k], \emptyset) \\ (v, \sigma) \; \mathcal{R} \; \hat{v} \quad &\Leftrightarrow \quad \begin{cases} \{v\} \; \mathcal{R} \; \hat{v} & v = r, \hat{v} = \pi \\ v \; \mathcal{R} \; \hat{v} & v = \{r_1, ..., r_n\}, \hat{v} = \pi \\ v \sqsubseteq eval(\hat{v}, \phi_\sigma) & v \in Basic, \hat{v} = k \\ \hat{v} = \top & \text{otherwise} \end{cases} \\ \sigma \; \mathcal{R} \; \hat{\sigma} \quad &\Leftrightarrow \quad \forall x \in Dom(\sigma) \cap Dom(\hat{\sigma}).(\sigma[x], \sigma) \; \mathcal{R} \; \hat{\sigma}[x] \end{aligned}$$

The relation between concrete and abstract values is defined only in the context of a store, because the store provides a binding for any iterator

field names that may appear in paths and conditions. When $\hat{v}$ is an abstract operation, evaluating $\hat{v}$ must approximate v. If L is the set of loop variables that appear in the entire program, $\phi_\sigma = \bigcup_{l \in L} [\iota^l \mapsto \sigma[l]]$, where $\sigma[l] = \top$ if $\sigma[l]$ is undefined.

The full proof [WC07] contains a standard subproof that query extraction is sound for paths without conditions. We include here only an extension of that proof that shows expression evaluation gives the same results, assuming that evaluating every path condition may give the value true. We then show that the analysis only constructs conditions that satisfy this assumption. Proof of soundness for command evaluation follows trivially. In this summarized proof, we appeal to the full proof where necessary.

Lemma 1 (Subcomputation compatibility). If $(\rho_1 \mathrel{\mathcal{R}} \pi_1)$ and $(\rho_2 \mathrel{\mathcal{R}} \pi_2)$, then $(\rho_1 \cup \rho_2) \mathrel{\mathcal{R}} (\pi_1 \sqcup \pi_2)$.

Proof. If $\pi_1 \sqcup \pi_2 = \top$, then the relation trivially holds. Otherwise, by the definition of $\mathcal{R}$, $\rho_1 \subseteq \bigcup_{p[k] \in \pi_1} \mathit{CLoad}_1(\mathsf{r_0}, p[k], \emptyset)$ and $\rho_2 \subseteq \bigcup_{p[k] \in \pi_2} \mathit{CLoad}_1(\mathsf{r_0}, p[k], \emptyset)$. Assuming all conditions k may be true, $\rho_1 \cup \rho_2 \subseteq \bigcup_{p[k] \in \pi_1 \cup \pi_2} \mathit{CLoad}_1(\mathsf{r_0}, p[k], \emptyset)$, so the relation holds. □

Theorem 1 (Soundness of expression evaluation). For all e, σ, $\hat{\sigma}$, k:

$$\frac{\langle e, \sigma \rangle \rightarrow \langle v, \rho \rangle \qquad k, I \vdash \langle e, \hat{\sigma} \rangle \hat{\rightarrow} \langle \hat{v}, \pi \rangle \qquad \sigma \mathrel{\mathcal{R}} \hat{\sigma} \qquad \mathit{true} \sqsubseteq \mathit{eval}(k, \phi_\sigma)}{((v, \sigma), \rho) \mathrel{\mathcal{R}} (\hat{v}, \pi)}$$

Proof. By induction on the structure of e.

Base case $e \equiv [\![l]\!]$ In this case, $(v, \rho) = (\sigma[l], \emptyset)$ and $(\hat{v}, \pi) = (\hat{\sigma}[l], \emptyset)$ The premise $\sigma\ \mathcal{R}\ \hat{\sigma}$ gives the desired result.

The induction hypothesis asserts that evaluating subexpressions under condition k produces sound results. It remains to show that evaluating operators and traversals under condition k produces sound results.

