Program Transformation for Asynchronous Query Submission

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

Indian Institute of Technology Bombay, Indian Institute of Technology Hyderabad

***Current Affiliation: Sybase Inc.**

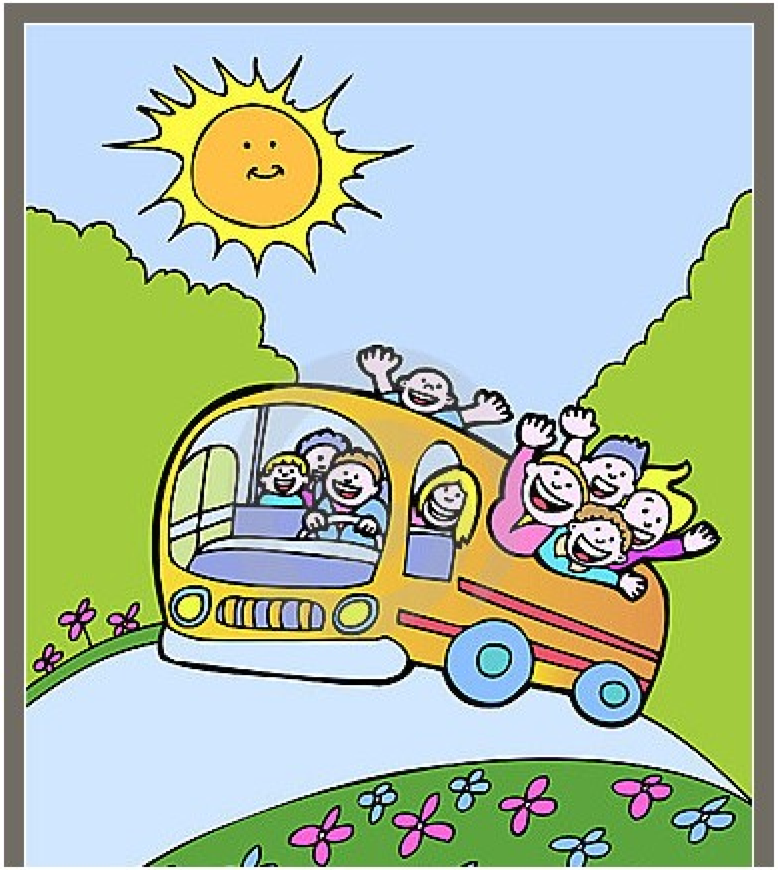
The Problem

And what if there is only one taxi?

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

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 a stored proc to accept a batch of bindings instead of a single binding.

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;
}
```

**

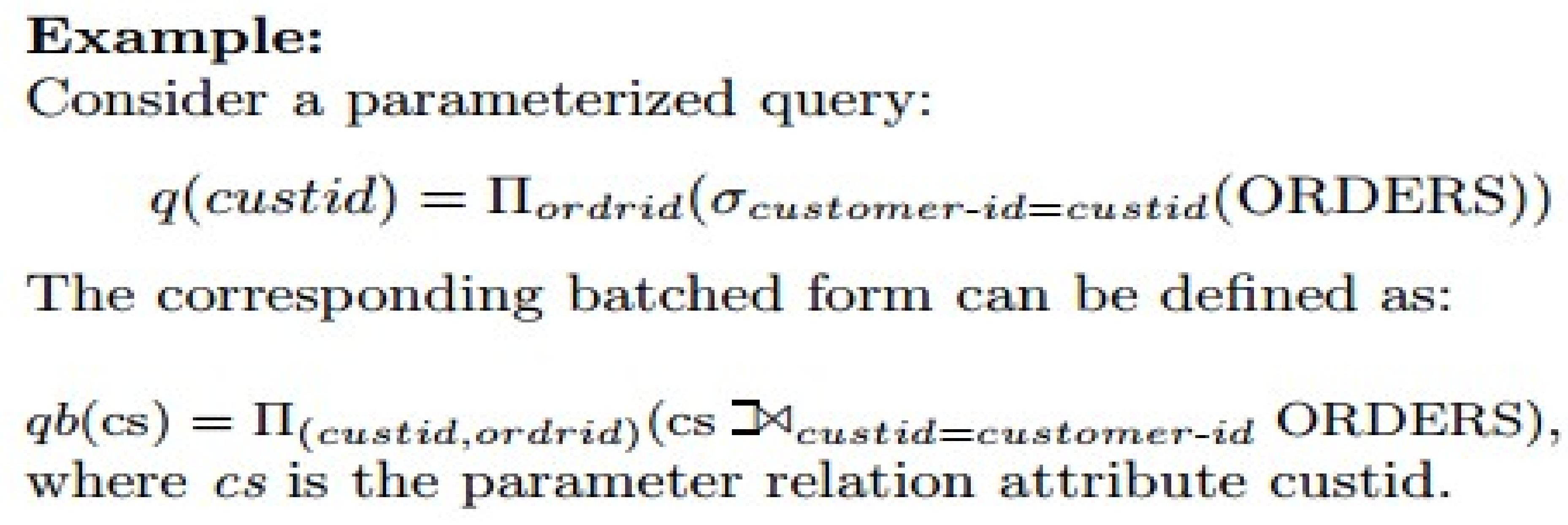
```
qt = con.Prepare(
    "SELECT count(partkey) " +
    "FROM part " +
    "WHERE p_category=?");
while(!
categoryList.isEmpty()) {
    category =
   categoryList.next();
    qt.bind(1, category);
    qt.addBatch();
}

qt.executeBatch();

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

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

Batched Forms of parameterized relational Queries

$$qb(p) = \bigcup_{p_t \in p} \{\{p_t\} \times q(p_t)\}$$

where ***q****(p1,p2,..pn) be a query with n parameters and* ***qb*** *as its batched form*



Example:

Consider a parameterized query:

$$q(custid) = \Pi_{ordrid}(\sigma_{customer\text{-}id=custid}(\mathrm{ORDERS}))$$

The corresponding batched form can be defined as:

$qb(\mathrm{cs}) = \Pi_{(custid,ordrid)}(\mathrm{cs} ⟕_{custid=customer\text{-}id} \mathrm{ORDERS})$,
where cs is the parameter relation attribute custid.

Batch Safe Operations

- Batched forms – no guaranteed order of parameter processing
- Can be a problem for operations having side-effects

Batch-Safe operations

- All operations that have no side effects
- Also a few operations with side effects
  - E.g.: INSERT on a table with no constraints
  - Operations inside unordered loops (e.g., cursor loops with no order-by)

Rule1: Rewriting a Simple Set Iteration Loop

1A(i). Basic Form

```
for each $t$ in $r$ loop
    $q(t.c_1, t.c_2, \ldots t.c_m);$        $\Longleftrightarrow$        $qb(\Pi^d_{c_1, c_2, \ldots c_m}(r));$
end loop;
```

*where **q** is any batch-safe operation with **qb** as its batched form*

1A(ii). Form with loop invariant parameters

```
for each $t$ in $r$ loop
    $q(t.c_1, t.c_2, \ldots t.c_m, v_1, v_2, \ldots v_n);$
end loop;
            $\Updownarrow$
$qb(\Pi^d_{c_1, c_2, \ldots c_m}(r) \times \{(v_1, v_2, \ldots v_n)\});$
```

Rule 1: Rewriting a Simple Set Iteration Loop

1B. Unconditional invocation with return value

```
for each $t$ by ref in $r$ loop
    $t.c_{w1}, t.c_{w2}, \ldots t.c_{wn} = q(t.c_{r1}, t.c_{r2}, \ldots t.c_{rm});$
end loop;
where $q$ is a pure function.
```

$$\Updownarrow$$

$$\mathcal{M}_{c_{w1}=c_{w1'} \ldots c_{wn}=c_{wn'}}(r, e)$$

where $e = \rho_x(c_{r1}, \ldots c_{rm}, c_{w1'}, \ldots c_{wn'})qb(\Pi_{c_{r1},\ldots c_{rm}}(r));$

1C. Conditional Invocation

```
for each $t$ by ref in $r$ loop
    $(t.cv == true)?\ t.c_{w1}, \ldots t.c_{wn} = q(t.c_{r1}, \ldots t.c_{rm});$
end loop;
where $q$ is a pure function.
```

