Program Transformations for Asynchronous and Batched Query Submission

Karthik Ramachandra, Mahendra Chavan, Ravindra Guravannavar, S Sudarshan

Abstract—The performance of database/Web-service backed applications can be significantly improved by asynchronous submission of queries/requests well ahead of the point where the results are needed, so that results are likely to have been fetched already when they are actually needed. However, manually writing applications to exploit asynchronous query submission is tedious and error-prone. 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 data flow 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 also present a novel approach that, at runtime, can combine multiple asynchronous requests into batches, thereby achieving the benefits of batching in addition to that of asynchronous submission. We have built a tool that implements our transformation techniques on Java programs that use 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, which shows the effectiveness of the proposed rewrite techniques, both in terms of their applicability and the performance gains achieved.

Index Terms—Query Optimization, Program Analysis, Program Transformation

1 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.

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.

The original program is likely to be slow since it makes multiple synchronous requests to the database, each of which

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)
}
```


- *Karthik Ramachandra, IIT Bombay.* *E-mail: karthiksr@cse.iitb.ac.in*
- *Mahendra Chavan, IIT Bombay (current affiliation: SAP, Pune)* *E-mail: mahendra.chavan@sap.com*
- *Ravindra Guravannavar, IIT Hyderabad (current: independent consultant)* *E-mail: ravig@acm.org*
- *S Sudarshan, IIT Bombay* *E-mail: sudarsha@cse.iitb.ac.in*

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;
}
```

incurs network round trip delays, as well as delays in the database. In contrast, the rewritten program allows the network round trips to be overlapped. It also allows the database to better use its resources (multiple CPUs and disks) to process multiple asynchronously submitted queries. Asynchronous calls have been long employed to make concurrent use of different system components, like CPU and disk.

In this paper our focus is on automated rewriting of application programs so as to submit multiple queries asynchronously, as illustrated in Example 3. 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.

The most closely related prior work to our paper is that of Guravannavar and Sudarshan [1], who describe how to rewrite loops in database applications to replace multiple executions of a query in a loop by a single execution of a set-oriented (batched) form of the query. Batching can provide significant benefits because it reduces the delay due to multiple synchronous round trips to the database, and because it allows more efficient query processing techniques to be used at the database. Our program transformation techniques are based on the techniques described by Guravannavar and Sudarshan [1], but unlike their work, we show how to exploit asynchronous query submission, instead of batching.

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. Batching also results in a delayed response time, since the initial results from a loop appear only after the complete execution of the batch. Also, batching may not be applicable altogether when there is no efficient set-oriented interface for the request invoked, as is the case for many Web services.

As compared to batching, asynchronous submission of queries can allow overlap of client computation with computation at the server; it can also allow initial results to be processed early, instead of waiting for an entire batch to be processed at the database, which can lead to better response times for initial results. Further, asynchronous submission is applicable to Web Services that do not support set-oriented access. On the other hand pure asynchronous submission can lead to higher network overheads, and extra cost at the database, as compared to batching. We present a technique which we call asynchronous batching, which combines the benefits of asynchronous submission and batching.

The following are the key contributions of this paper:

1) We show (in Section 3) 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) Section 4 describes the design of our implementation. We first describe (in Section 4.1) the design challenges of such a program transformation tool. Since programmers may need to debug a rewritten version of their program, we present several techniques to make the rewritten program more readable.
   We then describe (in Section 4.2) the design of a framework that supports asynchronous query submission. Our framework provides a common API that can be configured to use either asynchronous submission or batching, or a combination of both.
3) In Section 5 we present extensions of the basic techniques described above. Specifically, we present (in Section 5.1) a modification of the code generated by the loop fission transformation that optimizes for response time by allowing early generation of initial results.
   We also present (in Section 5.2) *asynchronous batching*, a novel technique that combines the benefits of asynchronous query submission and batching by combining, at run time, multiple pending asynchronous requests into one or more batched requests.
4) These techniques have been incorporated into the DBridge holistic optimization tool [2], [3] to optimize Java programs that use JDBC.
   We present (in Section 6) a detailed experimental study of the proposed transformations on several real world applications. The experimental study shows significant performance gains due to our techniques.

This article is an extended version of our earlier conference paper [4]. Contribution 3, which is entirely novel, along with the related performance study in Section 6.3, are the key additions made in this journal version.

The rest of the paper is organized as follows. The model of asynchronous submission used in this paper is described in Section 2. Sections 3 through 6 describe our key contributions, as outlined above. Related work is described in Section 7. We discuss possible extensions of our techniques in Section 8 and conclude in Section 9.

2 Asynchronous Execution Model

We use the following model in this paper for coordinating asynchronous calls. 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. Example 1 of Section 1 shows a program using this 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.

More details regarding the design of these methods are given in Section 4. It is possible to extend our approach to make use of other models of asynchronous execution, but such extensions are not part of this paper.

3 Basic Transformations

Guravannavar and Sudarshan [1] 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 [5]. 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.

3.1 Data Dependence Graph

Inter-statement dependencies are best represented in the form of a data dependence graph [6] or its variant called the program dependence graph [7]. 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.

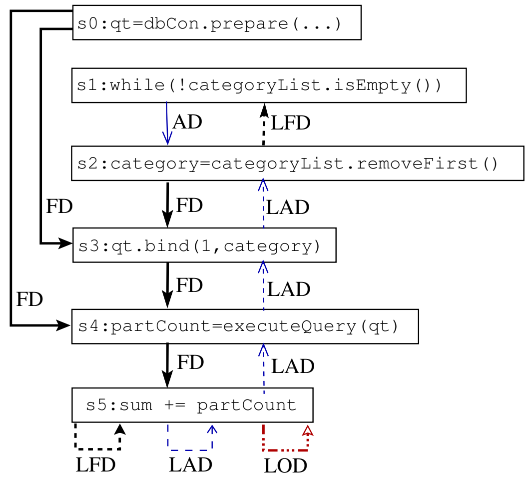

Fig. 1. Data Dependence Graph for Example 2

- 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 data flow 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.

