Program Transformations for Asynchronous Query Submission

Mahendra Chavan [#], Ravindra Guravannavar [*], Karthik Ramachandra [#], S. Sudarshan [#]

[#]*Indian Institute of Technology, Bombay*
{mahcha, karthiksr, sudarsha}@cse.iitb.ac.in

[*]*Indian Institute of Technology, Hyderabad*
ravig@iith.ac.in

***Abstract*— 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, which allows the application to perform other processing instead of blocking while the query is executed, and to concurrently issue multiple queries. Concurrent submission of multiple queries can allow the query execution engine to better utilize multiple processors and disks, and to reorder disk IO requests to minimize seeks. Concurrent submission also reduces the impact of network round-trip latency and delays at the database, when processing multiple queries. However, manually writing applications to exploit asynchronous query submission is tedious.**

In this paper we address the issue of automatically transforming a program written assuming synchronous query submission, to one that exploits asynchronous query submission. Our program transformation method is based on dataflow analysis and is framed as a set of transformation rules. Our rules can handle query executions within loops, unlike some of the earlier work in this area. We have built a tool that implements our transformation techniques on Java code that uses JDBC calls; our tool can be extended to handle Web service calls. We have carried out a detailed experimental study on several real-life applications rewritten using our transformation techniques. The experimental study shows the effectiveness of the proposed rewrite techniques, both in terms of their applicability and performance gains achieved.

I. Introduction

In many applications calls made to execute database queries or to invoke web services are often the main causes of latency. Asynchronous or non-blocking calls allow applications to reduce such latency by overlapping CPU operations with network or disk IO requests, and by overlapping local and remote computation. Consider the program fragment shown in Example 1. In the example, it is easy to see that by making a non-blocking call to the database we can overlap the execution of method *foo()* with the execution of the query, and thereby reduce latency.

Many applications are however not designed to exploit the full potential of non-blocking calls. Manual rewrite of such applications although possible, is time consuming and error prone. Further, opportunities for asynchronous query submission are often not very explicit in the code. For instance, consider the program fragment shown in Example 2. In the program, the result of the query, assigned to the variable *partCount*, is needed by the statement that immediately follows the statement executing the query. For the code in the given form there would be no gain in replacing the blocking query execution call by a non-blocking call, as the execution will have to block on a *fetchResult* call immediately after making the *submitQuery* call. It is however possible to transform the given loop, as shown in Example 3, and thereby enable asynchronous query submission.

Example 1 *A simple opportunity for asynchronous query submission*

```
r = executeQuery(query1);
s = foo(); // Some computation not dependent on r
bar(r, s)   // Computation dependent on r and s
```

Code with Asynchronous Query Submission

```
handle = submitQuery(query1); // Non-blocking query submit
s = foo();
r = fetchResult(handle); // Blocking call to fetch query result
bar(r, s)
```

Example 2 *Hidden opportunity for asynchronous query submission*

```
qt = dbCon.prepare( "select count(partkey)                    (s0)
                     from part where p_category=?");
while(!categoryList.isEmpty()) {                              (s1)
    category = categoryList.removeFirst();                    (s2)
    qt.bind(1, category);                                     (s3)
    partCount = executeQuery(qt);                             (s4)
    sum += partCount;                                         (s5)
}
```

The rewritten program in Example 3 contains two loops; the first loop submits queries in a non-blocking mode and the second loop uses a blocking call to fetch the results and then executes the statements that depend on the query results.

Asynchronous calls have been long employed to make concurrent use of different system components, like CPU and disk. In contrast to earlier work on exploiting asynchronous execution, described in Section II, our work focusses on rewriting programs external to the database so as to submit

Example 3 *Loop Transformation to Enable Asynchronous Query Submission*

```
qt = dbCon.prepare( "select count(partkey)
                     from part where p_category=?");
int handle[MAX_SIZE], n=0;
while(!categoryList.isEmpty()) {
      category = categoryList.removeFirst();
      qt.bind(1, category);
      handle[n++] = submitQuery(qt);
}
for(int i = 0; i < n; i++) {
      partCount = fetchResult(handle[i]);
      sum += partCount;
}
```

multiple queries asynchronously. In general, automatically transforming a given loop so as to make asynchronous query submissions is a non-trivial task, and we address the problem in this paper.

Asynchronous execution enabled by the transformations presented in this paper can improve application performance significantly for several reasons: *(a)* the database server and the application run on different machines, allowing query execution to overlap with application program execution, *(b)* the database server typically runs on a multi-core system with large caches, and multiple disks, allowing better use of resources and higher throughput if multiple queries are submitted concurrently, *(c)* database query execution engines today support techniques such as shared scans and RID ordering prior to fetch which can allow queries to execute faster if submitted concurrently than if they are submitted one at a time.

The following are the key technical contributions we make in this paper:

1) We show (in Section III) how a basic set of program transformations, such as loop fission, enable complex programs to be rewritten to make use of asynchronous query submission. Although loop fission is a well known transformation in compiler optimizations and batching, to the best of our knowledge no prior work shows its use for asynchronous submission of database queries.
2) In many cases the data dependencies between program statements do not permit loop fission, which is a key transformation to enable asynchronous calls. We show (in Section IV) that in many cases it is possible to reorder the program statements so as to enable loop fission and give an algorithm to do so. The classical works in static program analysis, which deal with loop fission [1], [2], do not consider statement reordering as a means to enable loop fission. The statement reordering algorithm increases the opportunities for asynchronous query submission significantly. We also prove a sufficient condition on the data dependence graph for a query execution statement to be made non-blocking.
3) Since programmers may need to debug a rewritten version of their program, we present (in Section V) several techniques to make the rewritten program more readable.
4) We present (in Section VI) a detailed experimental study of the proposed transformations on several real world applications. The experimental study shows significant performance gains due to the program transformations.

Guravannavar et.al. [3] describe how to rewrite loops in database applications and stored procedures, to transform iterative execution of queries and updates into a single execution of a set-oriented (batched) form of the query or update. Our program transformation techniques for asynchronous query submission are based on the techniques described in [3]. Both asynchronous query submission and batching are important techniques to improve the performance of database applications. Although batching reduces round-trip delays and allows efficient set-oriented execution of queries, it does not overlap client computation with that of the server, as the client completely blocks after submitting the batch. Also, batching may not be applicable altogether when there is no efficient set-oriented interface for the request invoked.

II. Related Work and Background

Most operating systems today allow applications to issue asynchronous IO requests [4]. Asynchronous calls are also used for data prefetch and overlapping operator execution inside query execution engines [5], [6], [7]. Asynchronous calls have also been used to hide memory access latency by issuing prefetch requests [8]. Yeung [9] proposes deferred execution of remote procedure calls and code shipping as a way of reducing latency, but the work does not consider asynchronous calls.

