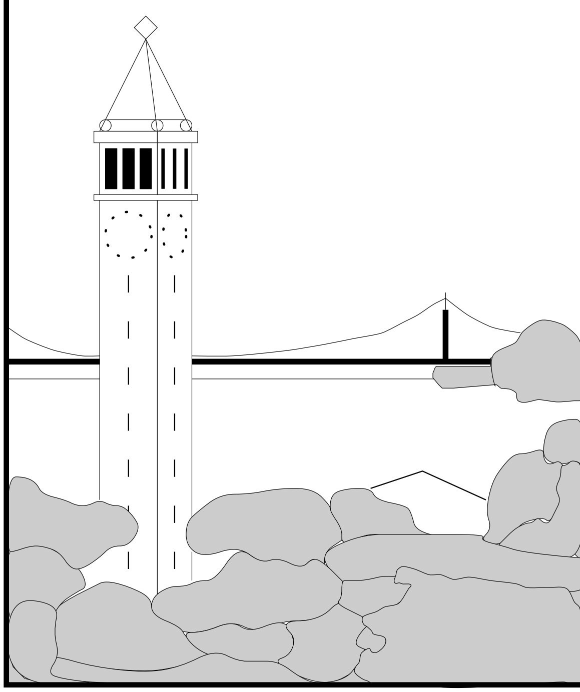
Light-Weight Instrumentation From Relational Queries Over Program Traces

Simon Goldsmith
University of California, Berkeley
sfg@eecs.berkeley.edu

Robert O'Callahan
IBM T.J. Watson Research Center
roca@us.ibm.com

Alex Aiken
Stanford University
aiken@cs.stanford.edu



Report No. UCB/CSD-4-1315

March 2004

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Light-Weight Instrumentation From Relational Queries Over Program Traces [*†]

Simon Goldsmith
University of California, Berkeley
sfg@eecs.berkeley.edu

Robert O'Callahan
IBM T.J. Watson Research Center
roca@us.ibm.com

Alex Aiken
Stanford University
aiken@cs.stanford.edu

ABSTRACT

Instrumenting programs with code to monitor their dynamic behaviour is a technique as old as computing. Today, most instrumentation is either inserted manually by programmers, which is tedious, or automatically by specialized tools, which are nontrivial to build and monitor particular properties. We introduce Program Trace Query Language (PTQL), a general language in which programmers can write expressive, declarative queries about program behaviour. PTQL is based on relational queries over program traces. We argue that PTQL is more amenable to human and machine understanding than competing languages. We also describe a compiler, Partiqle, that takes a PTQL query and a Java program and produces an instrumented program. This instrumented program runs normally but also evaluates the PTQL query on-line. We explain some novel optimizations required to compile relational queries into efficient instrumentation. To help evaluate our work, we present the results of applying a variety of PTQL queries to a set of benchmark programs, including the Apache Tomcat Web server. The results show that our prototype system already has usable performance, and that our optimizations are critical to obtaining this performance. Our queries also revealed significant (and apparently unknown) performance bugs in the *jack* SpecJVM98 benchmark, in Tomcat, and in the IBM Java class library, and some uncomfortably clever code in the Xerces XML parser.

1. INTRODUCTION

Dynamic analysis is an important technique for measuring program performance and checking program correctness. Full blown dynamic analyses are difficult to write and almost certainly not worth the trouble for small questions. Often, programmers resort to ad hoc dynamic analysis: inserting extra fields and print statements. This manual instrumentation is labor intensive and makes code harder to read and maintain.