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.,* LFD 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.

$$\Updownarrow$$

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 which are written in ss_1 and read in ss_2.

1) Table t and record r have attributes corresponding to each variable in *SV* and a key.
2) ss'_1 is the 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*).

Note: The names of variables/fields for the table(T), record(r), and *loopkey* have to be chosen so as to avoid any name conflict with existing program variables.

3.2 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*) [8]. Guravannavar and Sudarshan [1] 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 in [1] can be extended to make use of asynchronous calls.

A formal specification of the transformation is given as Rule A. 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.

Example 4 *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)
}
```

Example 5 *After reordering the statements in Example 4*

```
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;
}
```

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 makes an improvement of the fundamental nature to the loop fission transformation proposed in [1]. Rule A significantly relaxes the pre-conditions (see Rule 2 in [1]). For instance, Rule A allows loop-carried output dependencies to cross the split boundaries of the loop. This rule can also be applied to perform batching, thereby increasing its applicability. In general, our transformations are such that the resulting program can be used either for batching or for asynchronous submission, and this choice can be made at runtime. Our transformations in fact blur the distinction between batching and asynchronous submission, and can be used to achieve the best of both, as described in Section 5.2.

Note that adding split variables introduces an additional overhead as compared to the original program, primarily in terms of the space required to store the contexts of each iteration of the loop. However, for loops that involve data access from a database/web service, the overheads of data access are typically higher than the overheads due to split variables. Approaches to minimize memory overheads are discussed briefly in Section 8.

Applicability

The pre-condition that no loop-carried flow dependencies cross the point of split can limit the applicability of Rule A in several practical cases. Consider the program in Example 4.

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

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).

However, in many cases it is 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 4, if reordered as shown in Example 5, permit loop fission. Note that in the transformed program of Example 5 there are no loop-carried flow dependencies, which prohibit the application of Rule A to split the loop at the query execution statement. An algorithm for statement reordering to enable loop fission, along with a sufficient condition for the applicability of the loop fission transformation are given in [4].

Further, Rule A is also not directly applicable when the query execution statement lies inside a compound statement such as an *if-then-else* block. We now present additional transformation rules which can be used to address this restriction.

3.3 Control Dependencies

We handle control dependencies using the approach of [1]. Consider the initial program shown in Example 6. 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 [8], by introducing boolean variables and guard statements. We define a transformation rule to perform this conversion.

The formal specification of the transformation, called Rule 4 in [1] is shown as Rule B in this paper. An *if-then-else* block is transformed into an assignment of the value of the predicate p to a boolean variable *cv*, followed by a sequence of statements guarded by the value (or the negation) of boolean variable *cv*. In Example 6, we apply Rule B and introduce a boolean variable c to remember the result of the predicate evaluation,

Example 6 *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);
    boolean c = (v == 0);
    c==true? v = executeQuery(q);
    c==true? log(v);
    print(v);
}
```

and then convert the statements inside the conditional block into guarded statements. We can then apply Rule A and split the loop as described earlier.

Note that Exceptions thrown during query execution in the original program will now be thrown as part of the asynchronous submission (*submitQuery* API). Our implementation catches these exceptions, and re-throws them at the corresponding *fetchResult* invocation. Control transfer statements such as *break*, *continue*, and *return* also lead to control dependencies. These statements lead to additional edges in the control flow graph, and with these additional edges, our transformation rules can be applied as described. Details are fairly straight-forward and hence omitted.

3.4 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 7. 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.

4 System Design and Implementation

The techniques we propose can be used with any language and data access API. We have implemented these ideas and incorporated them into the DBridge holistic optimization tool [2], [3]. A system that can support asynchronous query submission would include two main components (i) a source-to-source

Example 7 *Dealing with nested loops*

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

After Transformation

Table t_1 = *new Table();*
while(pred1){
  Table t_2 = *new Table(); Record* r_1 = *new Record();*
  while(pred2){
    Record r_2 = *new Record();*
    r_2*.handle* = *submitQuery(q);*
    t_2*.addRecord(*r_2*);*
  }
  r_1*.child* = t_2*;* t_1*.addRecord(*r_1*);*
}
for each r_1 *in* t_1 {
  for each r_2 *in* r_1*.child* {
    x = *fetchResult(*r_2*.handle); process(x);*
  }
}

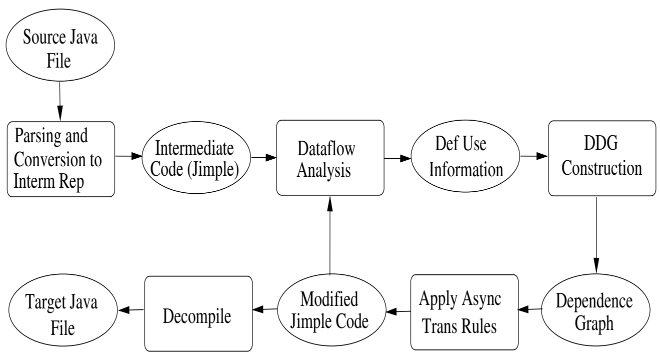

Fig. 2. Program Transformation Phases

program transformer, and (ii) a runtime asynchronous submission framework. The runtime infrastructure also supports asynchronous batching, a technique that supports batching of asynchronous requests, described in Section 5.2. We now describe each of these components in detail.

4.1 Program Transformer

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 data flow analysis of the given program and build the data dependence graph. We used the SOOT optimization framework [9]. 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 2. 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 data flow information each time the code changes. The rule application process stops when all query execution statements (or the user specified ones) which do not lie on a true-dependence cycle, are converted to asynchronous calls.

Our tool has been implemented with 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 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.

4.2 Runtime Asynchronous Submission Framework

The runtime library works as a layer between the actual data access API (such as JDBC) and the application code. It provides asynchronous submission methods in addition to wrapping the underlying API. Features such thread management and cache management are handled by this library. The transformed programs in our implementation use the *Executor* framework of the *java.util.concurrent* package for thread scheduling and management [10].

