Holistic Optimization by Prefetching Query Results

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

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

ABSTRACT

In this paper we address the problem of optimizing performance of database/web-service backed applications by means of automatically prefetching query results. Prefetching has been performed in earlier work based on predicting query access patterns; however such prediction is often of limited value, and can perform unnecessary prefetches. There has been some earlier work on program analysis and rewriting to automatically insert prefetch requests; however, such work has been restricted to rewriting of single procedures. In many cases, the query is in a procedure which does not offer much scope for prefetching within the procedure; in contrast, our approach can perform prefetching in a calling procedure, even when the actual query is in a called procedure, thereby greatly improving the benefits due to prefetching. Our approach does not perform any intrusive changes to the source code, and places prefetch instructions at the earliest possible points while avoiding wasteful prefetches. We have incorporated our techniques into a tool for holistic optimization called DBridge, to prefetch query results in Java programs that use JDBC. Our tool can be easily extended to handle Hibernate API calls as well as Web service requests. Our experiments on several real world applications demonstrate the applicability and significant performance gains due to our techniques.

Categories and Subject Descriptors

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

1. INTRODUCTION

Most applications on the Web today spend part of their execution time on local computation and spend the rest in accessing databases, Web services or other applications remotely. Typically, for any remote access, there is a conversation between an application server and say, a database in the form of a series of requests (SQL queries/HTTP requests) and responses. In such applications, the time taken for remote access is split between (a) preparing requests, (b) transmitting them over the network, (c) actual computation at the database, to serve the request (involving processing and disk IO), (d) preparing responses, (e) transmitting responses back over the network.

Performing these actions synchronously results in a lot of latency since the calling application blocks during stages (b) through (e). Much of the effects of latency can be reduced if these actions are overlapped with local computations or other requests. Such overlap can be achieved by issuing asynchronous requests in advance, while the application continues performing other tasks. In many cases, the results can be made available by the time they are actually required, thereby completely hiding the effect of latency. This idea of making query results available before they are actually needed by the application, is called *query result prefetching*.

Prefetching can help greatly in reducing response times. Consider the program shown in Figure 1. The *generateReport* method accepts a customer id (*custId*), a currency code (*curr*), and a date (*fromDate*), and performs the following tasks in sequence: (i) Retrieves information about all accounts of that customer and processes them in a loop (n_1 to n_5), (ii) Retrieves and processes customer information (n_6 and n_7), (iii) If the supplied currency code doesn't match the default (DEFAULT_CURR), it fetches and displays the current exchange rate between the two (n_8 to n_{10}). (iv) The loop that processes accounts also invokes a method *processTransactions* for every account, which retrieves all transactions after the *fromDate* for processing, after retrieving the balance as of *fromDate* (n_{11} to n_{15}). In order to keep the listing simple, we use methods *processAccount*, and *processCustomer* to denote all the processing that happens on account and customer data. We use this example throughout the paper.

There are many opportunities for prefetching in Figure 1, some of which are not very obvious, which are exploited by our transformations; Figure 2 shows the transformed program with prefetch submissions. For brevity, Figure 2 uses symbols $q1$, $q2$ etc. to denote actual query strings, and omits lines of code that remain unchanged.

First consider query $q2$, whose parameter is available at the very beginning of *generateReport*. Our transformations prefetch the query result by invoking *submit*($q2$,custId) at the beginning of *generateReport* as shown in Figure 2; *submit* is a non-blocking call which initiates prefetching of query results to a cache, and returns immediately. Thereby, exe-

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.

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

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

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

void processTransactions(int accId, String from){
(n11)  int startingBalance = getBalanceAsOf(from, accId);
(n12)  int balance = startingBalance;

(n13)  ResultSet t=executeQuery("SELECT *
                 FROM transactions
                 WHERE accountId=? AND date > ?
                 ORDER BY date", {accId, from}); // q4
(n14)  while(t.next()){
(n15)      balance = processTransaction(t, balance);
       }
}
```

Figure 1: Program with prefetching opportunities

```
void generateReport(int custId, int curr, String fromDate){
    submit(q2, custId);
    submitChain({q1, q4'}, {{custId}, {fromDate}});
    // q4' denotes query q4 with its first ? replaced
    // by :q1.accountId.

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

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

Figure 2: Program with prefetch requests

cution of $q2$ gets overlapped with the execution of the loop starting at line n_2.

Next, consider query $q3$, which is executed only if the predicate in line n_8 is true. Our transformations deal with this situation by issuing a prefetch conditional on the predicate in line n_8.

Similarly, query $q4$ in the method *processTransactions* can be prefetched in the method *generateReport* just after line n_3, which is the earliest point where its parameters are available. In this case, our transformations allow prefetch to be done in a calling procedure. As a further optimization, we note that the parameter *accountId* of $q4$, which becomes available in line n_3, is really a result of query $q1$, and $q4$ is executed for every row in the result of $q1$. Our transformations therefore combine the prefetching of $q1$ and $q4$ by invoking the *submitChain* procedure as shown in Figure 2. The procedure prefetches multiple queries, where parameters of later queries come from results of earlier queries; it initiates prefetch of a query once queries that it depends on have been prefetched. As yet another optimization, the procedure *submitChain* can combine the queries into a single prefetch query to the database.

Manually identifying the best points in the code to perform prefetching is hard and time consuming, because of the presence of loops and conditional branches; it is even harder in situations where the query is invoked in some procedure P, but to get any benefit the prefetch should be done in another procedure Q which calls P. Manually inserted prefetching is also hard to maintain as code changes occur.

Much of the earlier work on optimization has focused on query optimization at the database, where the given query is rewritten to the optimal form. But as can be seen from the above scenario, optimization of the application requires not just database query optimization, but also optimization of database access in the application, by means of rewriting the application code. Such optimization spanning the application and the database has been referred to as holistic optimization [13].

Prefetching has been used in databases and many other areas to hide latency and make concurrent use of resources. However, in contrast to work on prefetching query results by [8, 15, 2] which are based on predicting query access patterns, our work focuses on automatic rewriting application programs to issue prefetches effectively. Manjhi et al. [13] and our earlier work (Chavan et al. [4]) also consider automatic rewriting of application code by means of inserting prefetches (asynchronous query submissions). However, those techniques only work within a single procedure, and cannot work across procedure boundaries; and even within a procedure, techniques described in this paper improve the applicability of prefetching. Related work is discussed in more detail in Section 7.

Our technical contributions in this paper are as follows:

1. We give a novel algorithm (in Section 4), which statically inserts prefetch instructions at the earliest possible point across procedure calls, in presence of conditional branching and loops. To this end, we extend a data flow analysis technique called anticipable expressions analysis, to analyze anticipability of queries (Section 3). Although anticipable expressions analysis is a known data flow analysis technique, to the best of our knowledge there is no prior work on its use for prefetching query results.
2. In many cases, the benefit of prefetching is limited due to the presence of assignment statements and conditional branches that precede the query execution statement. We propose (in Section 5) enhancements such as code motion, chaining, and rewriting prefetch requests to increase benefits of prefetching.
3. We describe (in Section 6) how our techniques integrate with our earlier work in this area [5, 4] thereby increasing their applicability. We also describe the ap-

plicability of our work in persistence frameworks such as Hibernate [7], and Web services.

