PROGRAM TRANSFORMATION FOR ASYNCHRONOUS QUERY SUBMISSION

(ICDE 2011)

Mahendra Chavan*, Ravindra Guravannavar,
Karthik Ramachandra, S Sudarshan

Indian Institute of Technology Bombay,
Indian Institute of Technology Hyderabad

*Current Affiliation: Sybase Inc.

THE PROBLEM

THE PROBLEM

- Applications often invoke Database queries/Web Service requests
  - repeatedly (with different parameters)
  - synchronously (blocking on every request)
- At the Database end:
  - Naive iterative execution of such queries is **inefficient**
    - No sharing of work (eg. Disk IO)
    - Network round-trip delays

The problem is **not** within the database engine!

The problem is **the way queries are invoked** from the application!!

SOLUTION 1: USE A BUS!

(OUR) EARLIER WORK: BATCHING

Rewriting Procedures for Batched Bindings
Guravannavar et. al. VLDB 2008

- Repeated invocation of a query **automatically** replaced by a single invocation of its batched form.
- Enables use of efficient set-oriented query execution plans
- Sharing of work (eg. Disk IO) etc.
- Avoids network round-trip delays

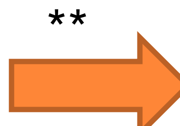
Approach

- Transform imperative programs using equivalence rules
- Rewrite queries using decorrelation, APPLY operator etc.

PROGRAM TRANSFORMATION FOR BATCHED BINDINGS (VLDB08 PAPER)

```
qt = con.prepare(
        "SELECT count(partkey) " +
         "FROM part " +
         "WHERE p_category=?");

while(!categoryList.isEmpty()) {
        category = categoryList.next();
        qt.bind(1, category);
        count = qt.executeQuery();
        sum += count;
}
```




```
qt = con.Prepare(
            "SELECT count(partkey) " +
            "FROM part " +
            "WHERE p_category=?");
while(!categoryList.isEmpty()) {
            category = categoryList.next();
            qt.bind(1, category);
            qt.addBatch();
}
```

```
qt.executeBatch();
```

```
while(qt.hasMoreResults()) {
            count = qt.getNextResult();
            sum += count;
}
```

** Conditions apply. See Guravannavar and Sudarshan, VLDB 2008

LIMITATIONS OF EARLIER WORK ON BATCHING

- Limitations (Opportunities?)
  - Some data sources e.g. Web Services may not provide a set oriented interface
  - Arbitrary inter-statement data dependencies may severely limit applicability of transformation rules
  - Multicore processing power on the client can be exploited better by using multiple threads of execution
- Our Approach
  - Exploit asynchronous query execution, through
    - New API
    - Automatic Program rewriting
  - Improved set of transformation rules
  - Increase applicability by reordering

ASYNCHRONOUS EXECUTION: MORE TAXIS!!

MOTIVATION

Fact 1: Performance of applications can be significantly improved by asynchronous submission of queries.

- Multiple queries could be issued concurrently
- Application can perform other processing while query is executing
- Allows the query execution engine to share work across multiple queries
- Reduces the impact of network round-trip latency

Fact 2: Manually writing applications to exploit asynchronous query submission is **HARD!!**

OUR CONTRIBUTIONS IN THIS PAPER

1. Automatically transform a program to exploit Asynchronous Query Submission
2. A novel Statement Reordering Algorithm that greatly increases the applicability of our transformations
3. An API that wraps any JDBC driver and performs these optimizations (DBridge)
4. System design challenges and a detailed experimental study on real world applications

AUTOMATIC PROGRAM TRANSFORMATION FOR ASYNCHRONOUS SUBMISSION

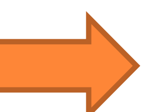
INCREASING THE APPLICABILITY OF TRANSFORMATIONS

SYSTEM DESIGN AND EXPERIMENTAL EVALUATION

PROGRAM TRANSFORMATION EXAMPLE

```
qt = con.prepare(
      "SELECT count(partkey) " +
       "FROM part " +
       "WHERE p_category=?");

while(!categoryList.isEmpty()) {
      category = categoryList.next();
      qt.bind(1, category);
      count = executeQuery(qt);
      sum += count;
}
```