Figure 3 shows the behaviour of the asynchronous submission API. The first loop in the transformed program submits the query to a queue in every iteration. The *stmt.addBatch(ctx)* invocation is a non blocking query submission, with the same semantics as the *submitQuery* API described in Section 2. This queue is monitored by a thread pool which manages a configurable number of threads. The requests are picked up by

```
LoopContextTable lct = new LoopContextTable();
while(!categoryList.isEmpty()){
    LoopContext ctx = lct.createContext();
    category = categoryList.next();
    stmt.setInt(1, category);
    ctx.setInt("category", category);
    stmt.addBatch(ctx);
}
stmt.executeBatch();
for (LoopContext ctx : lct) {
    category = ctx.getInt("category");
    ResultSet rs = stmt.getResultSet(ctx);
    rs.next();
    int count = rs.getInt("count");
    sum += count;
    print(category + ": " + count);
}
```

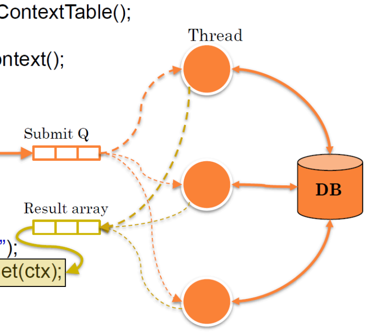

Fig. 3. Asynchronous query submission API

free threads which maintain open connections to the database. The individual threads execute the query in a synchronous manner i.e., the thread blocks till the query results are returned. The results are then placed in a cache keyed by the loop context(ctx).

The second loop accesses the results corresponding to the loop context using the *stmt.getResultSet(ctx)* which has the same semantics as the *fetchResult* API described in Section 2. Subsequently, it executes statements that depend on the query results. The *LoopContextTable* ensures the following: (i) it preserves the order of execution between the two loops and (ii) for each iteration of the first loop, it captures the values of all variables updated, and restores those values in the corresponding iteration of the second loop.

5 Extensions and Optimizations

We now describe two extensions to our basic technique of asynchronous query submission. These extensions can significantly improve performance as shown by our experiments.

5.1 Overlapping the Generation and Consumption of Asynchronous Requests

Consider the basic loop fission transformation Rule A. On applying this rule, a loop will result in two loops, the first that generates asynchronous requests (hereafter referred to as the *producer* loop), and the second that processes, or consumes results (hereafter referred to as the *consumer* loop).

According to Rule A, the processing of query results (the consumer loop) starts only after all asynchronous submissions are completed (i.e, after the producer loop completes). Although this transformation significantly reduces the total execution time, it results in a situation where results start appearing much later than in the original program. In other words, for a loop of n iterations, the time to k-th response ($1 \leq k \leq n$) for small k is more as compared to the original program, even though the time may be less for larger k. This could be a limitation for applications that need to show some results early, or that only fetch the first few results and discard the rest.

This limitation can be overcome by overlapping the consumption of query results with the submission of requests. The transformation Rule A can be extended to run the producer loop (the loop that makes asynchronous submissions) as a separate thread. That is, the main program spawns a thread to execute the producer loop, and continues onto the consumer loop immediately. Since the loop context table (*Table t* in Rule A) may be empty when the consumer loop starts, and may get more tuples as the consumer loop progresses, we implement the loop context table as a blocking (producer-consumer) queue. The producer thread submits requests onto this queue, which are picked up by the consumer loop.

Note that this transformation is safe, and does not lead to race conditions since there are no data dependences between the producer and consumer loop other than through the loop context table. This is because the values of all variables updated in the producer loop are captured, and restored in the consumer loop via the loop context table. The blocking queue implementation of the loop context table avoids race conditions on the table. The details of this extension are straightforward and hence a formal specification is omitted. We evaluate the benefits of this extension and show the results in Section 6.3.

5.2 Asynchronous Submission of Batched queries

As mentioned earlier, the transformation rules proposed in this paper can be used for either batching or asynchronous submission. However, there are key differences in the approaches due to which their performance characteristics vary. In this section we compare the relative benefits and drawbacks of batching and asynchronous query submission, and propose a new strategy that can combine the benefits of both strategies.

5.2.1 Asynchronous Submission vs. Batching

The following are some of the drawbacks of batching as compared to asynchronous submission:

- 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 blocks after submitting the batch.
- Although batching reduces the overall execution time of the program, for initial results it typically results in a worse response time, since the result of the first query is available only when the result set of the large batch is returned.
- Since batching retrieves the results for the whole loop at once, it may significantly increase the memory requirement at the client.
- Batching may not be applicable when there is no (efficient) set-oriented interface for the request invoked.

The asynchronous query submission technique presented in Section 3 avoids the problems mentioned above for batching, but has a few drawbacks of its own, as compared to batching:

- Asynchronous query submission does not reduce the number of network round trips but only overlaps them. This may increase network congestion.
- The database still receives individual queries and hence this may result in a lot of random IO at the database.

- As a result of the above, whenever batching is applicable, and the number of iterations of the loop is large, batching leads to much better performance improvements (in terms of total execution time) than asynchronous submission.

More details are described in our experiments in Section 6.3. We now describe how to combine both these approaches.

5.2.2 Asynchronous Batching: Best of Both Worlds

Batching and Asynchronous submission can be seen as two ends of a spectrum. Batching, at one end, combines all requests into one big request with no overlapping execution, where as asynchronous submission retains individual requests as is, while completely overlapping their execution. Clearly, there is a range of possibilities between these two, that can be achieved by *asynchronous* submission of multiple, smaller *batches* of queries. This approach, which we call *asynchronous batching*, retains the advantages of batching and asynchronous submission, while avoiding their drawbacks.

Consider the example in Figure 3. As mentioned earlier, the first loop in this program submits a query to a queue in each iteration. This request queue is monitored by a thread pool. In pure asynchronous submission, each free thread picks up an individual request from the queue. In contrast, with asynchronous batching, the thread can observe the whole queue, and pick up one, or more, or all requests from the queue. If a thread picks up a single request, it executes the query as described earlier. However. if a thread picks up more than one request, it performs a query rewrite as done in batching, and executes those requests as a batch. Once the result of the batch arrives, it is split into multiple result sets corresponding to each individual query, which are then placed in the cache.