4. We discuss (in Section 8) the design considerations of incorporating our techniques into the DBridge tool [3] which rewrites Java programs to perform prefetching. We present a detailed experimental evaluation of the proposed techniques on several real world applications. The results of our experiments show significant performance gains of more than 50% in many cases.

2. BACKGROUND

In this section, we provide background material on the terminology used in the rest of the paper.

2.1 Prefetch Execution model

We assume the following execution model for prefetching: There exists a cache of query results. This cache is keyed by the tuple *(queryString, parameterBindings)*, and contains a result set. Every prefetch instruction puts its query results into this cache. If the prefetch query execution results in an exception, the exception is cached. Now we define the semantics of the methods we use.

- **executeQuery:** This is a blocking function which first checks if the *(queryString, parameterBindings)* being issued, already exists in the cache. If so, it returns the cached results, else blocks till the results are available. If the cached result is an exception, it is thrown at this point.
- **submit:** This is a non-blocking function which issues a prefetch. It accepts a query and its parameters, submits the query for execution and returns immediately.

Additionally, we define a variant of *submit* called *submitChain*, which handles chaining of prefetch requests; *submitChain* is discussed in Section 5.2.

2.2 Data Structures used

We use the following data structures in our analysis:

2.2.1 Control Flow Graph

The *Control Flow Graph* (CFG) is a directed graph that represents all paths that might be traversed by a program during its execution [14]. In a control flow graph each node represents a basic block (a straight-line piece of code without branches). There are two specially designated nodes: the *Start* node, through which control enters into the flow graph, and the *End* node, through which all control flow leaves. Additionally, for each node n, $Entry(n)$ and $Exit(n)$ represent the program points just before the execution of the first statement, and just after the execution of the last statement of n. Directed edges represent control flow; sets *pred(n)* and *succ(n)* denote the predecessors and successors of a node n respectively.

CFGs are usually built on intermediate representations such as Java bytecode. Our techniques apply to any CFG; our implementation uses CFGs built on a representation called Jimple, provided by the SOOT optimization framework [19]. The CFG for Figure 1 is shown in Figure 3.

2.2.2 Call Graph

The *call graph* (also known as a call multi-graph) is a directed graph that represents calling relationships between methods in a program. Specifically, each node represents a method and each edge (f, g) indicates that method f calls method g. In addition, each edge also stores the program point of invocation of g in f and has the mapping of formal to actual variables. A cycle in the graph indicates recursive method calls. We currently assume that the Call Graph is a directed acyclic graph (DAG), as our algorithms do not handle recursive method calls.

2.3 Data flow analysis

We now briefly describe the general framework of data flow analysis that we use in this paper. Data flow analysis is a program analysis technique that is used to derive information about the run time behaviour of a program [12]. For a given program entity e, such as an expression $a * b$, data flow analysis of a program involves two steps:
(i) Discovering the effect of individual program statements on e (called *local* data flow analysis). This is expressed in terms of sets Gen_n and $Kill_n$ for each node n in the CFG of the program. Gen_n denotes the data flow information generated within node n. For eg., the set Gen_n contains the expression $a * b$ if node n computes $a * b$. $Kill_n$ denotes the information which becomes invalid in node n. For eg., the expression $a * b$ is said to be killed in node n if n has an assignment to a or b. The values of Gen_n and $Kill_n$ are computed once per node, and they remain unchanged.
(ii) Relating these effects across statements in the program (called *global* data flow analysis) by propagating data flow information from one node to another. This is expressed in terms of sets In_n and Out_n, which represent the data flow information at $Entry(n)$ and $Exit(n)$ respectively.

The specific definitions of sets Gen_n, $Kill_n$, In_n and Out_n depend upon the analysis, and we define them for our analysis in Section 3. The relationship between local and global data flow information is captured by a system of *data flow equations*. The nodes of the CFG are traversed and these equations are iteratively solved until the system stabilizes, i.e., reaches a fixpoint. Data flow analysis captures all the necessary interstatement data and control dependences about e through the sets In_n and Out_n. The results of the analysis are then used to infer information about e.

3. QUERY ANTICIPABILITY ANALYSIS

As we saw in Figure 1, we consider programs with query executions embedded within them, along with loops, branching, and other imperative constructs. Prefetching of queries involves inserting query submission requests at program points where they were not present in the original program. The goal is to insert asynchronous query prefetch requests at the earliest possible points in the program so that the latency of network and query execution can be maximally overlapped with local computation. Suppose a query q is executed with parameter values v at point p in the program. The earliest possible points e where query q could be issued are the set of points where the following conditions hold: (a) all the parameters of q are available, (b) the results of executing q at points e and p are the same, and (c) conditions (a) and (b) do not hold for predecessors of e. For efficiency reasons, we impose an additional constraint that no prefetch request should be wasted. In other words, a prefetch request for query q with parameters v should only be inserted at earliest points where it can be guaranteed that q will be executed subsequently with parameters v.

Detecting earliest possible points for queries in the presence of multiple query execution statements, while satisfying the above constraints, requires a detailed analysis of the program. The presence of conditional branching, loops and method invocations lead to complex interstatement data and control dependences which are often not explicit in the program. We approach this problem using a data flow analysis framework called *anticipable expressions analysis* and extend it to compute *query anticipability*.

Anticipable expressions analysis [12] is a data flow analysis technique that is used for eliminating redundant computations of expressions. It can facilitate *expression motion*, which involves advancing computation of an expression to earlier points in control flow paths. This analysis is typically used for expressions with binary operators to detect earlier points in the program where they can be moved.

Our analysis differs from anticipable expressions analysis in the following aspects: (a) our goal is the insertion of prefetch instructions, not code motion (b) we compute and propagate data flow information for query execution statements as against expressions. Although anticipable expressions analysis is a known data flow analysis technique, to the best of our knowledge, no prior work shows its use for prefetching query results. The scope of this analysis is intraprocedural i.e., we use this analysis to find query anticipability within a procedure. (Moving of prefetches across procedures is discussed later, in Section 4.2.) We now formally define our analysis:

DEFINITION 3.1. *A query execution statement q is **anticipable** at a program point u if every path from u to End contains an execution of q which is not preceded by any statement that modifies the parameters of q or affects the results of q.* □

Query anticipability analysis is a data flow framework with query execution statements being the data flow values (program entities of interest). All required data flow information for this analysis are compactly represented using bit vectors, where each bit represents a query execution statement. For a query execution statement q, we define the sets (bit vectors) Gen_n and $Kill_n$ as follows: Gen_n is 1 at bit q if n is the query execution statement q. $Kill_n$ is 1 at bit q if either n contains an assignment to a parameter of q, or performs an update to the database that may affect the results of q. Conservatively, we assume that any update to the database affects the results of q. This assumption can be removed by performing a more precise interquery dependence analysis [16], taking into account the indirect effects due to views and triggers.

