Program Analysis and Transformation for Holistic Optimization of Database Applications

Karthik Ramachandra
Indian Institute of Technology Bombay
karthiksr@cse.iitb.ac.in

Ravindra Guravannavar
Indian Institute of Technology Hyderabad
ravig@acm.org

S Sudarshan
Indian Institute of Technology Bombay
sudarsha@cse.iitb.ac.in

Abstract

We describe DBridge, a novel program analysis and transformation tool to optimize database and web service access. Traditionally, rewrite of queries and programs are done independently, by the database query optimizer and the language compiler respectively, leaving out many optimization opportunities. Our tool aims to bridge this gap by performing holistic transformations, which include both program and query rewrite.

There has been earlier research in this area involving program analysis and transformation for automatically rewriting database applications to perform optimizations; for example, our earlier work has addressed batching or asynchronous submission of iterative queries, and prefetching query results. DBridge implements these techniques for Java programs and internally uses Soot, a Java optimization framework, for static analysis and transformation. DBridge can perform such optimizations on Java programs that use the JDBC API to access the database. It is currently being extended to handle the Hibernate API, and Web Services.

In this paper, we describe the program transformations that DBridge can perform. We then discuss the design and implementation of DBridge with a focus on how the Soot framework has been used to achieve these goals. Finally, we conclude by discussing some of the future directions for our tool.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program Analysis; H.2.4 [*Database Management Systems*]: Query processing

1. Introduction

Database applications, typically written in languages such as Java, PHP, C#, perform declarative queries and updates from within imperative code that encodes business logic. Such applications use a mix of procedural constructs and SQL. In such applications, poor performance is often observed due to (a) repeated execution of parameterized queries leading to network round-trip delays and server-side random IO, and (b) synchronous (blocking) execution of queries leading to ineffective use of computing resources Some of the approaches that have been proposed to avoid or reduce the effects of these delays are as follows.

- **Set Oriented Query Execution:**
  Iterative execution of parameterized SQL queries can be replaced by a *batched form* or set oriented form of the query, which is often far more efficient. The importance of set oriented execution is well known in the database community, especially in the context of nested subqueries. Query decorrelation [5, 12, 15] addresses the problem of iterative execution of nested subqueries, by rewriting them using set operations such as joins, thereby reducing random I/O. However, decorrelation techniques are not directly applicable to imperative program loops. Guravannavar et. al. [7] propose a set of program transformation rules for automatically rewriting loops containing query invocations to use batched parameter bindings, thereby enabling set oriented execution of database queries. The transformation rules make use of inter-statement data dependencies gathered from static analysis of the program. The proposed program transformation rules are powerful enough to rewrite a large class of loops involving complex control flow and arbitrary level of nesting.
- **Asynchronous Query Submission:**
  Synchronous execution of queries or Web service requests forces the calling application to block until the query/request is satisfied. The performance of applications can be significantly improved by asynchronous submission of queries. Manjhi et al. [13] consider rewriting of application code by means of inserting prefetches within a procedure. Yeung [17] proposes deferred execution of remote procedure calls and code shipping by using program rewrite techniques. Chavan et. al. [1] address the issue of automatically transforming a program written assuming synchronous query submission, to one that exploits asynchronous query submission. Their approach is based on dataflow analysis and is framed as a set of transformation rules based on the work of Guravannavar et. al. [7].
- **Prefetching Query Results:**
  Recently, Ramachandra et. al. [14] proposed techniques based on program analysis and rewriting to prefetch the results of queries or Web service requests that are subsequently issued by a program. These techniques statically place asynchronous prefetch instructions for query execution statements at the earliest possible points. To find more prefetching opportunities, techniques such as loop fission are employed, and the proposed approach works even when calls to the database are deep within functions called within a loop. This work integrates well with [1, 7] and enables those transformations to work with interprocedural code.