$$\Updownarrow$$

$\mathcal{M}_{c_{w1}=c_{w1'} \ldots c_{wn}=c_{wn'}}(r, e)$, where

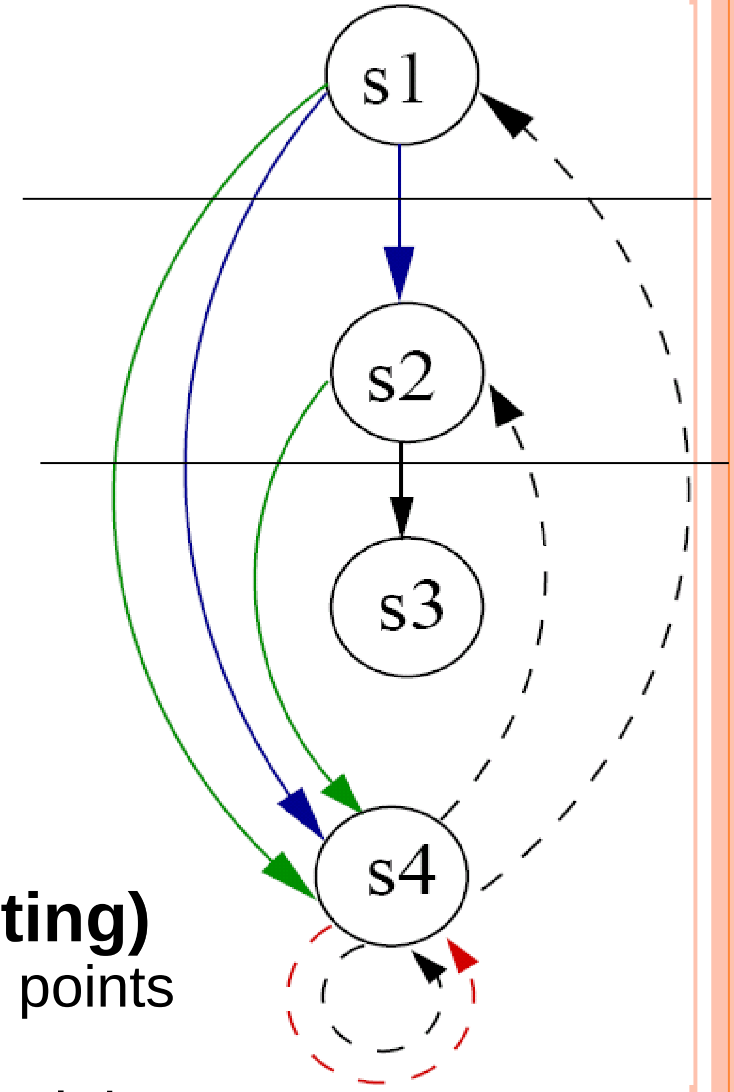
$e = \rho_x(c_{r1}, \ldots c_{rm}, c_{w1'}, \ldots c_{wn'})qb(\Pi_{c_{r1},\ldots c_{rm}}(\sigma_{cv=true}\ r));$

Data Dependency Graph

```
(S1)while (category != null) {
(s2)    item-count =q1(category);
(s3)    sum = sum + item-count;
(s4)   category =getParent(category);
      }
```

W->R	→	Flow Dependence
R->W	🟩 →	Anti Dependence
W->W	🟥 →	Output Dependence
	🟦 →	Control Dependence
	⇢	Loop-Carried

Data Dependencies



- **Pre-conditions for Rule-2 (Loop splitting)**
  - No loop-carried flow dependencies cross the points at which the loop is split
  - No loop-carried dependencies through external data (e.g., DB)

Rule 2: Splitting a Loop

```
while (p) {
    ss1;
    $s_q$;
    ss2;
}
```

```
Table(T) t;
while(p) {
    ss1 modified to save local variables as a tuple in t
}
```

Collect the parameters

```
for each r in t {
    $s_q$ modified to use attributes of r;
}
```

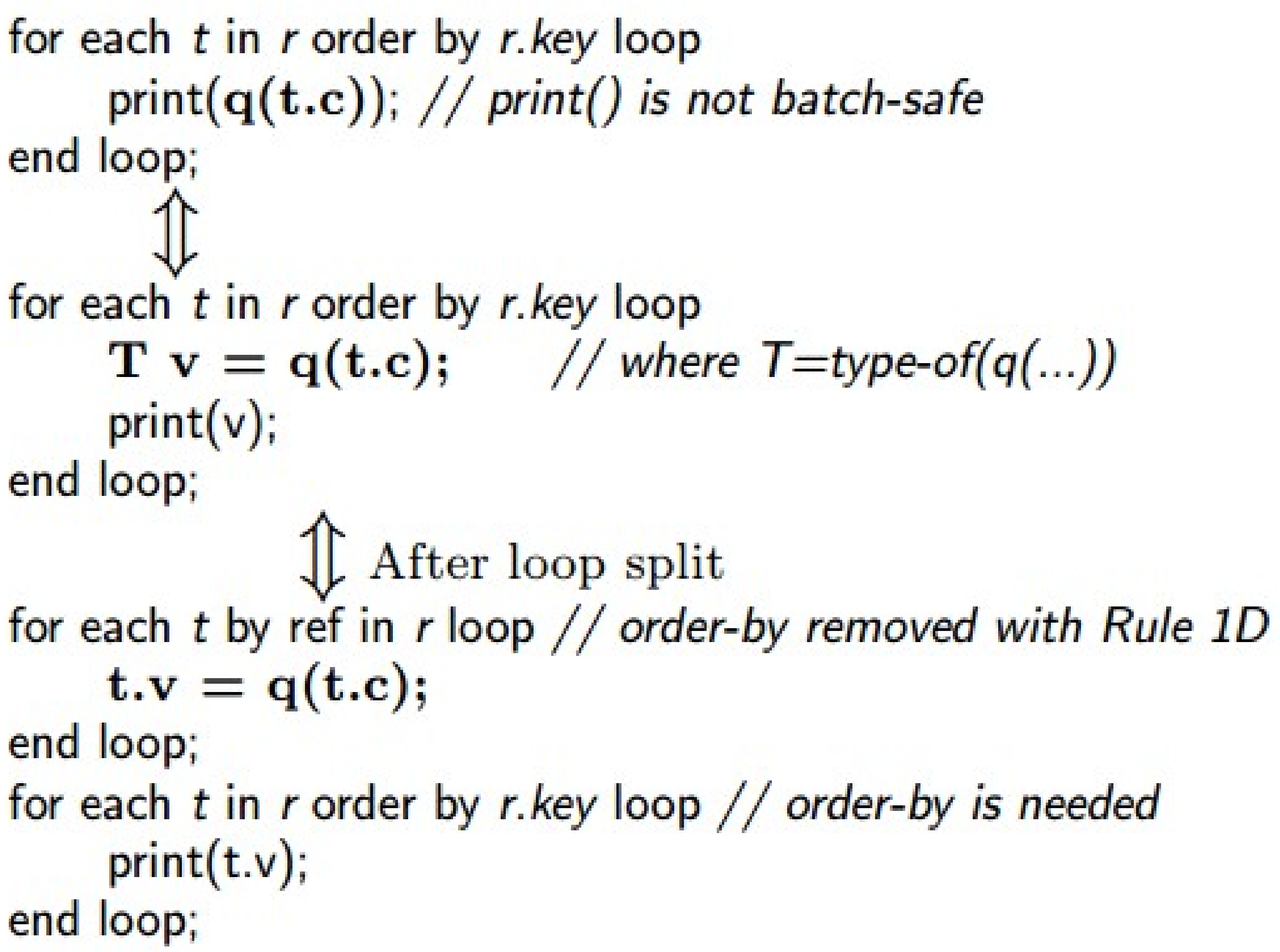
Can apply Rule 1A-1C and batch.

```
for each r in t {
    ss2 modified to use attributes of r;
}
```

Process the results

* Conditions Apply

Rule 3: Isolating batch safe operation



```
for each t in r order by r.key loop
    print(q(t.c)); // print() is not batch-safe
end loop;
       ⇕
for each t in r order by r.key loop
    T v = q(t.c);      // where T=type-of(q(...))
    print(v);
end loop;
              ⇕ After loop split
for each t by ref in r loop // order-by removed with Rule 1D
    t.v = q(t.c);
end loop;
for each t in r order by r.key loop // order-by is needed
    print(t.v);
end loop;
```