Case $e \equiv [\![\mathsf{op}^t_n(e_1 \ldots e_n)]\!]$ If the abstract semantics gives $\hat{v} = \top$ for e, then this case is trivially proved. Otherwise, it must be shown that if, for each e_i, the concrete semantics gives (v_i, ρ_i) and the abstract semantics gives $(\hat{v}_i, \pi_i)$, then:

$$\begin{array}{c} f_{\mathsf{op}_n}(v_1, \ldots, v_n) = f'_{\mathsf{op}_n}(v_1, \ldots, v_n) \\ \sqsubseteq \\ eval(\mathsf{op}_n(\hat{v}_1, \ldots, \hat{v}_n), \phi_\sigma) = f'_{\mathsf{op}_n}(eval(\hat{v}_i, \phi_\sigma), \ldots, eval(\hat{v}_n, \phi_\sigma)) \end{array}$$

Because f' is monotonic, it suffices to show that if $(v_i, \sigma)\ \mathcal{R}\ \hat{v}_i$, then $v_i \sqsubseteq eval(\hat{v}_i, \phi_\sigma)$. If v_i is a basic value, then the definition of $\mathcal{R}$ suffices. It remains to show that if v_i is a record identifier,

$$v_i \sqsubseteq \bigsqcup_{p[k] \in \hat{v}_i} \left\{ \bigsqcup CLoad_2(\mathsf{r}_0, p[k], \phi_\sigma) \right\}$$

Because $v_i\ \mathcal{R}\ \hat{v}_i$, there exists some paths $\pi' \subseteq \hat{v}_i$ such that $v_i \in CLoad_1(\mathsf{r}_0, p', \emptyset)$, where $p' \in \pi'$. Calling $CLoad_1$ on these paths generates of set of iterator field bindings Φ that includes ϕ_σ; therefore $r \in \bigcup_{p[k] \in \hat{v}} CLoad_2(\mathsf{r}_0, p[k], \phi_\sigma)$. Hence, the desired result that evaluating all paths in $\hat{v}_i$ with bindings ϕ_σ approximates r. Lemma 1 gives $\bigcup \rho_i\ \mathcal{R}\ \bigsqcup \pi_i$.

Case $e \equiv [\![e.f^t]\!]$ Rules U-Traverse and K-Traverse and the induction hypothesis give $(r, \rho_e)\ \mathcal{R}\ (\pi_e, \pi)$ for the subexpression e. For the entire expression, the rules give $\rho_f = \mathit{Load}(r, f)$, $\pi_f = \{p.f^t[k] \mid p \in \pi_e\}$. If $\mathit{true} \sqsubseteq \mathit{eval}(k, \phi_\sigma)$, then $\pi_f = \{p.f^t \mid p \in \pi_e\}$, which is sound [WC07]. Lemma 1 gives $(\rho_e \cup \rho_f)\ \mathcal{R}\ (\pi \sqcup \pi_f)$. □

Theorem 2 (Condition evaluation approximates *true*). For all σ, $\hat{\sigma}$, conditions k produced by the analysis:

$$\frac{\sigma\ \mathcal{R}\ \hat{\sigma}}{\mathit{true} \sqsubseteq \mathit{eval}(k, \phi_\sigma)}$$

Proof. By induction on the structure of k.

Base case $k = \mathit{true}$ Trivial, because $\mathit{eval}(\mathit{true}, _) = \mathit{true}$.

The induction hypothesis asserts that evaluating subconditions approximates *true*. It remains to prove the theorem for any condition k' the analysis creates.