While our transformation rules are based on [3], we make the following novel contributions. First, we present a novel statement reordering algorithm to enable loop fission. The statement reordering algorithm greatly increases the applicability of the other transformation rules as shown by our experimental study involving several real-world applications. The statement reordering algorithm presented in this paper is useful not only for asynchronous query submission but also for batching. Second, we show how the transformation rules presented in [3] can be adapted for asynchronous query submission. Third, we formally characterize the programs that can be rewritten for asynchronous query submission, which we believe is an important theoretical contribution.

More recently, Manjhi [10] considers prefetching of query results by employing non-blocking database calls. Non-blocking query execution requests are made eagerly, as soon as the values for the query parameters are known. A blocking call is subsequently issued when the results of the query are needed, and this call is likely to take much less time as the query results would be already computed and available in the cache.

Similar to the work of Manjhi [10] our work considers rewriting database application code for prefetching query results. Manjhi [10] considers only straight-line code while

exploiting opportunities for prefetching. In many practical applications, the results of a query are consumed by the very next statement that follows the query execution statement (see Example 2), which forces immediate blocking if one considers only straight-line code. Such opportunities can only be exploited by loop transformations, which is the main focus of this paper.

Two models are prevalent for coordinating asynchronous calls: the *observer model* and the *callback model*.

The Observer Model: In this model, the calling program explicitly polls the status of the asynchronous call it has made. When the results of the call are strictly necessary to make any further computation, the calling program blocks until the results are available. The observer model is suitable when the results of the calls must be processed in the order in which the calls are made. Example 1 of Section I shows a program making use of the observer model to coordinate the asynchronous query execution. We now formally define the semantics of the methods we use.

- *executeQuery*: Submits a query to the database system for execution, and returns the results. The call blocks until the query execution completes.
- *submitQuery*: Submits a query to the database system for execution, but the call returns immediately with a handle (without waiting for the query execution to finish).
- *fetchResult*: Given a handle to an already issued query execution request, this method returns the results of the query. If the query execution is in progress, this call blocks until the query execution completes.

The Callback Model: In this model, the calling program registers a callback function as part of the non-blocking call. When the request completes, the callback function is invoked to process the results of the call. The event driven model is suitable when the program logic to process the call results is small and the order of processing the results is unimportant.

The program transformations presented in this paper make use of the observer model for asynchronous query submission. It is possible to extend the proposed approach to make use of the callback model for programs in which the order of processing the query result is unimportant. However, the details of such extensions are not part of this paper.

III. Basic Transformations

Guravannavar et.al. [3] present a set of program transformation rules to rewrite program loops so as to enable batched bindings for queries. In this section, we show how some of these transformation rules can be extended for asynchronous query submission. We then present a novel statement reordering algorithm, in the next section, which significantly improves the applicability of the transformation rules.

The program transformation rules we present, like the equivalence rules of relational algebra, allow us to repeatedly 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 replacing a blocking query execution statement with a non-blocking statement. Applying any rule results in an equivalent program and hence the rule application process can be stopped at any time. We omit a formal proof of correctness for our transformation rules, and refer the interested reader to [11]. Each program transformation rule has not only a syntactic pattern to match, but also certain pre-conditions to be satisfied. The pre-conditions make use of the inter-statement data dependencies obtained by static analysis of the program. Before presenting the formal transformation rules, we briefly describe the *data dependence graph*, which captures the various types of inter-statement data dependencies.

A. Data Dependence Graph

Inter-statement dependencies are best represented in the form of a data dependence graph [1] or its variant called the program dependence graph [12]. The *Data Dependence Graph* (DDG) of a program is a directed multi-graph in which program statements are nodes, and the edges represent data dependencies between the statements. The data dependence graph for the program of Example 2 is shown in Figure 1. The types of data dependence edges are explained below.

- A *flow-dependence* edge ($\xrightarrow{FD}$) exists from statement (node) s_a to statement s_b if s_a writes a location that s_b may read, and s_b follows s_a in the forward control-flow. For example, in Figure 1, a flow-dependence edge exists from node s2 to node s3 because statement s2 writes *category* and statement s3 reads it.
- An *anti-dependence* edge ($\xrightarrow{AD}$) exists from statement s_a to statement s_b if s_a reads a location that s_b may write, and s_b follows s_a in the forward control flow. For example, in Figure 1, an anti-dependence edge exists from node s1 to node s2 because statement s1 reads *categoryList* and statement s3 writes it.
- An *output-dependence* edge ($\xrightarrow{OD}$) exists from statement s_a to s_b if both s_a and s_b may write to the same location, and s_b follows s_a in the forward control flow.
- A *loop-carried flow-dependence* edge ($\xrightarrow{LFD_L}$) exists from s_a to s_b if s_a writes a value in some iteration of a loop L and s_b may read the value in a later iteration. For example, in Figure 1, a loop-carried flow-dependence edge exists from node s2 to node s1 because statement s2 writes *categoryList* and statement s1 reads it in a subsequent iteration. Similarly, there are loop carried counter parts of *anti* and *output* dependencies, which are denoted by ($\xrightarrow{LAD_L}$) and ($\xrightarrow{LOD_L}$) respectively.
- External data dependencies: Program statements may have dependencies not only through program variables but also through the database and other external resources like files. For example, we have $s_1 \xrightarrow{FD} s_2$ if s_1 writes a value to the database, which s_2 may read subsequently. Though standard dataflow analysis performed by compilers considers only dependencies through program

variables, it is not hard to extend the techniques to consider external dependencies, at least in a conservative manner. For instance, we could model the entire database (or file system) as a single program variable and thereby assume every query/read operation on a database/file to be conflicting with an update/write of the database/file. In practice, it is possible to perform a more accurate analysis on the external writes and reads.