All the above optimization techniques [1, 3, 7, 13, 14, 17] require the understanding of the query or the Web service API in addition to the standard data and control flow information of

SOAP'12 June 14, Beijing, China.
Copyright © 2012 ACM ISBN 978-1-4503-1490-9/12/06…$10.00

```
Connection con = DriverManager.getConnection( url );
PreparedStatement  pstmt  = con. prepareStatement (
        ‘‘ SELECT count(partkey) FROM part WHERE category=?”);

while( category  !=  −1) {
    pstmt . setInt  (1,   category );
    ResultSet  rs  = pstmt.executeQuery ();
    if  ( rs . next ())  {
        partCount  =  rs . getInt  (0);
        total   += partCount;
            print ( category  +   ‘‘:’’   +   partCount );
    }
    category  =  getParent ( category );
}
```

Figure 1. A Program Snippet with JDBC Calls [2])

```
Connection con = DBridgeDriverManager.getConnection( url );
PreparedStatement  pstmt  = con.dBridgePrepareStatement(
        ‘‘ SELECT count(partkey) FROM part WHERE category=?”);

LoopContextTable lct = new LoopContextTable();
while( category  !=  −1) {
  LoopContext ctx = lct.createContext();
  pstmt . setInt  (1,   category );
  ctx.setInt(“category”, category);
  category  =  getParent ( category );
  pstmt . addBatch( ctx );
}

pstmt . executeBatch ();

for  (LoopContext ctx :   lct )  {
  category = ctx.getInt(“category”);
  ResultSet  rs  = pstmt . getResultSet( ctx );
  if  ( rs . next ())  {
    partCount  =  rs . getInt  (0);
    total   += partCount;
        print ( category  +   ‘‘:’’   +   partCount );
  }
}
```

Figure 2. Program snippet after Transformation [2]

the program. Such techniques that span the application and the database are referred to as *holistic optimization* techniques [13].

DBridge [2] is a holistic optimization tool based on the techniques presented in [1, 7] and [14]. DBridge performs these optimizations on Java programs that use the JDBC API [9] to access the database. It is currently being extended to handle the Hibernate API [8], and Web services. DBridge uses the Soot framework [16] extensively for code analysis and transformation. Soot provides a convenient intermediate representation called Jimple, and also provides some of the data flow analyses required by DBridge.

DBridge statically analyzes the input program and identifies opportunities for the above mentioned techniques; it then rewrites the program accordingly. DBridge is designed to be a source-to-source transformation tool, and to this end, it ensures readability of the transformed code. The tool is thus best suited for integration into an application development environment (IDE). DBridge can also be used as a preprocessing step inside a language compiler, thus making the compiler “database access aware” [2].

2. Overview of DBridge

In this paper we outline how program analysis and transformation is performed in DBridge, using the Soot framework. DBridge comprises of two components - a program transformer, and a runtime library. These have been described in detail in [1, 2, 7, 14]. We now briefly summarize these components.

2.1 Program Analysis and Transformation

DBridge primarily performs two kinds of transformations: loop fission and prefetch statement insertion.

2.1.1 Loop Fission

Techniques presented in both [7] and [1] depend on loop fission as the basic transformation to enable set oriented or asynchronous submission of queries. As an illustration of the kind of loop fission DBridge can perform, consider the Java program snippet shown in Figure 1 (reproduced from [2]. The program computes the total number of parts in a given list of categories. Note the repeated execution of a parameterized aggregate query inside the *while* loop. The program snippet after transformation by DBridge is shown in Figure 2. Note that the repeated execution of the parameterized query has been replaced by the single execution of a batched version of the query which is not part of any loop. The transformed code is a result of the application of several transformation rules. An overview of the important transformations performed by DBridge are presented in [2]; we briefly describe them here.