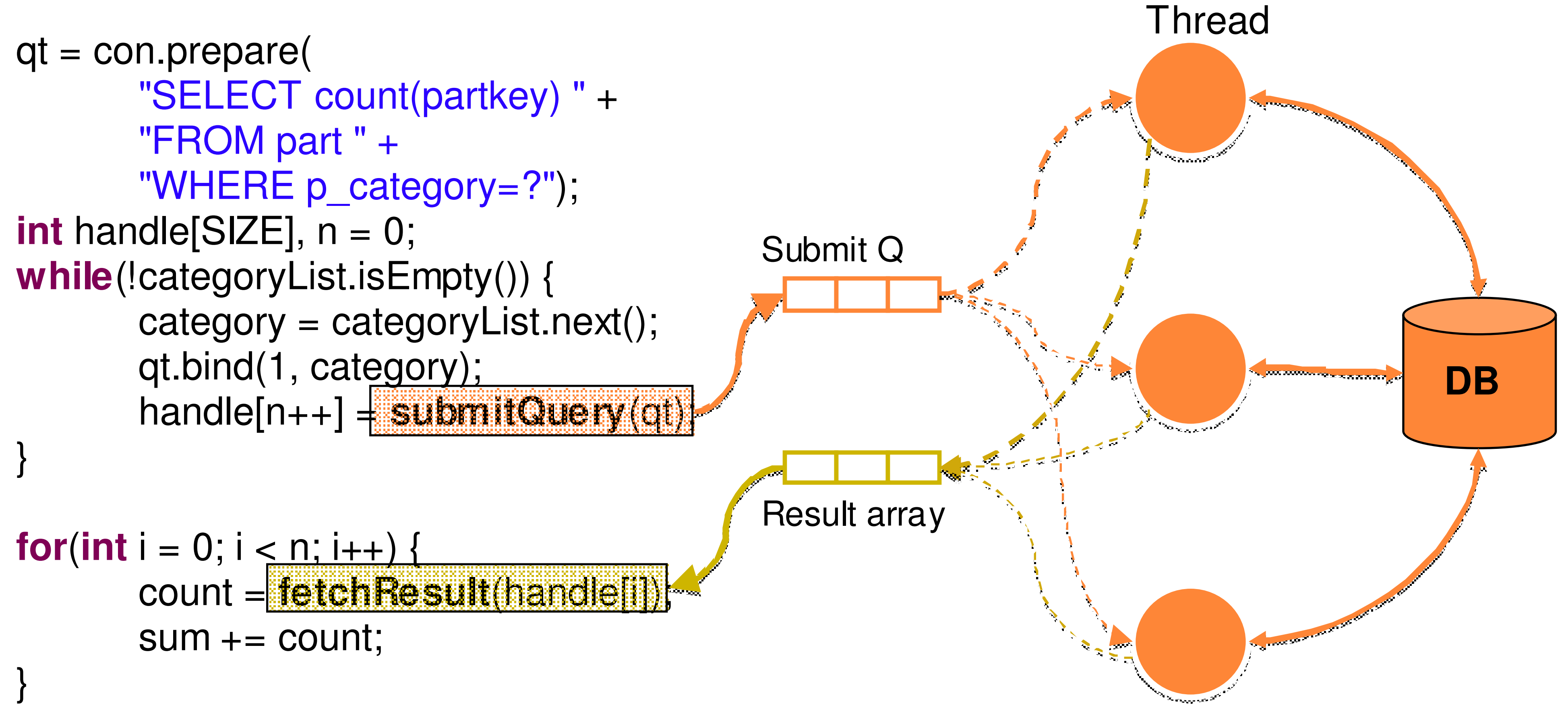


```
qt = con.Prepare(
            "SELECT count(partkey) " +
            "FROM part " +
            "WHERE p_category=?");
int handle[SIZE], n = 0;
while(!categoryList.isEmpty()) {
            category = categoryList.next();
            qt.bind(1, category);
            handle[n++] = submitQuery(qt);
}

for(int i = 0; i < n; i++) {
            count = fetchResult(handle[i]);
            sum += count;
}
```

- Conceptual API for asynchronous execution
  - executeQuery() – blocking call
  - submitQuery() – initiates query and returns immediately
  - fetchResult() – blocking wait

ASYNCHRONOUS QUERY SUBMISSION MODEL

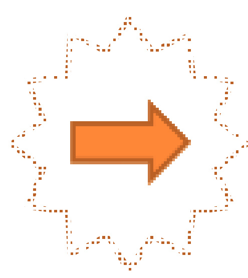


- submitQuery() – returns immediately
- fetchResult() – blocking call

PROGRAM TRANSFORMATION

- Possible to rewrite manually, but tedious.
- Challenge:
  - Complex programs with arbitrary control flow
  - Arbitrary inter-statement data dependencies
  - Loop splitting requires variable values to be stored and restored
- **Our contribution 1: Automatically rewrite to enable asynchrony.**

```
while(!categoryList.isEmpty()) {
        category = categoryList.next();
        qt.bind(1, category);
        count = executeQuery(qt);
        sum += count;
}
```



```
int handle[SIZE], n = 0;
while(!categoryList.isEmpty()) {
        category = categoryList.next();
        qt.bind(1, category);
        handle[n++] = submitQuery(qt);
}
for(int i = 0; i < n; i++) {
        count = fetchResult(handle[i]);
        sum += count;
}
```

PROGRAM TRANSFORMATION RULES

- Rule A: Equivalence rule for Loop fission
  - Minimal pre-conditions
  - Simplified handling of nested loops
- Rule B: Converting control dependencies to flow dependencies
  - Enables handling conditional branching(if-then-else) structures
- Rule C1, C2, C3: Rules to facilitate reordering of statements
  - Used by our statement reordering algorithm (coming up next)

- All the above simplify and generalize the transformation rules of our VLDB08 paper
- For details, refer to our paper