Case $k' \equiv [\![k \wedge \hat{v}]\!]$**,** $\hat{v}$ **is a query condition** In this case, the analysis attaches k' to all paths generated by the true-branch of an **if**. So, it must be shown:

$$\frac{\langle e, \sigma\rangle \rightarrow \langle \mathit{true}, \rho\rangle \qquad k, I \vdash \langle e, \hat{\sigma}\rangle \hat{\rightarrow} \langle \hat{v}, \pi\rangle \qquad \sigma\ \mathcal{R}\ \hat{\sigma}}{\mathit{true} \sqsubseteq \mathit{eval}(k \wedge \hat{v}, \phi_\sigma)}$$

The induction hypothesis states $\mathit{true} \sqsubseteq \mathit{eval}(k, \phi_\sigma)$, so it remains to show $\mathit{true} \sqsubseteq \mathit{eval}(\hat{v}, \phi_\sigma)$. The induction hypothesis also enables the invocation of the full proof, which gives $(\mathit{true}, \sigma)\ \mathcal{R}\ \hat{v}$ which is defined to mean $\mathit{true} \sqsubseteq \mathit{eval}(\hat{v}, \phi_\sigma)$.

Case $k' \equiv [\![k \wedge \neg\hat{v}]\!]$, $\hat{v}$ is a query condition In this case, the analysis attaches k' to all paths generated by the false-branch of an **if**. So, it must be shown:

$$\frac{\langle e,\sigma\rangle \rightarrow \langle \mathit{false},\rho\rangle \qquad k, I \vdash \langle e,\hat{\sigma}\rangle \hat{\rightarrow} \langle \hat{v},\pi\rangle \qquad \sigma\ \mathcal{R}\ \hat{\sigma}}{\mathit{true} \sqsubseteq \mathit{eval}(k \wedge \neg\hat{v}, \phi_\sigma)}$$

Proceeding as above, the full proof gives $\mathit{false} \sqsubseteq \mathit{eval}(\hat{v}, \phi_\sigma)$. Since $f'_\neg$ is monotonic, $\mathit{true} \sqsubseteq \neg\mathit{eval}(\hat{v}, \phi_\sigma)$. A simple analysis on the domain of $f'_\neg$ gives $\neg\mathit{eval}(\hat{v}, \phi) = \mathit{eval}(\neg\hat{v}, \phi)$, and reaches the desired conclusion. □

Theorem 3 (Soundness of command evaluation). For all σ, $\hat{\sigma}$, c:

$$\frac{\langle c,\sigma\rangle \rightarrow \langle \sigma',\rho\rangle \qquad k, I \vdash \langle c,\hat{\sigma}\rangle \hat{\rightarrow} \langle \hat{\sigma}',\pi\rangle \qquad \sigma\ \mathcal{R}\ \hat{\sigma}}{(\sigma',\rho)\ \mathcal{R}\ (\hat{\sigma}',\pi)}$$

Proof. Note that the transfer functions of this extended semantics are also monotonic. Thus the proof of soundness for commands is similar to that of basic commands in the full proof, with appropriate applications of Theorems 1 and 2. □

Bibliography

[ABD+89] Malcolm Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The Object-Oriented Database System Manifesto. In *Deductive and Object-Oriented Databases*, 1989.

[AM95] M. P. Atkinson and R. Morrison. Orthogonally Persistent Object Systems. *The VLDB Journal*, 4(3), 1995.

[ANS99] ANSI/INCITS. Database Languages - SQLJ - Part 1: SQL Routines using the Java Programming Language. Technical Report 331.1-1999, ANSI/INCITS, 1999.

[Atk78] Malcolm P. Atkinson. Programming Languages and Databases. In *The International Conference on Very Large Data Bases*, 1978.

[Atk01] Malcolm P. Atkinson. Persistence and Java - A Balancing Act. In *The International Symposium on Objects and Databases*, 2001.

[AU77] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1977.

[Ban88] Utpal K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988.

[BEJ+00] Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[BH07] Don Box and Anders Hejlsberg. LINQ: .NET Language-Integrated Query. `http://msdn.microsoft.com/en-us/library/bb308959.aspx`, 2007.

[Bie03] G. M. Bierman. Formal semantics and analysis of object queries. In *The ACM SIGMOD International Conference on Management of Data*, 2003.

[BK98] N. Brown and C. Kindel. Distributed Component Object Model Protocol–DCOM/1.0, 1998. Redmond, WA, 1996.