Consider the following program fragment:

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

public class B {
    void y() {
        sleep();
    }
    void sleep() { }
}
```

Can method `DB.doTransaction()` transitively call method `sleep()`? While the answer to this question is clearly "yes" for our contrived example, understanding the who-calls-whom relation in a large, object-oriented program can be a non-trivial task. A programmer might try to answer the question by instrumenting the code in the following way:

```
public class DB {
    B b;
    public static boolean doTransActive = false;
    void doTransaction() {
        doTransActive = true;
        b.y();
        doTransActive = false;
    }
}

public class B {
  void y() {
    sleep();
  }
```

[*]This work continues under the terms of joint study agreement W0135710 between IBM and UC Berkeley.
[†]This research was supported in part by the National Science Foundation under grant no. NSF CCR-0085949, and by Subcontract no. PY-1099 to Stanford, from the Dept. of the Air Force, prime contract no. F33615-00-C-1693. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

```
  void sleep() {
    if (DB.doTransActive) {
      System.out.println("call to sleep()!");
    }
  }
}
```

For only five lines of code, this instrumentation adds considerable complexity. We have added a new field (`doTransActive`) to `class DB` — necessary to communicate to `sleep()` the fact that `doTransaction()` is executing. Furthermore, we have added logic to both `sleep()` and `doTransaction()` which, without documentation, is not obviously separate from the primary function of these methods. Keeping this instrumentation in the code and turning it on and off becomes a matter of commenting or uncommenting (hopefully all of) it.

Worst of all, this instrumentation is not even correct. If `doTransaction()` terminates in an exception, `doTransActive` is never unset. If `doTransaction()` is a recursive function, `doTransActive` is set to `false` too soon (when the first activation of `doTransaction()` returns). The situation is quite a bit more complex in a multithreaded program. Each thread needs to keep track of whether it is executing `DB.doTransaction()` and care must be taken to avoid data races. Fortunately, these complexities are similar for all analyses and inserting the necessary supporting instrumentation could be automated.

In this paper we describe our design of Program Trace Query Language (PTQL), a language for writing queries over program traces.

We also describe our implementation and evaluation of Partiqle, a tool to compile a PTQL query into light-weight instrumentation on Java programs to answer that query.

Expressing the question above, *"Can method* `DB.doTransaction()` *transitively call method* `sleep()`*?"*, in PTQL avoids the problems that come with manual instrumentation. A query is written in one place and thus is much easier to understand and maintain, and furthermore does not clutter the program. Queries are also declarative. Finally, the programmer does not need not consider issues such as thread safety and recursion as those are left to Partiqle. Consider:

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

This PTQL query is looking for two method invocations, `doTrans` and `sleep`, where `doTrans` is a method named `doTransaction` defined in class `DB` and `sleep` is method named `sleep` defined in class `B`. Furthermore, `doTrans` and `sleep` should happen in the same thread and `sleep` should happen during `doTrans`. We discuss the details of PTQL in Section 2.

The contributions of this paper are as follows:

- We introduce PTQL (Section 2). PTQL is a declarative language, similar in spirit to SQL. With PTQL the user need only specify what data she wants and not worry about how to gather it. As in relational databases, this decision leaves the implementor free to choose efficient data representations and query evaluation plans.
- We describe a number of optimizations that we implemented in Partiqle (Section 3). These optimizations are critical to reducing the time and space overhead of evaluating queries as the program runs.
- We identify a class of queries that are amenable to online evaluation and describe how other queries can be split into several queries in this class.
- We report our preliminary experience with an implementation (Section 4). We used Partiqle to run several queries on 20 real Java programs, including Apache Tomcat [4]. Our queries also revealed significant (and apparently unknown) performance bugs in the *jack* SpecJVM98 [17] benchmark, in Tomcat, and in the IBM Java class library, and some uncomfortably clever code in the Xerces XML parser.

We examine related work in Section 5, discuss future work in Section 6, and conclude in Section 7.

2. Program Trace Query Language (PTQL)

This section describes PTQL, our SQL-like query language over Java program traces. A relational data model for program traces and an SQL-like language for querying them have several advantages:

- Program traces are naturally viewed as sets of records. Each record corresponds to a program event where the record's fields are properties of that event. Each type of event is a relation in the PTQL schema.
- Interesting properties of a program's execution lie in correlations of different events (i.e., relational joins).