Asynchronous batching aims to achieve the best of batching and asynchronous submission, since it has the following characteristics.

- Like batching, it reduces network round trips, since multiple requests may be batched together.
- Like asynchronous submission, it overlaps client computation with that of the server, since batches are submitted asynchronously.
- Like batching, it reduces random IO at the database, due to use of set oriented plans.
- Although the total execution time of this approach might be comparable to that of batching, this approach results in a much better latency comparable to asynchronous submission, since the results of queries become available much earlier than in batching.
- Memory requirements do not grow as much as with pure batching, since we deal with smaller batches.

The key challenge in engineering such a system is to identify the sweet spot in the spectrum between batching and asynchronous submission. This primarily involves deciding the size of each batch and the number of threads to use, which would result in the best performance. This decision cannot be made statically during program transformation, since it depends on runtime factors such as (i) the number of iterations in the loop, (ii) the query processing time and the size of its results, (iii) the capacity and load on the client machine and the database server, (iv) network bandwidth availability.

Asynchronous batching is a completely runtime decision; the program transformation is performed in accordance with the rewrite rules in this paper, and requires no additional rewriting. The runtime library makes decisions on asynchronous calls vs. partial batching in a dynamic fashion. We now discuss strategies to tune parameters for asynchronous batching.

5.2.3 Adaptive tuning of batch size

The runtime library is extended to allow a thread to pick up one or more requests from the queue. However the key problem is the following: given a queue of n requests, how many requests should a free thread pick up? Note that the only information available for a thread is the current state of the queue. We now propose strategies to automatically vary the batch size at runtime. Assuming that the number of threads T available on the client is fixed, these strategies are affected by the following metrics:

1) *The request arrival rate:* This is the rate at which the program submits requests onto the queue. In our example of Figure 3, the first loop submits one request per iteration. Arrival rate essentially captures the time taken between consecutive submissions.
2) *The request processing rate:* This is the rate at which requests in the queue are processed. Processing a request includes the query processing time at the database and the network round trip time.

The request arrival rate would be higher than the requests processing rate if any of the following are true: (a) the producer loop has no expensive operations, (b) network round trips are very expensive, (c) query processing time is high. We now propose three possible strategies for asynchronous batching.

One-or-all Strategy: This is a simple strategy to combine asynchronous submission and batching. Given a queue with n requests, the *One-or-all* strategy for a free thread is as follows: If $n = 1$, then pick up the request from the queue, and execute it as an individual request. If $n > 1$, pick up all the n requests in the queue and batch them. In other words, (i) insert the parameters of the n requests into a temporary parameter table, (ii) rewrite the query using the technique given in [1], (iii) execute this rewritten query. If $n = 0$, wait for new requests. In this strategy, a free thread will always clear the queue by picking up all pending requests from the queue.

Lower Threshold Strategy: The *One-or-all* strategy can be improved based on an observation regarding batching. Batching results in 3 network round trips, one each for (a) inserting parameters into a temporary table, (b) executing the batched query, and (c) clearing the temporary table. In fact each thread incurs another round trip while batching for the first time in order to create the temporary table. This means that the time taken to process one batch is roughly equivalent to the time taken to process *at least* three individual requests sequentially, since there are 3 network round trips and 3 queries being executed for every batch. We verified this in our experiments, and found that very small batches perform poorly as compared to asynchronous submission.

Therefore, we use the following strategy. We define a *batching threshold* $bt \geq 3$. If $n > bt$, then pick up all the n requests in the queue and batch them. If $1 \leq n \leq bt$, then pick up one request from the queue, and execute it as an individual request. If $n = 0$, wait for new requests. Observe that in this strategy, a free thread does not necessarily clear the queue.

Consider the situation where the request arrival rate is higher than the request processing rate. In this setting, the first few (about T) requests would be sent as individual requests asynchronously. Since the queue builds up much faster than it is consumed, after the first few iterations, the requests would be submitted in batches with increasing sizes.

On the other hand, consider the case where the request processing rate is higher than the rate of arrival of requests onto the queue. In this situation, the queue would not grow in size since the requests keep getting consumed at a higher rate, and hence n would remain below (or close to) the *batching threshold*. This implies that most requests would be sent individually, mimicking the behaviour of asynchronous query submission.

Thus we can see that the lower threshold strategy is actually quite adaptive. Batch sizes vary in accordance with the queue size, which in turn depends upon the arrival rate of requests, the rate at which requests get processed, and the number of threads working concurrently on processing requests.

Growing upper-threshold based Strategy: Although the above approach improves response time and adapts the batch size according to the queue size, in situations where the arrival rate of requests is high, it may lead to a situation where a single large batch is submitted while the remaining threads are idle. This could lead to a slower response time for initial results, since the database would take a longer time to process a large batch, and higher memory consumption due to a large request queue, although the larger batch size may reduce overall work at the database server, and reduce the time to process all requests.

For applications that need better response times for initial results, we use an upper-threshold strategy. We use a *growing upper threshold* that bounds the maximum batch size. This upper threshold is not a constant but is initially small, so that batch sizes are small initially, but grows as more requests are submitted, so that response times for later results are not unduly affected due to very small batch sizes.

The growing upper-threshold strategy works as follows. If the number of requests in the queue is less than the current upper threshold, all requests in the queue are added to a single batch. However, if the number of requests in the queue is more than the current upper threshold, the batch size that is generated is equal to the current threshold; however, for future batches, the upper threshold is increased; in our current implementation of the growing upper-threshold strategy, we double the upper threshold whenever a batch of size equal to the current upper threshold is created.

Note that the upper threshold strategy is orthogonal to the lower-threshold strategy, and each may be used with or without the other.