AUTOMATIC PROGRAM TRANSFORMATION FOR ASYNCHRONOUS SUBMISSION

INCREASING THE APPLICABILITY OF TRANSFORMATIONS

SYSTEM DESIGN AND EXPERIMENTAL EVALUATION

APPLICABILITY OF TRANSFORMATIONS

- Pre-conditions due to inter statement dependencies restrict applicability
- **Our Contribution 2: A Statement Reordering algorithm that**
  - **Removes dependencies that prevent transformation**
  - **Enables loop fission at the boundaries of the query execution statement**

```
while (category != null) {
    qt.bind(1, category);
    int count = executeQuery(qt);
    sum = sum + count;
    category = getParent(category);
}
```

Loop fission not possible due to dependency (→)

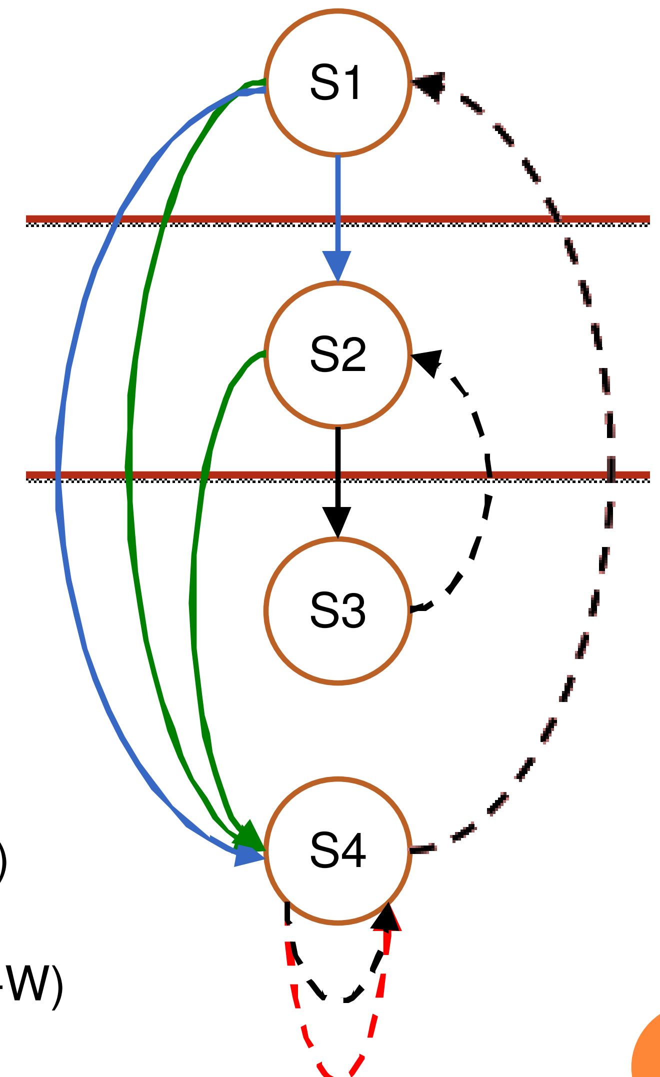
```
while (category != null) {
    int temp = category;
    category = getParent(category);
    qt.bind(1, temp);
    int count = executeQuery(qt);
    sum = sum + count;
}
```

Loop fission enabled by safe reordering

BEFORE

```
S1: while (category != null) {
S2:     qt.bind(1, category);
        int count = executeQuery(qt);
S3:     sum = sum + count;
S4:     category = getParent(category);
    }
```

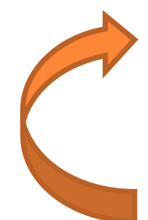
Data Dependence Graph (DDG)



- Flow Dependence (W-R)
- Anti Dependence (R-W)
- Output Dependence (W-W)
- Control Dependence
- Loop-Carried

BEFORE

S1:	`while (category != null) {`
S2:	`qt.bind(1, category);` `int count = executeQuery(qt);`
S3:	`sum = sum + count;`
S4:	`category = getParent(category);` `}`