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

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

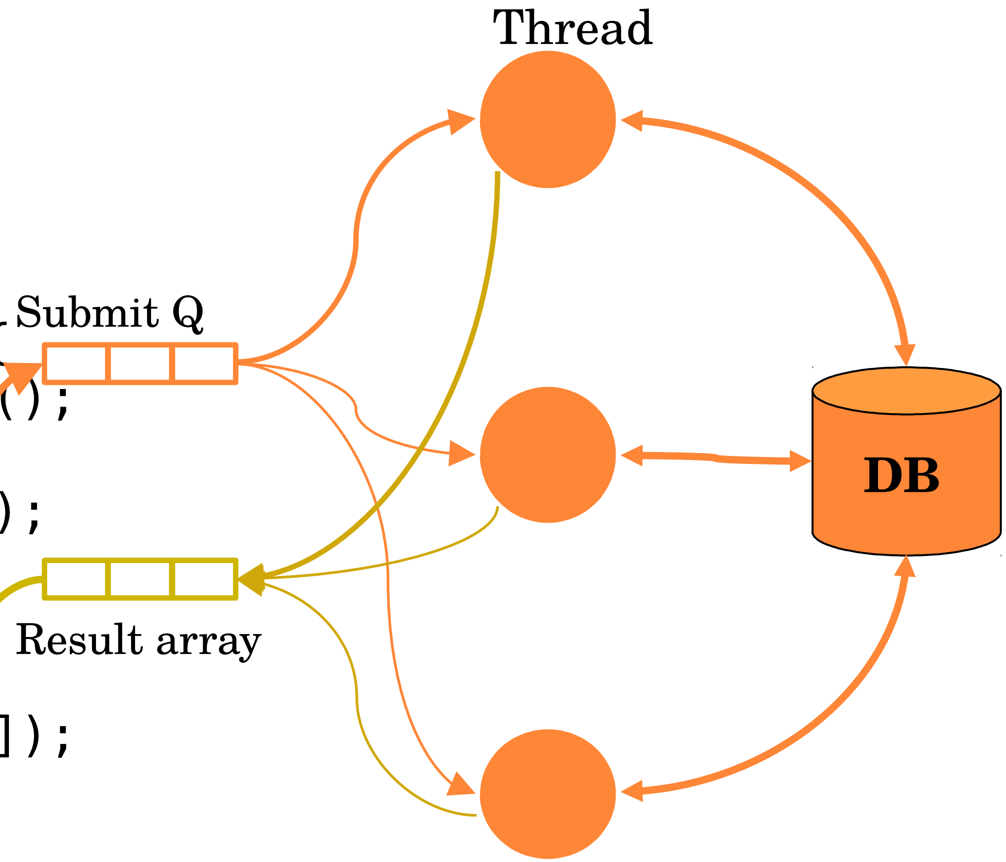
```
qt = con.prepare(
    "SELECT count(partkey)
     " + "FROM part " +
     "WHERE
     p_category=?");

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

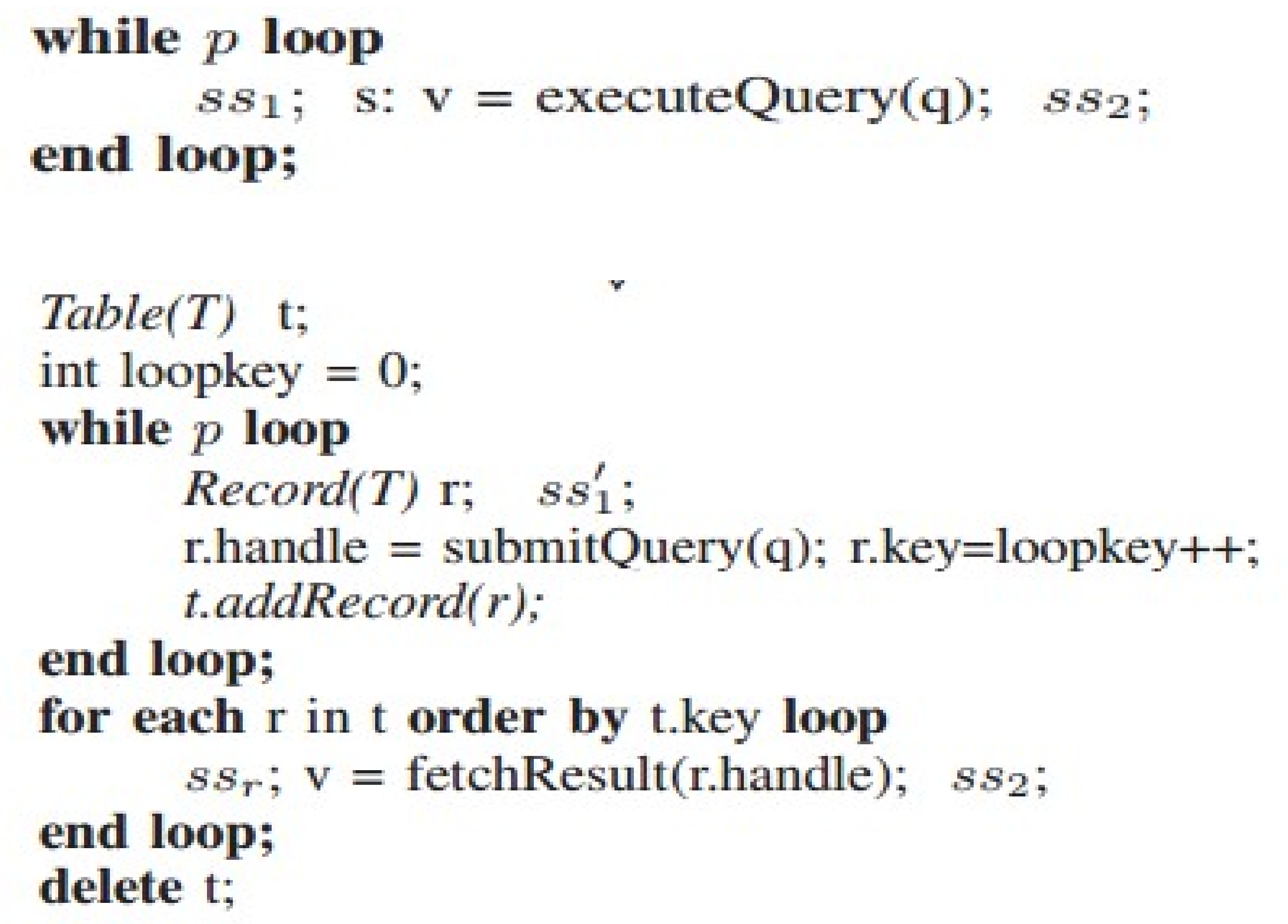
Asynchronous query submission model

```
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;
}
```



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

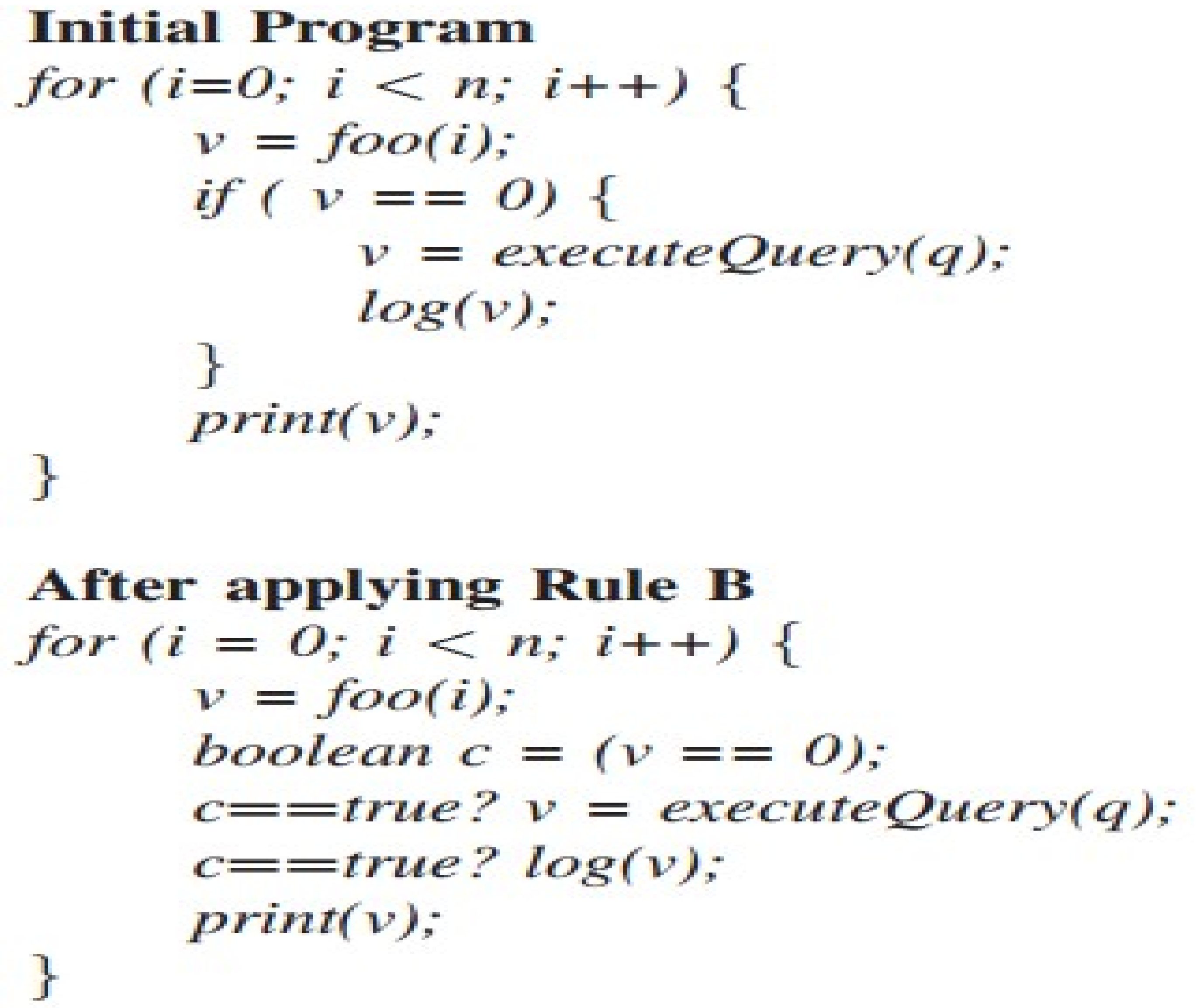
Rule A : Basic Equivalence Rule for Loop Fission



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