- *Statement Reordering*: DBridge applies a set of transformation rules to reorder the statements within the loop body, so as to permit loop splitting at the query execution statement. Statement reordering is performed taking inter-statement data dependencies into account. In order to achieve this, we define the *ReorderingUnit* interface, and model the loop body as a nested sequence of *ReorderingUnit*s. These *ReorderingUnit*s are swapped based on rules that preserve all the true dependencies. Pseudo dependencies (also known as anti and output dependencies [10]) that prohibit loop fission, are broken by introducing temporary variables; see [6, 7] for details.
- *Loop Splitting:* This is the key transformation, described in [1, 7] to enable set oriented execution or asynchronous submission. As a result of this transformation, a loop is split into two parts as described in [2]; Figure 2 shows the result of loop splitting applied to the program in Figure 1. The loop fission transformation has certain preconditions for its applicability [7], and may require prior application of statement reordering so that the preconditions are met. The preconditions are defined on the Data Dependence Graph.
- *Rewrite of Conditional Blocks:* DBridge can deal with conditional control transfer statements (*if-then-else*), and query execution statements inside conditional blocks. DBridge also handles *order-sensitive* operations within the loop correctly. Order-sensitive operations are operations whose order of execution is important for the correctness of the program. DBridge first transforms conditional blocks into a sequence of guarded statements by introducing a boolean variable to remember the branching decision. Each member in this sequence is a *ReorderingUnit*. It then applies the loop splitting transformation. Finally, the sequence of *ReorderingUnit*s are merged back and conditional blocks are regenerated.
- *Nested Loops:* DBridge works with arbitrary levels of loop nesting. Also, in general, the body of a loop may contain more than one parameterized query. Repeated application of the loop splitting transformation allows rewriting of any number of queries that lie inside the loop.

2.1.2 Prefetch Statement Insertion

Techniques for query result prefetching are described in [14], which are implemented in DBridge. These techniques consider

```
void generateReport(int custId, int curr, String fromDate){
(n1)    ResultSet a=executeQuery("SELECT * FROM accounts
                       WHERE custId=?", custId); // q1
(n2)    while(a.next()){
(n3)      int accountId = a.getInt("accountId");
(n4)      processAccount(a);
(n5)      processTransactions(accountId, fromDate);
        }

(n6)    ResultSet c = executeQuery("SELECT * FROM customers
                   WHERE custId=?", custId); // q2
(n7)    processCustomer(c);

(n8)    if(curr != DEFAULT_CURR){
(n9)      ResultSet s=executeQuery("SELECT exchgRate
                     FROM exchange WHERE src=? AND dest=?",
                     {curr, DEFAULT_CURR}); // q3
(n10)     printExchangeRate(s, curr);
        }
}
```

Figure 3. Program with prefetching opportunities [14]

```
void generateReport(int custId, int curr, String fromDate){
    submit(q2, custId);
    submit(q1, custId);

    boolean b = (curr != DEFAULT_CURR);
    if(b) submit(q3, {curr, DEFAULT_CURR});

     ... // code unchanged (lines n1 to n7)
    if(b){
      ... // code unchanged (lines n9, n10)
    }
}
```

Figure 4. Program with prefetch requests [14]

programs with query execution statements embedded within them, and statically insert prefetch instructions for those queries at the earliest possible points in the program across method invocations.

An example from [14] is reproduced here in Figure 3 in order to illustrate the kind of prefetching performed by DBridge. The *generateReport* method accepts a customer id (*custId*), a currency code (*curr*), and a date (*fromDate*), and performs the following tasks in sequence: (i) Retrieves information about all accounts of that customer and processes them in a loop (n_1 to n_5), (ii) Retrieves and processes customer information (n_6 and n_7), (iii) If the supplied currency code does not match the default (DEFAULT_CURR), it fetches and displays the current exchange rate between the two (n_8 to n_{10}). The program after insertion of prefetch instructions is shown in Figure 4. The *submit* method is the instruction that issues an asynchronous prefetch. For brevity, Figure 4 uses symbols $q1$, $q2$ etc. to denote actual query strings, and omits lines of code that remain unchanged.

In order to detect such opportunities, Ramachandra et. al. [14] extend anticipable expressions analysis [11], to analyze anticipability of queries. DBridge performs *query anticipability analysis*, and insert prefetch instructions as described in [14]. We briefly describe the analysis here; see [14] for details.