Query anticipability computation requires propagation of data flow information against the direction of control flow. The data flow information at $Exit(n)$ (i.e., Out_n) is computed by merging information at $Entry$ of all successors of n. The data flow equations for query anticipability analysis are:

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

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

Equation 1 defines In_n in terms of Out_n, Gen_n and $Kill_n$. Out_n is defined in Equation 2 by merging the In values of all successors of n using set intersection ($\cap$) as the merge operator. Out_{End} is initialized to be ϕ as query executions are not anticipated at $Exit(End)$. We use $\cap$ to capture the notion that the query execution statement is anticipable at Out_n only if it is anticipable along *every path* from n to End. In_n and Out_n for all other nodes are initialized to the universal set.

The nodes of the CFG are traversed in reverse topological order and the values of Out_n and In_n are calculated for each node; this process is repeated until the system reaches a fixpoint. (An example of query anticipability analysis is discussed in detail later, in Section 4.)

For a given query execution statement q, query anticipability analysis discovers a set of anticipability paths. Each such path is a sequence of nodes $(n_1, n_2, ..., n_k)$ such that:

- n_k is the query execution statement q,
- n_1 is either $Start$, or contains an assignment to some parameter of q, or performs an update to the database,
- no other node in the path contains an execution of q, or an assignment to any parameter of q, or an update to the database
- q is anticipable at every node in the path.

Anticipability can be blocked by the presence of *critical edges* in the CFG. A *critical edge* is an edge that runs from a *fork node* (a node with more than one successor) to a *join node* (a node with more than one predecessor). Such a critical edge is removed introducing a new node along the edge such that the new node has no other predecessor other than the *fork node*. Removal of critical edges is a standard technique used in code motion optimization [12], and it increases anticipability atleast along one path.

In the next section, we illustrate query anticipability analysis with an example, and describe how this analysis feeds into our prefetching algorithm, to identify earliest points for issuing prefetches across methods.

4. PREFETCH INSERTION ALGORITHM

We now present our novel algorithm for inserting query prefetch instructions in a program. We first describe an algorithm to place prefetch requests at earliest points within a procedure. Subsequently, we describe an algorithm that inserts prefetches across procedures.

4.1 Intraprocedural prefetch insertion

We begin with the algorithm for inserting prefetch instructions within a procedure, which we call intraprocedural prefetching. We make the following assumptions about the input procedure:

- Statements have no hidden side-effects. All reads and writes performed by a statement are captured in the Gen_n and $Kill_n$ sets. Importantly the reads and writes of variables within method invocations, the effects of global variables and shared data structures, are also captured. This assumption is easy to verify for program variables, but harder in presence of complex data structures and objects with aliasing. Therefore, as done by most optimizing compilers, a safe but less precise analysis could be performed to verify the assumption [14, 12]. This may introduce spurious dependences that reduce opportunities for prefetching, but will continue to guarantee program equivalence.

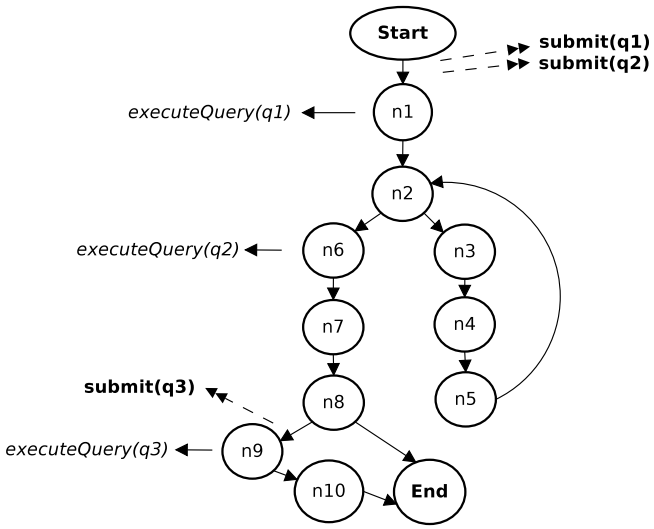

Figure 3: CFG for method *generateReport* of Figure 1

```
procedure InsertPrefetchRequests(CFG G)
begin
  remove all critical edges by edge splitting
  Q = {all query execution statements in G}
  perform QueryAnticipabilityAnalysis on G w.r.t Q

  for each Stmt q ∈ Q do begin
    for each Node n ∈ CFG G do begin
      if (Out_{n,q} ∩ ¬In_{n,q}) then begin
        //Case 1: data dependence barrier
        appendPrefetchRequest(n, q)
      else if (In_{n,q} ∩ ¬⋃_{m∈pred(n)} Out_{m,q})
        //Case 2: control dependence barrier
        prependPrefetchRequest(n, q)
      endif
    endfor
  endfor
end
```

Figure 4: Intraprocedural prefetch insertion algorithm

- The parameters to the query are primitive data types (int, float etc.) or strings. However, our techniques can be extended to arrays and objects.
- For simplicity of notation, the query execution statement is assumed to be of the form *executeQuery(sqlQuery, parameterBindings)* where the *sqlQuery* is a string and the *parameterBindings* is an array of primitive data types. This is a simplifying assumption and can be easily removed to make the underlying CFG aware of the data access API used. Our implementation works on Java programs that use JDBC API, and hence is JDBC-API aware.
- The prefetch instruction and the query execution statement execute within a single transaction; we discuss issues in enforcing this later, in Section 8.1.

The algorithm, shown in Figure 4 accepts the CFG of a procedure as input and returns a modified CFG with prefetch requests inserted. We illustrate its operation with the example in Figure 1. Consider the method *generateReport*, which

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

Table 1: Query anticipability analysis for method *generateReport* of Figure 1

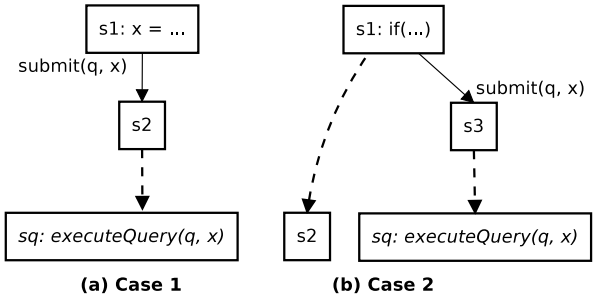

Figure 5: Barriers for prefetching

embeds 3 queries labeled $q1, q2$ and $q3$. The corresponding CFG, which indicates *executeQuery* nodes is shown in Figure 3. The methods *processAccount*, *processCustomer* and *printExchangeRate* do not involve any updates to the database that may invalidate the query results.

Algorithm *InsertPrefetchRequests* proceeds as follows: As a preprocessing step, critical edges are removed by introducing new nodes along them as described in [12]. In our example, the CFG remains unchanged as there are no critical edges. Then, we collect all query execution statements in a set Q, which forms the structure of our bit vector used for query anticipability analysis. In the example, $Q = \{q1, q2, q3\}$.