```
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;
```

Transforming Control-Dependencies to Flow Dependencies



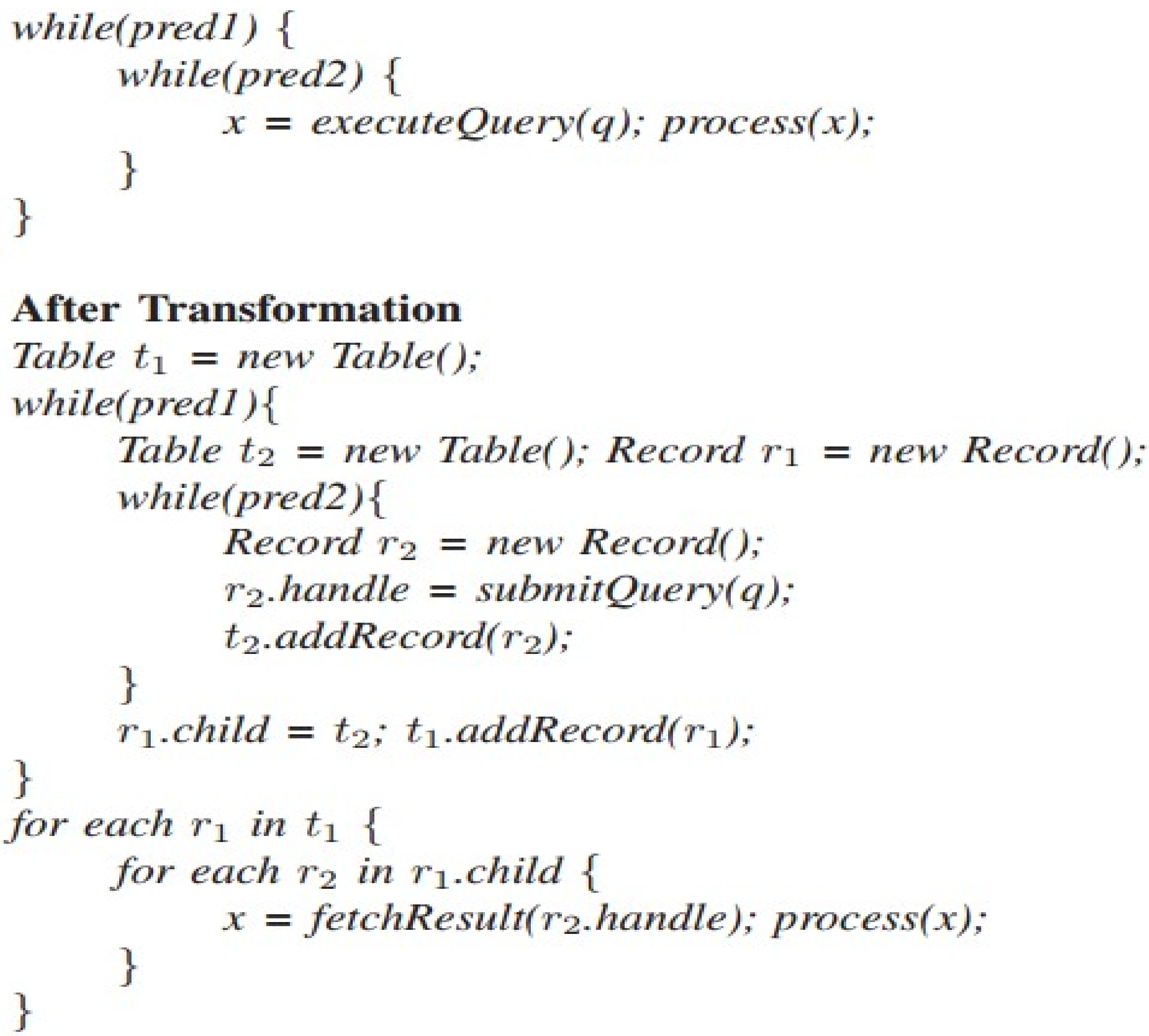
Initial Program

```
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);
}
```

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);
    }
}
```

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
- **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;
}
```

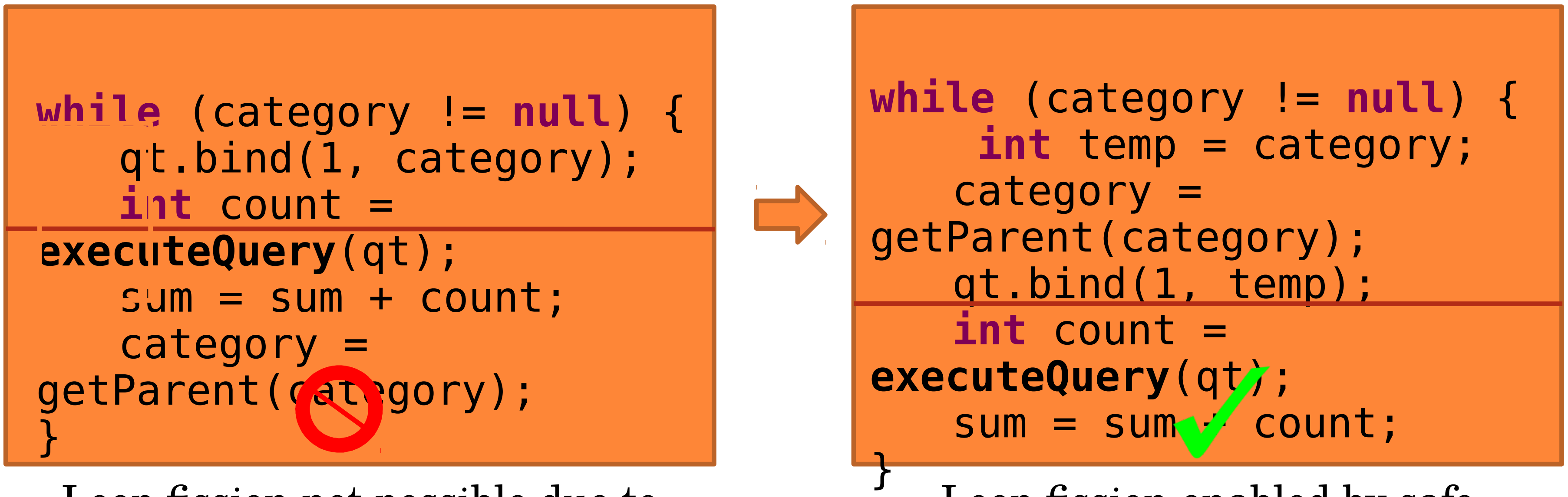
Automatic Program Transformation for asynchronous submission

Increasing the applicability of transformations

System design and experimental evaluation

Applicability of transformations

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



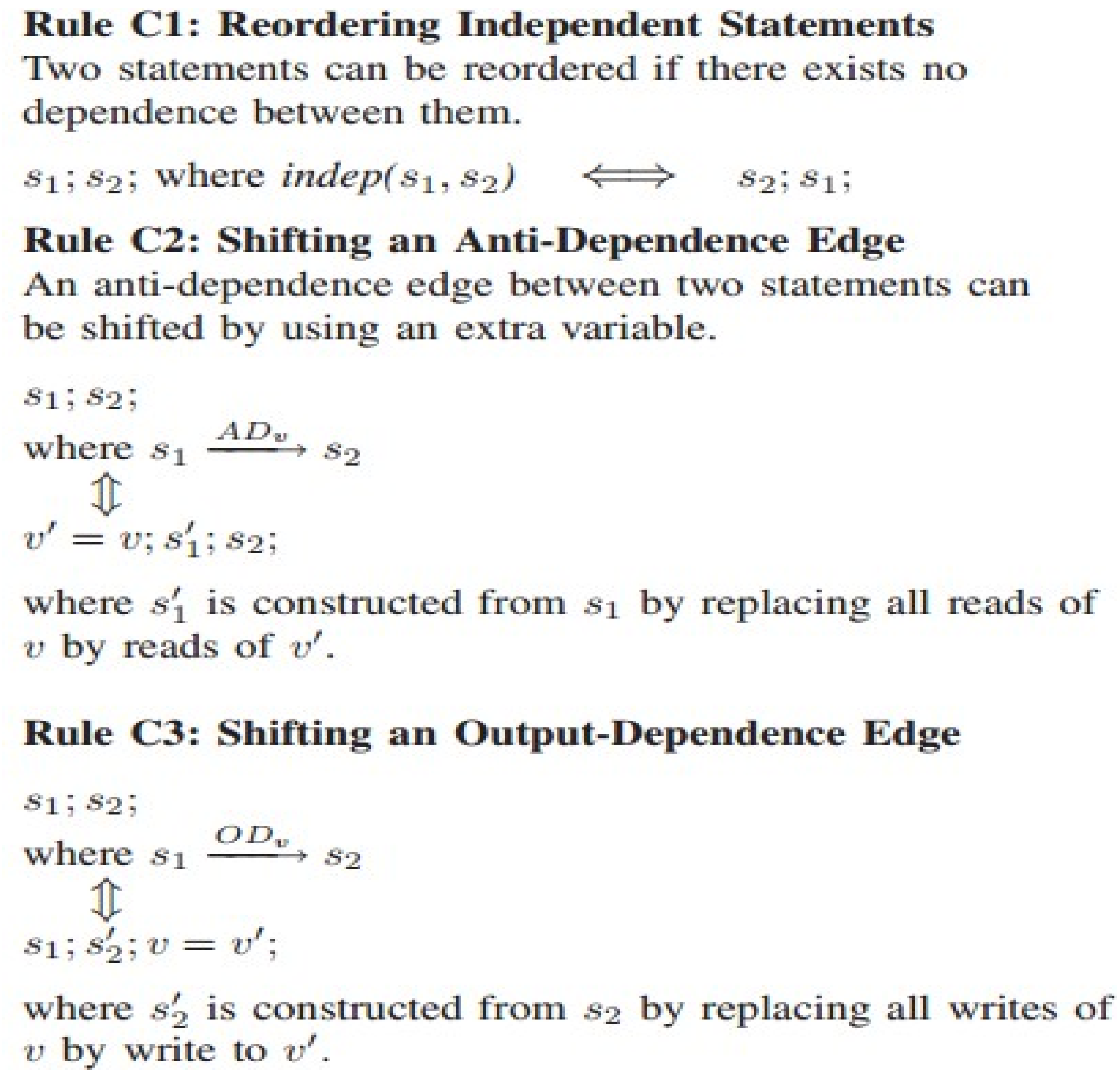
```
while (category != null) {
    qt.bind(1, category);
    int count =
executeQuery(qt);
    sum = sum + count;
    category =
getParent(category);
}
```

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

Basic Rules that Facilitates Reordering of Statements



Rule C1: Reordering Independent Statements

Two statements can be reordered if there exists no dependence between them.

$s_1; s_2;$ where $indep(s_1, s_2) \quad \Longleftrightarrow \quad s_2; s_1;$

Rule C2: Shifting an Anti-Dependence Edge

An anti-dependence edge between two statements can be shifted by using an extra variable.

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

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

Rule C3: Shifting an Output-Dependence Edge

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

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

The Statement Reordering Algorithm

- **Goal:** Reorder statements such that no LCFD edges cross the program point immediately succeeding s_q.

- **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

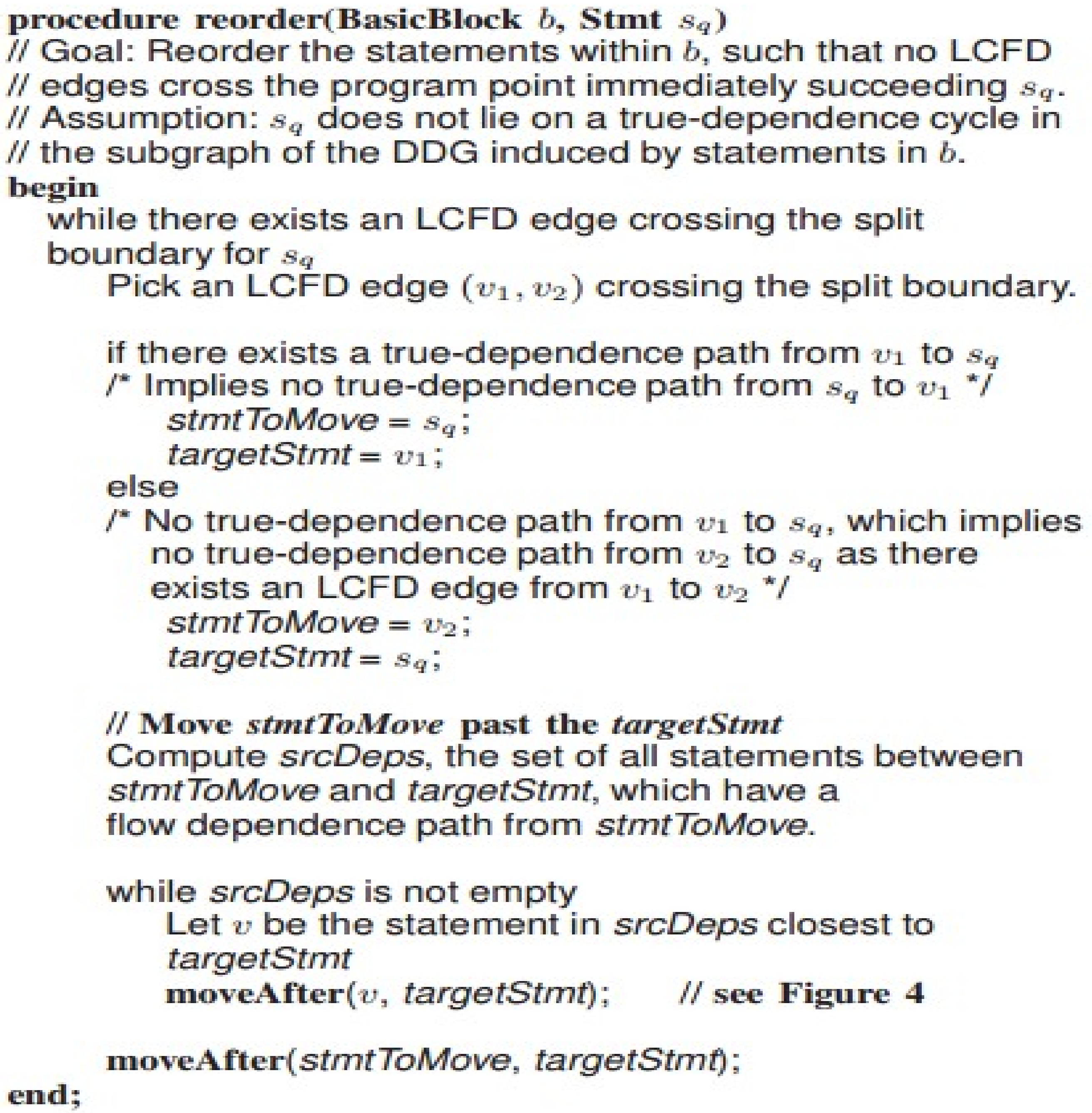
Definition: A True-dependence cycle in a DDG is a directed cycle made up of only FD and LCFD edges.

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

The Statement Reordering Algorithm

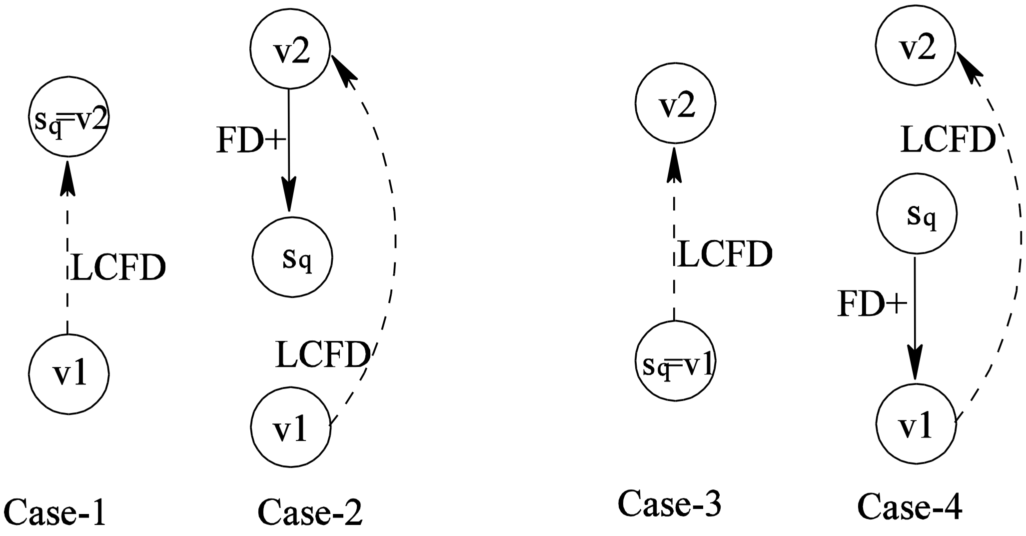


