Compiling Relational Queries Over Program Traces to Instrumentation
-or-
Beyond `printf()` Debugging

Simon Goldsmith
joint work with Robert O'Callahan and Alex Aiken

OSQ lunch

April 5, 2004

Overview

- PTQL
  - expressive, declarative query language over program traces
- Partiqle
  - compiles PTQL query to instrumentation of Java bytecode
- better than manual instrumentation
- simpler than creating new dynamic analysis tool
- efficient enough to run interesting queries on real world Java programs

Motivation

- Program specific questions
  - Does my program do X?
  - How many times?
  - How long does it take?
  - e.g. want a histogram of calls to `foo()` in third param
- Existing dynamic analysis tools
  - have a question hard wired
- How to answer these questions?

State of the Art

- manual instrumentation: extra fields, globals, etc.
- a bunch of calls to `printf()`
- hack until trace size < 500 MB
- grep/sed/perl out the info you want

Does `doTransaction()` call `sleep()`?

```
public class DB {
  B b;
  void doTransaction() {
    b.y();
} }
public class B {
  void y() { sleep(); }
  void sleep() {
} }
```

- Obviously yes for this example
- How might one manually instrument to find out?

```java
public class DB {
  B b;
  public static boolean active = false;
  void doTransaction() {
    active = true;
    b.y();
    active = false;
} }
public class B {
  void y() { sleep(); }
  void sleep() {
    if (DB.active) {
      System.out.println("call to sleep()!");
    }
} }
```

Failings of Manual Instrumentation

- adds complexity
- non-local
- wrong
  - recursion
  - exceptions
  - threads

Solution

- We claim: such ad hoc dynamic analyses are naturally represented as *queries* over the *program trace*
- advantages:
  - all in one place
  - declarative
  - tool handles recursion, threads, exceptions

Terminology

- A **program trace** is a sequence of time-stamped *events* that happen during program execution
- Each method invocation, object allocation, etc. that occurs during program execution is an **event**.
- A **query** specifies a combination of events.

Outline

- ✔ Mom and Apple Pie
- ✔ Knock Down the Strawman
- **Program Trace Query Language (PTQL)**
- PTQL compiler: Partiqle
- Overhead of Partiqle's Instrumentation
- Related Work
- Future Work

Program Trace Query Language (PTQL)

- basically SQL query over program trace
- tables:
  - `MethodInvocation`
  - `ObjectAllocation`
- event happens => record in table
  - e.g. call to `foo()` adds record to `MethodInvocation`
- records have start/end timestamps
- interesting queries join several records together

PTQL: What's in the Records?

- `MethodInvocation`
  - `methodName`
  - `implementingClass, declaringClass`
  - `startTime, endTime`
  - `receiver`
  - `thread`
  - `param0, param1, ...`
  - `result`
- `ObjectAllocation`
  - `allocTime, collectTime`
  - `dynamicType`

Example PTQL Query I

- Give me all the return values of method `foo`.

```
SELECT foo.result
  FROM MethodInvocation foo
 WHERE foo.methodName = “foo”
```

Does `doTransaction()` call `sleep()`?

```
SELECT  doTrans.startTime, sleep.startTime
  FROM  MethodInvocation doTrans,
        MethodInvocation sleep
 WHERE  doTrans.methodName = 'doTransaction'
   AND  doTrans.definingClass = 'DB'
   AND  sleep.methodName = 'sleep'
   AND  sleep.definingClass = 'B'
   AND  doTrans.thread = sleep.thread
   AND  doTrans.startTime < sleep.startTime
   AND  sleep.endTime < doTrans.endTime
```

Some Java Anti-Pattern Finding Queries

- `hashCode()` agrees with `equals()`
- calls to `hashCode()` on same receiver return same value
- no string `s = s + ...;` in a loop
- streams are closed <1000 ms after last read/write
- `compareTo()` is reflexive and transitive
- `x.compareTo(y) > 0` iff `y.compareTo(x) < 0`

Outline

- ✔ Mom and Apple Pie
- ✔ Knock Down the Strawman
- ✔ Program Trace Query Language (PTQL)
- **PTQL compiler: Partiqle**
- Overhead of Partiqle's Instrumentation
- Related Work
- Future Work