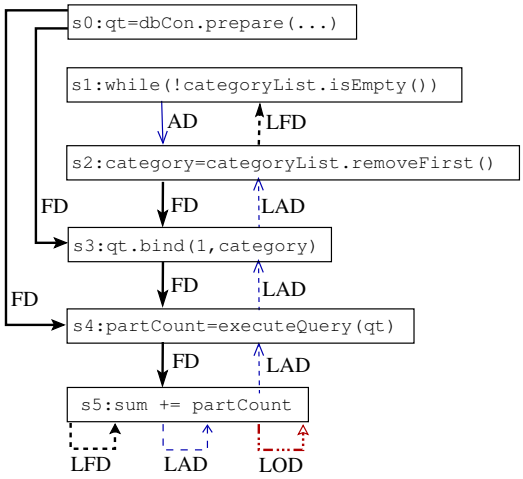

Fig. 1. Data Dependence Graph for Example 2

B. Basic Loop Fission Transformation

Consider the program fragment shown in Example 2 and its rewritten form shown in Example 3. The key transformation, to enable such a program rewriting is *loop fission* (or *loop distribution*) [2]. Guravannavar et.al. [3] make use of *loop fission* to replace iterative query executions with a batched (or set-oriented) query execution. In this section, we show how the program transformation rules proposed by Guravannavar et.al. [3] can be extended for rewriting programs to make use of asynchronous calls. A formal specification of the transformation is given as Rule A, which is a variant of the loop fission transformation presented in Guravannavar et.al. [3]. The LHS of the rule is a generic *while* loop containing a blocking query execution statement s. ss_1 and ss_2 are sequences of statements, which respectively precede and succeed the query execution statement in the loop body. The LHS of the rule then lists two pre-conditions, which are necessary for the rule to be applicable. The RHS of the rule contains two loops, the first one making asynchronous query submissions and the second one performing a blocking fetch followed by execution of statements that process the query results.

Note that any number of query execution statements within a loop can be replaced by non-blocking calls by repeatedly applying the loop fission transformation. Although we present the loop fission transformation rule *w.r.t.* a *while* loop, variants of the same transformation rule can be used to split set iteration loops (such as the second loop in the RHS of the Rule A).

Rule A Basic Equivalence Rule for Loop Fission

while p **loop**
  ss_1; s: v = executeQuery(q); ss_2;
end loop;

such that:

(a) No loop-carried flow dependencies (*i.e.,* LCFD edges, external or otherwise) cross the points before and after the query execution statement s.
(b) No loop-carried *external* anti or output dependencies cross the points before and after s.

⇕

Table(T) t;
int loopkey = 0;
while p **loop**
  Record(T) r; ss'_1;
  r.handle = submitQuery(q); r.key=loopkey++;
  t.addRecord(r);
end loop;
for each r in t **order by** t.key **loop**
  ss_r; v = fetchResult(r.handle); ss_2;
end loop;
delete t;

where the schema T and statement sequences ss'_1, ss_r are constructed as follows.
Let *SV* (split variables) be the set of variables for which either an LCAD or LCOD edge crosses the split boundaries (the edge is incident from ss_2 to s or ss_1, or from s to ss_1).

1) Table t and record r have attributes corresponding to each variable in *SV* and a key.
2) ss'_1 is same as ss_1 but with additional assignment statements to attributes of r. Each write to a split variable v is followed by an assignment statement $r.v = v$;. If the write is conditional, then the newly added statement is also conditional on the same guard variable.
3) ss_r is a statement sequence assigning attributes of r to corresponding variables. Each assignment in ss_r is conditional; the assignment is made only if the attribute of r is non-null (*assigned*).

Rule A makes an improvement of the fundamental nature to the loop fission transformation proposed by Guravannavar et.al. [3]. Rule A significantly relaxes the pre-conditions (see Rule 2 in [3]). For instance, Rule A allows loop-carried output dependencies to cross the split boundaries of the loop.

Applicability

The pre-condition that no loop-carried flow dependencies cross the point of split can seriously limit the applicability of Rule A. In the next section, we show examples to illustrate this limitation, and then present a solution to address the issue. Further, Rule A is also not directly applicable when the query execution statement lies inside a compound statement. We now present additional transformation rules which can be used to address this restriction.

C. Control Dependencies

Consider the initial program shown in Example 4. The query execution statement appears in a conditional block. This prohibits direct application of Rule A to split the loop at

the program point immediately following the query execution statement.

Conditional branching (*if-then-else*) and *while* loops lead to *control dependencies*. If the predicate evaluated at a conditional branching statement $s1$ determines whether or not control reaches statement $s2$, then $s2$ is said to be control dependent on $s1$. During loop split, it may be necessary to convert the control dependencies into flow dependencies [2], by introducing boolean variables and guard statements.

In Example 4, we apply Rule B and introduce a boolean variable c to remember the result of the predicate evaluation, and then convert the statements inside the conditional block into guarded statements. We can then apply Rule A and split the loop, as shown in the last part of Example 4. The formal specification of the transformation is given as Rule B.

Rule B Converting control-dependencies to flow-dependencies

if (p) { ss_1 } else { ss_2 }

$\Updownarrow$

boolean $cv = p$;
ss

where $ss[i] = (cv == true)?ss_1[i], 1 \leq i \leq |ss_1|$ and
$ss[k+j] = (cv == false)?ss_2[j], 1 \leq j \leq |ss_2|, k = |ss_1|$

D. Nested Loops

A query execution statement may be present in an inner loop that is nested within an outer loop. In such a case, it may be possible to split both the inner and the outer loops, thereby increasing the number of asynchronous query submissions before a blocking fetch is issued. To achieve this, we first split the inner loop and then the outer loop. Such a transformation is illustrated in Example 5. Note that the temporary table introduced during the inner loop's fission becomes a nested table for the temporary table introduced during the outer loop's fission. As the idea is straight-forward, we omit a formal specification of this rule.

IV. Statement Reordering

For several practical cases, the loop fission transformation given in Rule A may not be applicable directly, as the preconditions for its applicability are too restrictive. Consider the program in Example 6. We cannot directly split the loop so as to make the query execution statement (s2) non-blocking, because there are loop-carried flow-dependencies from statement s4 to s1 and to the loop predicate, which violate pre-condition (a) of Rule A. Statement s4, which appears after s1, writes a value and statement s1 reads it in a subsequent iteration. Such cases are very common in practice (*e.g.,* in most *while* loops the last statement affects the loop predicate, introducing a loop-carried flow dependency).

Fortunately, in many cases it is be possible to reorder the statements within a loop so as to make loop fission possible, without affecting the correctness of the program. For example, the statements within the loop of Example 6, if reordered as shown in Example 7, permit loop fission. Note that in the transformed program of Example 7 there are no loop-carried flow dependencies, which prohibit the application of Rule A to split the loop at the query execution statement.

In general, reordering of statements to enable loop fission is a non-trivial task as there can be arbitrary inter-statement dependencies in the loop. In this section, we present an algorithm for reordering statements within a loop so as to enable splitting of the loop at the desired statement boundary. Our reordering algorithm succeeds in enabling loop fission at the boundaries of the query execution statement if the statement does not lie on a cycle of flow (and loop-carried flow) dependencies.