[BL94] Phillip Bogle and Barbara Liskov. Reducing cross domain call overhead using batched futures. *ACM SIGPLAN Notices*, 29(10), 1994.

[Bla05] Michael Blaha. The Dilemma of Encapsulation vs. Query Optimization. `http://www.odbms.org/download/007.01%20Blaha%20The%20Dilemma%20of%20Encapsulation%20vs%20Query%20Optimization%20July%202005.pdf`, 2005.

[BLS+94] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension syntax. *SIGMOD Record*, 23(1),

1994.

[BMS05] Gavin M. Bierman, Erik Meijer, and Wolfram Schulte. The Essence of Data Access in Cω. In *The European Conference on Object-Oriented Programming*, 2005.

[BNHC99] Kumar Brahnmath, Nathaniel Nystrom, Antony L. Hosking, and Quintin I. Cutts. Swizzle Barrier Optimizations for Orthogonal Persistence in Java. 1999.

[BPS99] Philip A. Bernstein, Shankar Pal, and David Shutt. Context-Based Prefetch for Implementing Objects on Relations. In *The International Conference on Very Large Data Bases*, 1999.

[BZ99] Stephen Blackburn and John N. Zigman. Concurrency — The Fly in the Ointment? 1999.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, 2001.

[CD96] Michael J. Carey and David J. DeWitt. Of Objects and Databases: A Decade of Turmoil. In *The International Conference on Very Large Data Bases*, 1996.

[CDKN94] Michael J. Carey, David J. DeWitt, Chander Kant, and Jeffrey F. Naughton. A status report on the OO7 OODBMS benchmarking

effort. In *The ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1994.

[Cen04] Davor Cengija. Hibernate Your Data. `http://www.onjava.com/pub/a/onjava/2004/01/14/hibernate.html`, 2004.

[CF03] Yossi Cohen and Yishai A. Feldman. Automatic high-quality reengineering of database programs by abstraction, transformation and reimplementation. *ACM Transactions on Software Engineering and Methodology*, 12(3), 2003.

[Cha98] Surajit Chaudhuri. An overview of query optimization in relational systems. In *The ACM Symposium on Principles of Database Systems*, 1998.

[Che76] Peter P. Chen. The Entity-Relationship Model — Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1), 1976.

[CI07] William R. Cook and Ali H. Ibrahim. Programming languages & databases: What's the problem? Technical report, The University of Texas at Austin, 2007.

[CKP91] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *The ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[CLM+07] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web application via automatic partitioning. In *The ACM Symposium on Operating Systems Principles*, 2007.

[CLWY06] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. In *The International Symposium on Formal Methods for Components and Objects*, 2006.

[CR05] William R. Cook and Siddhartha Rai. Safe query objects: Statically typed objects as remotely executable queries. In *The International Conference on Software Engineering*, 2005.

[CTIW09] William R. Cook, Eli Tilevich, Ali Ibrahim, and Ben Wiedermann. Language design for distributed objects. In *The International Workshop on Distributed Objects for the 21st Century*, 2009.

[Dem92] Serge Demeyer. A survey of object-oriented databases. Technical report, Vrije Universiteit Brussel, 1992.

[DFO06] Debzani Deb, M. Muztaba Fuad, and Michael J. Oudshoorn. Towards Autonomic Distribution of Existing Object Oriented Programs. In *The International Conference on Autonomic and Autonomous Systems*, 2006.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis.

In *The European Conference on Object-Oriented Programming*, 1995.

[DSP03] Jacques-Antoine Dubé, Rick Sapir, and Peter Purich. Oracle Application Server TopLink Application Developer's Guide, 10g (9.0.4). Oracle Corporation, 2003.

[EH07] Torbjörn Ekman and Görel Hedin. The JastAdd Extensible Java Compiler. *ACM SIGPLAN Notices*, 42(10), 2007.