- This view allows PTQL to be declarative, thus freeing the user from specifying how to gather data and freeing the implementor to choose efficient data representations and query evaluation plans. There are many well-known and successful optimizations for SQL which can aid us in optimizing PTQL. Optimization is very important as many natural queries produce overwhelming amounts of data if naively implemented.

Section 2.1 describes in more detail the relational schema over which PTQL interprets queries, Section 2.2 gives a formal semantics of PTQL, and Section 2.3 provides some example queries.

2.1 Data Model: Tables and Fields

Our current schema for a program trace consists of two relations:

- `MethodInvocation` contains a record for each method invocation that occurs during program execution.
- `ObjectAllocation` contains a record for each object allocated during program execution.

The fields currently defined in `MethodInvocation` and `ObjectAllocation` are listed along with their types in Figures 1 and 2 respectively. Fields of type `object` contain references to records in `ObjectAllocation` (i.e. anything that the Java type system could type as `Object`). Fields of type `variant` may contain values of any type. Fields are assigned this type because the type of values that they contain cannot be determined until query evaluation. Fields may be compared using any of `<`, `=`, or `>`. Records from `ObjectAllocation` may be compared with fields of type `object` or `variant`.

As we demonstrate in Section 4, our current data model is rich enough to express useful queries. Nonetheless, we designed it with extensibility in mind. In future work, we plan to investigate two dimensions of extensibility. First, adding other relations will allow PTQL to talk about different sorts of events like reads or writes to object fields, lock acquires and releases, and thread start and stop. Second, adding fields to existing relations will allow more inspection of program state when events fire. Examples include investigation of local variables on method end, and values of object fields at method start, method end, and object collection.

2.2 Formal Definition

In this section we sketch a formal semantics for PTQL. This section can be safely skipped, as subsequent discussion does not rely on it.

A query consists of three clauses (see Figure 3): a `FROM` clause, a `WHERE` clause and a `SELECT` clause. Query results are drawn from the cartesian product of the relations in the `FROM` clause. Let z be a tuple from this cartesian product. The *identifier*s in the `FROM` clause give each position in z a unique name. Using these names, the `WHERE` clause gives predicates that z must satisfy if it is to be included in query results. Finally, the `SELECT` clause specifies the fields from z to be output with each query result.

More formally, we define the semantics of a PTQL query applied to a program P in terms of two sets of records: $\texttt{MethodInvocation}^P$ and $\texttt{ObjectAllocation}^P$. The set $\texttt{MethodInvocation}^P$ contains one record, with all fields defined in Section 2.1, for each method call that occurs in evaluating program P. Similarly, $\texttt{ObjectAllocation}^P$ contains one record for each object allocated during the run of P. The timestamp fields guarantee that each record is unique and thus that $\texttt{MethodInvocation}^P$ and $\texttt{ObjectAllocation}^P$ are indeed sets.

We define the comparison operators so that fields with incompatible types are not equal, greater, nor less than each other and fields of type `object` are neither less than nor greater than each other. If the field `x.paramI` of "`MethodInvocation` x" is used in a query, only invocations of methods with at least `I+1` parameters can match `x`. Similarly, use of `x.result` means only methods whose return type is not `void` can match `x` and use of `x.receiver` means only non-static methods can match `x`.

Figure 4 gives the semantics of a PTQL query applied to program P. F is a vector of "*identifier.field*" pairs and is used in defining ψ. The helper functions field and pred are parameterized by ψ, the mapping from *identifier.field* to positions in the flattened tuple $z \in [\![T_1]\!]^P \times \cdots \times [\![T_n]\!]^P$. In equation 5, field takes $x.f$, an *identifier.field* pair and a flattened tuple z from $[\![T_1]\!]^P \times \cdots \times [\![T_n]\!]^P$; it returns the value of the field f from the table named x (at position $\psi(x.f)$) from z. In equations 6 and 7, pred takes a predicate from the `WHERE` clause and a flattened tuple from $[\![T_1]\!]^P \times \cdots \times [\![T_n]\!]^P$; it finds the semantic values of the left and right hand sides of the predicate, compares them according to the comparison operator, and finally returns a boolean value indicating whether they have the specified relationship. The final equation gives the semantics of a query applied to program P. For all $z \in [\![T_1]\!]^P \times \cdots \times [\![T_n]\!]^P$ that satisfy all the predicates in the `WHERE` clause, the projection of z, as specified in the `SELECT` clause, appears in the set of query results.

2.3 Example Queries

We conclude our discussion of PTQL with a few example queries.

2.3.1 *Actual parameters for each call to* `Foo.y`