Query Anticipability Analysis:

Prefetching of queries involves inserting query submission requests at program points where they were not present in the original program. The goal is to insert asynchronous query prefetch requests

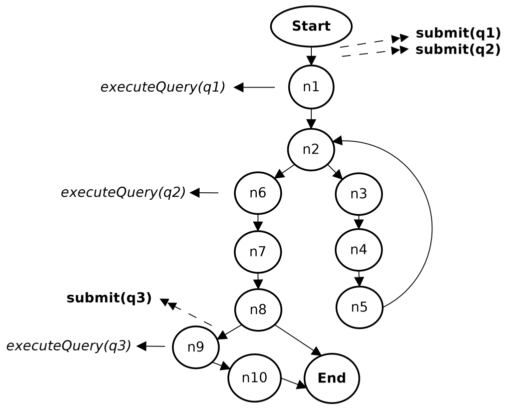

Figure 5. CFG for Figure 3 [14]

at the earliest possible points in the program so that the latency of network and query execution can be maximally overlapped with local computation.

Suppose a query q is executed with parameter values v at point p in the program. The earliest possible points e where query q could be issued are the set of points where the following conditions hold: (a) all parameters of q are available, (b) the results of executing q at points e and p are the same, (c) conditions (a) and (b) do not hold for predecessors of e, and (d) no prefetch request should be wasted. The scope of this analysis is intraprocedural, but the prefetching algorithm combines the results of this analysis to insert prefetches across method calls. Query anticipability is defined as follows [14]:

DEFINITION 2.1. A query execution statement q is ***anticipable*** at a program point u if every path from u to *End* contains an execution of q which is not preceded by any statement that modifies the parameters of q or affects the results of q. □

Query anticipability analysis is a bit vector backward data flow framework with query execution statements being the data flow values. Gen_n is 1 at bit q if n is the query execution statement q. $Kill_n$ is 1 at bit q if either n contains an assignment to a parameter of q, or performs an update to the database that may affect the results of q. The equations that describe query anticipability are:

$$In_n = (Out_n - Kill_n) \cup Gen_n \tag{1}$$

$$Out_n = \begin{cases} \phi & \text{if } n \text{ is } End \text{ node} \\ \bigcap_{s \in succ(n)} In_s & \text{otherwise} \end{cases} \tag{2}$$

The CFG corresponding to Figure 3 is shown in Figure 5, and the results of performing query anticipability analysis on Figure 3 is shown in Table 1. The table shows only the changed values in iteration #2.

DBridge implements this analysis by implementing a new *BackwardFlowAnalysis* which uses the *ArrayPackedSet* to hold the bit vectors. Anticipability can be blocked by the presence of *critical edges* in the CFG. DBridge therefore uses Soot's built in *CriticalEdgeRemover* as a preprocessing step on the *UnitGraph*.

2.2 The DBridge Runtime Library

The DBridge runtime library works as a layer between the actual data access API (JDBC/Hibernate etc.) and the application code. It provides set oriented execution, asynchronous submission and prefetch submission methods in addition to wrapping the underlying API. Features such as query rewriting, thread management and cache management are handled by this library. The API is designed in such a way that it can be configured to either use parameter batching(set oriented execution) or asynchronous submission

Node	Local Information		Global Information			
			Iteration #1		Iteration #2	
	Gen_n	$Kill_n$	Out_n	In_n	Out_n	In_n
End	000	000	000	000		
n_{10}	000	000	000	000		
n_9	001	000	000	001		
n_8	000	000	000	000		
n_7	000	000	000	000		
n_6	010	000	000	010		
n_5	000	000	111	111	010	010
n_4	000	000	111	111	010	010
n_3	000	000	111	111	010	010
n_2	000	000	010	010		
n_1	100	000	010	110		
$Start$	000	111	110	000		

Table 1. Query anticipability analysis for Figure 3 [14]

of queries on the transformed code with split loops, and Figure 2 highlights the corresponding DBridge API calls.