Example 4 *Transforming Control-Dependencies to Flow-Dependencies*

Initial Program

```
for (i=0; i < n; i++) {
    v = foo(i);
    if ( v == 0) {
        v = executeQuery(q);
        log(v);
    }
    print(v);
}
```

After applying Rule B

```
for (i = 0; i < n; i++) {
    v = foo(i);
    // Convert control deps to flow deps by
    // making use of a guard variable.
    boolean c = (v == 0);
    c==true? v = executeQuery(q);
    c==true? log(v);
    print(v);
}
```

After applying Rule A

```
Table(key, v, c, handle) t;
for (i = 0; i < n; i++) {
    Record r;
    v = foo(i); r.v = v;
    boolean c = (v == 0); r.c = c;
    c==true? r.handle = submitQuery(q);
    r.key = loopkey++;
    t.addRecord(r);
}
for each r in t order by key loop
    v = r.v; c = r.c; handle = r.handle;
    c==true? v = fetchResult(handle);
    c==true? log(v);
    print(v);
}
```

Example 5 *Dealing with nested loops*

```
while(pred1) {
    while(pred2) {
        x = executeQuery(q); process(x);
    }
}
```

After Transformation

```
Table tp;
while(pred1){
    Table tc; Record rp;
    while(pred2){
        Record rc;
        rc.handle = submitQuery(q);
        tc.addRecord(rc);
    }
    rp.tc = tc; tp.addRecord(rp);
}
for each rp in tp {
    for each rc in rp.tc {
        x = fetchResult(rc.handle); process(x);
    }
}
```

Example 6 *An example where loop fission is not directly applicable due to loop-carried dependencies*

```
qt = dbCon.prepare( "select count(partkey)
                     from part where p_category=?");
category = readInputCategory();
while(category != null) {
    qt.bind(1, category);                          (s1)
    partCount = executeQuery(qt);                  (s2)
    sum += partCount;                              (s3)
    category = getParentCategory(category);        (s4)
}
```

The preliminary idea of reordering statements by introducing temporary variables is presented in [3]. The basic rules that allow us to reorder statements are specified in Rule C, which is a minor variant of Rule 5 in [3]. However, to be able to split a loop at a desired point, multiple applications of Rule C may be needed. It is important that Rule C be applied in an appropriate sequence so as to achieve the desired reordering. We now give a novel algorithm to do so. The goal is to reorder the statements such that no loop-carried flow dependencies cross the desired split boundary. We make use of the following definition in the description to follow.

Definition 4.1: A true-dependence path (or cycle) in a data dependence graph is a directed path (or cycle) where each edge represents either a flow-dependence (FD) or a loop-carried flow-dependence (LCFD).

Note that a true-dependence path excludes anti, output, loop-carried anti and loop-carried output dependence edges. □

Example 7 *After reordering the statements in Example 6*

```
qt = dbCon.prepare( "select count(partkey)
                     from part where p_category=?");
category = readInputCategory();
while(category != null) {
    temp_category = category;
    category = getParentCategory(category);
    qt.bind(1, temp_category);
    partCount = executeQuery(qt);
    sum += partCount;
}
```

Rule C Basic Rules that Facilitate Reordering of Statements

Rule C1: Reordering Independent Statements
Two statements can be reordered if there exists no dependence between them.

$s_1; s_2;$ where *indep*(s_1, s_2) $\iff$ $s_2; s_1;$

Rule C2: Shifting an Anti-Dependence Edge
An anti-dependence edge between two statements can be shifted by using an extra variable.

$s_1; s_2;$
where $s_1 \xrightarrow{AD_v} s_2$
$\Updownarrow$
$v' = v; s_1'; s_2;$

where s_1' is constructed from s_1 by replacing all reads of v by reads of v'.

Rule C3: Shifting an Output-Dependence Edge

$s_1; s_2;$
where $s_1 \xrightarrow{OD_v} s_2$
$\Updownarrow$
$s_1; s_2'; v = v';$

where s_2' is constructed from s_2 by replacing all writes of v by write to v'.

The algorithm *reorder*, shown in Figure 2, works as follows. For each loop-carried flow dependence edge that crosses the split boundary (the program point in the basic block that immediately succeeds the blocking query execution statement), the algorithm decides the statement to move, and its target position. There are four cases to consider while deciding the statement to move and its target position. The cases are shown in Figure 3. The way in which we choose the *stmtToMove* and *targetStmt* ensures the following. If s_q, the blocking query execution statement, does not lie on a true-dependency cycle, then there exists no true-dependence path from the *stmtToMove* to the *targetStmt*. We then compute the set *srcDeps*, which comprises of all statements present between *stmtToMove* and *targetStmt*, and which have a path of flow-dependence edges from *stmtToMove*. Each statement in

```
procedure reorder(BasicBlock b, Stmt s_q)
// Goal: Reorder the statements within b, such that no LCFD
// edges cross the program point immediately succeeding s_q.
// Assumption: s_q does not lie on a true-dependence cycle in
// the subgraph of the DDG induced by statements in b.
begin
  while there exists an LCFD edge crossing the split
  boundary for s_q
    Pick an LCFD edge (v1, v2) crossing the split boundary.

    if there exists a true-dependence path from v1 to s_q
    /* Implies no true-dependence path from s_q to v1 */
      stmtToMove = s_q;
      targetStmt = v1;
    else
    /* No true-dependence path from v1 to s_q, which implies
       no true-dependence path from v2 to s_q as there
       exists an LCFD edge from v1 to v2 */
      stmtToMove = v2;
      targetStmt = s_q;

    // Move stmtToMove past the targetStmt
    Compute srcDeps, the set of all statements between
    stmtToMove and targetStmt, which have a
    flow dependence path from stmtToMove.

    while srcDeps is not empty
      Let v be the statement in srcDeps closest to
      targetStmt
      moveAfter(v, targetStmt);      // see Figure 4

    moveAfter(stmtToMove, targetStmt);
end;
```

Fig. 2. Procedure *reorder*

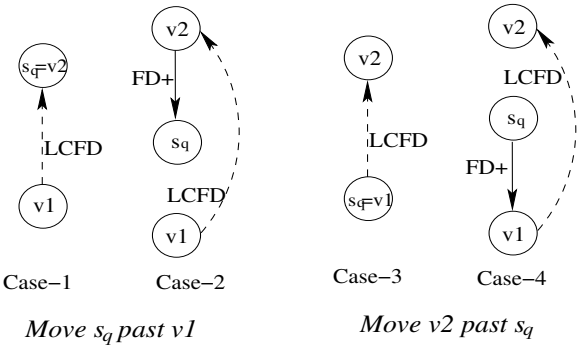

Fig. 3. Cases for Reordering Statements

srcDeps is then moved past the *targetStmt* using the *moveAfter* procedure. The procedure *moveAfter* (shown in Figure 4) performs the required reordering by swapping pairs of adjacent statements. While doing so, the procedure resolves any anti and output dependencies by creating stub statements, which make use of temporary variables.