The values of the sets Gen_n and $Kill_n$ for each node are computed first. $Kill_{Start}$ is defined as 111 in our example since the *Start* node assigns values to all parameters of the procedure. Out_{End} is initialized to ϕ (000), with In_n and Out_n initialized to the universal set (111 in our example) for all other nodes. Then the fixpoint is computed as described in Section 3. In our example, the values converge in 2 iterations. The results of performing query anticipability analysis on Figure 1 is shown in Table 1. The table shows only the changed values in iteration #2.

This analysis provides us with information about all points in the procedure where queries are anticipable. However, as stated earlier, we are interested in the earliest point where the query is anticipable. There are two cases to consider in order to arrive at the earliest point of anticipability, shown

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

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

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

(n8)    if(curr != DEFAULT_CURR){
            // prefetch q3
            submit(q3, {curr, DEFAULT_CURR});

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

void processTransactions(int accId, String from){
        submit(q4, {accId, from});
(n11)   int startingBalance = getBalanceAsOf(from, accId);
(n12)   int balance = startingBalance;

(n13)   ResultSet t=executeQuery("SELECT *
                FROM transactions
                WHERE accountId=? AND date > ?
                ORDER BY date", {accId, from}); // q4
(n14)   while(t.next()){
(n15)       balance = processTransaction(t, balance);
        }
}
```

Figure 6: Result of *InsertPrefetchRequests* on Figure 1

in Figure 5 with statements denoted by $s1$, $s2$, $s3$, and sq and paths in the CFG denoted by dashed arrows.

Case 1: As shown in Figure 5(a), $s1$ prevents the prefetch from being placed above it, due to an assignment to x. Such barriers can be due to assignments to query parameters or updates to the database that affect the query result. Update statements can be seen as external data dependences, where the dependence exists through the database. These barriers are called *data dependence barriers*, since it is a data dependence that prevents the prefetch to be placed before this barrier.
More formally, if query q is not anticipable at $Entry(n)$, but is anticipable at $Exit(n)$, the prefetch statement is inserted immediately after n. This indicates that n is the initial node of an anticipability path for q. In our example, this case applies for queries $q1$ and $q2$ since they are not anticipable at In_{Start} but become anticipable at Out_{Start} as indicated by Table 1.

Case 2: This case is shown in Figure 5(b). Here, sq is control dependent on $s1$, since the predicate evaluated at $s1$ determines whether or not control reaches sq. The prefetch for q cannot be moved to $s1$ since the path $s1 \rightarrow s2$ does not issue the query subsequently. Such barriers due to conditional branching (*if-then-else*) are called *control dependence barriers*, since it is a control dependence that prevents the prefetch from being moved earlier.
Formally, if query q is anticipable at $Entry(n)$, and not anticipable at the $Exit$ of any of the predecessors of n, the prefetch statement is inserted immediately before n. In our example, this case applies for query $q3$, as indicated by Table 1 at Out_{n_8} and In_{n_9}.

In our algorithm, $In_{n,q}$ and $Out_{n,q}$ represent the anticipability of q at In_n and Out_n respectively. The procedures *appendPrefetchRequest* and *prependPrefetchRequest* accept a node n and a query execution statement q, prepare the prefetch instruction for q, and insert it immediately after or before n respectively. Note that there can be multiple points in a program where a prefetch can be issued for a query q as there could be multiple paths reaching q from $Start$. We omit the details of running query anticipability analysis on the method *processTransactions* due to lack of space. The output of Algorithm *InsertPrefetchRequests* on Figure 1 is shown in Figure 6. The CFG of method *generateReport* shown in Figure 3 indicates the points where prefetch submissions are placed relative to the points where they are in the original program. The prefetch for query $q2$ has been placed at the beginning of the procedure beyond the loop. Query $q3$ cannot be moved before n_8 as it is not anticipable along the path $n_8 \rightarrow End$. We describe techniques to move these prefetches further ahead, in Section 5. Note that the parameters to the *submit()* call in Figure 6 are actual query strings, and the symbols $q1$, $q2$ etc. have been used only as a shorthand for readability in this paper.

4.2 Interprocedural prefetching

The benefit of prefetching can be greatly increased by moving prefetches across method invocations. For instance, consider the query $q4$ in method *processTransactions* in Figure 1, which is executed with parameters *accId* and *from*. The method *processTransactions* is invoked from *generateReport* (line n_5), and the *accountId* used in $q4$ is available right after n_3. Query $q4$ can be submitted for prefetch in *generateReport*, right after n_3, thereby overlapping it with the method *processAccount*. The potential benefit here is much greater than if the prefetch could only be done in the method where the query is executed.

We now show how to extend our algorithm to perform interprocedural prefetching. Our algorithm can handle arbitrary levels of nesting, and can move the prefetch instruction across these levels while preserving the preconditions given in Section 3. Our algorithm currently cannot handle recursive method calls i.e., we assume that the Call Graph is a DAG.

The interprocedural prefetch insertion algorithm *InsertInterproceduralPrefetchRequests* is shown in Figure 7. The input to the algorithm is the call graph of the program, along with the CFGs of all the procedures involved. The intraprocedural algorithm *InsertPrefetchRequests* is used as a subroutine, after modifying it as follows: (i) The set Q of query execution statements now additionally considers prefetch requests of the form *submit(sqlQuery, parameterBindings)* where the *sqlQuery* is a string and the *parameterBindings* is an array of primitive data types. (ii) Be-

```
procedure InsertInterproceduralPrefetchRequests
(CallGraph CG)
begin
  rts = {Vertices of CG sorted in reverse topological order}
  for each Vertex v ∈ rts do begin
    // run the modified intraprocedural algorithm
    InsertPrefetchRequests(cfg(v))

    s = Entry(cfg(v)) // the first statement of procedure v
    while isPrefetchStatement(s) do begin
      remove(s,v) //remove s from procedure v
      callSites = {cfg(src(e)) | e ∈ CG and dest(e) == v}
      for each CFG cs ∈ callSites do begin
          t = {all invocations of v in cs}