```
procedure reorder(BasicBlock $b$, Stmt $s_q$)
// Goal: Reorder the statements within $b$, such that no LCFD
// edges cross the program point immediately succeeding $s_q$.
// Assumption: $s_q$ does not lie on a true-dependence cycle in
// the subgraph of the DDG induced by statements in $b$.
begin
  while there exists an LCFD edge crossing the split
  boundary for $s_q$
      Pick an LCFD edge $(v_1, v_2)$ crossing the split boundary.

      if there exists a true-dependence path from $v_1$ to $s_q$
      /* Implies no true-dependence path from $s_q$ to $v_1$ */
            stmtToMove = $s_q$;
            targetStmt = $v_1$;
      else
      /* No true-dependence path from $v_1$ to $s_q$, which implies
          no true-dependence path from $v_2$ to $s_q$ as there
          exists an LCFD edge from $v_1$ to $v_2$ */
            stmtToMove = $v_2$;
            targetStmt = $s_q$;

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

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

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

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

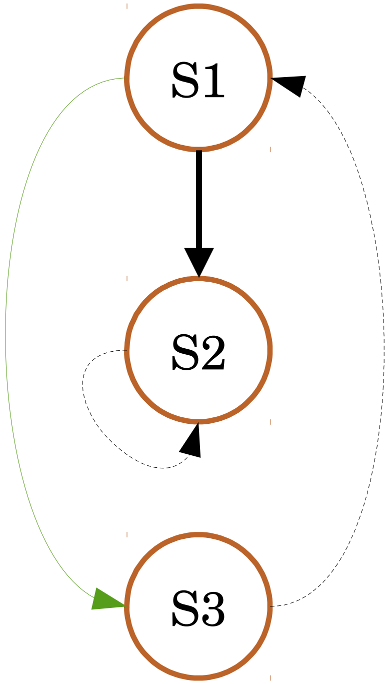
- Step 2: Compute statements dependent on the ***stm*** (***stmdeps***)
- Step 3: Move each of ***stmdeps*** past ***target***
- Step 4: Move ***stm*** past ***target***

*heavily simplified; refer to paper for details

Before

```
While (category != null) loop
    (s1) icount = q(category)
    (s2) sum = sum + icount
    (s3) category = getParent(category)
End loop
```

Data Dependence Graph (DDG)



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

Before

```
While (category != null) loop
    (s1) icount = q(category)
    (s2) sum = sum + icount
    (s3) category = getParent(category)
End loop
```

Intuition: Move s1 pass s3

After

```
While (category != null) loop
    (ts1) category1 = category
    (s3) category = getParent(category)
    (s1) icount = q(category1)
    (s2) sum = sum + icount
End loop
```