Intuition: Move S4 before S2

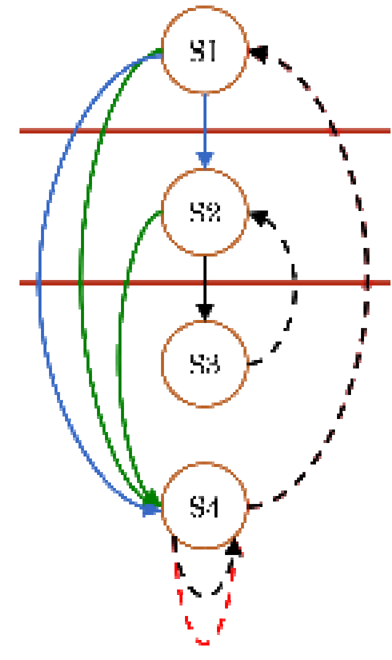
AFTER

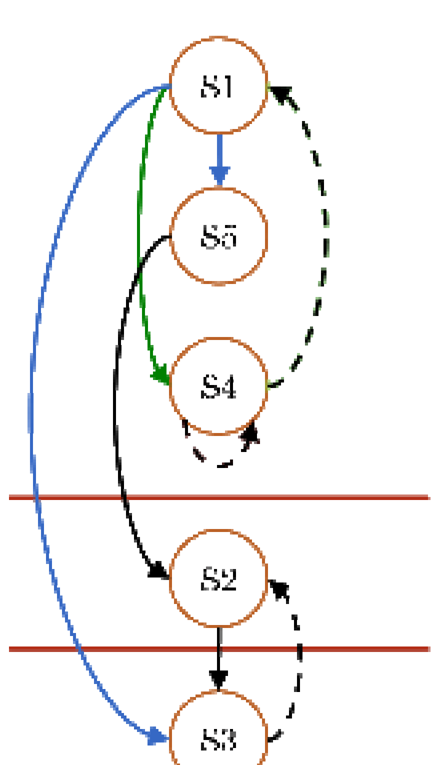
S1:	`while (category != null) {`
S5:	`int temp = category;`
S4:	`category = getParent(category);`
S2:	`qt.bind(1, temp);` `int count = executeQuery(qt);`
S3:	`sum = sum + count;` `}`

BEFORE

S1:	`while (category != null) {`
S2:	`qt.bind(1, category);` `int count = executeQuery(qt);`

S3:
S4:
	`sum = sum + count;`
	`category = getParent(category);`
	`}`





S1:
S5:
S4:
	`while (category != null) {`
	`int temp = category;`
	`category = getParent(category);`

S2:	`qt.bind(1, temp);` `int count = executeQuery(qt);`
S3:	`sum = sum + count;` `}`

AFTER

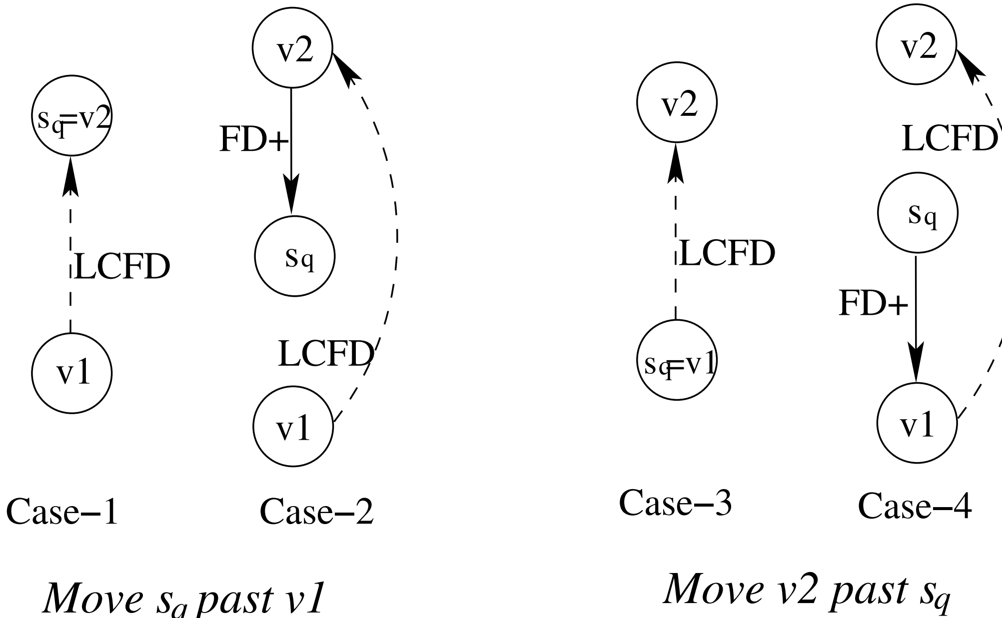
THE STATEMENT REORDERING ALGORITHM

- **Goal:** Reorder statements such that no loop-carried flow dependencies cross the desired split boundary.
- **Input:**
  - The blocking query execution statement Sq
  - The basic block b representing the loop
- **Output:** Where possible, a reordering of b such that:
  - No LCFD edges cross the split boundary Sq
  - Program equivalence is preserved

THE STATEMENT REORDERING ALGORITHM*

For every loop carried dependency that crosses the query execution statement

- Step 1: Identify which statement to move(***stm***) past which one (***target***)



Move s_q past v1　　*Move v2 past s_q*

- Step 2: Compute statements dependent on the ***stm*** (***stmdeps***)
- Step 3: Move each of ***stmdeps*** past ***target***
- Step 4: Move ***stm*** past ***target***

*HEAVILY SIMPLIFIED; REFER TO PAPER FOR DETAILS

THE STATEMENT REORDERING ALGORITHM

Definition: A True-dependence cycle in a DDG is a directed cycle made up of only FD and LFD edges.

THEOREM:

If a query execution statement doesn't lie on a **true-dependence cycle** in the DDG, then algorithm reorder always reorders the statements such that the loop can be split.