Examples 8, 9 and 10 illustrate the working of the statement reordering algorithm. Figure 5 shows the data dependence graph for the original and reordered code of Example 10. In Figure 5, for each flow-dependence (FD) edge from x to y, there exists a corresponding loop-carried anti-dependence (LCAD) edge from y to x, but these edges are not shown.

```
procedure moveAfter(Stmt s, Stmt t)
External variables used:
    List srcDeps, Stmt s_q      // Variables assigned in reorder
begin
    if s succeeds t in the basic block
        return;
    Stmt next = successor(s);
    do {
        if no flow/anti/output dependence edges between
            s and next
            /* Reorder the statements by applying Rule-C1 */
            swap s and next;
        else {
            // Let OD_v: denote output dependence on variable v
            for each OD_v edge from s to next {
                /* Shift the OD edge by applying Rule-C3 */
                Replace writes to v in next by writes to a new
                variable v';
                Insert a new statement as'_v that assigns v' to
                v immediately after next;
                moveAfter(as'_v, t);
            }

            // Let AD_v denote anti-dependence on variable v
            for each AD_v edge from s to next {
                /* Shift the AD edge by applying Rule-C2 */
                if there exists an AD_v edge from s_q to next
                  // Use a reader stub
                    Insert a new statement as'_v that assigns v to a
                    new temp variable v' immediately before s;
                    Replace all read references to v in s by v';
                else      // Use writer stub
                    Replace write of v in next by write to a new
                    temp var v';
                    Insert a new statement as_v that assigns v' to
                    v immediately after next;
                    moveAfter(as_v, t);
            }
            swap s and next;
        }
        lastStmt = next;
        if (lastStmt != t)
            next = successor(s);
    }
    while(lastStmt != t) ;
end
```

Fig. 4. Procedure *moveAfter*

Similarly, AD and OD edges have corresponding LCFD and LCOD edges respectively, which are not shown. In this example, $s1$ is the blocking query execution statement. The LCFD edge from $s4$ to $s1$ crosses the split boundary and hence $s1$ must be moved past $s4$. As can be seen in Figure 5, after the reordering, no LCFD edges cross the split boundary.

A. Applicability of Transformation Rules

Although our program transformation algorithm succeeds in rewriting fairly complex programs for asynchronous query submission, not every program can be rewritten this way. The inter-statement data dependencies may prohibit a blocking query execution statement from being converted to a non-blocking statement. In this section, we formally identify the condition for such a transformation to be possible.

Example 8 *Illustration 1 of Statement Reordering*

while(*category != null*) **loop**
(s1) icount = **q(category)**;
(s2) sum = sum + icount;
(s3) category = getParent(category);
end loop;

After moving s1 past s3

while(*category != null*) **loop**
(ts1) category1 = category;
(s3) category = getParent(category);
(s1) icount = **q(category1)**;
(s2) sum = sum + icount;
end loop;

Example 9 *Illustration 2 of Statement Reordering*

while(*top > 0)* **loop**
(s6) top = top-1;
(s7) curcat = stack[top];
(s8) catitems = **q(curcat)**;
(s9) totalcount = totalcount + catitems;
(s10') stack, top = block(curcat, top);
end loop;

After moving s8 past s10'

while(*top > 0)* **loop**
(s6) top = top-1;
(s7) curcat = stack[top];
(s10') stack, top = block(curcat, top);
(s8) catitems = **q(curcat)**;
(s9) totalcount = totalcount + catitems;
end loop;

As an example, consider the program shown in Example 11, and its DDG shown in Figure 6 (this DDG is obtained after transforming the control dependencies to flow dependencies using Rule B). The query invocation in statement `s2` can be made non-blocking but not the one in statement `s1`. The query invocation in statement `s1` lies on the true-dependence cycle $\texttt{s1} \xrightarrow{FD} \texttt{s4} \xrightarrow{LFD} \texttt{s1}$, and hence we cannot reorder the statements so as to satisfy the pre-conditions of Rule A.

Note that flow dependencies that result from control-dependencies (Rule B) must be taken into account while checking for the presence of a true-dependence cycle. Intuitively, a call cannot be converted to a non-blocking call if its execution in any iteration depends on the value it returned in a previous iteration.

Theorem 4.1: Given a basic block of code b and statement s_q in b such that s_q does not lie on a true-dependence cycle in the DDG, procedure reorder terminates, reordering the statements of b such that:

Example 10 *Illustration 3 of Statement Reordering*

Original Program	**After moving s1 past s4**
while(*pred(c)*) **loop**	**while**(*pred(c)*) **loop**
(s1) cv1? a = **q(b)**;	*(s2) cv2? a3,c = f(x);*
(s2) cv2? a,c = f(x);	*(n1) b2 = b;*
(s3) d = g(a, b);	*(n2) b5 = b;*
(s4) cv3? a,b = h(c);	*(s4) cv3? a1,b = h(c);*
end loop;	*(s1) cv1? a* = **q(b5)**;
	(n3) cv2? a = a3;
	(s3) d = g(a, b2);
	(n4) cv3? a = a1;
	end loop;

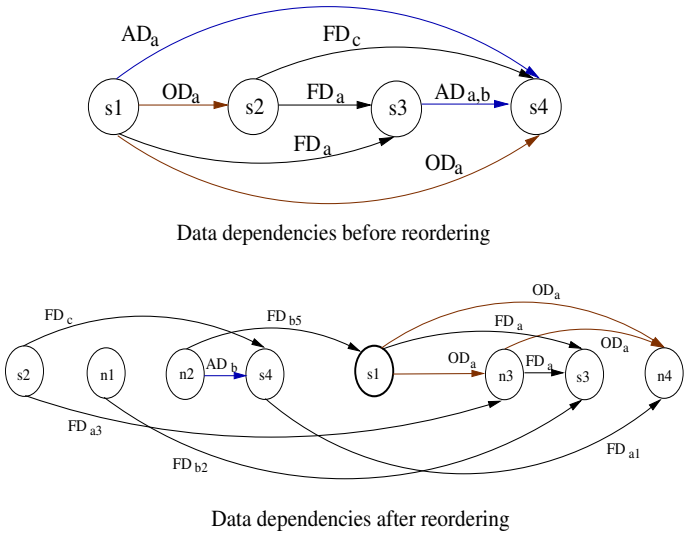

Fig. 5. Data Dependence Graphs for Example 10

(a) No LCFD edges cross the program points that immediately precede and succeed s_q.
(b) Program correctness is preserved (*i.e.*, the reordered block is equivalent to the original)

The proof of Theorem 4.1 can be found in [11].