```
SELECT   Y.param0, Y.param1
  FROM   MethodInvocation Y
 WHERE   Y.methodName = 'y'
   AND   Y.declaringClass = 'Foo'
```

For each call to methods named `y` declared in class `Foo`, this query returns a result containing the first two actual parameters of the call.

2.3.2 *Consistency of* `hashCode()` *with* `equals()`

The documentation for `java.util.Hashcode` [3] requires that implementations of the `hashCode()` method agree `equals()`. In particular, if `x.equals(y)` returns `true`, `x.hashCode() == y.hashCode()` should hold. This query checks that any calls to `hashCode()` and `equals()` follow this specification.

```
SELECT   *
  FROM   MethodInvocation eq, MethodInvocation xhc,
         MethodInvocation yhc
 WHERE   eq.methodName = 'equals'
   AND   eq.declaringClass = 'Object'
   AND   xhc.methodName = 'hashCode'
   AND   xhc.declaringClass = 'Object'
   AND   yhc.methodName = 'hashCode'
   AND   yhc.declaringClass = 'Object'
   AND   eq.receiver = xhc.receiver
   AND   eq.param0 = yhc.receiver
   AND   eq.result = true
   AND   xhc.result != yhc.result
```

In this query `eq` matches calls to `equals()`, and `xhc` and

- `startTime` : long - a unique timestamp for the start of the method invocation
- `endTime` : long - a unique timestamp for the end of the method invocation
- `methodName` : string - name of the method
- `declaringClass` : string - name of the class in which the method is first defined
- `implementingClass` : string - name of the class which implements this version of the method
- `receiver` : object - `this` parameter to the method (if non-static)
- `thread` : object - the thread in which the method is invoked
- `result` : variant- value returned by method
- `param0`, `param1`, ... : variant- values of the actual parameters to the method

Figure 1: Fields of `MethodInvocation`

- `startTime` : long - a unique timestamp for the allocation time of the object
- `endTime` : long - a unique timestamp for the collection of an object
- `allocThread` : object - the thread in which the object is allocated
- `dynamicType` : string - the class name of the object's runtime type
- `receiver` : object - the object

Figure 2: Fields of `ObjectAllocation`

$$
\begin{array}{lcl}
\langle \mathit{query} \rangle & ::= & \texttt{SELECT}\ \langle \mathit{selectitem} \rangle\ \ [,\ \langle \mathit{selectitem} \rangle]^* \\
 & & \texttt{FROM}\ \ \langle \mathit{fromitem} \rangle\ \ [,\ \langle \mathit{fromitem} \rangle]^* \\
 & & \texttt{WHERE}\ \ \langle \mathit{whereitem} \rangle\ \ [\texttt{AND}\ \langle \mathit{whereitem} \rangle]^* \\
\langle \mathit{selectitem} \rangle & ::= & \mathit{identifier.field} \\
\langle \mathit{fromitem} \rangle & ::= & \langle \mathit{relation} \rangle\ \mathit{identifier} \\
\langle \mathit{whereitem} \rangle & ::= & \mathit{identifier.field}\ \langle \mathit{op} \rangle\ \mathit{identifier.field} \\
 & | & \mathit{identifier.field}\ \texttt{=}\ \text{'}\mathit{string}\text{'} \\
\langle \mathit{relation} \rangle & ::= & \texttt{MethodInvocation}\ \mid\ \texttt{ObjectAllocation} \\
\langle \mathit{op} \rangle & ::= & \texttt{<}\ \mid\ \texttt{=}\ \mid\ \texttt{!=}\ \mid\ \texttt{>}
\end{array}
$$

Figure 3: Syntax of Query Language

$$
\begin{aligned}
[\![\texttt{MethodInvocation}]\!]^P &= \texttt{MethodInvocation}^P && (1)\\
[\![\texttt{ObjectAllocation}]\!]^P &= \texttt{ObjectAllocation}^P && (2)\\
F_x &= \langle x.f_1, \ldots, x.f_n \rangle && (3)\\
& \quad \text{where the fields of } T \text{ are } f_1, \ldots, f_n && (4)\\
\mathsf{field}^\psi\,(x.f,\ z) &= z.\psi(x.f) && (5)\\
\mathsf{pred}^\psi\,(x_1.f_1 < x_2.f_2,\ z) &= \mathsf{field}^\psi\,(x_1.f_1,\ z) < \mathsf{field}^\psi\,(x_2.f_2,\ z) && (6)\\
\mathsf{pred}^\psi\left(x_1.f_1 = {}'\texttt{string}',\ z\right) &= \mathsf{field}^\psi\,(x_1.f_1,\ z) = \text{“}\mathit{string}\text{”} && (7)