          for each Stmt c ∈ t do begin
            // replace formal parameters in s with their
            // actual counterparts in c
            replaceParameters(s, c)
            prependPrefetchRequest(s, t)
          endfor
      endfor
      s = Entry(cfg(v))
    endwhile
  endfor
end
```

Figure 7: Interprocedural prefetch insertion

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

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

(n1)    ResultSet a=executeQuery("SELECT *
                        FROM accounts
                        WHERE custId=?", custId); // q1
(n2)    while(a.next()){
(n3)        int accountId = a.getInt ("accountId ");
            submit(q4, {accountId, fromDate});//prefetch q4
(n4)        processAccount(a);
(n5)        processTransactions(accountId, fromDate);
        }
(n6)    ResultSet c = executeQuery("SELECT *
                FROM customers
                WHERE custId=?", custId); // q2
(n7)    processCustomer(c);

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

Figure 8: Output of InsertInterprocedural-PrefetchRequests on Figure 1 (with code motion)

fore the point of invoking *appendPrefetchRequest* or *prepend-PrefetchRequest*, if q is a prefetch statement, it is removed from its original point.

The key intuition behind the interprocedural algorithm is the fact that if a prefetch can be submitted at the beginning of a procedure, it can instead be moved to all its call sites. The vertices of the call graph are traversed in the reverse topological order, and *InsertPrefetchRequests* is invoked for the CFG of each vertex. Then, the first statement of the CFG is examined to see if it is a prefetch submission. If so, then the prefetch statement is inserted just before the method invocation of interest in all its call sites. This additionally requires the replacement of formal parameters in the prefetch statement, with the actual parameters in the call site. Traversing the call graph in reverse topological order ensures that no prefetch opportunities are lost, since, all successors of a vertex v are processed before processing v.

In our example of Figure 1, the modified intraprocedural algorithm is first run on *processTransactions*, which brings the prefetch submission of query $q4$ to the beginning of the method as shown in Figure 6. Then the call graph is looked up and the prefetch instruction is prepended to the method invocation on line n_5 of *generateReport*. As part of this, the parameters *accId* and *from* in *processTransactions* are replaced by *accountId* and *fromDate* in *generateReport*. Subsequently, the run of the modified intraprocedural algorithm on *generateReport* moves the prefetch of $q4$ to the point immediatly after line n_3, as shown in Figure 8. Also, the intraprocedural algorithm inserts prefetch requests for queries $q1$ and $q2$ at the beginning of *generateReport*, as described in Section 4.1, and indicated in Figure 3. Therefore, queries $q1$ and $q2$ can be moved to call sites of *generateReport*.

4.3 Discussion

We presented two algorithms for statically inserting prefetch instructions. Algorithm *InsertPrefetchRequests* uses query anticipability analysis and inserts prefetch instructions at the earliest points within a procedure, in the presence of conditional branching, loops and other intraprocedural constructs. Algorithm *InsertInterproceduralPrefetchRequests* combines this analysis and inserts prefetch instructions at the earliest possible points in the whole program, across method invocations. Our algorithms ensure the following: (i) No prefetch request is wasted. A prefetch is inserted at a point only if the query is executed with the same parameter bindings subsequently. (ii) All existing statements of the program remain unchanged. These algorithms only insert prefetch requests at specific program points and hence they are very non-intrusive. (iii) The transformed program preserves equivalence with the original program.

The presence of conditional exits due to exceptions may result in query execution not being anticipable earlier in the program. Since such exits are rare, we can choose to ignore such exits when deciding where to prefetch a query, at the cost of occassional wasted prefetches. We chose this as the default option in our implementation.

5. ENHANCEMENTS

We now discuss three enhancements to our prefetching algorithm to increase the benefits of prefetching. For instance, consider query $q3$ in Figure 6, where the prefetch submission just precedes the query invocation, which would not provide any performance benefit. The enhancements described in this section allow the prefetch to be done earlier, allowing better performance. The enhancements are based on equivalence preserving program and query transformations.

void ***submitChain***(*queries, params*)
where

- *queries*: Array of query strings
- *params*: Array of arrays of parameters for the queries, other than chaining parameters.

Figure 9: The submitChain interface

```
submitChain ({" SELECT * FROM accounts
                 WHERE custId=?",
               " SELECT * FROM transactions
                 WHERE accountId = :q1.accountId
                       AND date > ?"},
             {{custId},{fromDate}});
```

Figure 10: Chaining for $q1$ and $q2$ of Figure 8

5.1 Transitive code motion

The goal of prefetching is to hide the latency of query execution (due to network and disk IO) by overlapping them with local computations or other requests. In terms of the CFG, this means that the longer the length of the paths from the prefetch request P to the query execution statement Q, the more the overlap, and the more beneficial the prefetch. The distance between P and Q can be increased by transitively applying the *code motion* optimization.

In Section 4.1, we described data and control dependence barriers that prevent the prefetch from being inserted earlier (Figure 5). We now present a technique to increase the benefits of prefetching in presence of these barriers. Whenever a barrier is encountered, we perform the following transformations: (i) If a control dependence barrier is encountered, that control dependence is transformed into a data dependence using '*if-conversion*' [11], or equivalently by transforming them to guarded statements as discussed in [5]; in either case, the control dependence barrier gets transformed into a data dependence barrier. (ii) If a data dependence barrier (assignment to a query parameter) is encountered, we transitively apply anticipability analysis on barrier statements and move them to their earliest possible point, and recompute the anticipability of the query. Such transitive movement of the barrier statement can allow the prefetch to be performed earlier.

We illustrate transitive code motion with $q3$ in Figure 6 as an example. Here, a control dependence barrier due to the predicate *(currCode != DEFAULT_CURR)* is encountered. This is transformed into a data dependence using a variable b to hold the value of the predicate. Now, an anticipability analysis of the predicate reveals that it could be placed at the beginning of the method, and a guarded prefetch submission is placed just after it. The output of transitive code motion on Figure 6 is shown in Figure 8. Among existing lines of code, only line n_8 is transformed to use the variable b. For lack of space we omit formal analysis and details of algorithms to perform transitive code motion for prefetching. We note however that transitive code motion has been partially implemented in the code used in our experiments.

5.2 Chaining Prefetch requests

A commonly encountered situation in practice is the case where the output of one query feeds into another. This is an example of a *data dependence barrier*, as described in Section 5.1, where the dependence arises due to another query. For example say a query $q1$ forms a barrier for submission of $q2$, but $q1$ itself has been submitted for prefetch as the first statement of the method. As soon as the results of $q1$ become available in the cache, the prefetch request for $q2$ can be issued. This way of connecting dependent prefetch requests is called chaining.

According to our execution model, the prefetch requests are asynchronously submitted. In this model, chaining can be visualised as a sequence of events and event handlers. A handler (or multiple handlers) is registered for every query. As soon as the results of a query are available, an event is fired, invoking all the handlers that subscribe to this event. These event handlers (which can be thought of as callback functions) pick up the results and issue prefetch submissions for subsequent queries in the chain. The event handlers themselves raise events which in turn trigger other handlers, which goes on till the chain is complete.

Such a chain is set up as follows: Suppose we have a set of queries (q_1, q_2, ..., q_k), such that q_i forms a barrier for q_{i+1}. Let the set p_i denote the results of q_i that form the parameters to q_{i+1}. Let p_0 denote the parameters for q_1. Now, at the point of prefetch of q_1, we initiate this chain of prefetches by registering handlers for each query. The set p_i (which we call as the set of chaining parameters) is passed to the handler that executes query q_{i+1}. We encapsulate all these details using a simple interface in order to preserve readability. The program is rewritten to use the API method *submitChain()* whenever a query result is found to be a data dependence barrier. This makes the rewrite straightforward. The signature and semantics of the *submitChain* method are shown in Figure 9. Chaining parameters (i.e. parameters that come from the result of an earlier query) are represented as :q_i.*attrname*, in the query string itself.

This kind of chaining can be extended to iterative execution of queries in a loop, where the following conditions hold: (i) the parameters of the query in the loop (say q_{loop}) are from the results of a previous query that is outside the loop (say q_{outer}), (ii) the loop iterates over the all the tuples in the results of q_{outer}, (iii) q_{loop} is unconditionally executed in every iteration of the loop. Such cases are commonly encountered in practice. For instance, queries $q1$ and $q2$ in Figure 8 satisfy these conditions. Once the dependence between $q2$ and $q1$ are identified, along with the chaining parameter, the *submitChain* API method is invoked as shown in Figure 10. Once the first query in the chain executes, the second query is issued for all the *accountId* values returned by the first query.