V. System Design

Our rewrite rules can conceptually be used with any language. We chose Java as the target language and JDBC as the interface for database access. To implement the rules we need to perform dataflow analysis of the given program and build the data dependence graph. We used the SOOT optimization framework [13]. SOOT uses an intermediate code representation called Jimple and provides dependency information on Jimple statements. Our implementation transforms the Jimple code using the dependence information. Finally, the Jimple code is translated back into a Java program.

The important phases in the program transformation process are shown in Figure 7. The main task of our program transformation tool appears in the *Apply Async Trans Rules* phase. The program transformation rules are applied in an iterative manner, updating the dataflow information each time the code

Example 11 *Statement with Cyclic True-Dependencies*

```
while(eid != NULL) loop                                   (s0)
    mgr =SELECT manager                                   (s1)
          FROM emp WHERE empid=eid;
    idx = SELECT perfindex FROM rating                    (s2)
          WHERE reviewer=mgr and reviewed=eid;
    sumidx += idx;                                        (s3)
    eid = mgr;                                            (s4)
end loop;
```

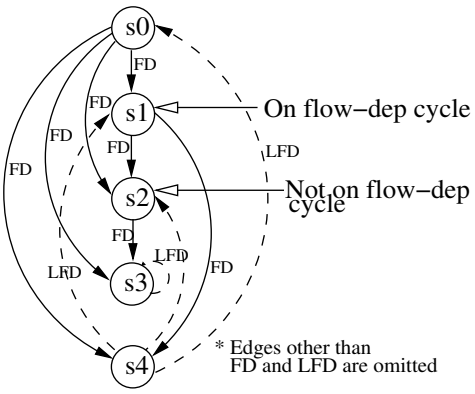

Fig. 6. DDG for Example 11

changes. The rule application process stops when all (or the user chosen) query execution statements, which do not lie on a true-dependence cycle, are converted to asynchronous calls.

There were several challenges in implementing our program transformation tool, which has the following design goals.

1) Readability of the transformed code
2) Robustness for variations in intermediate code
3) Extensibility

Since our program transformations are source-to-source, maintaining readability of the transformed code is important. We achieve this goal through several measures. (a) The transformed code mostly uses standard JDBC calls and very few calls to our custom runtime library. This is achieved by providing a set of JDBC wrapper classes. The JDBC wrapper classes and our custom runtime library hide the complexity of asynchronous calls. (b) When we apply Rule B followed by Rule A to split a loop, the resulting code will have many guarded statements. This leads to a very different control structure as compared to the original program. 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.

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. We have designed our program transformation tool for robust matching of desired program fragments. The tool can handle several variations in the

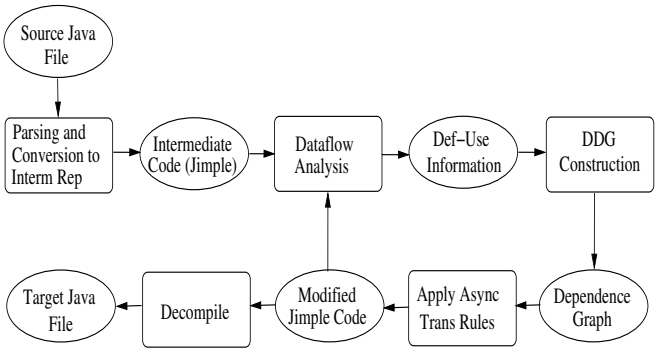

Fig. 7. Program Transformation Phases

intermediate (Jimple) code.

One of our design goals has been extensibility. Each of the transformation rules has been coded as a separate class. Application of any transformation rule independently must preserve the correctness of the program. Such a design makes it easy to add new program transformation rules.

VI. Experimental Results

For evaluating the applicability and benefits of the proposed transformations, we consider five Java applications: two publicly available benchmarks (which were also considered by Manjhi et.al. [14]) and three other real-world applications we encountered. Our current implementation does not support all the transformation rules presented in this paper, and does not support exception handling code. Hence, in some cases part of the rewriting was performed manually in accordance with the transformation rules. We performed the experiments with two widely used database systems - a commercial system we call SYS1, and PostgreSQL. The SYS1 database server was running on a 64 bit dual-core machine with 4 GB of RAM, and PostgreSQL was running on a machine with two Xeon 3 GHz processors and 4 GB of RAM. Since disk IO is an important parameter that affects the performance of applications, we report the results for both warm cache and cold cache. The Java applications were run from a remote machine connected to the database servers over a 100 Mbps LAN. The applications used JDBC API for database connectivity. The transformed programs use the *Executor* framework of the *java.util.concurrent* package for thread scheduling and management.

Experiment 1: Auction Application: We consider a benchmark application called RUBiS [15] that represents a real world auction system modeled after *ebay.com*. The application has a loop that iterates over a collection of comments, and for each comment loads the information about the author of the comment. The *comments* table had close to 600,000 rows, and the *users* table had 1 million rows. First, we consider the impact of our transformations as we vary the number of loop iterations, fixing the number of threads at 10. Figure 8 shows the performance of this program before and after the transformations with warm and cold caches in log scale. The y-