- Proof in [Guravannavar 09]
- Theorem and Algorithm applicable for both Batching and Asynchronous submission transformations

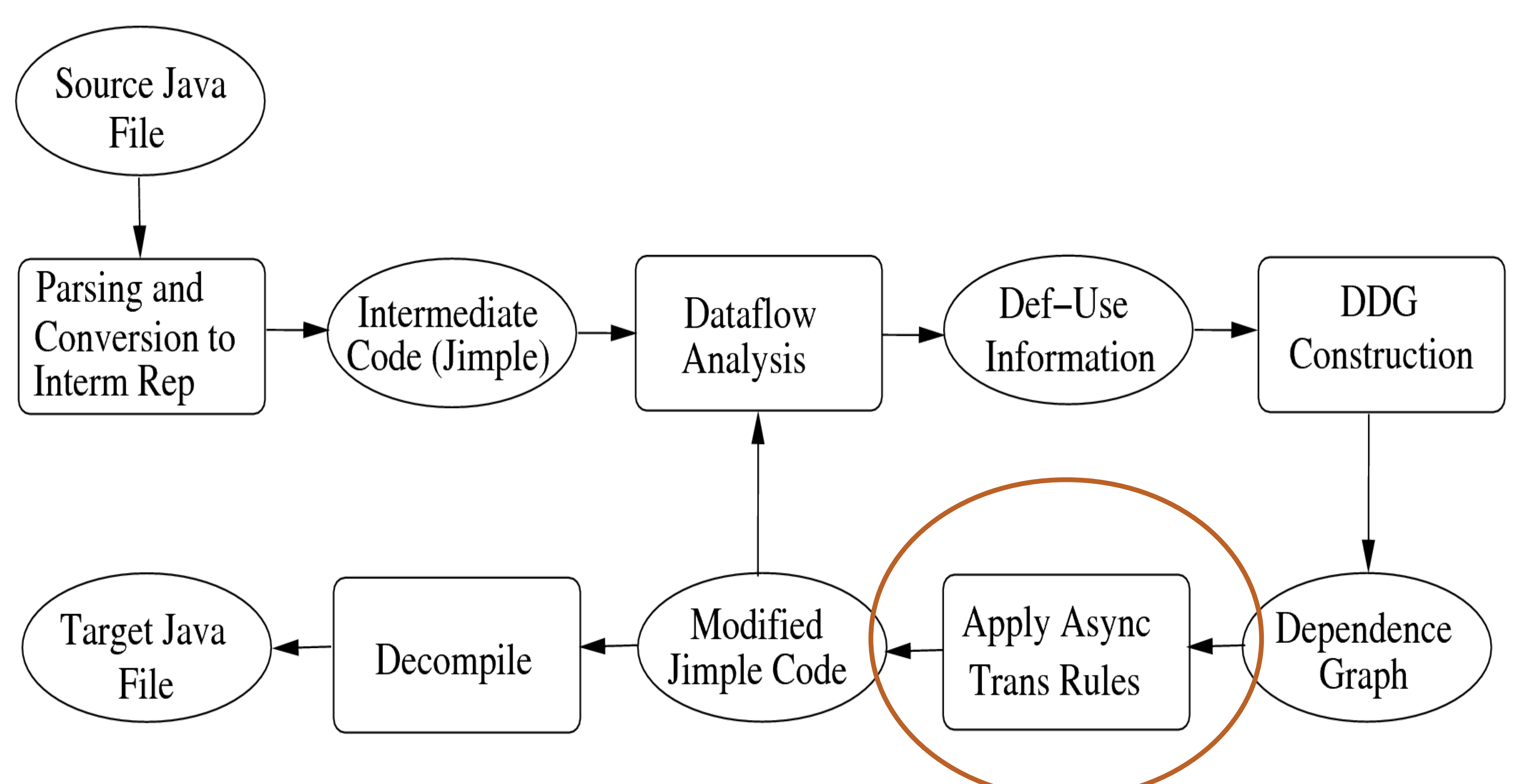
AUTOMATIC PROGRAM TRANSFORMATION FOR ASYNCHRONOUS SUBMISSION

INCREASING THE APPLICABILITY OF TRANSFORMATIONS

SYSTEM DESIGN AND EXPERIMENTAL EVALUATION

SYSTEM DESIGN: DBRIDGE

- For Java applications using JDBC
- SOOT framework for analysis and transformation



DBRIDGE API

- Java API that extends the JDBC interface, and can wrap any JDBC driver
- Can be used with:
  - Manual rewriting (LoopContext structure helps deal with loop local variables)
  - Automat ic rewriting
- Hides details of thread scheduling and management
- Same API for both batching and asynchronous submission