[FFKD87] Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce F. Duba. A Syntactic Theory of Sequential Control. *Theoretical Computer Science*, 52(3), 1987.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3), July 1987.

[Fow03] Martin Fowler. *Patterns of Enterprise Application Architecture*. Adison Wesley, 2003.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *The ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2007.

[GL07] Joseph (Yossi) Gil and Keren Lenz. Simple and safe SQL queries with C++ templates. In *The International Conference on Generative Programming and Component Engineering*, 2007.

[GM00] A. Gawecki and F. Matthes. Integrating Query and Program Optimization Using Persistent CPS Representations. In Malcom P. Atkinson and Ray Welland, editors, *Fully Integrated Data Environments*, ESPRIT Basic Research Series, pages 496–501. 2000.

[GSD04] Carl Gould, Zhendong Su, and Premkumar Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In *The International Conference on Software Engineering*, 2004.

[Ham76] Griffith Hamlin, Jr. Configurable applications for satellite graphics. *ACM SIGGRAPH Computer Graphics*, 10(2), 1976.

[HE07] Görel Hedin and Torbjörn Ekman. The JastAdd system — modular extensible compiler construction. *Science of Computer Programming*, 69, 2007.

[HGBM08] Jungwoo Ha, Magnus Gustafsson, Stephen M. Blackburn, and Kathryn S. McKinley. Microarchitectural Characterization of Production JVMs and Java Workloads. Technical Report TR-08-13, The University of Texas at Austin, Department of Computer Sciences, 2008.

[Hib05] Hibernate reference documentation. http://www.hibernate.org/hib_docs/v3/reference/en/html, May 2005.

[HM90] Antony L. Hosking and J. Eliot B. Moss. Towards Compile-Time Optimisations for Persistence. In *The International Workshop on Persistent Object Systems*, September 1990.

[HM91] Antony L. Hosking and J. E Moss. Compiler support for persistent programming. Technical report, University of Massachusetts, Amherst, Amherst, MA, USA, 1991.

[HMW03] Wook-Shin Han, Yang-Sae Moon, and Kyu-Young Whang. PrefetchGuide: capturing navigational access patterns for prefetching in client/server object-oriented/object-relational DBMSs. *Information Sciences*, 152(1), 2003.

[HNCB99] Antony L. Hosking, Nathaniel Nystrom, Quintin I. Cutts, and Kumar Brahnmath. Optimizing the read and write barriers for orthogonal persistence. In *The International Workshop on Persistent Object Systems*, 1999.

[Hos96] Antony Hosking. Residency check elimination for object-oriented persistent languages. In *The International Workshop on Persistent Object Systems*, 1996.

[HS99] Galen C. Hunt and Michael L. Scott. The Coign automatic distributed partitioning system. In *The USENIX Symposium on*

Operating Systems Design and Implementation, 1999.

[Ibr09] Ali Ibrahim. *Practical Transparent Persistence*. PhD thesis, The University of Texas at Austin, 2009.

[IC06] A. Ibrahim and W. Cook. Automatic prefetching by traversal profiling in object persistence architectures. In *The European Conference on Object-Oriented Programming*, 2006.

[IJCT09] Ali Ibrahim, Yang Jiao, William R. Cook, and Eli Tilevich. Remote batch invocation for compositional object services. In *The European Conference on Object-Oriented Programming*, 2009.

[IJI+09] Ali Ibrahim, Yang Jiao, Marc Fisher II, William R. Cook, and Eli Tilevich. Remote batch invocation for web services: Document-oriented web services with object-oriented interfaces. In *The European Conference on Web Services*, 2009.

[IKY+00] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a Java Just-In-Time compiler. In *The ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2000.

[ISO03] ISO/IEC. Information technology — Database languages — SQL — Part 3: Call-Level Interface (SQL/CLI). Technical Report 9075-3:2003, ISO/IEC, 2003.

[IZ06] Ming-Yee Iu and Willy Zwaenepoel. Queryll: Java database queries through bytecode rewriting. In *The ACM/IFIP/USENIX International Middleware Conference*, 2006.