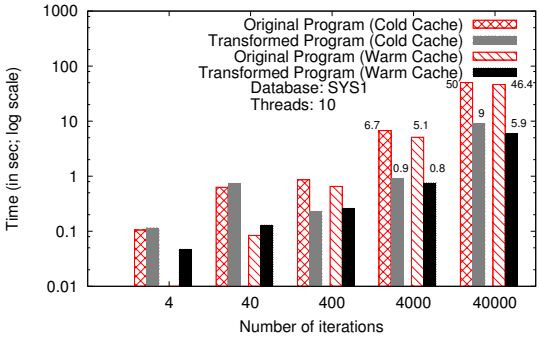

Fig. 8. Experiment 1 with varying number of iterations

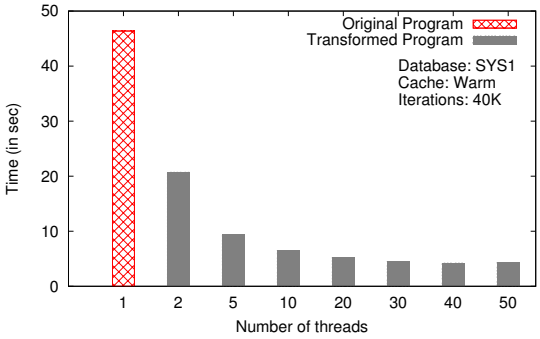

Fig. 9. Experiment 1 with varying number of threads

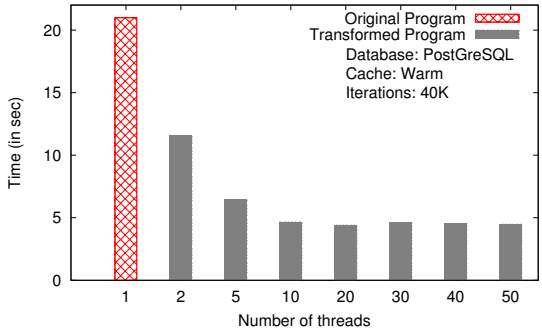

Fig. 10. Experiment 1 with varying number of threads

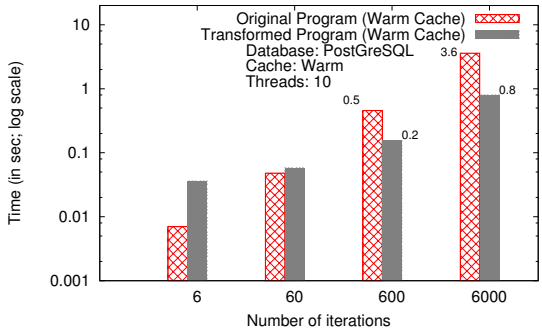

Fig. 11. Experiment 2 with varying number of iterations

axis denotes the end to end time taken for the loop to execute, which includes the application time and the query execution time.

For a small number of iterations, the transformed program is slower than the original program. The overhead of thread creation and scheduling overshoots the query execution time. However, as the number of iterations increases, the benefits of our transformations increase. For the case of 40,000 iterations, we see an improvement of a factor of 8.

Next, we keep the number of iterations constant (at 40,000) and vary the number of threads. The results of this experiment are shown in Figure 9. The execution time (for both the warm and cold cache) drops sharply as the number of threads is increased, but gradually reaches a point where the addition of threads does not improve the execution time.

The results of the above experiment on PostgreSQL are shown in Figure 10, which follow the same pattern as in the case of SYS1.

Experiment 2: Bulletin Board Application: RUBBoS [15] is a benchmark bulletin board-like system inspired by *slashdot.org*. For our experiments we consider the scenario of listing the top stories of the day, along with details of the users who posted them. Figure 11 shows the results of our transformations with different number of iterations. Although the transformed program takes slightly longer time for small number of iterations, the benefits increase with the number of iterations (note the log scale of y-axis).

Experiment 3: Category Traversal: This program, taken from [3], finds the part with maximum size under a given category (including all its sub-categories) by performing a DFS of the category hierarchy. For each node (category) visited, the program queries the item table. The TPC-H part table, augmented with a new column category-id and populated with 10 million rows, was used as the item table. The category table had 1000 rows - 900 leaf level, 90 middle level and 10 top level categories (approximately). A clustering index was present on the category-id column of the category table and a secondary index was present on the category-id column of the item table.

Figure 12 shows the performance of this program before and after applying our transformation rules. As in the earlier example, we first fix the number of threads and vary the number of iterations. We perform this experiment with ten threads, on a warm cache on SYS1. The results are in accordance with our earlier experiments. In addition, we observe that the number of threads is an important parameter in such scenarios. This parameter is influenced by several factors, such as the number of processor cores available for the database server and the client, the load on the database server, the amount of disk IO, CPU utilization etc. The effect of varying number of threads can be more clearly observed in Figure 13, where we keep the number of iterations constant (at 100) and vary the number of threads from 1 to 50.

The trends in Figure 13 are very similar for both the warm and cold cache, though the actual numbers differ. When the program is run with a cold cache, the amount of disk IO involved in running the queries is substantially higher than with a warm cache. But the bottleneck of disk IO can be reduced by issuing overlapping requests. Such overlapping

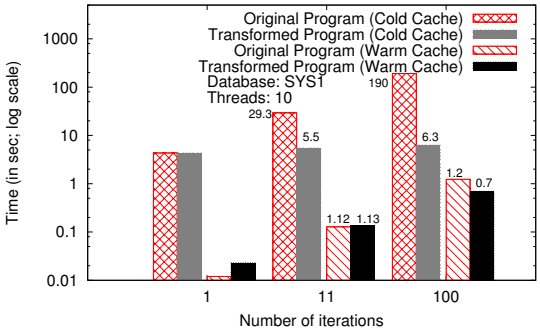

Fig. 12. Experiment 3 with varying iterations

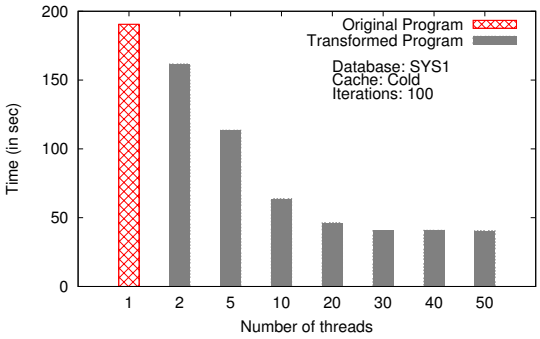

Fig. 13. Experiment 3 with varying number of threads

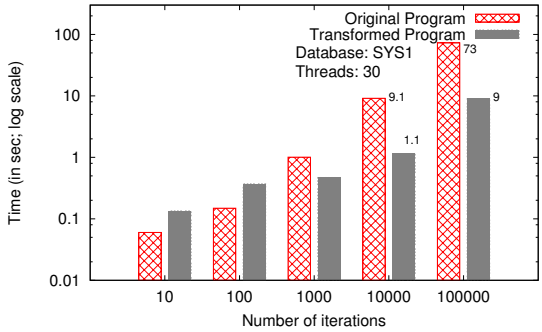

Fig. 14. Experiment 4 with varying number of iterations

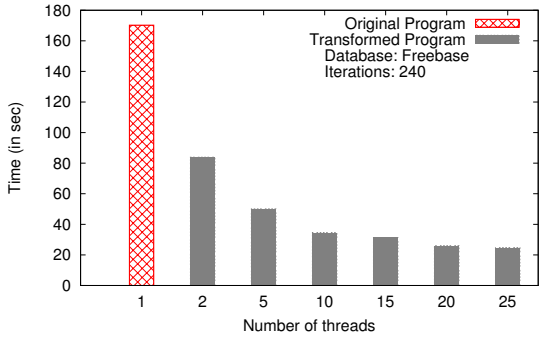

Fig. 15. Experiment 5 with varying number of threads

query submissions enable the database system to choose plan strategies such as shared scan.

In transforming this program, the reordering algorithm was first applied and then the loop was split using Rule A.