6 Experimental Results

We have conducted a detailed experimental evaluation of our techniques using the DBridge tool. In Section 6.1, we present our experiments on asynchronous query submission and its benefits. Next, in Section 6.3, we compare basic asynchronous submission with the extensions and optimizations described in Section 5, and discuss the results.

6.1 Asynchronous query submission

For evaluating the applicability and benefits of the proposed transformations, we consider four Java applications: two publicly available benchmarks (which were also considered by Manjhi et al. [11]) and two 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 cache of results was maintained using the *ehcache* library [12].

Experiment 1: Auction Application: We consider a benchmark application called RUBiS [13] 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 (by choosing user ids with appropriate number of associated comments), fixing the number of threads at 10. Figure 4 shows the performance of this program before and after the transformations with warm and cold caches in log scale. The y-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. Although it is unrealistic to load 40,000 comments in a web application,

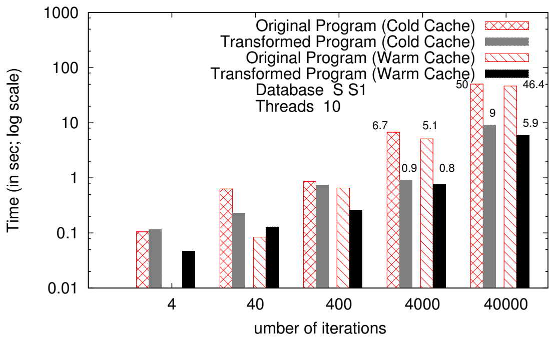

Fig. 4. Experiment 1 with varying number of iterations

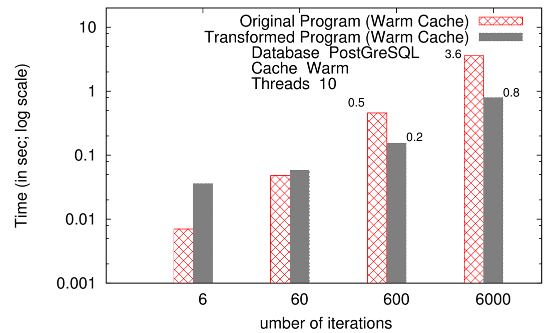

Fig. 6. Experiment 2 with varying number of iterations

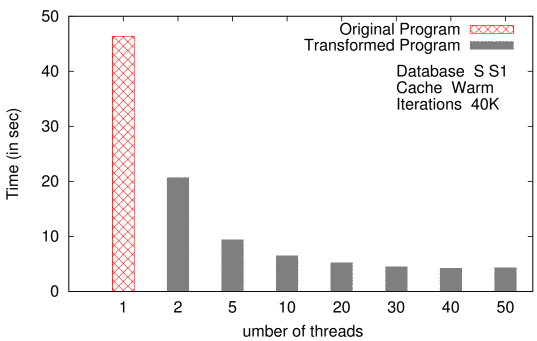

Fig. 5. Experiment 1 with varying number of threads

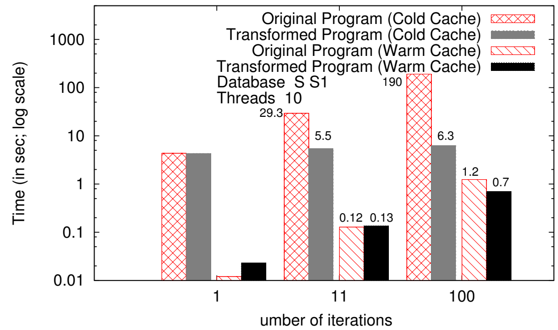

Fig. 7. Experiment 3 with varying iterations

we have performed this experiment to understand the extent of improvement that could be gained by our transformations, and to study the behaviour of our rewritten program, while increasing the amount of data it handles.

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 5. 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 follow the same pattern as in the case of SYS1, and the results are given in [4].

Experiment 2: Bulletin Board Application: RUBBoS [13] 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 6 shows the results of our transformations with different number of iterations. Although the transformed loop in the 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 [1], 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 7 shows the performance of this loop in the 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.

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

The effect of varying the number of threads shows similar trends as that of Experiment 1, though the actual numbers differ. The results can be found in [4]. In transforming this program, the reordering algorithm was first applied and then the loop was split using Rule A.

Experiment 4: 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 [14], a social database about entities, spanning millions of topics in thousands of categories. It is an entity graph which

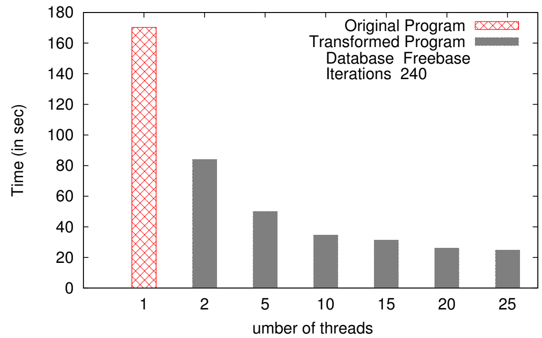

Fig. 8. Experiment 4 with varying number of threads

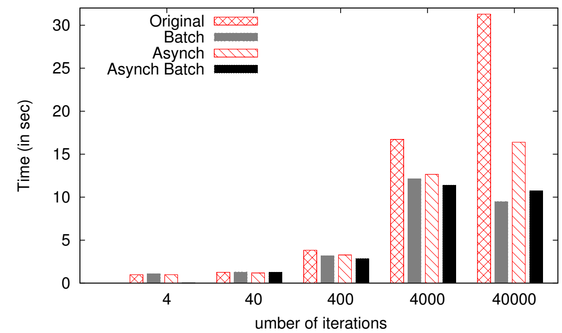

Fig. 9. Total execution time with no. of iterations

Application	# Servlets	# Opp	# Trans	Applicability (%)
Auction	16	9	9	100
BulletinBoard	18	8	6	75

TABLE 1
Applicability of transformation rules

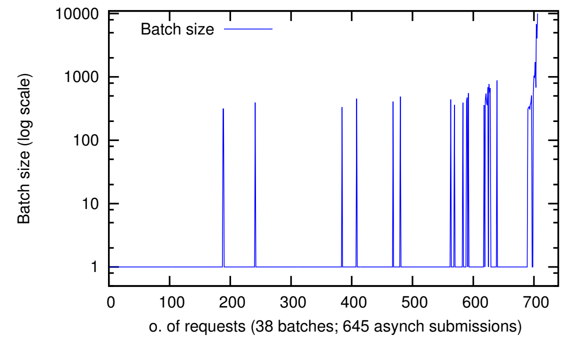

Fig. 10. Batch sizes during an execution

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 support set oriented queries.