If the API is configured to use parameter batching, query rewrite is performed at runtime within DBridge's implementation of the *executeBatch* method, as described in [2]. The *executeBatch* call is present in between the two loops as shown in Figure 2. This method internally transforms the query statement into a set oriented form, which is often more efficient. For example, the scalar aggregate query in the example would be transformed into the following query, where *pb* is a temporary table in which the parameter bindings are materialized.

```
SELECT pb.category, le.c1
FROM pbatch pb,
      OUTER APPLY (SELECT count(partkey) as c1
                  FROM part
                  WHERE category=pb.category) le;
```

The rewritten query shown here uses the OUTER APPLY construct of Microsoft SQL Server. The query can alternatively be written using the left outer join combined with LATERAL construct, depending on which construct the underlying database supports. Such a rewriting enables the use of efficient set oriented query processing algorithms such as hash or merge join.

In the asynchronous submission mode, loop splitting is done as shown in Figure 2, but instead of waiting for the first loop to finish before sending a batched query, queries are submitted asynchronously within the first loop. The *stmt.addBatch(ctx)* invocation is a non blocking query submission. This request queue is monitored by a thread pool which manages multiple threads (number of threads being configurable). The requests are picked up by free threads which maintain open connections to the database, and execute the query. The results are then placed in an array indexed by the loop context (ctx). The second loop accesses the results corresponding to the loop context and executes statements that depend on the query results.

Queries, being side-effect free, can be executed in any order of the parameter bindings. However, the loop can contain other order-sensitive operations, which must be executed in the same order as in the original program. To ensure this, the loop splitting transformation of DBridge maintains the loop context in an ordered table (*LoopContextTable*), and iterates over the loop context records in the order in which they are produced.

In the case of prefetching, the transformed program contains *submit* calls which are nonblocking query submissions. Figure 4 highlights the *submit* calls of the DBridge API. The library takes care of issuing a prefetch in the background at this point, while the program continues to execute. The results of the query are put into a cache keyed by *(queryString, parameterBindings)*. DBridge's implementation of the *executeQuery* method looks up the cache, and blocks till the results become available.

One of the design goals for this API is to make it extensible in order to add support for different data access APIs. It is currently being extended to handle web services. A more detailed description of our API is available in [1, 2] and [14].

3. System Design

We now discuss some of the design considerations and challenges in engineering a program transformation tool such as DBridge. As described in [2], ensuring that the transformed program is *equivalent* in its functionality to the original program is a *strict requirement* for DBridge. DBridge makes conservative assumptions about data dependencies that cannot be statically determined, at the cost of losing optimization opportunities, in order to preserve equivalence. The transformations we perform are based on a set of formally defined equivalence rules, whose correctness proofs can be found in [6].

DBridge is built with the following design goals:

- **Robustness**: It is not always possible to split every loop that involves a query execution statement. Inter-statement data dependencies may prohibit such a rewrite. Similarly, prefetch instructions are blocked by certain statements which form barriers for prefetching, due to data or control dependences. Hence, identifying desired program patterns is very important.

  The intermediate code has the advantage of being simple and suitable for data flow analysis, but it makes the task of recognizing desired program patterns difficult. Each high level language construct translates to several instructions in the intermediate representation. DBridge has been designed for robust matching of desired program fragments and can handle several variations in programs.

- **Readability**: The prefetching transformation is very less intrusive as it only places prefetch instructions at certain points in the program and hardly modifies existing lines of code. The loop fission transformation, though, can end up making quite intrusive changes depending on the complexity of the original program. The presence of nested loops, and complex control flow can make the resulting code less readable.

  Programmers may need to read the transformed code to debug a program, or even to gain confidence in the correctness of the transformed code. Therefore, maintaining readability of the transformed code is very important. We achieve this goal through several measures. For instance, when we rewrite conditional blocks and then split a loop, the resulting code will have many guarded statements. We therefore introduce a pass where such guarded statements are grouped back in each of the two generated loops, so that the resulting code resembles the original code [1, 2].