[JL96] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[Knu68] D.E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, 1968.

[KS04] Kristian Kvilekval and Ambuj Singh. SPREE: Object Prefetching for Mobile Computers. In Robert Meersman and Zahir Tari, editors, *CoopIS/DOA/ODBASE*, volume 3291 of *Lecture Notes in Computer Science*, pages 1340–1357. Springer, 2004.

[LAC+96] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *The ACM SIGMOD International Conference on Management of Data*, 1996.

[Lea00] Neal Leavitt. Whatever Happened to Object-Oriented Databases? `http://www.leavcom.com/pdf/DBpdf.pdf`, 2000.

[LM96] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *The ACM International Confer-*

ence on Architectural Support for Programming Languages and Operating Systems, 1996.

[LM99] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *The USENIX Conference on Domain Specific Languages*, 1999.

[Mai87] David Maier. Representing database programs as objects. In *The ACM SIGMOD Workshop on Database Programming Languages*, 1987.

[Mar05] Bruce E. Martin. Uncovering database access optimizations in the middle tier with TORPEDO. In *The IEEE International Conference on Data Engineering*, 2005.

[MBMZ01] Alonso Marquez, Stephen Blackburn, Gavin Mercer, and John N. Zigman. Implementing Orthogonally Persistent Java. In *The International Workshop on Persistent Object Systems*, pages 247–261, 2001.

[McN86] David J. McNally. *Models for Persistence in Lazy Functional Programming Systems*. PhD thesis, University of St. Andrews, October 1986.

[MEW08] Fabrice Marguerie, Steve Eichert, and Jim Wooley. *Linq in action*. Manning Publications Co., Greenwich, CT, USA, 2008.

[MH94] J. Eliot B. Moss and Antony L. Hosking. Expressing object residency optimizations using pointer type annotations. In *The International Workshop on Persistent Object Systems*, September 1994.

[MH96] J. Eliot B. Moss and T. Hosking. Approaches to Adding Persistence to Java. In *The International Workshop on Persistence and Java*, 1996.

[MH98] V. Matena and M. Hapner. Enterprise Java Beans Specification 1.0. Sun Microsystems, 1998.

[MH07] Eva Magnusson and Görel Hedin. Circular reference attributed grammars — their evaluation and applications. *Science of Computer Programming*, 68(1), 2007.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *The ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[MSS95] F. Matthes, G. Schroder, and J.W. Schmidt. Tycoon: A scalable and interoperable persistent system environment. In M.P. Atkinson, editor, *Fully Integrated Data Environments*. Springer-Verlag, 1995.

[MvD76] Janet Michel and Andries van Dam. Experience with distributed processing on a host/satellite graphics system. In *ACM SIGGRAPH Computer Graphics*, 1976.

[New06] Ted Neward. The Vietnam of Computer Science. `http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx`, June 2006.

[NT05] Matthias Neubauer and Peter Thiemann. From sequential programs to multi-tier applications by program transformation. In *The ACM Symposium on Principles of Programming Languages*, 2005.

[Obj97] The Object Management Group (OMG). *The Common Object Request Broker: Architecture and Specification*, 1997.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PTDL07] Mike P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. *IEEE Computer*, 40(11), 2007.

[PZ97] M. Philippsen and M. Zenger. JavaParty– transparent remote objects in Java. *Concurrency Practice and Experience*, 9(11), 1997.

[Ric89] Joel Edward Richardson. *E: a persistent systems implementation language*. PhD thesis, The University of Wisconsin, 1989. Supervisor-Michael James Carey.

[SAKM09] Yee Jiun Song, Marcos K. Aguilera, Ramakrishna Kotla, and Dahlia Malkhi. RPC chains: efficient client-server communication in geodistributed systems. In *The ACM/USENIX Symposium on Networked Systems Design and Implementation*, 2009.

[SG90] James W. Stamos and David K. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4), 1990.