Since our current implementation supports only JDBC API, we manually applied the transformations to the code that executes the Web service requests. The results of this experiment are shown in Figure 8. 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.

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 transformation took very little time (less than a second) for programs with about 1000 lines of code. Our implementation only analyzes user code, and avoids analyzing system/third party libraries. It can also be configured to run on specific parts of the code rather than the entire codebase, while dealing with large programs.

6.2 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 by our transformation rules. The results of the analysis is presented in Table 1.

We consider all servlets that have SQL queries (# Servlets). Among those, we consider all kinds of loop structures which include a query execution statement in the loop body as potential opportunities (# Opp). Such loops are an important cause of performance issues in database applications, including those that are backed by object relational mapping tools such as Hibernate [15]. Among such potential opportunities, those that satisfy the preconditions for our rules, are transformed (# Trans).

We see that all such opportunities present in the auction system indeed satisfy the preconditions and can be transformed. Although our preconditions are more general than those proposed in [1], the opportunities satisfied both. In the bulletin board application a few of the loops performed recursive method invocations which prevent them from being transformed. Out of the programs seen earlier, the remaining were too small for this analysis, and hence omitted.

6.3 Effect of Optimizations

We have performed experiments to compare the following approaches: (i) *Original*: the original program (ii) *Batch*: the program rewritten using query batching, (iii) *Asynch*: the program rewritten according to our technique of asynchronous submission, (iv) *Asynch Batch*: our technique of combining batching and asynchronous submission (Section 5.2), using the simple threshold based strategy (v) *Asynch Overlap*: asynchronous submission with concurrent generation of requests (Section 5.1), (vi) *Asynch Batch Overlap*: asynchronous batching with concurrent generation of requests, and (vii) *Asynch Batch Grow*: asynchronous batching with concurrent generation of requests and the growing-upper-threshold strategy. Our current implementation does not support the *Async Overlap*

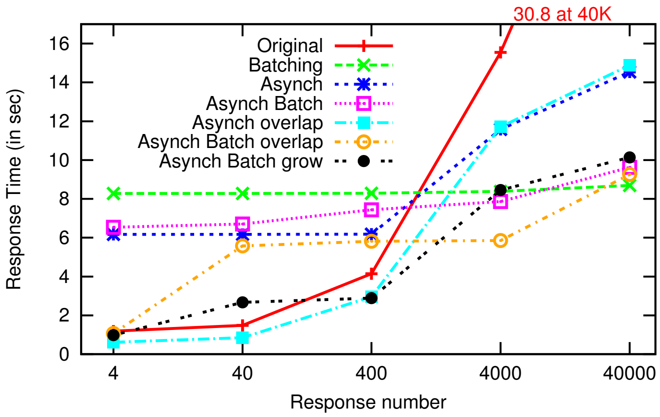

Fig. 11. Time to k-th response (with 40000 iterations)

transformation, and hence we have rewritten the code manually as described in Section 5.1.

The experiments have been conducted on a widely used commercial database system SYS3. The SYS3 database server was running on a 64 bit 2.3 GHz quad-core machine with 4 GB of RAM. The Java applications were run from a remote 3.3 Ghz quad-core machine connected to the database servers over a 100 MBps LAN. For approach (iv), we used a lower batching threshold of 300 with 48 threads, and for approach (vii), we used a doubling growth rate for the upper threshold. We again consider the benchmark auction application RUBiS [13] and the scenario described in Experiment 1 of Section 6.1.

6.3.1 Total execution time

First, we compare the total execution time of this program according to the approaches (i) through (iv), since the optimizations in (v), (vi) and (vii) have minimal impact on the total execution time. The results of this experiment with cold cache are shown in Figure 9. The x-axis shows the number of iterations and the y-axis shows the total execution time.

It can be observed from Figure 9 that at smaller number of iterations, all approaches behave very similarly, and differences can be observed at larger number of iterations. Asynchronous submission (with 12 threads) gives about 50% improvement, while batching leads to about 75% at 40000 iterations. Asynchronous batching, with 48 threads and a lower batching threshold of 300 leads to about 70% improvement.

At 40000 iterations, we have recorded the behaviour of one run of asynchronous batching, shown in Figure 10. The x-axis shows the number of requests (either batched or individual), and the y-axis shows the batch sizes in log scale. Overall, there were 38 batch submissions and 645 asynchronous submissions, and among the 38 batches, the average batch size was 1019. Initially, many requests are sent individually since the lower batching threshold was set to 300. But the queue builds up quite fast and hence there are a few intermittent batch submissions. As the execution progresses, there are more and more batch submissions, and batch sizes also start growing. Towards the end, there are batches of upto 10000 requests. This behaviour is in accordance with our expectation as described in Section 5.2.

6.3.2 Time to k-th response

Next, we compare the response time of the program according to approaches (i) through (vii) described earlier. Here, by response time we mean the duration between the start of the program and the arrival (or the output) of the k-th response from the program. In our auction system experiment, records are printed when the information about the author of the comment is retrieved. Therefore, the response time is measured at the instant where the author information of the k-th comment is output. We fix the number of iterations at 40000, and record the time taken for the k-th response, with k varying from 1 to 40000. The results of this experiment are shown in Figure 11. The x-axis shows the response number k, and the y-axis shows time in seconds. For this experiment, the *Async Batch Grow* approach used a lower batching threshold of 100, and an upper threshold that doubles, initially set to 200.