  Another challenge encountered in ensuring readability was the limitation of decompilation, which at times resulted in code that was quite unreadable. We use the Dava decompiler that is bundled with Soot, and we have encountered instances where the whole body of a loop is folded into the header of the *for* loop. We have currently worked around this by carefully generating Jimple code to avoid such results.

- **Extensibility**: DBridge is designed in a way that provides an elegant framework for introducing new transformation rules or extending existing rules. Each rule is encapsulated as an object, and all the information necessary to apply a rule is provided by the framework via the *DBridgeDependenceGraph*. The runtime library API is designed to allow different data sources to be plugged in with less effort.

4. Implementation

The important phases in the program transformation process are shown in Figure 6. The input is a standard Java program written using the JDBC API for database access. The input file is first converted into Jimple, using which we construct a *DBridge Dependence Graph*, which is the basic data structure on which our transformation rules rely. The *DBridge Dependence Graph* essentially encapsulates the *DataDependencyGraph*, *UnitGraph*, the *LoopNestTree* and other supporting data structures. While building the data dependency graph, Soot analyzes all the library classes as well, which is time and memory intensive. If library classes are excluded, we would prefer Soot to be conservative and add dependencies which may be spurious. For a tool like DBridge, where preserving semantics is of prime importance, it is acceptable to not perform certain optimizations rather than changing semantics.

4.1 Applying transformations

The main task of DBridge appears in the *Apply Trans Rules* phase. The program transformation rules are applied in an iterative manner, updating the dataflow information each time the code changes. The rule application process stops when all (or the user chosen) transformations are done. The intermediate representation is then converted back to a target Java source file.

4.1.1 Loop Fission

The program transformation rules presented in [1, 6] can be applied repeatedly to refine a given program. Applying a rule to a program involves substituting a program fragment that matches the antecedent (LHS) of the rule with the program fragment instantiated by the consequent (RHS) of the rule. Some rules facilitate the application of other rules and together achieve the goal of splitting a loop.

In DBridge, the transformation rules are applied iteratively as shown in Figure 7. A query execution statement present within a loop body is considered as a *candidate for batching or asynchronous submission*, and for each such candidate, the rules are applied. First, conditional blocks if any, are converted into a sequence of guarded statements. Then the statement reordering algorithm is run, which enables loop fission. Finally, the control structure of the program is restored by merging back guarded statements into conditional blocks.

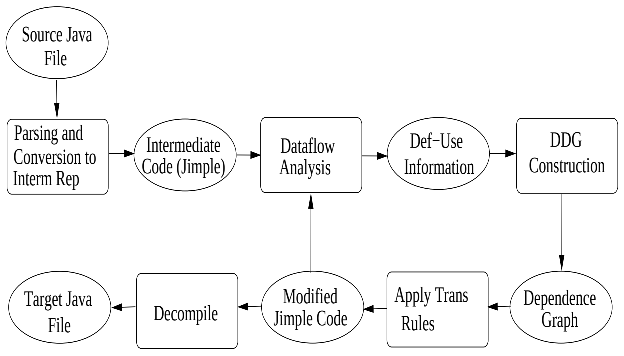

Figure 6. Program transformation phases [2]

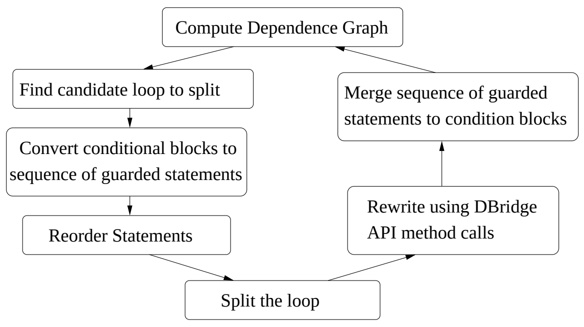
Compute Dependence Graph
Find candidate loop to split
Convert conditional blocks to sequence of guarded statements
Reorder Statements
Split the loop
Rewrite using DBridge API method calls
Merge sequence of guarded statements to condition blocks

Figure 7. Application of transformation rules for loop fission [2]

4.1.2 Prefetch insertion

DBridge uses the fixed point iteration provided by *BackwardFlowAnalysis* in order to perform query anticipability analysis. We define a *Value* type called *QueryExecutionValue* that stores the query statement and its parameters in the *FlowSet*. The Soot *CallGraph* construction takes considerably long time, and we essentially require a simplified call graph which only deals with methods of interest to us. Hence we have extended Soot's *CallGraph* and built our own, over which we traverse in reverse topological order (using Soot's *PseudoTopologicalOrderer*), as per the interprocedural algorithm in [14].