[SGL06] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop: a language for programming the web 2.0. In *The ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2006.

[SM94] Joachim W. Schmidt and Florian Matthes. The DBPL project: advances in modular database programming. *Information Systems*, 19(2), 1994.

[SMV93] J.W. Schmidt, F. Matthes, and P. Valduriez. Building Persistent Application Systems in Fully Integrated Data Environments: Modularization, Abstraction and Interoperability. In *The Proceedings of the Euro-Arch'93 Congress*, October 1993.

[Sof00] Rational Software. Whitepaper on the UML and Data Modeling, 2000.

[Sun97] Sun Microsystems. *Java Remote Method Invocation Specification*, 1997.

[TA90] B. H. Tay and A. L. Ananda. A survey of remote procedure calls. *The ACM SIGOPS Operating Systems Review*, 24(3), 1990.

[TCJ09] Eli Tilevich, William R. Cook, and Yang Jiao. Explicit batching for distributed objects. In *The IEEE International Conference on Distributed Computing Systems*, 2009.

[Tri92] Phil Trinder. Comprehensions, a query notation for DBPLs. In *The ACM SIGMOD Workshop on Database Programming Languages*, 1992.

[TS09] Eli Tilevich and Yannis Smaragdakis. J-Orchestra: Enhancing Java programs with distribution capabilities. *ACM Transactions on Software Engineering and Methodology*, 19(1), 2009.

[TTS+08] Zachary Tatlock, Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. Deep typechecking and refactoring. In *The ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2008.

[VKS04] Roman Vitenberg, Kristian Kvilekval, and Ambuj K. Singh. Increasing concurrency in databases using program analysis. In *The European Conference on Object-Oriented Programming*, 2004.

[VL00] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2), 2000.

[Wad] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2).

[WC07] Ben Wiedermann and William R. Cook. Extracting queries by static analysis of transparent persistence. In *The ACM Symposium on Principles of Programming Languages*, 2007.

[Wei07] Dan Weinreb. Object-Oriented Database Management Systems Succeeded. `http://www.danweinreb.org/blog/object+oriented+database+management+systems+succeeded`, 2007.

[WF08] Lei Wang and Michael Franz. Automatic Partitioning of Object-Oriented Programs for Resource-Constrained Mobile Devices with Multiple Distribution Objectives. In *The International Conference on Parallel and Distributed Systems*, 2008.

[WGSD07] Gary Wassermann, Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. *ACM Transactions on Software Engineering and Methodology*, 16(4), 2007.

[WIC08] Ben Wiedermann, Ali Ibrahim, and William R. Cook. Interprocedural query extraction for transparent persistence. In *The ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2008.

[Won00] Limsoon Wong. Kleisli, a functional query system. *Journal of Functional Programming*, 10(1), 2000.

[WPN06] Darren Willis, David J. Pearce, and James Noble. Efficient object querying in Java. In *The European Conference on Object-Oriented Programming*, 2006.

[WPN08] Darren Willis, David J. Pearce, and James Noble. Caching and incrementalisation in the Java query language. *ACM SIGPLAN Notices*, 43(10), 2008.

[YK03] Kwok Cheung Yeung and Paul H. J. Kelly. Optimising Java RMI Programs by Communication Restructuring. In *The ACM/IFIP/USENIX International Middleware Conference*, 2003.

[Zon95] Q. Y. Zondervan. Increasing cross-domain call batching using promises and batched control structures. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1995.

[ZZNM02] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3), 2002.

Vita

Benjamin Alan Wiedermann graduated from Olathe East High School in Olathe, Kansas in 1997. In 2001, he graduated *magna cum laude* from Boston University with a Bachelor of Arts in Computer Science and a minor in Theatre Arts. He worked as a computer science textbook author until 2003, when he entered the Computer Science Ph.D. program at the University of Texas at Austin.

Permanent address: 701 North Loop #110
Austin, Texas 78751

This dissertation was typeset with LaTeX† by the author.

† LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.