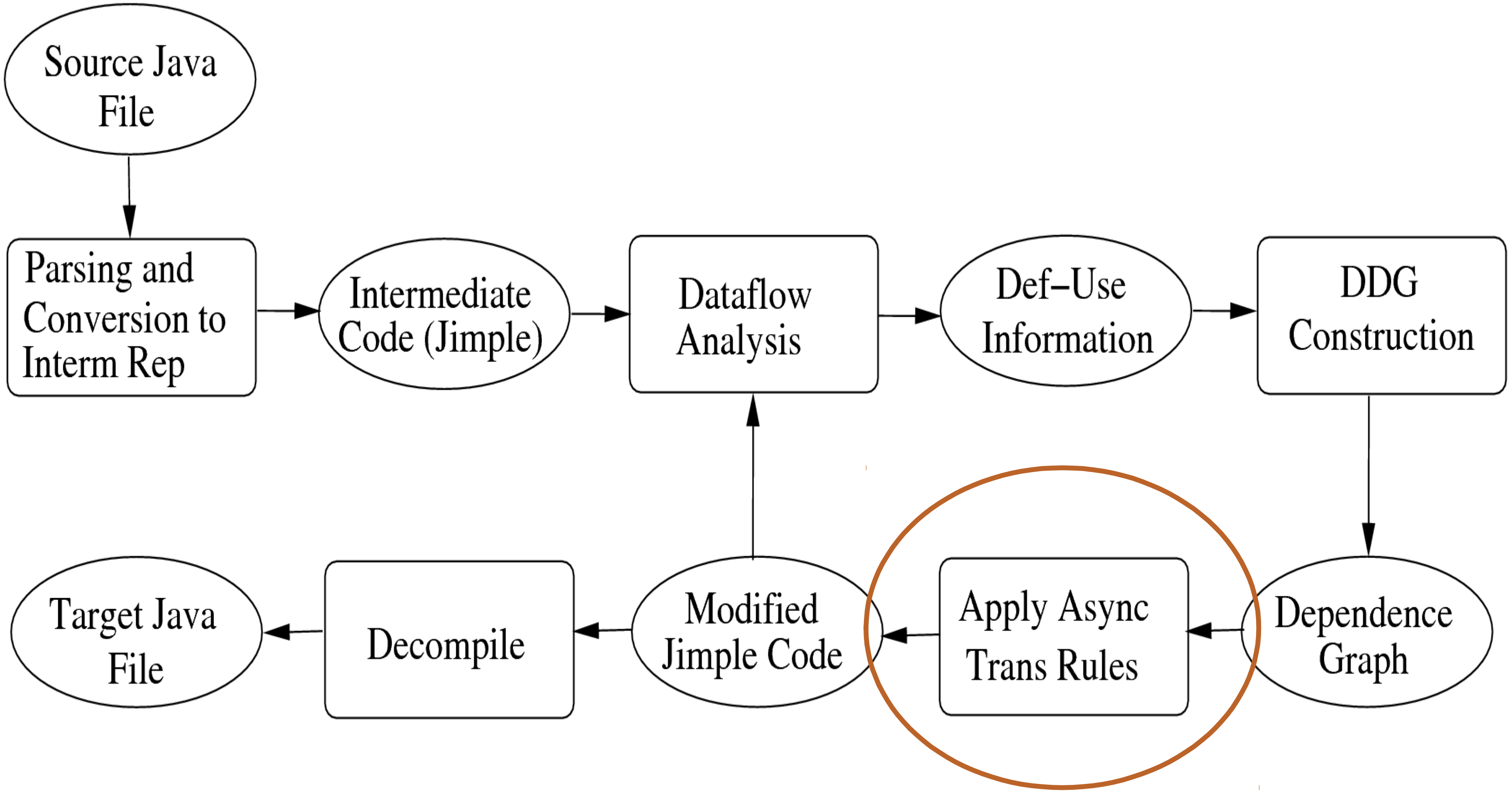
Automatic Program Transformation for asynchronous submission

Increasing the applicability of transformations

System design and Experimental evaluation

System Design: DBridge

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



Note: Rule application will stop when all query execution statement which don't lie on true dependence cycles are converted to asyn calls.

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

Extensions And Optimizations

- 1. Overlapping the Generation and Consumption of Asynchronous Requests
  - On applying the basic loop fission , a loop will result in two loops.
  - First loop generates asynchronous requests – Producer loop
  - Second loop that processes the result – Consumer loop

  - Problem: First producer loop will complete then only consumer loop will start processing , high response time.

  - Solution : Overlapping the consumption of query result with the submission of requests.

Extensions And Optimizations

- 2. Asynchronous Submission of Batched queries
  - Asynchronous submission of multiple, smaller
  - batches of queries.
  - With asynchronous batching, the thread can observe the whole queue, and pick up one, or more, or all requests from the queue

  - Advantages:
  - Reduces network round trip delays
  - Overlaps client computation with that of server
  - Reduces random IO at database
  - Memory requirement do not grow as much as with pure batching due to small batch size.

Adaptive tuning of batch size

1. One or all Strategy:

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.

2. Lower Threshold Strategy:

- Batching results in three network round trip and very small batches perform poorly as compared to asynchronous submission.
- If n > bt, then pick up all the n requests in the queue and batch them.
- If 1 ≤ n ≤ bt, then pick up one request from the queue, and execute it as an individual request.

bt>=3

Adaptive tuning of batch size

3. Growing upper-threshold based Strategy:

- Problem in Lower threshold approach: 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.

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.
- 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.

Experiments

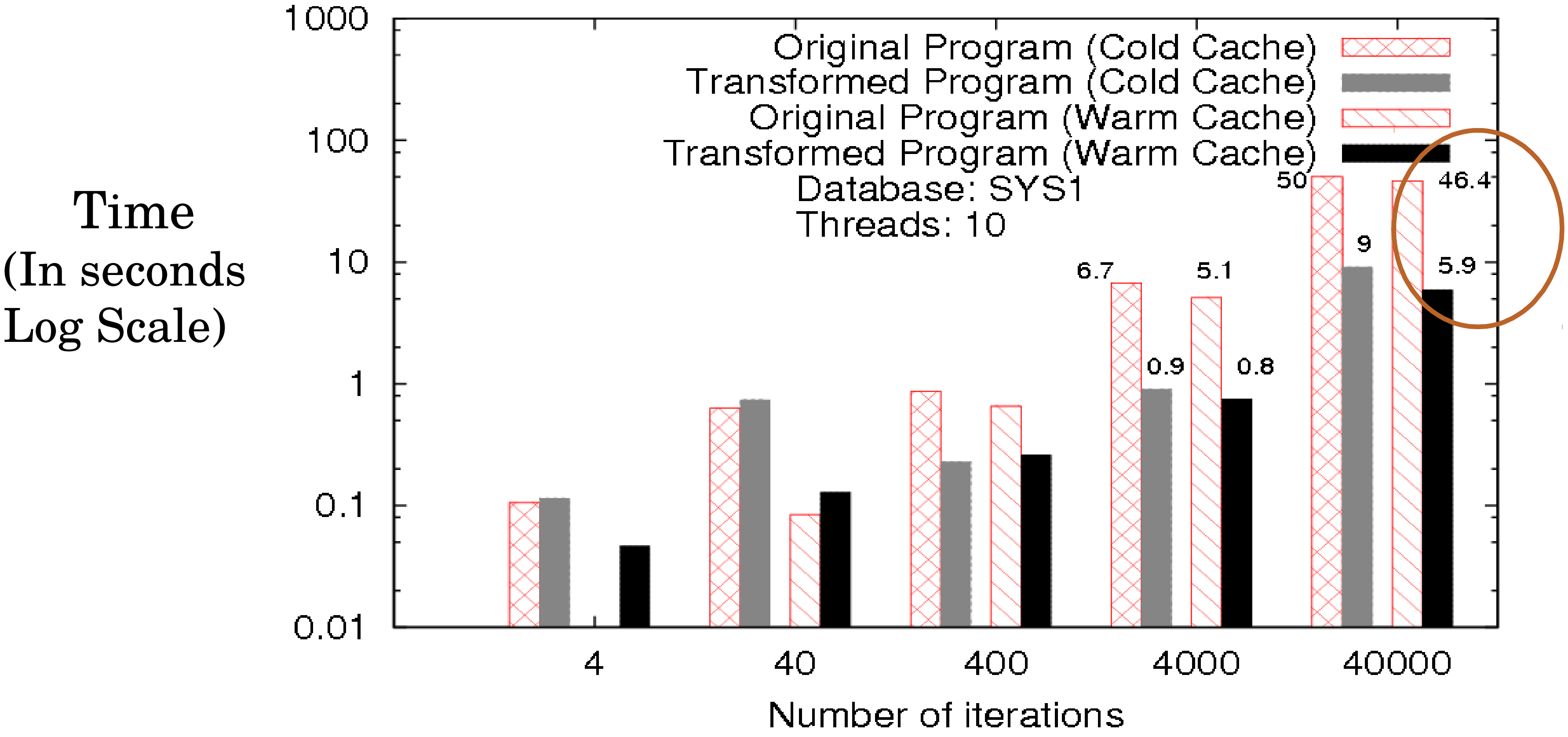
- Conducted on 5 applications
  - Two public benchmark applications (Java/JDBC)
  - Two real world applications (Java/JDBC)
  - Freebase web service client (Java/JSON)