\end{aligned}
$$

$$
\left[\!\!\left[\begin{array}{ll}\texttt{SELECT} & s_1, \ldots, s_m \\ \texttt{FROM} & T_1\ x_1, \ldots, T_n\ x_n \\ \texttt{WHERE} & w_1, \ldots, w_k\end{array}\right]\!\!\right]^P = \left\{ \left\langle \mathsf{field}^\psi\,(s_1,\ z), \ldots, \mathsf{field}^\psi\,(s_m,\ z) \right\rangle \;\middle|\; \begin{array}{l} F = \mathit{concatenate}(F_{x_1}, \ldots, F_{x_n}) \\ \psi(y.f) = i \ \text{ iff } F.i = y.f \\ z \in [\![T_1]\!]^P \times \cdots \times [\![T_n]\!]^P \wedge \bigwedge_j \mathsf{pred}^\psi\,(w_j,\ z) \end{array} \right\}
$$

Figure 4: Semantics of Query on Program P

yhc match calls to hashCode(). The query is interested in a concordance of events such that that the receivers of the calls to hashCode() are the receiver and first parameter to the call to equals(). For such a group of events, the specification requires that if the call to equals() returns true, the calls to hashCode must agree. This query returns results where the calls to hashCode do not agree.

3. Partiqle: INSTRUMENTATION AND OPTIMIZATIONS

This section discusses Partiqle, our tool to compile a PTQL query into light-weight instrumentation to answer the query while the program runs. We outline our instrumentation strategy, describe runtime support structures needed to evaluate queries online, and discuss optimizations that reduce execution time overhead and memory footprint.

In designing Partiqle, we had to choose between offline analysis (logging events to a trace file, and analyzing the trace file post-mortem) and online analysis (and design points in between).

Offline evaluation of the query allows for a constant sized memory footprint as events are gathered during program execution. However, in practice, the post-mortem processing of large traces usually requires similar resources to simply doing the analysis on-line; in particular, because random accesses to disk are very slow, efficient analyses must read traces sequentially. The main advantages of offline evaluation are that the analysis need not compete with the application for space, and offline analyses can read the trace sequentially multiple times — clever algorithms can sometimes be designed to take advantage of this [16].

We prefer on-line processing whenever reasonable performance can be obtained from an on-line algorithm. Even though disk is cheap, managing large volumes of trace data can impose considerable overhead. On-line query evaluation presents a simpler model to the user by eliminating post-processing steps. Online evaluation can also provide the quickest feedback, keeping the code-debug cycle short. Another advantage is the ability to stop the program when certain behaviours are detected and start a debugger or dump a stack trace. For these reasons, plus the fact that we saw opportunities to optimize and reduce our overhead to below the minimum overhead of off-line analysis, we chose to implement Partiqle as an online analysis engine. However, one can easily imagine implementing PTQL queries using offline analysis, or even extending Partiqle with automatic or manual selection of the degree of off-line-ness.

To ease development and deployment of Partiqle, we instrument the program at the level of Java bytecodes and write our instrumentation in Java. This creates reentrancy issues, which we resolve by avoiding the use of most Java library classes and refusing to instrument those library classes we do use. In theory Partiqle is usable in conjunction with any Java virtual machine, and in practice we use it with both Sun and IBM VMs.

Section 3.1 outlines Partiqle's basic instrumentation strategy, runtime data structures, and query evaluation strategy. Sections 3.2, 3.3 and 3.5 describe optimizations. Section 3.8 applies Partiqle's instrumentation, runtime data structures, and optimizations to an example query.