DBridge: A Program Rewrite tool for Set-oriented Query Execution

Demonstrations Track 1, ICDE 2011

EXPERIMENTS

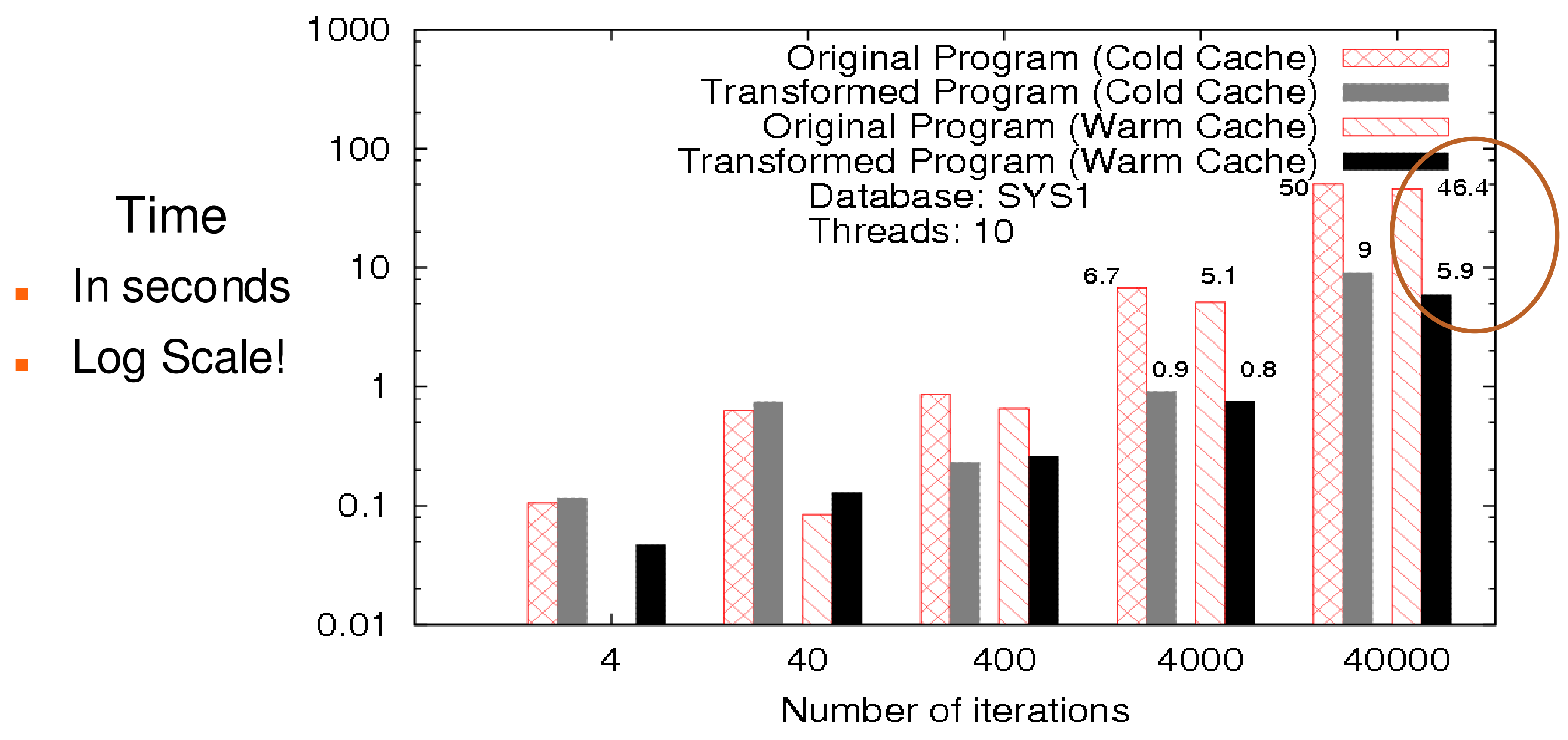
- Conducted on 5 applications
  - Two public benchmark applications (Java/JDBC)
  - Two real world applications (Java/JDBC)
  - Freebase web service client (Java/JSON)

- Environments
  - A widely used commercial database system – SYS1
    - 64 bit dual-core machine with 4 GB of RAM
  - PostgreSQL
    - Dual Xeon 3 GHz processors and 4 GB of RAM