The original program has the best response time initially. However, the response time increases quite steeply with k, and reaches about 31 seconds for the 40000th response. Batching, in contrast, has a constant curve. This is because even the first response is output only after (i) all parameters are added to the parameter batch table, and (ii) the transformed (set oriented) query is executed. Essentially, the time to k-th response in batching is very close to the total execution time, since all the results are returned together.

The *Asynch* approach starts off with a better (lower) response time as compared to batching, but increases beyond batching for larger values of k. Asynchronous batching initially behaves like asynchronous submission, and slowly deviates from it. At larger number of iterations, it behaves more like batching. In other words, it always tends towards the better of *Asynch* and *Batch*.

The *Overlap* versions of *Asynch* and *Asynch Batch* show much better response times compared to the earlier approaches. The *Async Batch Grow* approach behaves the best in balancing response time vs total execution time. It initially shows response times similar to the original program, and does even better than the *asynch* and *Batch* at larger iterations. At $k = 40000$, it results in the response time comparable to *Batch*.

6.3.3 Discussion

In summary, our experimental study shows that batching and asynchronous submission are beneficial techniques with different trade offs, and the combined technique of asynchronous batching with optimizations aims at balancing these trade offs. Some of the trade offs are (a) total execution time vs. time to k-th response, (b) reducing network round trips (by batching multiple requests) vs. overlapping execution of queries, (c) reducing memory consumption (by using iterative query execution) vs. set oriented execution of the query.

These trade offs are essentially controlled by parameters used in asynchronous batching, such as the batching threshold, number of threads etc. Based on the use case, the parameters have to be tuned in order to achieve desired behaviour. Our contribution in this paper has been to expose these trade offs to the developer, and allow manual tuning of such parameters. We have also presented some initial approaches for automatic tuning of parameters in Section 5.2.3, but we believe that there is scope for more work in this area.

7 Related Work

Most operating systems today allow applications to issue asynchronous IO requests [16]. Asynchronous calls are also used for data prefetch and overlapping operator execution inside query execution engines [17], [18], [19]. Asynchronous calls have also been used to hide memory access latency by issuing prefetch requests [20]. Asynchronous calls are widely used in the communication between the web browser and the server using manually placed AJAX requests.

Yeung [21] proposes an approach to automatically optimize distributed applications written using Java RMI based on the concept of deferred execution, where remote calls are delayed for as long as possible. However, this work does not consider asynchronous calls and and query executions within loops. Dasgupta et al. [22] and Chaudhuri et al. [23] propose an architecture and techniques for a general static analysis framework to analyze database application binaries that use the ADO.NET API, with the goals of identifying security, correctness and performance problems.

Many Prefetching techniques based on prediction and read aheads based on data access patterns have been proposed [24], [25]. There has been work on prediction based prefetching of query results [26], by analyzing logs and trace files, but this work does not consider asynchronous prefetching. There has been very recent work by Cheung et al. [27] which considers the use of various program analysis techniques to improve database applications. Unlike the Sloth system [28] which uses lazy evaluation, we use early evaluation (or asynchronous prefetching), with the same goal of exposing query batching opportunities in database applications.

Guravannavar and Sudarshan [1] consider rewriting loops in database applications and stored procedures, to transform iterative executions of queries into a single execution of a set-oriented form of the query. We use a similar framework of program transformation, but for asynchronous query submission. While our transformation rules are based on [1], we make the following novel contributions. First, we show how the transformation rules presented in [1] in the context of batching, can be adapted for asynchronous query submission. Second, we describe an extension to our transformation that enables overlapping of generation and consumption of asynchronous requests, thereby greatly improving the response time. Third, we present a technique to combine batching and asynchronous query submission into a common framework.

Manjhi et al. [29] consider prefetching of query results by employing non-blocking database calls, made at the beginning of a function. A blocking call is subsequently issued when the results of the query are needed. However, they do not describe details to automate this task. Ramachandra et al. [30] propose a technique to insert prefetch requests for queries/web services at the earliest possible point in the program across procedure invocations. However, both [30] and [29] do not consider loop transformations for queries within loops while exploiting opportunities for prefetching, and this forms the main focus of this paper.

8 Extensions

We now discuss some system design considerations and extensions of techniques described in this paper.

Ensuring transaction properties: In our implementation, we have used one connection per thread in order to achieve overlapping query execution. This is because in JDBC, (a) a database connection allows only one open query at a time, (b) there are no API methods that allow asynchronous submission. ADO.NET provides asynchronous API (such as the *BeginExecuteReader* and *EndExecuteReader* APIs), which allow overlapping of query execution with local computation. However, even these APIs do not support overlapping query executions through a single connection.

In order to fully preserve transaction properties and achieve true asynchronous submission, individual threads in the thread pool should be part of a single shared transaction. Such an infrastructure is not currently supported by any database vendor to the best of our knowledge. Although databases support distributed transactions (such as JDBC XA transactions), their goal is to allow transactions across multiple data sources.

One way to implement this (if snapshot queries are supported) is to allow multiple connections to share a snapshot point. Such a feature, if supported, would allow multiple threads (with their own connections) to share and execute transactions on the same snapshot. We believe that this would be a minor change in databases that already support snapshot isolation, and would be a useful feature to have. Such a built in support would not only simplify application development, but also lead to significant improvement in performance, as compared to our current implementation.

Rewriting loops containing update transactions needs to consider dependencies between update statements and program variables. A conservative approach is to assume that update statements are dependent on other update or select statements in a loop, and model them as data dependencies which factor in to the preconditions for our transformation rules. This can be improved by using more precise inter query dependence analyses [31].

Minimizing memory overheads: If the number of loop iterations is large, the transformed program may incur high memory overhead, in order to store the handle and the state associated with each iteration. Storing such state on disk increases the IO cost. Our technique can be extended such that, based on memory usage, the producer thread backs off and waits while results are consumed and memory freed, and then generates more requests.