Partiqle: Goal

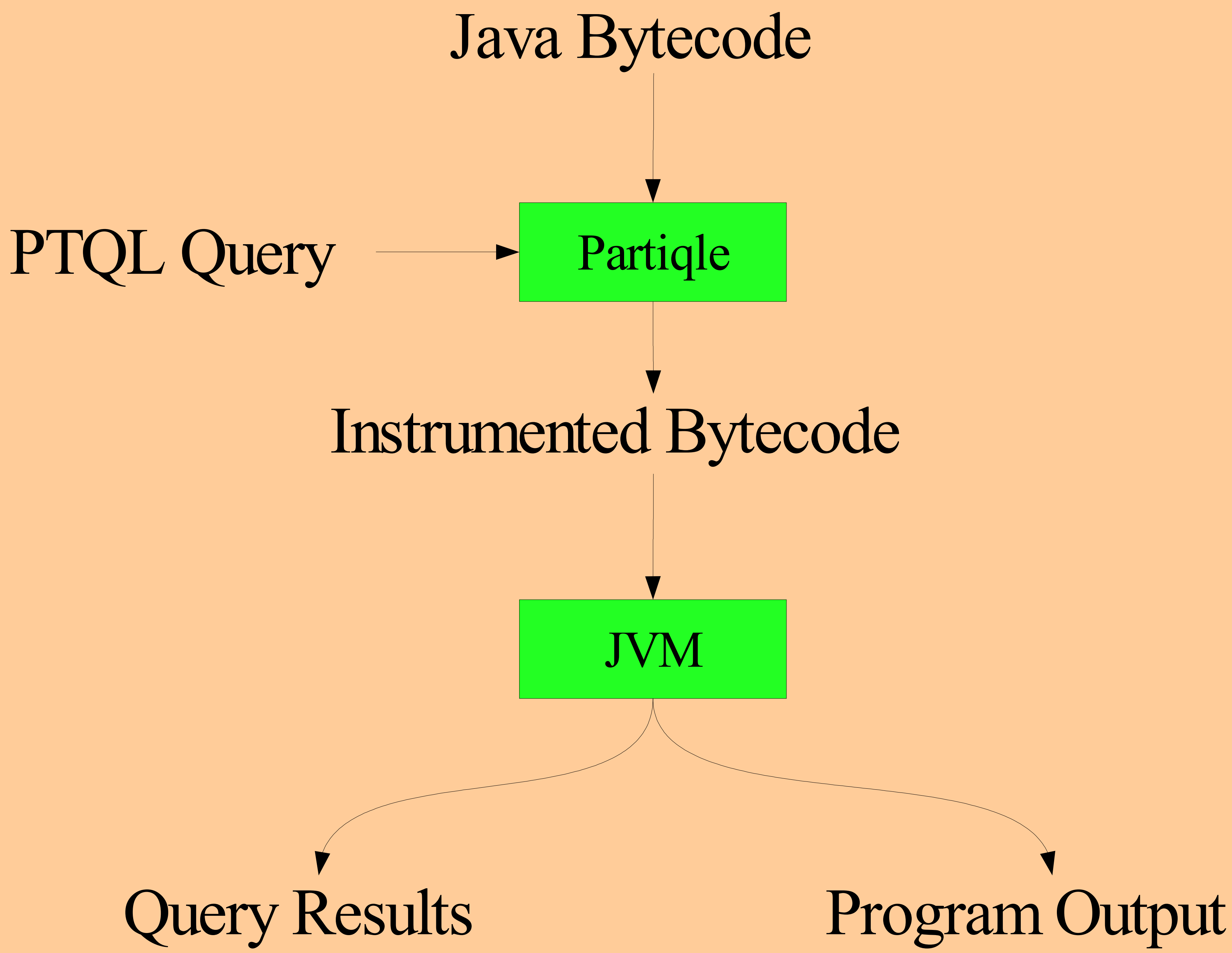
- **in:** PTQL query + program + program input
- **out:** program output + set of query results
  - one result = a tuple of events' fields

One Approach

- log program trace to a relational database
  - add instrumentation to log events
- query database
- problem: does not scale
  - too many events => traces too big

Partiqle: Approach

- ...like that but without the database
- push query evaluation into instrumentation
- evaluate the query online
  - intermediate data kept in memory
- optimizations to minimize
  - amount of data kept
  - duration data kept


Java Bytecode
PTQL Query
Partiqle
Instrumented Bytecode
JVM
Query Results
Program Output

Partiqle: Query Evaluation Strategy

- one runtime table per FROM item
- instrumentation where events happen
  - create record
  - fill out fields
  - add to runtime table
- last event in query result triggers query evaluation

Partiqle: Optimization

- central tenet: discard events as early as possible
  - static: no instrumentation to record event
  - admission check: don't record irrelevant events
  - retention check: discard event record when no longer relevant

Consider the example from the intro...