4.2 Challenges

We now discuss a few implementation challenges and describe some reusable patterns that evolved in our implementation.

4.2.1 Detecting complex patterns in code

In order to perform transformations such as loop fission or prefetch statement insertion, we need to identify patterns in the input program where these transformations are applicable. We refer to these patterns as *candidates* for transformation. For example, any query execution statement is a potential *candidate* for the prefetching transformation. A query execution statement present within a loop body is considered as a potential *candidate* for batching or asynchronous submission. More formally, a query execution statement present within a loop body, and which does not lie on a cycle of true data dependencies is a *candidate* for our loop fission transformation (Refer to [1, 6] for formal definitions and proofs). Detecting such *candidates* for JDBC programs involves combining information from the *UnitGraph* (i.e., the CFG) and the *DataDependencyGraph* (i.e., the DDG), and involves the following steps:

1. The SQL query string is typically found as an argument to the *prepareStatement* method, and the *executeQuery* method is the query execution statement. Methods named *setInt*, *setFloat* etc. that are invoked on the JDBC *PreparedStatement* interface are the ones that bind query parameters. We combine information from all these statements in order to get the query string and the variables (or constants) that are bound as its parameters.
2. To identify query execution statements present in loops, we need to look for cycles in the CFG containing an *executeQuery* call. Soot's *LoopNestTree* is a convenient way to iterate over loops instead of directly working with the *UnitGraph*.
3. Subsequently, we use the *DataDependencyGraph* to identify whether the *executeQuery* call lies on a cycle of true data dependencies.

In general, in order to perform transformations on large programs, we feel that a tool like Soot should provide (a) a concise and perhaps declarative way to specify such candidate patterns, and

(b) an efficient mechanism that accepts the pattern specification as input and detects these patterns in large programs.

4.2.2 *ReorderingUnit*s for transformation

As described in Section 2.1.1, the kind of transformations performed by DBridge involves equivalence-preserving reordering of statements. The statement reordering algorithm [1] enables splitting of the loop at the desired statement boundary. Wherever necessary, pseudo dependencies (anti and output dependencies [10]) are broken by introducing local variables.

This is implemented by modeling the loop body as a nested sequence of *ReorderingUnit*s. A *ReorderingUnit* comprises of a list of *Unit*s which may be statements or other *ReorderingUnit*s. This way, we represent (i) single Jimple statements (Stmt), (ii) block of statements (Block), (iii) conditional blocks (*if-else* constructs), (iv) loops, and (v) ternary statements as *ReorderingUnit*s. Now, these *ReorderingUnit*s are swapped based on rules that preserve all the true dependencies. This abstraction helps deal with nested structures such as nested loops and nested conditional blocks in a uniform manner.

4.2.3 Channelizing transformations

The transformations performed by DBridge are of the following types:

- Insertion of new statements,
- Deletion of statements,
- Replacement of statements,
- Modifying targets of *goto* statements.

As described earlier, our program transformation rules are applied in an iterative manner, and each application of a rule results in many transformations to the body. Also, we use data structures such as *ReorderingUnit* and *DBridgeDependenceGraph* which wrap and compose the Body, UnitGraph and other Soot structures. We need to make sure that our data structures are always consistent with changes made to the underlying body. Hence all our rule objects adhere to a pattern that simplifies the management of complex transformations. Instead of making changes directly to the unit chain from the rule objects, transformations are channelized through a *TransformationsCollector* object that builds up a collection of transformations of the above types. At the end of a rule application, a '*commit*' is performed to apply all the collected transformations to the *Body* in an appropriate order, and all dependent data structures are updated to reflect the changes.