9 Conclusion

We propose a program analysis and transformation based approach to automatically rewrite database applications to exploit the benefits of asynchronous query submission. The techniques presented in this paper significantly increase the applicability of known techniques to address this problem. We also described a novel approach to combine asynchronous submission with our earlier work on batching in order to achieve a balance between the trade offs of batching and asynchronous query submission.

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.

Acknowledgements: We thank Yatish Turakhia for his help in the implementation of asynchronous batching in DBridge.

REFERENCES

[1] R. Guravannavar and S. Sudarshan, "Rewriting Procedures for Batched Bindings," in *Intl. Conf. on Very Large Databases*, 2008.
[2] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan, "DBridge: A program rewrite tool for set-oriented query execution," in *ICDE*, 2011, pp. 1284–1287.
[3] K. Ramachandra, R. Guravannavar, and S. Sudarshan, "Program analysis and transformation for holistic optimization of database applications," in *ACM SIGPLAN SOAP 2012*, 2012, pp. 39–44.
[4] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan, "Program transformations for asynchronous query submission," in *ICDE*, 2011, pp. 375–386.
[5] R. Guravannavar, "Optimization and evaluation of nested queries and procedures," Ph.D. Thesis, IIT Bombay, 2009.
[6] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.
[7] 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.
[8] K. Kennedy and K. S. McKinley, "Loop Distribution with Arbitrary Control Flow," in *Proceedings of Supercomputing*, 1990.
[9] "Soot: A Java Optimization Framework http://www.sable.mcgill.ca/soot."
[10] "The Java executors framework." [Online]. Available: http://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html
[11] 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.
[12] "The ehcache Java caching library." [Online]. Available: http://ehcache.org/
[13] "ObjectWeb Consortium-JMOB (Java middleware open benchmarking)." [Online]. Available: http://jmob.ow2.org/
[14] "The Freebase repository: http://www.freebase.com/."
[15] "Hibernate OR Mapping tool." [Online]. Available: http://www.hibernate.org
[16] "Kernel Asynchronous I/O (AIO) Support for Linux http://lse.sourceforge.net/io/aio.html."
[17] G. Graefe, "Executing Nested Queries," in *10th Conference on Database Systems for Business, Technology and the Web*, 2003.
[18] M. Elhemali, C. A. Galindo-Legaria, T. Grabs, and M. M. Joshi, "Execution Strategies for SQL Subqueries," in *ACM SIGMOD*, 2007.
[19] 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.
[20] S. P. Vanderwiel and D. J. Lilja, "Data Prefetch Mechanisms," *ACM Computing Surveys*, vol. 32, no. 2, 2000.
[21] K. C. Yeung, "Dynamic Performance Optimisation of Distributed Java Applications," Ph.D. dissertation, Imperial College of Science, Technology and Medicine, 2004.
[22] A. Dasgupta, V. Narasayya, and M. Syamala, "A static analysis framework for database applications," in *ICDE '09*, 2009, pp. 1403–1414.
[23] S. Chaudhuri, V. Narasayya, and M. Syamala, "Bridging the application and dbms divide using static analysis and dynamic profiling," in *SIGMOD*, 2009, pp. 1039–1042.
[24] P. A. Bernstein, S. Pal, and D. Shutt, "Context-based prefetch – an optimization for implementing objects on relations," *The VLDB Journal*, vol. 9, no. 3, pp. 177–189, Dec. 2000.
[25] A. Bilgin, R. Chirkova, T. Salo, and M. Singh, "Deriving efficient sql sequences via read-aheads," in *Data Warehousing and Knowledge Discovery*, ser. LNCS, 2004, vol. 3181.
[26] I. T. Bowman and K. Salem, "Semantic prefetching of correlated query sequences," in *ICDE*, 2007.
[27] A. Cheung, S. Madden, A. Solar-Lezama, O. Arden, and A. C. Myers, "Using program analysis to improve database applications," *IEEE Data Eng. Bull.*, vol. 37, no. 1, pp. 48–59, 2014.
[28] A. Cheung, S. Madden, O. Arden, and A. C. Myers, "Sloth: Being lazy is a virtue (when issuing database queries)," in *SIGMOD*, 2014.
[29] A. Manjhi, "Increasing the Scalability of Dynamic Web Applications," Ph.D. dissertation, Carnegie Mellon University, 2008.
[30] K. Ramachandra and S. Sudarshan, "Holistic optimization by prefetching query results," in *ACM SIGMOD 2012*, 2012, pp. 133–144.
[31] S. Parthasarathy, W. Li, M. Cierniak, and M. J. Zaki, "Compile-time inter-query dependence analysis," in *IEEE Symp. on Parallel and Distr. Processing*, 1996.

Karthik Ramachandra is a PhD student in the Computer Science department at IIT Bombay. His research areas include query processing and optimization and holistic optimization that spans the boundary of databases and compilers. Prior to this, he has had several years of experience in software development at ThoughtWorks Inc. He received his bachelors degree in Information Science from the Visweswaraiah Technological University, India.

Mahendra Chavan is presently working with SAP Labs as a senior developer on database products' synergies. He has been working on query optimization and query processing in scale-out scenarios. Prior to joining SAP Labs Pune, he was part of the holistic optimization group at IIT Bombay during his masters degree. He received his masters degree in Computer Science from IIT Bombay and bachelor's degree in Computer Science from Pune Institute of Computer Technology, Pune.

Ravindra Guravannavar received his MTech and PhD in Computer Science from IIT Bombay, and his bachelor's degree in Computer Science from Karanatak University, India. He was an Assistant Professor in the Computer Science and Engineering department at IIT Hyderabad. He also has several years of experience in software product development in the industry. His research interests are in query processing and automated performance improvement of database applications. He is a member of ACM.

S. Sudarshan received his B.Tech. from IIT Madras, and his PhD from Univ. of Wisconsin, Madison. He is a Professor in the Computer Science and Engineering department at IIT Bombay. His research areas include query processing and optimization, holistic optimization, keyword querying on semi-structured data, and test data generation for SQL queries. He has authored over 90 papers, and is a co-author of a leading textbook, Database System Concepts, currently in its 6th edition. He is a member of ACM and IEEE.