Does `doTransaction()` call `sleep()`?

```
SELECT  doTrans.startTime, sleep.startTime
  FROM  MethodInvocation doTrans,
        MethodInvocation sleep
 WHERE  doTrans.methodName = 'doTransaction'
   AND  doTrans.definingClass = 'DB'
   AND  sleep.methodName = 'sleep'
   AND  sleep.definingClass = 'B'
   AND  doTrans.thread = sleep.thread
   AND  doTrans.startTime < sleep.startTime
   AND  sleep.endTime < doTrans.endTime
```

Baseline Instrumentation

- two run-time tables:
  - `dts` for `doTrans`
  - `ss` for `sleep`
- instrumentation at start of each method:
  - create record
  - add it to tables `dts` and `ss`
- find all pairs in `dts` $\times$ `ss` that satisfy the query

Static Filtering of Instrumentation Sites

- use these conditions to filter instrumentation sites:

```
doTrans.methodName = 'doTransaction'
doTrans.definingClass = 'DB'
sleep.methodName = 'sleep'
sleep.definingClass = 'B'
```

- at start of `DB.doTransaction()`
  - add a new record to `dts`
- at start of `B.sleep()`
  - add a new record to `ss`
- check all pairs (`doTrans`, `sleep`) in `dts` × `ss`

Admission Check

- Only some calls to `sleep()` are interesting

```
doTrans.thread = sleep.thread
doTrans.startTime < sleep.startTime
sleep.endTime < doTrans.endTime
```

  - when record `sleep` added to `ss`, `dts` must contain
    - a call to `DB.doTransaction()`
    - that has already started but not ended
    - on the same thread
- instrumentation at `sleep()` does an *admission check*
  - if no suitable `doTrans` in `dts` drop this `sleep`

Output Query Results Incrementally

- At the start of `sleep()` we have a record `sleep` and all `doTrans` records that could match with it
  - we can output all results involving this `sleep` now
- No need to record the `sleep`, we are done with it
- Benefits:
  - incremental output
  - reduces size of tables
- Note: `ss` table always empty!
  - intuition: table contains only records that might contribute to future results

Retention Check

```
doTrans.startTime < sleep.startTime
sleep.endTime < doTrans.endTime
```

- At end of `doTransaction()`, all matching calls to `sleep()` must have already started and ended
- instrumentation at end of `doTransaction()` does a *retention check*
  - if there is no suitable `sleep` in `ss`, drop this `doTrans`
  - `ss` is always empty; check always fails; always drop `doTrans`
  - intuition: we can discard `doTrans` because no `sleeps` need it anymore

Final Picture of Example

- start of `doTransaction()` : add record to `dts`
- end of `doTransaction()` : remove record from `dts`
- start of `sleep()` : output query result for matching records in `dts` (if any)

Summary of Our Approach

- each item in `FROM` clause => table at runtime

```
FROM MethodInvocation doTrans => dts
     MethodInvocation sleep => ss
```

- each event => record in appropriate table
  - call to `doTransaction()` => add record to `dts`
- static predicates filter instrumentation sites
- admission/retention checks prune tables
- timing analysis tells us when to remove records from tables

