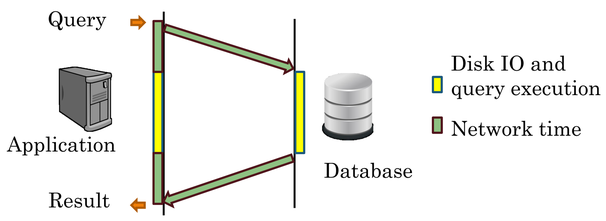
Holistic Optimization by Prefetching Query Results

Karthik Ramachandra & S. Sudarshan, IIT Bombay

The Latency Problem

- Database applications experience lot of latency due to
  - Network round trips to the database
  - Disk IO at the database



Prefetching Query Results

Advantages

- Multiple queries could be issued concurrently
  - Allows the database to share work across multiple queries
- Application performs other processing while query executes
- Significantly reduces the impact of latency

Challenges

- Hard to identify earliest and safe points in the code to perform prefetching
- Complex interprocedural code with queries deep inside
- Hard to manually maintain as code changes occur

Our Solution

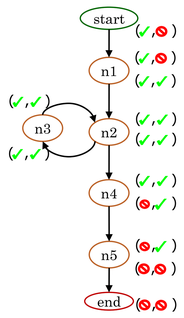
Prefetching based on Static analysis

- Inserts prefetches at **earliest possible point** in the program
- Works in the presence of loops and **interprocedural code**
- **No wasted prefetches except due to exceptions**
- Code motion, chaining and rewriting to optimize prefetches
- Applicable to JDBC, Hibernate, Web Services, and similar data access APIs
- Being implemented in the DBridge Holistic optimization tool http://www.cse.iitb.ac.in/infolab/dbridge

PREFETCHING WALKTHROUGH

1. Query Anticipability Analysis: Find Valid points of prefetch insertion

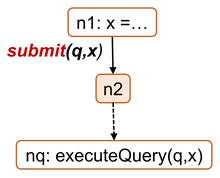
```
void genAllReports(){
  for (…) {
    …
    genReport(custId, city);
  }
}
void genReport(int cId, String city) {
  city = …
  while (…){
    …
  }
  rs1 = executeQuery(q1, cId);
  rs2= executeQuery(q2, city);
  …
}
```



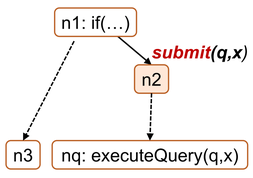
CFG of method genReport()

- Bit vector = (q1,q2)
- ✓ = anticipable (valid)
- ⃠ = not anticipable (invalid)

2. Identify earliest point and insert prefetch statement



Data Dependence Barrier



Control Dependence Barrier

- All query parameters should be available, with no intervening assignments
- No intervening updates to the database
- Should be guaranteed that the query will be executed subsequently

```
void genAllReports(){
  for (…) {
    …
    genReport(custId, city);
  }
}
void genReport(int cId, String city) {
  submit(q1, cId);
  city = …
  submit(q2, city);
  while (…){
    …
  }
  rs1 = executeQuery(q1, cId);
  rs2 = executeQuery(q2, city);
  …
}
```

3. Prefetch at the beginning of a method can be moved to all its callers

```
void genAllReports(){
  for (…) {
    submit(q1, cId);
    …
    genReport(custId, city);
  }
}
void genReport(int cId, String city) {
  city = …
  submit(q2, city);
  while (…){
    …
  }
  rs1 = executeQuery(q1, cId);
  rs2 = executeQuery(q2, city);
  …
}
```

- ***executeQuery* ():** normal execute query
- ***submit*():** non-blocking call that initiates query and returns immediately; once the results arrive, they are stored in a cache
- ***executeQuery*():** checks the cache and blocks if results are not yet available

✓ Equivalence with original program is preserved
✓ All existing statements remain unchanged
✓ Prefetch is not wasted

Prefetching Enhancements

Optimizing prefetches in presence of barriers
- Using program and query transformations
- Preserving program equivalence

1. Code Motion with Strong Anticipability

- Control dependence barrier:
  - Transform it into a data dependence barrier by rewriting it as a guarded statement
- Data dependence barrier:
  - Apply anticipability analysis on the barrier statements
  - Move the barrier to its earliest point followed by the prefetch

```
void genReport(int cId){
  int x = …;
  while (…){
    …
  }
  if (x > 10)
    rs1 = executeQuery(q1, cId);
}
```

```
void genReport(int cId){
  int x = …;
  boolean b = (x > 10);
  if (b) submit(q1, cId);
  while (…){
    …
  }
  if (b)
    rs1 = executeQuery(q1, cId);
}
```

2. Chaining and rewriting Prefetch requests

- Output of a query forms a parameter to another – commonly encountered
- Prefetch of query 2 can be issued soon after results of query 1 are available.

```
void report(int cId,String city){
  …
  c = executeQuery(q1, cId);
  while (c.next()){
    accId = c.getString("accId");
    d = executeQuery(q2, accId);
  }
}
```

```
void report(int cId,String city){
  submitChain({q1, q2'}, {{cId}, {}});
  …
  c = executeQuery(q1, cId);
  while (c.next()){
    accId = c.getString("accId");
    d = executeQuery(q2, accId);
  }
}
```

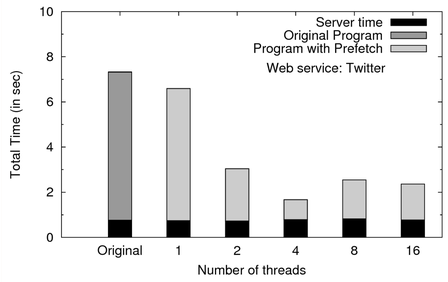
- Chained SQL queries can be rewritten into one query using known techniques
- Reduces network round trips, aids in selection of set oriented query plans

```
submitChain({"SELECT * FROM accounts WHERE custid=?",
      "SELECT * FROM transactions WHERE accId=:q1.accId"}, {{cId}, {}});
```

```
SELECT *
FROM (SELECT * FROM accounts WHERE custId = ?)
     OUTER APPLY
     (SELECT * FROM transactions
      WHERE transactions.accId = account.accId)
```

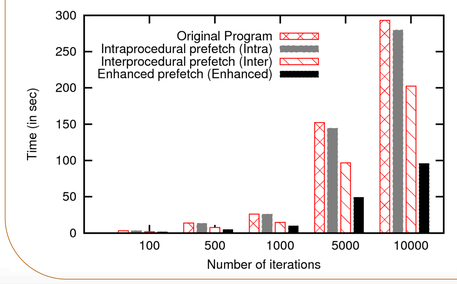
Experiments (upto 75% improvement)

1. Twitter Dashboard



- Web Service: HTTP/JSON with Twitter4j client
- Monitors 4 keywords for new tweets
- Interprocedural prefetching; no rewrite possible
- **75%** improvement at 4 threads
- Server time constant; network overlap leads to significant gain

2. ERP Application: Impact of our techniques



- Java/JDBC application
- Intraprocedural: moderate gains
- Interprocedural: substantial gains (25-30%)
- Enhanced (with rewrite): significant gain (**50%** over Interprocedural)

Contact: karthiksr@cse.iitb.ac.in

ACM SIGMOD 2012, Scottsdale, AZ

This work supported by: Microsoft Research India PhD Fellowship and Yahoo! Key Scientific Challenges Award 2011