```
SELECT *
FROM (SELECT * FROM accounts WHERE custId = ?)
     OUTER APPLY
     (SELECT * FROM transactions
      WHERE transactions.accountId = account.accountId
            AND date > ?)
```

Figure 11: Prefetch query rewrite for Figure 10

5.3 Rewriting Prefetch requests

Chaining by itself can lead to substantial performance gains, especially in the context of iterative query execu-

tion whose parameters are from a result of a previous query. Chaining collects prefetch requests together, resulting in a set of queries with correlations between them. Such queries can be combined and rewritten using known query decorrelation techniques [17]. In order to preserve the structure of the program, the results of the merged rewritten query are then split into individual result sets and stored in the cache according to the individual queries.

Figure 11 shows the rewritten query for queries in the chain of Figure 10, using the OUTER APPLY syntax of SQL Server. This kind of query rewrite has been done earlier in [17, 5, 2] etc. by the use of the LEFT OUTER LATERAL or OUTER APPLY operators. Rewriting not only reduces round trips of query execution, but also aids the database in choosing better execution plans. The resulting code achieves the advantages of batching without having to split the loop. In some cases it can perform better than batching, since the overhead of creating a parameter batch [5] is avoided.

6. DISCUSSION

We now consider how our transformation can work in conjunction with existing transformation techniques, and with Hibernate and Web services.

6.1 Integration with loop fission

In our earlier work (Guravannavar et. al. [5] and Chavan et. al. [4]) we had proposed program transformation methods to exploit set oriented query execution or asynchronous submission to improve performance of iterative execution of parameterized queries. Both asynchronous query submission and batching depend on the loop fission transformation which is effective within a procedure, but not effective in optimizing iterative execution of procedures containing queries. For example, consider a query which is executed in method M, which is invoked from within a loop in method N. Performing loop fission to enable batching (or asynchronous submission) requires one of the following transformations to M: (i) a set-oriented (or asynchronous) version of M, (ii) fission of method M into two at the point of query execution, (iii) inlining of M in N. All these transformations are very intrusive and complex.

Our prefetching algorithm can be used as a preprocessing step to apply the loop fission transformation as follows: Consider the case where a query execution is deeply nested within a method chain, with a loop in the outermost method. Algorithm *InsertInterproceduralPrefetchRequests* brings the prefetch statement up the method call hierarchy into the method with the loop whenever possible. At this point, if the preconditions for prefetch chaining and rewrite (Section 5.2) are not satisfied, the loop fission transformation can be applied as described in [5] or [4]. Also, loop fission is not applicable if the query execution is part of a cycle of true dependencies [5], and is very restricted in the presence of exception handling code. In many such cases, our techniques are applicable and beneficial.

6.2 Hibernate and Web services

A lot of real world applications are backed by persistence frameworks such as Hibernate [7], or by data sources exposed through Web services (eg. Amazon, Twitter etc.). Programs that use Hibernate rarely contain SQL queries directly embedded in application code. They invoke the API methods of Hibernate, which in turn generate the necessary SQL queries using the O/R mapping information provided in configuration files or annotations. Web services are typically accessed using APIs that wrap the HTTP requests and responses defined by the Web service.

Although we describe our algorithms in the context of programs that embed SQL queries, the algorithms are more generic and applicable for a wider class of applications. To apply our techniques for such applications effectively, (i) our CFG has to be aware of the data access API in order to place prefetches appropriately, and (ii) there has to be runtime support to issue asynchronous prefetches for these data access methods. For example, querying in Hibernate is primarily done either by the use of HQL (Hibernate Query Language), or by the QBC (Query by Criteria) API, apart from native SQL queries. With some analysis, these API methods can be incorporated into our CFG before we run our prefetch insertion algorithm.

Our implementation currently supports prefetching through asynchronous submission for JDBC API, a subset of the Hibernate API and the Twitter API, used in our experments. Some databases and Web services provide asynchronous APIs for data access. Our transformation techniques can be tailored to use these APIs for prefetching.

7. RELATED WORK

The idea of prefetching has been used in many areas of computer science. Prefetching has long been supported for device IO in operating systems, especially when it is sequential. Databases internally use prefetching extensively to improve performance of query processing. Sequential scans can be speeded up to a large extent by prefetching as shown by Smith [18]. Even if the access pattern is not strictly sequential, it exhibits spatial locality in many cases, and prefetching is achieved by fetching blocks or pages at a time.

There has been earlier work where the prefetch is not based on physical layout and spatial locality, but on request patterns. Fido [15] recognizes patterns of object references in order to predict future references, but does not apply to queries. Haas et. al. [6] propose an approach which allowed a query to prefetch (return) extra attributes that would have been fetched by a subsequent query. The Scalpel system [2] detects patterns in request streams of client applications during a training phase, and generates rules which are used for prefetching at runtime. In contrast to the above systems, our approach has no overhead of training since it is based on static analysis, never performs wasteful prefetches, and can exploit code motion to prefetch at an earlier point.

More recently, approaches based on static analysis have been proposed to address problems with similar goals. Manjhi et. al. [13] consider prefetching of query results by employing non-blocking database calls. For every query, they place a copy of all variable initializations that the query uses directly or indirectly (through some other variable) at the beginning of the program. Next, they put a non-blocking-execute function call for the query after all these variable initializations. However, as we demonstrate in this paper, this problem requires a detailed analysis of the program. Firstly, placing copies of all variable initializations at the beginning of the program may not only duplicate many computations, but worse, it can lead to incorrect behaviour in the presence of side effects, global variables, and conditional assignments. Secondly, they do not consider interprocedural prefetch, which restricts the benefits of their algorithm.

Our work guarantees correctness and places prefetches at the earliest possible point across method calls.