SKIP: More on Timing Analysis

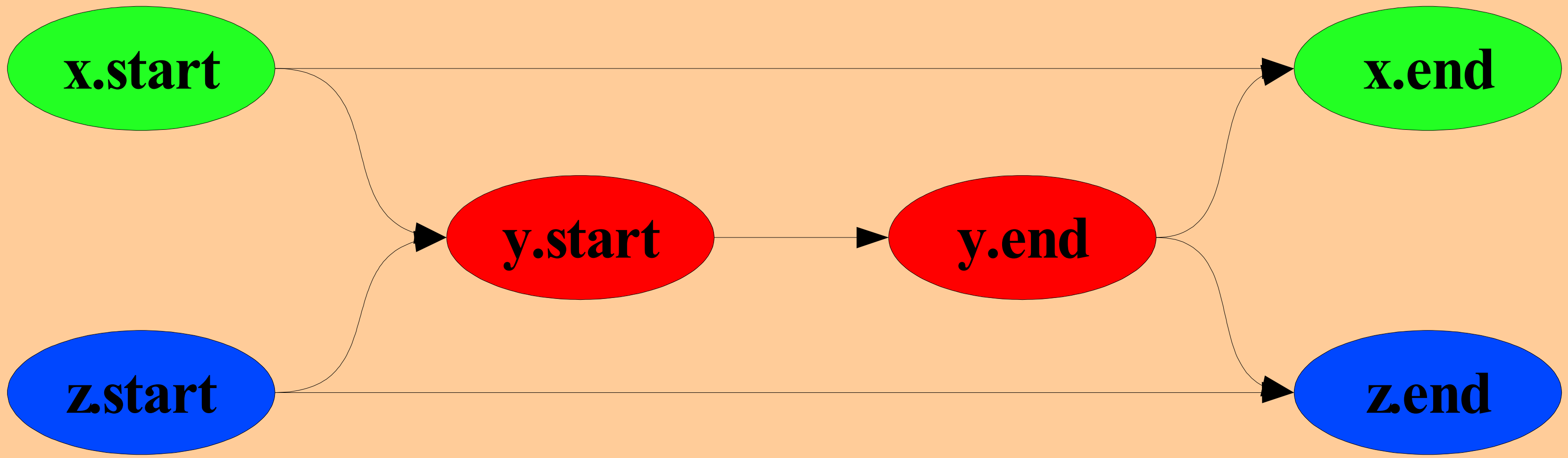
- Notice the time constraints from our example

```
x.startTime < y.startTime
y.endTime < x.endTime
```

- time constraints determine
  - which tables to check in admission/retention checks
    - when y starts, x must have already started
    - when x ends, y must have ended
  - when we have enough info to output results
- Let's look at how...

SKIP: Timing Graph

- Explicit and implicit constraints give us a partial ordering of start and end events
- e.g. x.start < y.start, z.start < y.start, y.end < x.end, y.end < z.end



- admission/retention checks examine predecessors in timing graph

SKIP: Post-dominator Nodes

- When do we have enough information to output a result tuple?
  - after all start events
  - after all output information is available
  - after all WHERE conditions can be verified
  - the *post-dominator node*
- If no such node exists, the query requires information to be held indefinitely and may be intrinsically costly