Experiment 4: Value Range Expansion: In this application, taken from [3], data about forms issued to various agents would arrive in the format *(agent-id, start-form-number, end-form-number)*. The program would iterate over all the form issue records, expand the issue range and populate the *forms-master* table with entries corresponding to each individual form. The purpose was to be able to update and track the status of each individual form subsequent to its issue. The original program had an outer loop iterating over the *form issue* records and an inner loop iterating over the range *(start-form-number, end-form-number)*. An INSERT operation was performed inside the inner loop. The transformed program could asynchronously submit the INSERT operations. The running times of the original and transformed program are shown in Figure 14 in log scale. Since this program performs no reads, the results are independent of the cache state.

This program required the reordering algorithm to be first applied for the loop to be split using Rule A.

Experiment 5: Web service invocation: Although we presented our program transformation techniques in the context of database queries, the techniques are more general in their applicability, and can be used with requests such as Web service calls. In this experiment, we consider an application that fetches data about directors and their movies from Freebase [16], a social database about entities, spanning millions of topics in thousands of categories. It is an entity graph which can be traversed using an API built using JSON over HTTP. The client application, written in Java, retrieves the movie and actor information for all actors associated with a director. Such applications usually require the execution of a sequence of queries from within a loop because (a) operations such as joins are not possible directly, and (b) the Web service API may not be supporting set oriented queries.

Since our current implementation supports only JDBC API, we manually applied the transformations for the code which fires the JSON queries. The results of this experiment are shown in Figure 15. As we vary the number of threads, overlapping HTTP requests are made by the client application which saves on network round-trip delays. Since our experiment used the publicly available Freebase sandbox over the Internet, the actual time taken can vary with network load. However, we expect the relative improvement of the transformed program to remain the same. This experiment demonstrates the applicability of our transformation rules beyond database query submission.

Applicability of Transformation rules: In order to evaluate the applicability of our transformation rules, we consider the two publicly available benchmark applications used above, the auction application and the bulletin board application. For each of these, we have analyzed the source code to find out (a) how many opportunities for asynchronous submission of queries exist, and (b) how many of those opportunities are exploited

TABLE I

APPLICABILITY OF TRANSFORMATION RULES

Application	# Opportunities	# Transformed	Applicability (%)
Auction	9	9	100
Bulletin Board	8	6	75

by our transformation rules. The results of the analysis is presented in Table I. We consider all kinds of loop structures which include a query execution statement in the loop body, as potential opportunities (# Opportunities). Among such potential opportunities, those which satisfy the preconditions for our rules, are exploited (# Transformed). This would involve reordering of statements in a lot of situations.

We see that all such opportunities present in the auction system indeed satisfy the preconditions and can be transformed. In the bulletin board application, few of the loops performed recursive method invocations which prevent them from being transformed. Out of the five programs seen earlier, the remaining three were too small for this analysis, and hence omitted.

Time Taken for Program Transformation: Although the time taken for program transformation is usually not a concern (as it is a one-time activity), we note that, in our experiments the program transformation took very little time (less than a second).

VII. DISCUSSION

We now discuss some future directions to our work.

Which calls to be transformed?: It may not be beneficial to transform every blocking query submission call to a non-blocking call. From our experimental study it is also evident that given a query execution statement, the benefit to be achieved by converting it to a non-blocking call depends on the number of iterations and other system parameters. In our current implementation we assume that user can specify which query submission statements to be transformed. Making this decision in a cost-based manner is a future work.

Minimizing memory overheads: If the number of loop iterations is large, the transformed program incurs high memory overhead, because we need to store the *handle* and the state associated with each loop iteration in an in-memory table. This problem can be addressed in two ways: (a) materialize part of the in-memory table to the disk, or (b) limit the number of loop iterations performed before the results are processed. It is possible to extend our loop fission transformation to allow the second loop (which consumes the query results) to begin after a specific number of asynchronous query submissions. This can be achieved by enclosing the two loops generated after the fission into a parent loop. We omit the details of this extension from this paper.

How many threads to use?: Our experiments show that the optimal number of threads differs from case to case. Identifying the optimal number of threads for a given case is a challenging problem. Several factors, specific to both the program and the system/deployment environment, influence the decision on the number of threads to use. This is another direction for our future work.

Updates and Transactions: In this paper, we have not addressed issues related to the interaction between asynchronous queries and transaction semantics. Although this is a non-issue for read-only queries, rewriting loops containing update transactions needs more thought.

VIII. CONCLUSION

We propose a program analysis and transformation based approach to automatically rewrite database applications to exploit the benefits of asynchronous query submission. The program transformation rules and algorithms presented in this paper significantly increase the applicability of known techniques to address this problem. We provide a sufficient condition on the data dependence graph, which characterizes the program statements that can be transformed with our approach. Although our program transformations are presented in the context of database queries, the techniques are general in their applicability, and can be used in other contexts such as calls to Web services, as shown by our experiments. We presented a detailed experimental study, carried out on real-world and publicly available benchmark applications. Our experimental results show performance gains to the extent of 75% in several cases. Finally, we identify some interesting directions along which this work can be extended.

REFERENCES

[1] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[2] K. Kennedy and K. S. McKinley, "Loop Distribution with Arbitrary Control Flow," in *Proceedings of Supercomputing*, 1990. [Online]. Available: citeseer.ist.psu.edu/kennedy90loop.html

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

[4] "Kernel Asynchronous I/O (AIO) Support for Linux http://lse.sourceforge.net/io/aio.html."

[5] G. Graefe, "Executing Nested Queries," in *10th Conference on Database Systems for Business, Technology and the Web*, 2003.

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

[7] S. Iyengar, S. Sudarshan, S. Kumar, and R. Agrawal, "Exploiting Asynchronous IO using the Asynchronous Iterator Model," in *Intl. Conf. on Management of Data (COMAD)*, 2008.

[8] S. P. Vanderwiel and D. J. Lilja, "Data Prefetch Mechanisms," *ACM Computing Surveys*, vol. 32, no. 2, 2000.

[9] K. C. Yeung, "Dynamic Performance Optimisation of Distributed Java Applications," Ph.D. dissertation, Imperial College of Science, Technology and Medicine, 2004.

[10] A. Manjhi, "Increasing the Scalability of Dynamic Web Applications," Ph.D. dissertation, Carnegie Mellon University, 2008.

[11] R. Guravannavar, "Optimization and evaluation of nested queries and procedures," Ph.D. Thesis, Indian Institute of Technology, Bombay, 2009.

[12] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[13] "Soot: A Java Optimization Framework http://www.sable.mcgill.ca/soot."

[14] A. Manjhi, C. Garrod, B. M. Maggs, T. C. Mowry, and A. Tomasic, "Holistic Query Transformations for Dynamic Web Applications," in *Intl. Conf. on Data Engineering*, 2009.

[15] "ObjectWeb Consortium-JMOB (Java middleware open benchmarking)." [Online]. Available: http://jmob.ow2.org/

[16] "The Freebase repository: http://www.freebase.com/."