4.3 Limitations and Future Work

Currently, DBridge does not fully support programs with all kinds of exception handling and other unconditional control transfer statements. External dependences i.e., dependences through files or other resources outside the program are not yet supported. Also, DBridge does not support language features such as reflection, dynamic class loading, and native code. This is not a big concern for database/web applications, since such features are rarely used inside core application functions that access the database.

We are currently extending DBridge to support other data access APIs and we intend to make it more extensible to be able to plug in different data sources. In future, we plan to implement more holistic optimization techniques in DBridge, and implement a cost model to decide which calls need to be transformed using which techniques. We also intend to scale DBridge to be able to run on large codebases; the time and memory consumption of Soot in whole program mode while performing interprocedural analyses is currently one of the issues we face in this direction.

5. Conclusion

We described the design and implementation of a tool called DBridge, that optimizes database/web service access by performing optimizations that span across the boundaries of the application and the data source. Experiments described in [1, 7, 14] show that these optimizations lead to significant gains in performance. More details about DBridge are available on the project website [4].

We believe that there are lots of opportunities for applying static analysis techniques in optimizing database/web applications. The availability of Soot has been one of the compelling reasons to build this tool in Java. Soot provides a convenient intermediate representation called Jimple, and also performs most of the necessary data flow analyses which we require. We found Soot to be an extremely valuable tool for prototyping of research ideas.

Acknowledgments

We would like to thank Mahendra Chavan and Prabhas Kumar Samanta for having implemented many features of DBridge.

References

[1] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. Program transformations for asynchronous query submission. In *IEEE International Conference on Data Engineering*, pages 375–386, 2011.

[2] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. Dbridge: A program rewrite tool for set-oriented query execution. In *IEEE International Conference on Data Engineering*, pages 1284–1287, 2011.

[3] A. Dasgupta, V. Narasayya, and M. Syamala. A static analysis framework for database applications. In *IEEE International Conference on Data Engineering*, 2009.

[4] DBridge. The DBridge Holistic Optimizer http://www.cse.iitb.ac.in/infolab/dbridge.

[5] M. Elhemali, C. A. Galindo-Legaria, T. Grabs, and M. M. Joshi. Execution Strategies for SQL Subqueries. In *ACM SIGMOD*, 2007.

[6] R. Guravannavar. *Optimization and Evaluation of Nested Queries and Procedures*. Ph.D. thesis, Indian Institute of Technology, Bombay, 2009.

[7] R. Guravannavar and S. Sudarshan. Rewriting Procedures for Batched Bindings. In *Intl. Conf. on Very Large Databases*, 2008.

[8] Hibernate. The Hibernate O/R mapping tool: http://hibernate.org.

[9] JDBC. Java Database Connectivity (JDBC) API http://java.sun.com/products/jdbc/overview.html.

[10] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. ISBN 1-55860-286-0.

[11] U. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., 1st edition, 2009. ISBN 0849328802.

[12] W. Kim. On Optimizing an SQL-like Nested Query. In *ACM Trans. on Database Systems, Vol 7, No.3*, 1982.

[13] A. Manjhi, C. Garrod, B. M. Maggs, T. C. Mowry, and A. Tomasic. Holistic Query Transformations for Dynamic Web Applications. In *IEEE International Conference on Data Engineering*, 2009.

[14] K. Ramachandra and S. Sudarshan. Holistic optimization by prefetching query results. In *ACM SIGMOD*, 2012(to appear).

[15] P. Seshadri, H. Pirahesh, and T. C. Leung. Complex Query Decorrelation. In *IEEE International Conference on Data Engineering*, 1996.

[16] SOOT. A Java Optimization Framework http://www.sable.mcgill.ca/soot.

[17] K. C. Yeung. *Dynamic Performance Optimisation of Distributed Java Applications*. PhD thesis, Imperial College of Science, Technology and Medicine, 2004.