SKIP: When a Record is Done

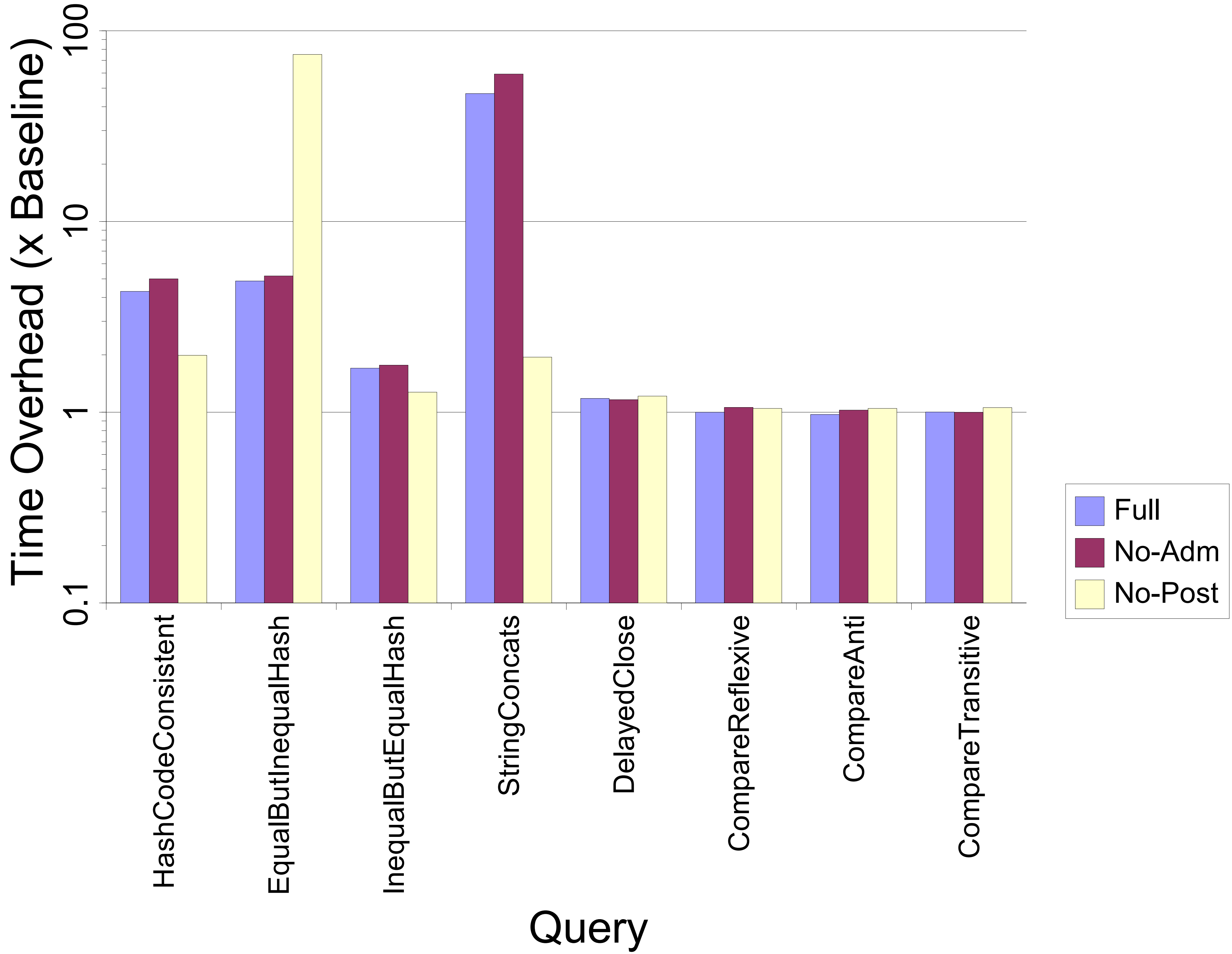
- At the post-dominator node
  - output all results involving the record
  - remove the record from its table
- At end event for x
  - do retention check to see if keeping x is necessary
  - sometimes can prove that retention check will always fail
    - E.g., events that are successors of post-dominator node in timing graph
- When a record is removed
  - may remove the last support from other records; their admission/retention checks should be repeated