EXPERIMENT SCENARIOS

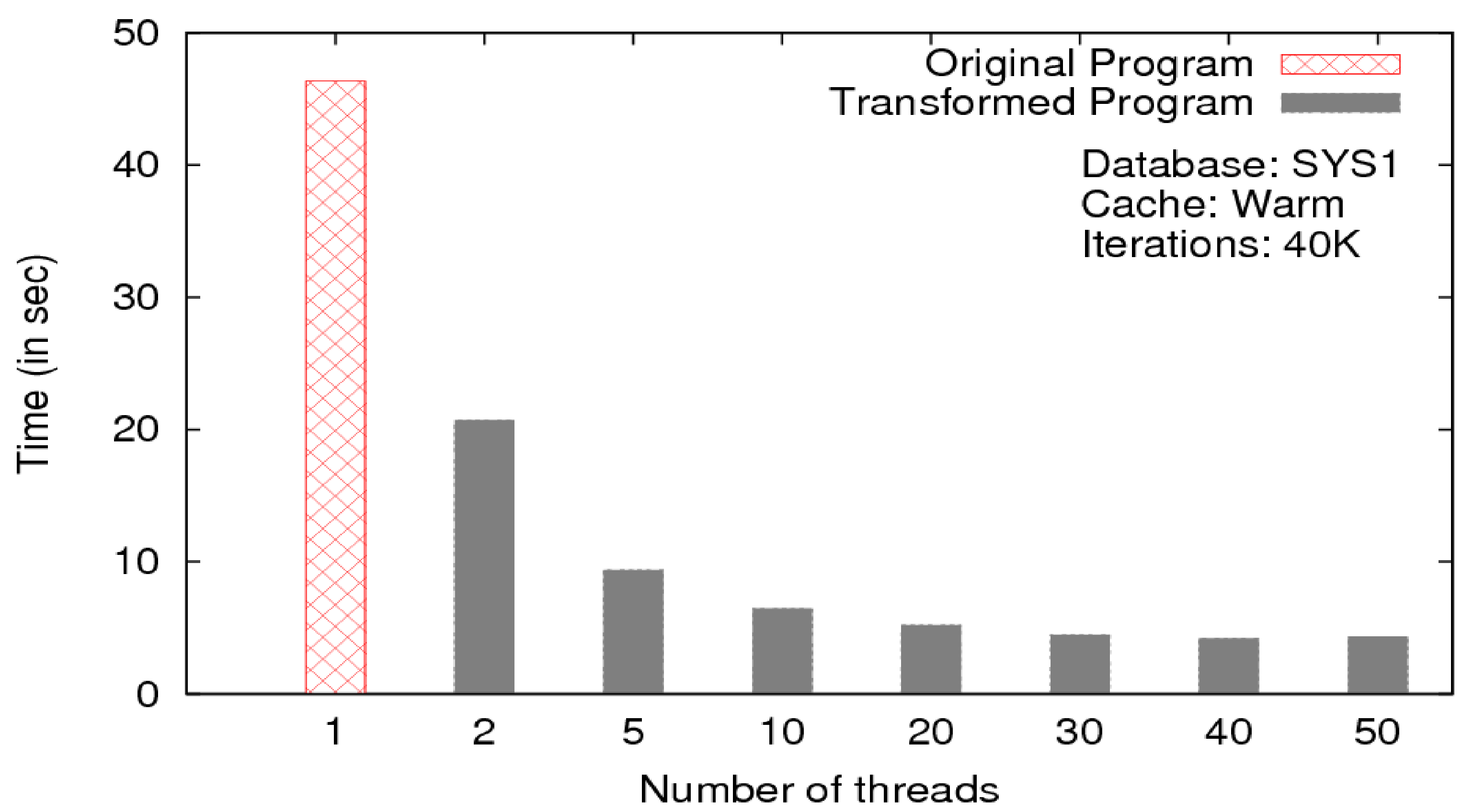
- Impact of iteration count
- Impact of number of threads
- Impact of Warm cache vs. Cold cache
  - Since Disk IO on the database is an important parameter

AUCTION APPLICATION: IMPACT OF ITERATION COUNT, WITH 10 THREADS



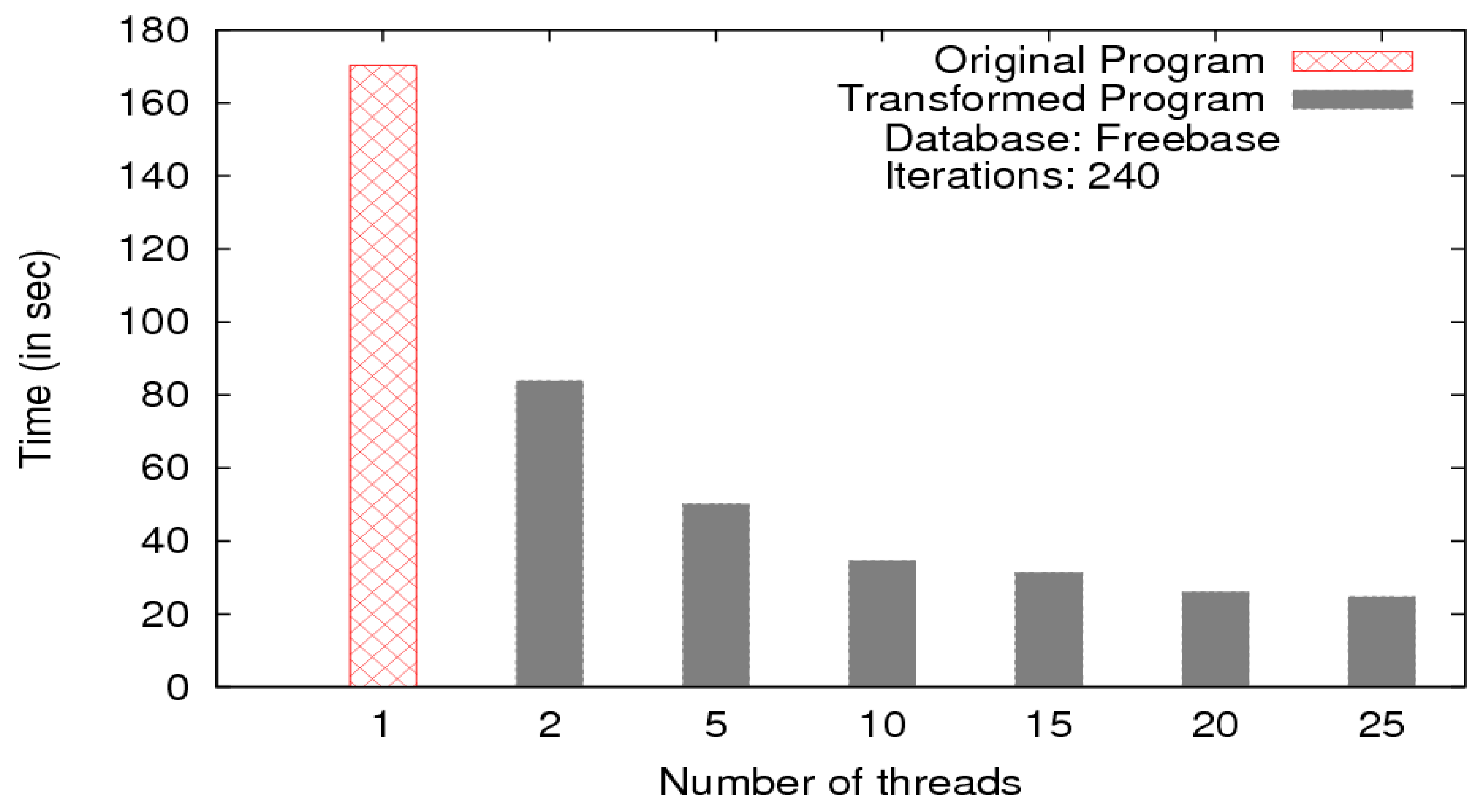
- For small no. (4-40) iterations, transformed program slower
- At 400-40000 iterations, **factor of 4-8 improvement**
- Similar for warm and cold cache