In our earlier work (Guravannavar et. al. [5] and Chavan et. al. [4]), we proposed program transformation methods to exploit set oriented query execution or asynchronous submission to improve performance of iterative execution of parameterized queries. Although batching reduces round-trip delays and allows 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 set-oriented interface for the request invoked. As discussed in Section 6.1, both [5] and [4] depend on loop fission which is very intrusive for interprocedural code; the techniques proposed here do not depend on loop fission, although as discussed in Section 6.1 the two approaches can be used together for maximum benefit.

8. SYSTEM DESIGN AND EXPERIMENTAL RESULTS

In this section we briefly discuss some system design considerations, and then present an experimental evaluation.

8.1 System Design

The techniques we propose can be used with any language and data access API. We have implemented these techniques, with Java as the target language, as part of the DBridge holistic optimization tool [3]. Our system has two components:

(i) A **runtime library (Prefetch API)** for issuing prefetch requests and managing the cache. This currently works with JDBC [9], a subset of Hibernate [7] API, and the Twitter [20] API. It uses the *Executor* framework of the *java.util.concurrent* package for thread scheduling and management. Obviously, using threads with separate connections to the database is not transaction safe, as it can only guarantee read committed level of isolation. However, higher levels of isolation can be achieved by either (a) using the asynchronous operation support provided by some databases (eg. MS SQL Server), or (b) using the Java Transaction API (JTA), if the underlying database allows multiple connections to work on a single transaction, or (c) on systems which support snapshot isolation, by sharing the same snapshot between connections (not currently suported by any system we know of, but this feature has been proposed for PostgreSQL). Also, in our current implementation, the cache is a simple hash map, with eviction under program control.

(ii) A **source-to-source program transformer**, which inserts prefetch API calls at appropriate points in a program. The transformer currently works with JDBC calls but can be easily extended for other data access APIs. Our prefetch insertion implementation operates on the CFG and the call graph of the input program, which is provided by the SOOT optimization framework [19]. SOOT uses an intermediate representation called Jimple and provides the CFG, the data dependence information and also a framework for performing fixed point iteration required for query anticipability analysis. Our implementation inserts prefetch instructions in Jimple code, which is then translated back into Java. Since our techniques cause minimal changes to the input program (mostly only insertion of prefetch method invocations), the readability of the transformed code is preserved.

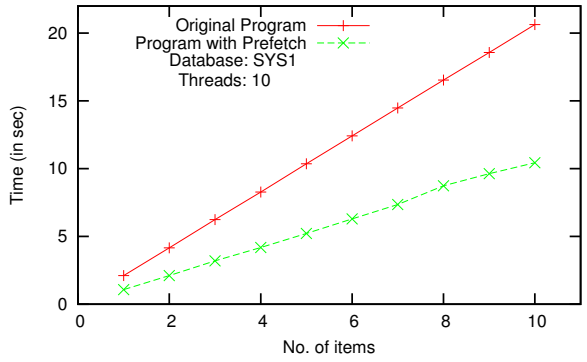

Figure 12: Experiment 1: Auction System (JDBC)

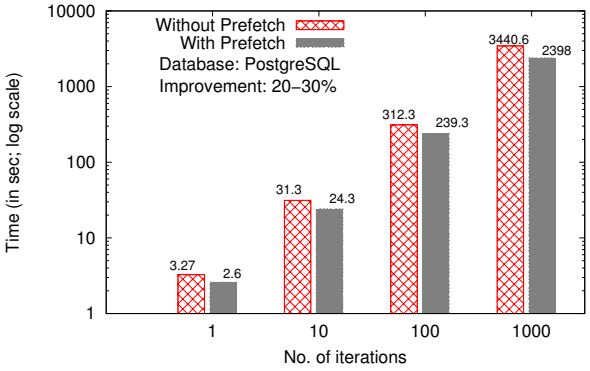

Figure 13: Experiment 2: Bulletin Board (Hibernate)

8.2 Experimental Results

We evaluate the benefits and applicability of the proposed techniques using four applications - two publicly available benchmarks for database applications, one real world commercial ERP application, and another real world application using a Web service. We have performed the experiments with two widely used database systems - a commercial system we call SYS1, and PostgreSQL. The database servers were run on 64 bit Intel Core-2 2.4 GHz machines with 4 GB of RAM. The Java applications were run from a remote machine with a similar processor and memory configuration, connected to the database servers over a 100Mbps LAN.

8.2.1 Experiment 1: Auction Application

We consider a benchmark JDBC application called RUBiS [10] that represents a real world auction system modeled after *ebay.com*. The application has a nested loop structure in which the outer loop iterates over a set of items, and loads all the review comments about it. The inner loop iterates over this collection of comments, and for each comment loads the information about the author of the comment. Finally the outer loop executes an aggregate query to maintain author and comment counts. The comments table had close to 600,000 rows, and the users table had 1 million rows. In this experiment, we only perform intraprocedural prefetching of the aggregate query. As a result, the prefetch instruction is placed before the inner loop, thereby achieving overlap of this loop. We consider the impact of our transformation as we vary the number of iterations of the outer loop, fixing

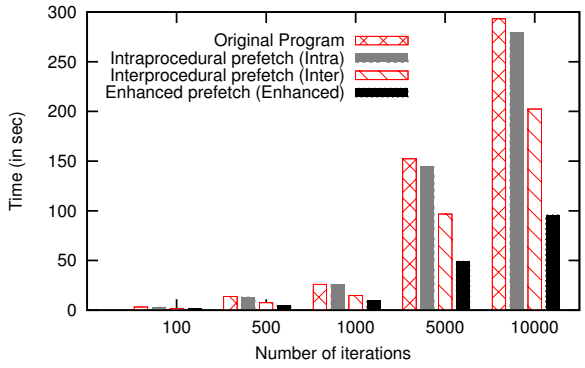

Figure 14: Experiment 3: ERP System (JDBC)

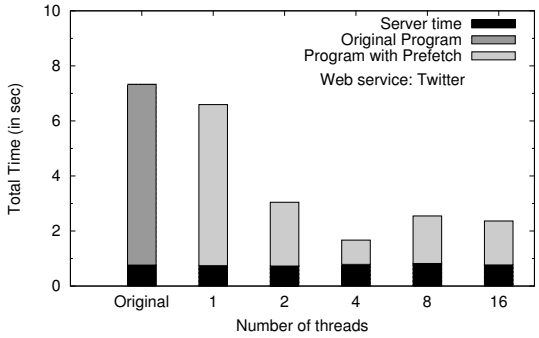

Figure 15: Experiment 4: Twitter (Web Service)

the number of threads at 10. Figure 12 shows the performance of this program before and after the transformation on SYS1. The x-axis denotes the number of items that are processed (the iterations of the outer loop), and the y-axis shows the total time taken. We observe that we consistently achieve about 50% improvement in the total time taken.

8.2.2 Experiment 2: Bulletin Board Application

RUBBoS [10] is a benchmark bulletin board-like system inspired by *slashdot.org*. For this experiment we consider the scenario of listing the top stories of the day, along with details of the comments made against them, using the Hibernate API for data access. The program loads the top stories, and iterates on each to load the details. Finally, it loads the comments on those stories. All these operations happen in different methods of a class. There were close to 10000 stories and more than 1.5 million comments in the database. We have manually inserted the prefetch requests according to the interprocedural prefetch insertion algorithm, since our implementation currently transforms only JDBC calls. However, we have extended our runtime API to handle a subset of the Hibernate API, to issue asynchronous prefetches. We consider the impact of prefetching as we vary the number of iterations, fixing the number of thread at 5. Figure 13 shows the results of this experiment in log scale on PostgreSQL. The y-axis denotes the end-to-end time taken (in seconds; log scale) by the program to execute. The actual values of the time taken are displayed along with the plot. The reduction in time taken ranges from 20% to 30%.

8.2.3 Experiment 3: ERP Application

We consider a popular commercial opensource Java ERP application called Adempiere [1],that uses JDBC. We consider the scenario of calculating taxes on orders with line items. Initially all the line items for an order are loaded (using a query $q1$) by invoking a method passing in the *orderId*. Then, for each item, a method is invoked to compute tax. This method loads the taxable amount and the type of tax applicable (using query $q2$), and returns the tax for that item. Finally, the tax for the order is computed by aggregating the taxes of all its line items. Here query $q2$ is present inside a method that is invoked from within a loop. The *lineitems* table had 150,000 rows. In this experiment, we analyze the impact of each of the techniques we present in this paper. First, we run the original program. Then we incrementally apply our techniques, namely *Intra* (intraprocedural prefetching Section 4.1), *Inter* (interprocedural prefetching Section 4.2), and finally *Enhanced* which includes code motion and query rewrite (Section 5). (For this experiment, code motion did not provide any benefits, so the benefit is purely due to query rewrite.) Query rewrite is currently done manually.