- 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

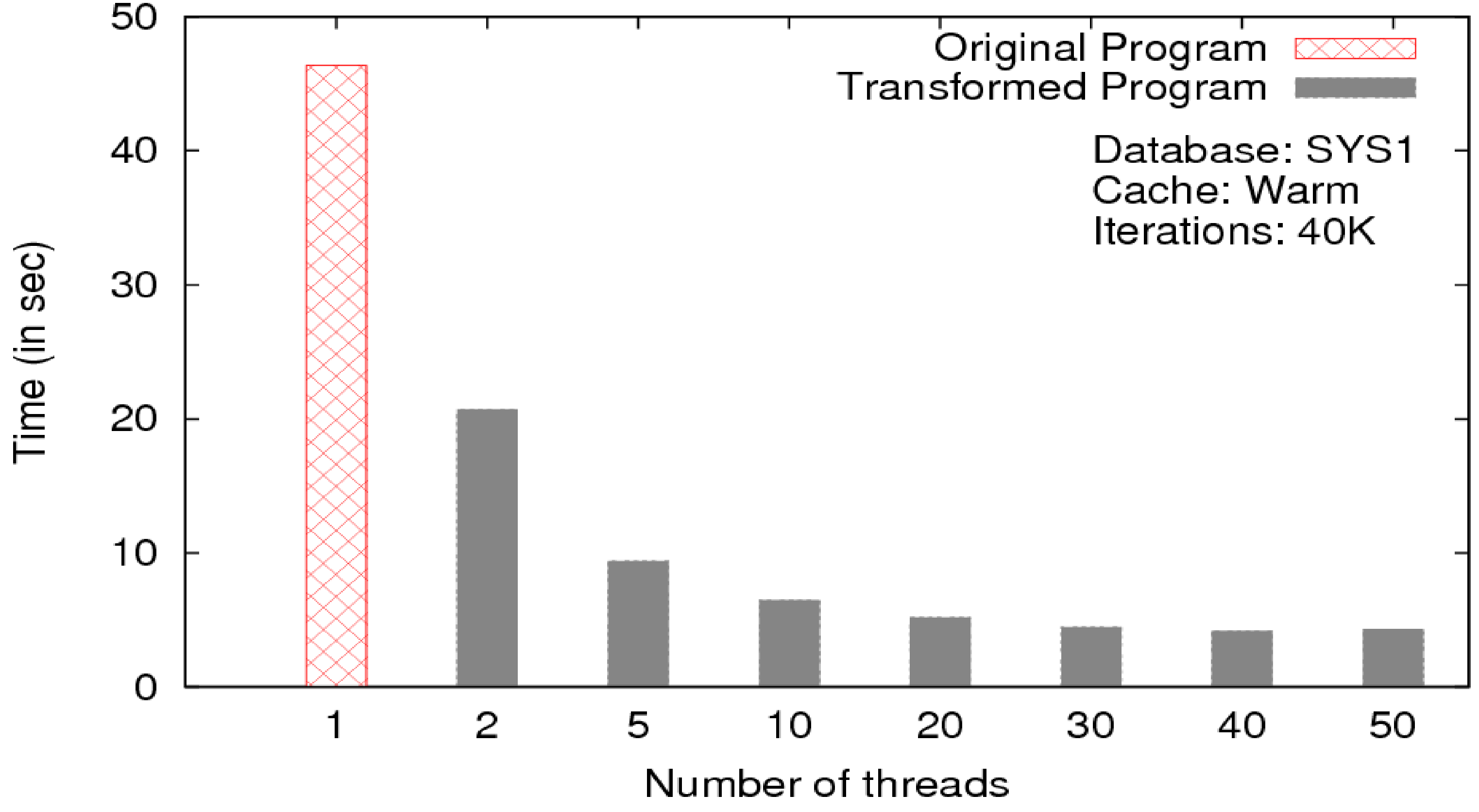
- Impact of iteration count
- Impact of number of threads
- Impact of Warm cache vs. Cold cache
  - Since Disk IO on the database is an important parameter

Auction Application: Impact of Iteration count, with 10 threads



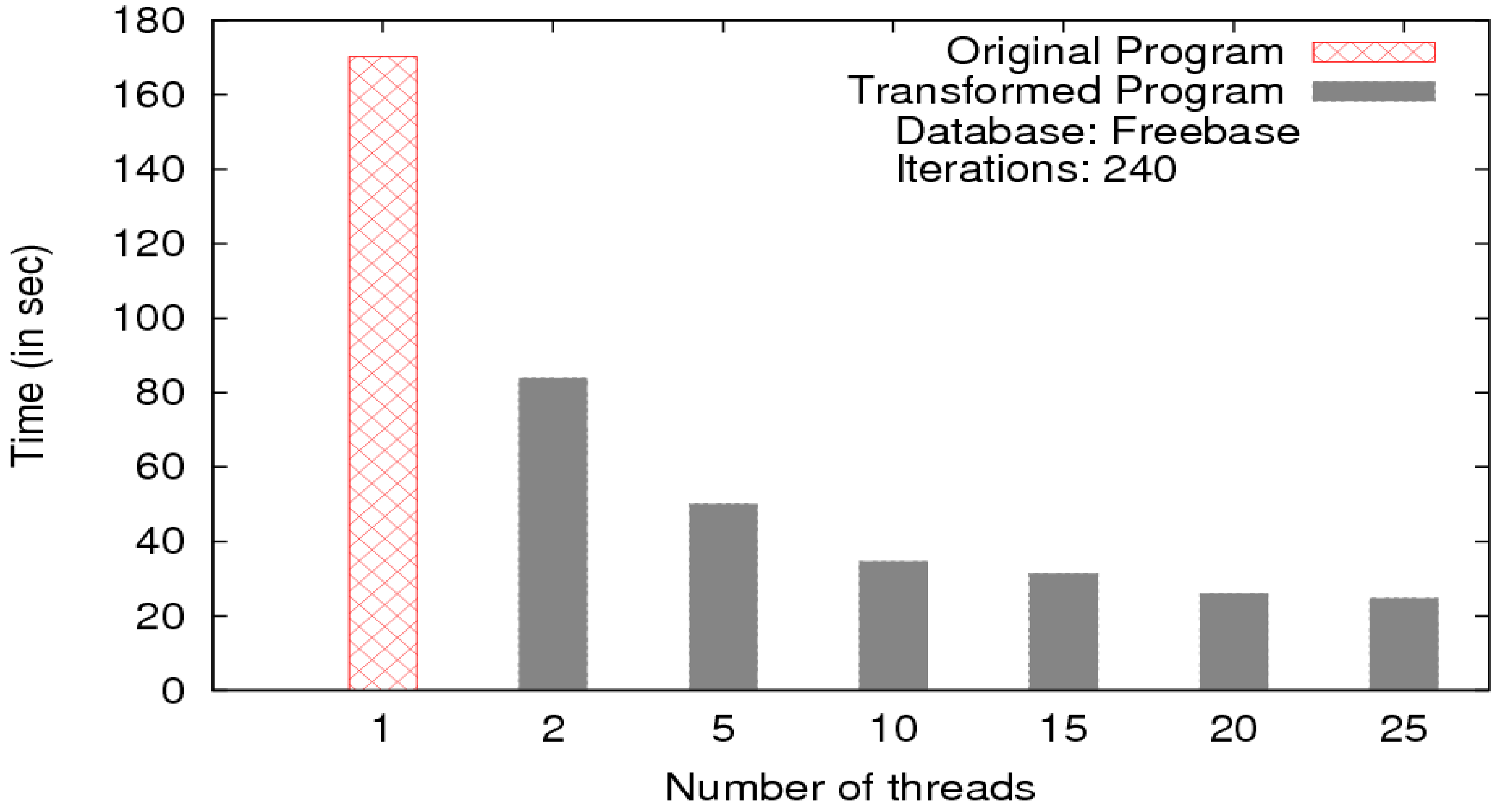
- For small no. (4-40) iterations, transformed program slower
- At 400-40000 iterations, **factor of 4-8 improvement**
- Similar for warm and cold cache

Auction Application: Impact of thread count, with 40K iterations



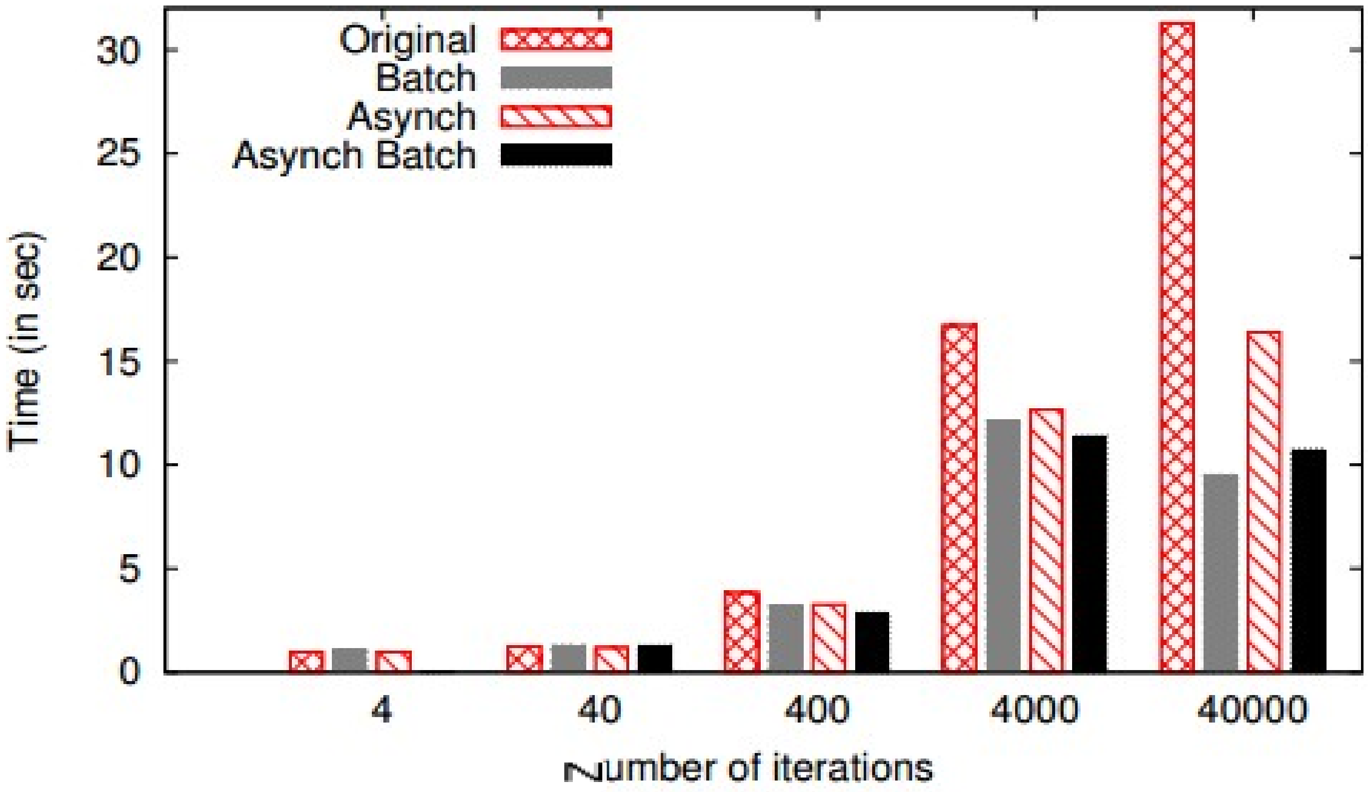
- Time taken reduces drastically as thread count increases
- No improvement after some point (30 in this example)