AUCTION APPLICATION: IMPACT OF THREAD COUNT, WITH 40K ITERATIONS



- Time taken reduces drastically as thread count increases
- No improvement after some point (30 in this example)

WEBSERVICE: IMPACT OF THREAD COUNT



- HTTP requests with JSON content
- Impact similar to earlier SQL example
- Note: Our system does not automatically rewrite web service programs, this example manually rewritten using our transformation rules

FUTURE DIRECTIONS?

- Which calls to be transformed?
- Minimizing memory overheads
- How many threads to use?
- Updates and transactions

ACKNOWLEDGEMENTS

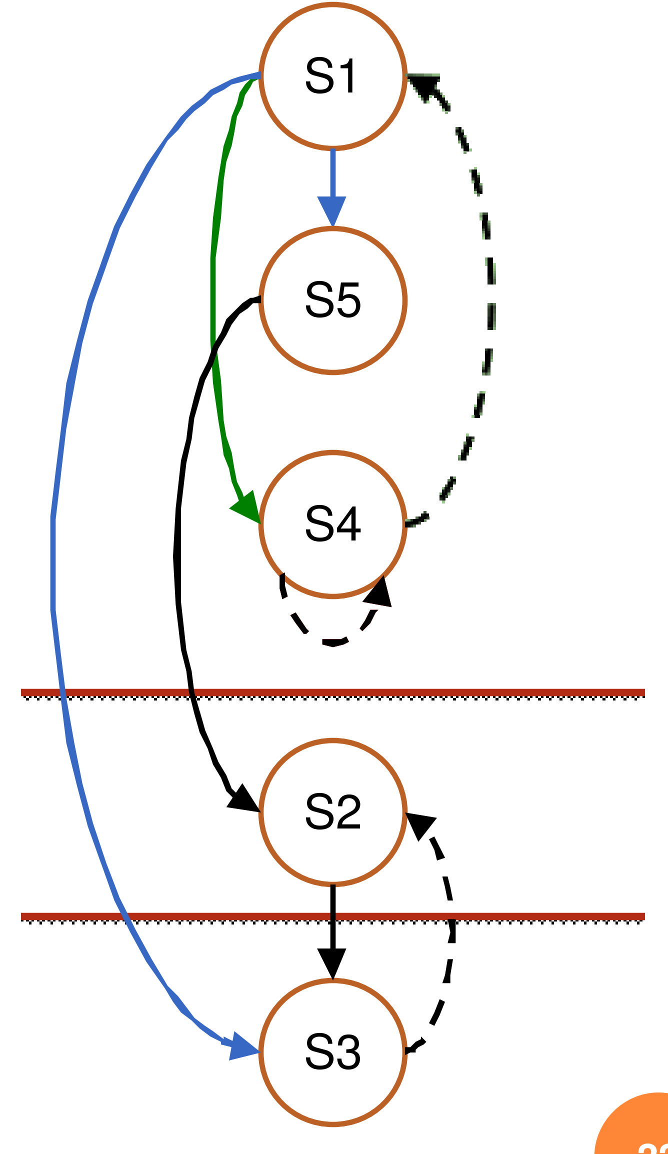
- Prabhas Samanta (IIT Bombay) – DBridge API implementation

AFTER

```
S1: while (category != null) {
S5:     int temp = category;
S4:     category = getParent(category);
        qt.bind(1, temp);
S2:     int count = executeQuery(qt);
S3:     sum = sum + count;
    }
```

Intuition: S2 moved past S4

Data Dependence Graph (DDG)



- Flow Dependence (W-R)
- Anti Dependence (R-W)
- Output Dependence (W-W)
- Control Dependence
- Loop-Carried