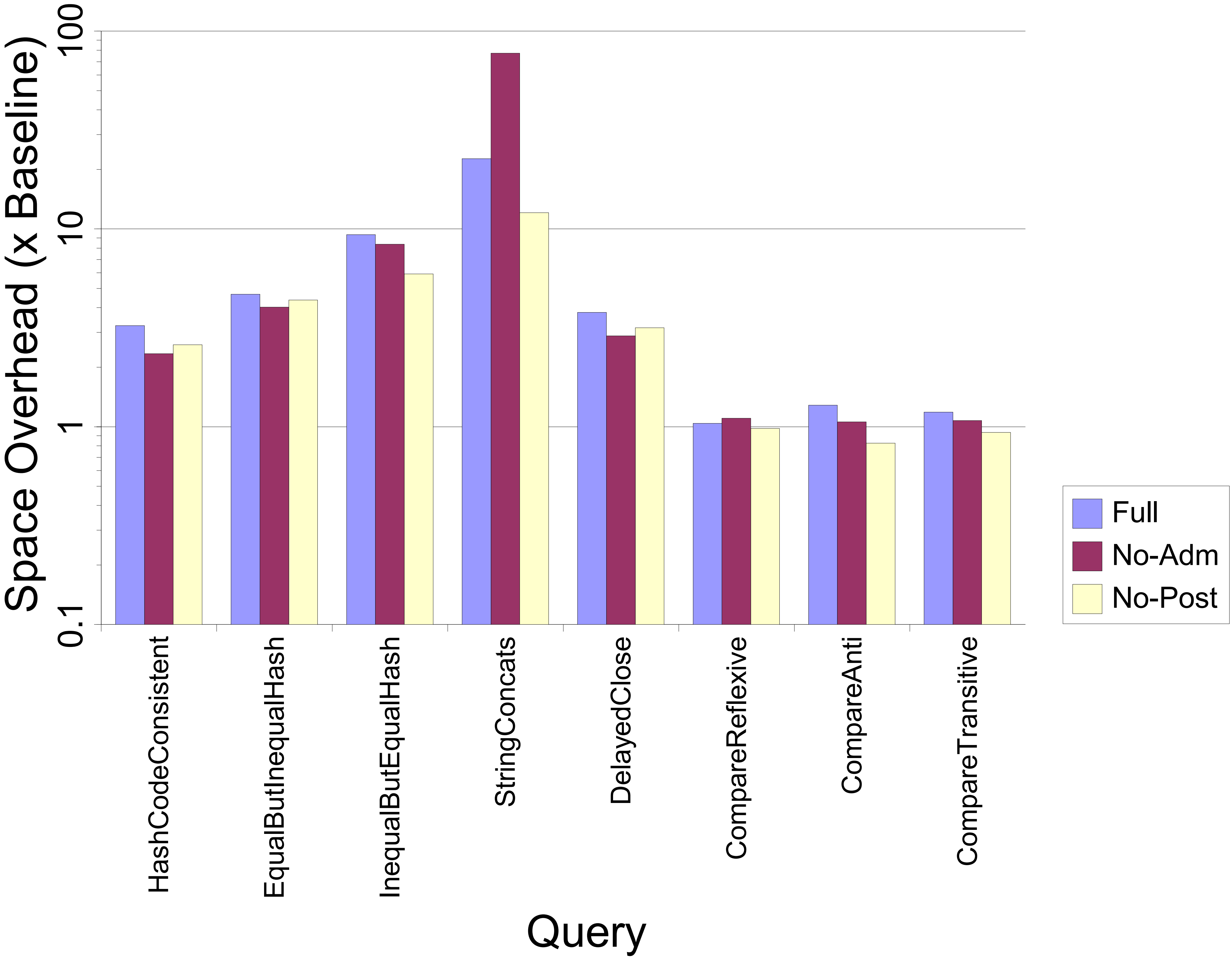
Outline

- ✔ Mom and Apple Pie
- ✔ Knock Down the Strawman
- ✔ Program Trace Query Language (PTQL)
- ✔ PTQL compiler: Partiqle
- **Overhead of Partiqle's Instrumentation**
- Related Work
- Future Work

Experiments

- ran anti-pattern queries (from before) on
  - Apache Tomcat (webserver / java servlets) (17k methods)
  - SpecJVM98
  - some microbenchmarks
- measured slowdown and memory footprint
- found some performance bugs
- show overhead for tomcat


Time Overhead (x Baseline)
100
10
1
0.1
HashCodeConsistent
EqualButInequalHash
InequalButEqualHash
StringConcats
DelayedClose
CompareReflexive
CompareAnti
CompareTransitive
Query
Full
No-Adm
No-Post



Space Overhead (x Baseline)
100
10
1
0.1
HashCodeConsistent
EqualButInequalHash
InequalButEqualHash
StringConcats
DelayedClose
CompareReflexive
CompareAnti
CompareTransitive
Query
Full
No-Adm
No-Post

Results

- Found several performance bugs (string concats)
  - Jack (SpecJVM98 benchmark)
  - Apache Tomcat's XML parser
  - IBM JDK
- Found correct, but subtle code
  - hash code consistency in Xerces XML parser

Future Work

- more thorough justification / case study
- representation change / performance issues
- subqueries / negation
- aggregation (ala SQL's `GROUP BY`)
- instrument for several queries at once
- add to the data model
- static analysis to prune instrumentation

Related Work

- Program Monitoring (e.g. PEDL/MEDL)
- DIDUCE / Daikon / Liblit
- Aspect Oriented Programming Languages
- Other trace-based query engines

Conclusion

- PTQL
  - expressive, declarative query language over program traces
- Partiqle
  - compiles PTQL query to instrumentation of Java bytecode
- better than manual instrumentation
- simpler than creating new dynamic analysis tool
- efficient enough to run interesting queries on real world Java programs