WebService: Impact of thread count



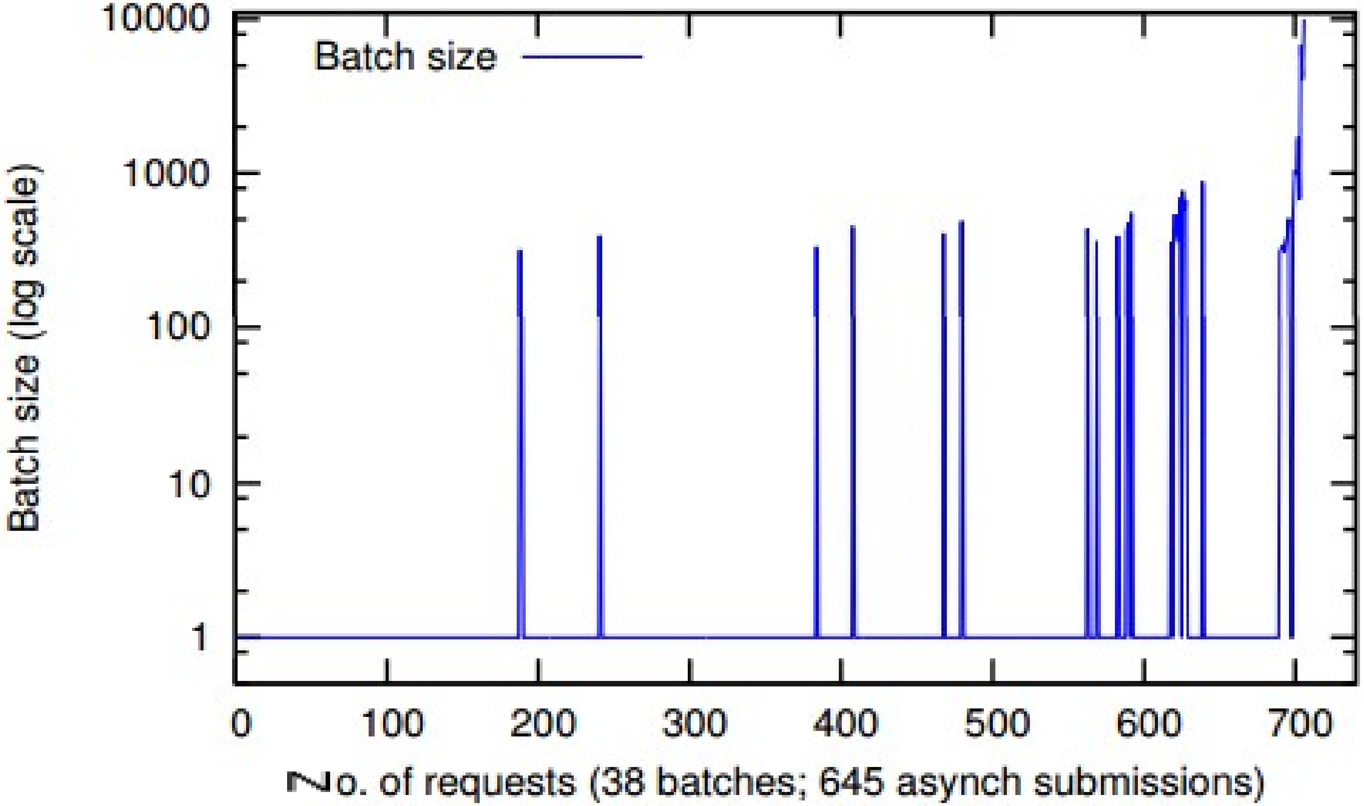
- HTTP requests with JSON content
- Impact similar to earlier SQL example
- Note: Our system does not automatically rewrite web service programs, this example manually rewritten using our transformation rules

Comparison of approaches



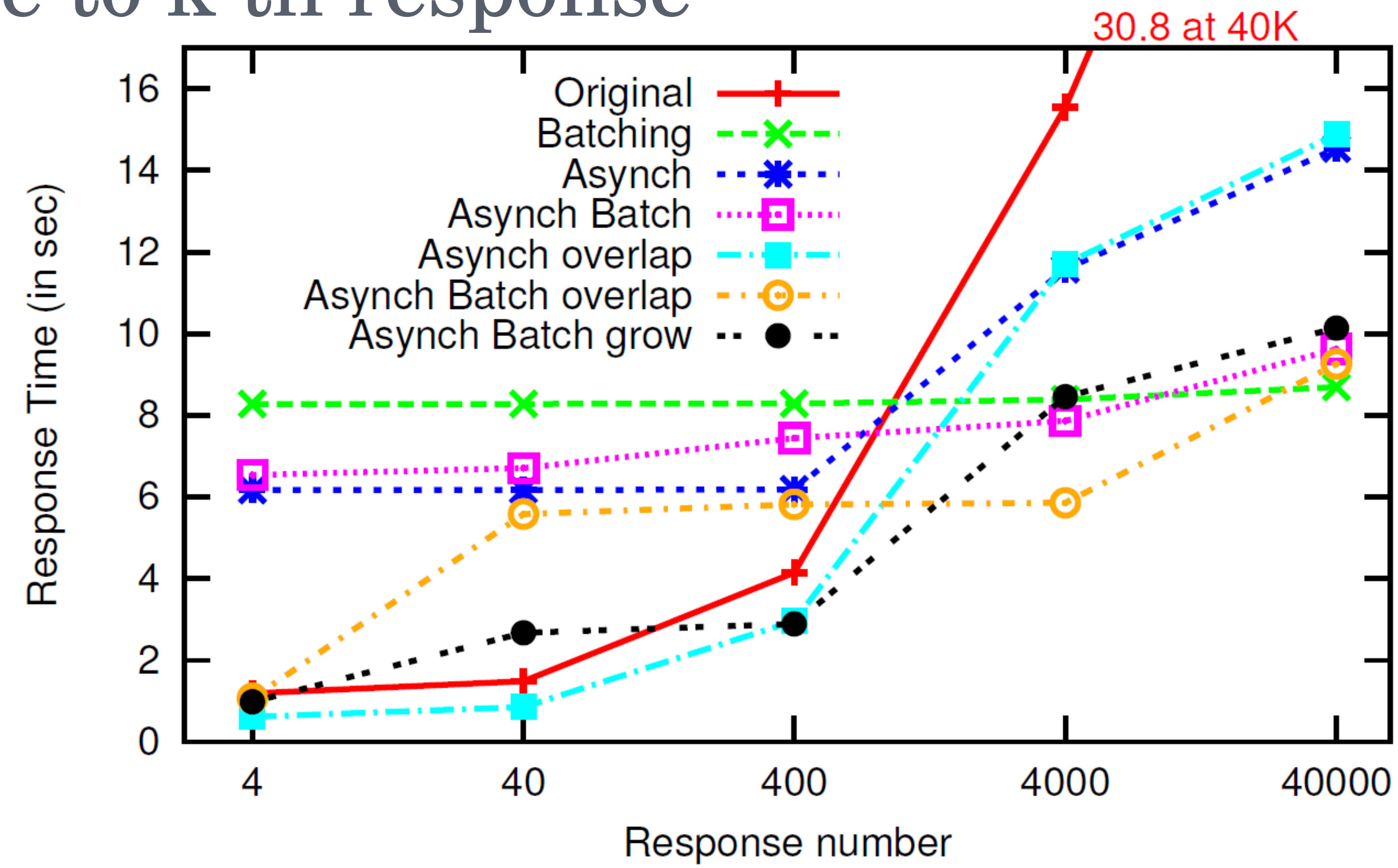
- At Small no of iterations, all approaches behaves similarly
- At 40000 iterations asynch submission with 12 threads gives 50 % improvement, batching gives 75 % impovement
- Asynch Batching with 48 threads and lower batching threshold of 300 leads to about 70% improvement.

Behaviour of one run of asynchonous batching



- Initially many requests are sent individually.
- As the execution progresses, there are more and more batch submission and batch size also start growing.

Time to k-th response



- Observe "Asynch Batch Grow" (black)
  - stays close to the original program (red) at smaller iterations
  - stays close to batching (green) at larger number of iterations.
  - The Async Batch Grow approach behaves the best in balancing response time vs total execution time

Thank You!