The results of this experiment on PostgreSQL are shown in Figure 14. The y-axis denotes the end to end time taken for the scenario to execute, which includes the application time and the query execution time. We measure the time taken for orders with number of line items (and thus the number of iterations) varying between 100 to 10000. We observe that the *Intra* approach provides only moderate gains. *Inter* provides substantial gains (between 25-30%) consistently. The reason for the improvement is that prefetches of $q1$ and $q2$ were moved to the calling methods, achieving more overlap. However, the use of the *Enhanced* approach leads to much bigger gains (about 50% improvement over *Inter*). *Enhanced* is able to achieve such gains as it reduces roundtrips by merging the two queries. This program was also a good example of how the *Intra* and *Inter* rewriting brought prefetch requests together, which then allowed query rewrite to merge the queries.

8.2.4 Experiment 4: Twitter Dashboard

In this experiment, we consider an application that monitors a few keywords (4 in our example), and fetches the latest tweets about those keywords from Twitter [20], a realtime information network. The public information in Twitter can be accessed using an API using JSON over HTTP. The application, written in Java, uses the Twitter4j library to connect to Twitter, and fetch the latest tweets with the necessary keywords.

We have extended our runtime prefetching library to work with Twitter requests, and manually inserted the prefetch instructions in the input program according to our interprocedural algorithm. The results of this experiment are shown in Figure 15. Since the Twitter requests are now prefetched asynchronously, they overlap with each other and hence save a lot of the network round trip delay. The actual time taken at the Twitter servers are also reported along with the response, and have been shown in Figure 15 as "Server time". The remaining time includes network latency and local computation time. Observe that the server time is almost the same for each case, but the total time taken decreases and reaches the minimum when we use 4 threads. At this point, we achieve more than 75% improvement in the total time

	Applicable Algorithm		
Query executions	Intra	Inter	Enhancements
100	32	63	16

Table 2: Applicability of prefetching

taken. As we increase the number of threads beyond 4, the total time taken increases. As our example monitors 4 keywords, there is an overhead to maintaining additional threads and Twitter connections. Since this experiment was conducted on the live Twitter API on 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 shows the applicability of our techniques beyond database query submission.

8.2.5 *Applicability of prefetching*

In order to evaluate the applicability of our prefetching techniques, we again consider Adempiere [1] (used in Experiment 8.2.3). We have analyzed a subset of the Adempiere source code to find out how many query execution statements can be prefetched using our techniques, and to what extent. (Finding the actual time benefits in these cases due to prefetching is an area of future work.) The results of the analysis are presented in Table 2. Out of 100 query execution statements, 32 were such that only intraprocedural prefetching was possible. In 63 cases, we were able to move prefetches across methods. The enhancements was applicable in 16 cases. Prefetching was not possible in 5 cases. Overall, our techniques are able to issue prefetches for 95% of the queries. The 32 cases where we were not able to move prefetches to calling methods were mainly due to conditional execution of queries, which prevents interprocedural prefetching.

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 150 lines of code.

9. CONCLUSION AND FUTURE WORK

We propose a program analysis based approach to automatically detect opportunities for prefetching query results in database applications. The algorithms presented in this paper significantly improve performance by prefetching across procedure calls, while avoiding wasteful prefetches. We also propose enhancements that transform programs and queries to increase applicability as well as benefits of prefetching. Although we present our techniques in the context of database queries, they are more general in applicability. We present a detailed experimental study, conducted on real world and benchmark applications, that show performance gains of more than 50% in many cases.

As part of future work, firstly, we plan to complete our implementation to handle all the API methods of Hibernate, and to provide extensibility features enabling easy addition of any Web service API. We also plan to implement a more sophisticated cache manager, supporting standard replacement policies as well as invalidation of cached results. We also plan to make the decision of which calls to prefetch, and the program point where it needs to be placed in order to maximize benefit, in a cost based manner. Also, our prefetching algorithm currently moves the prefetch instruction to call sites only if it can be pushed to the entry of a method. However, in many cases, there could be assignment statements that only the query depends on, which could also be moved to call sites along with the prefetch. This requires our code motion algorithm to be extended for the interprocedural case.

Acknowledgements: The work of Karthik Ramachandra was supported by Microsoft Research India PhD Fellowship and Yahoo! Key Scientific Challenges Program award.

10. REFERENCES

[1] Adempiere http://www.adempiere.org/.
[2] I. T. Bowman and K. Salem. Semantic prefetching of correlated query sequences. In *ICDE*, 2007.
[3] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. Dbridge: A program rewrite tool for set-oriented query execution. In *ICDE*, pages 1284–1287, 2011.
[4] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. Program transformations for asynchronous query submission. In *ICDE*, 2011.
[5] R. Guravannavar and S. Sudarshan. Rewriting procedures for batched bindings. *Proc. VLDB Endow.*, 1(1):1107–1123, 2008.
[6] L. M. Haas, D. Kossmann, and I. Ursu. Loading a cache with query results. In *VLDB*, 1999.
[7] Hibernate http://www.hibernate.org.
[8] A. Ibrahim and W. R. Cook. Automatic prefetching by traversal profiling in object persistence architectures. In *Proc. of the ECOOP*, 2006.
[9] Java Database Connectivity (JDBC) API http://java.sun.com/products/jdbc/overview.html.
[10] ObjectWeb Consortium-JMOB (Java middleware open benchmarking).
[11] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Procs. ACM/IEEE conference on Supercomputing*, pages 407–416, 1990.
[12] U. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., 1st edition, 2009.
[13] A. Manjhi, C. Garrod, B. M. Maggs, T. C. Mowry, and A. Tomasic. Holistic Query Transformations for Dynamic Web Applications. In *ICDE*, 2009.
[14] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
[15] M. Palmer. Fido: A cache that learns to fetch. In *VLDB*, pages 255–264, 1991.
[16] 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.
[17] P. Seshadri, H. Pirahesh, and T. C. Leung. Complex Query Decorrelation. In *ICDE*, 1996.
[18] A. J. Smith. Sequentiality and prefetching in database systems. *ACM Trans. Database Syst.*, 3:223–247, Sept. 1978.
[19] Soot: A Java Optimization Framework http://www.sable.mcgill.ca/soot.
[20] Twitter http://www.twitter.com.