```
Object method(Object arg0, Object arg1) {
    get global lock;
    MethodDescriptor mdescr = new MethodDescriptor(
        this,
        array of arguments to method,
        statically determined method id for method
    );
    add mdescr to runtime tables;
    release global lock;

    /* method may terminate with an exception */
    try {
        method body
        store return value in retval;
    } catch (Throwable e) {
        /* end-of-method code for exception case */
        get global lock;
        mdescr.setEndTimeExceptionResult();
        release global lock;
        throw e; /* rethrow e */
    }

    /* end-of-method code for regular termination */
    get global lock;
    mdescr.setEndTimeAndResult(returnValue);
    release global lock;
    return retval;
}
```

Figure 6: Baseline instrumentation for a method

3.1 Instrumentation

In order to answer a PTQL query over program P, Partiqle must instrument P to gather records that match the various events specified in the query. For each event, Partiqle must include instrumentation to record the fields PTQL specifies. In practice, many records never need to be generated and many fields never need to be set. Subsequent sections discuss optimizations that will augment, change or discard this instrumentation to take advantage of this situation.

3.1.1 Method Invocations

Figure 6 shows a baseline for instrumentation to gather method invocations (i.e., records in MethodInvocation). This instrumentation is thread safe: operations on shared data structures are protected by a global lock. At the start of the method, this instrumentation records the start time (field startTime), thread (thread), actual parameters (param0 and param1) and this pointer (receiver). At the end, it records the return value (result) and end time (endTime). If the method invocation ends with an exception, the instrumentation records that fact, sets endTime and rethrows the exception.

Java dictates that in a constructor, the 'this' reference is not accessible until after the superclass constructor has been called. This is also enforced at the bytecode level. Thus, in constructors, the receiver field is not available until some time during the invocation of the method. This unfortunately complicates some of the analyses described below, although the details are tedious and beyond the scope of this paper.

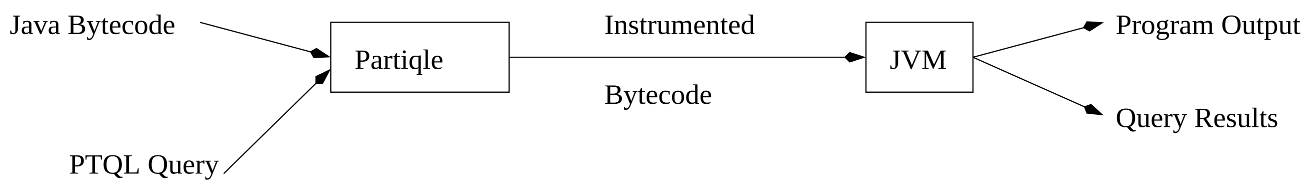


Figure 5: Partiqle Architecture

3.1.2 Objects

Gathering information about object lifetimes is harder than for method invocations because more code locations are involved. The semantics of Java also adds some complications.

We want to add a record to our table(s) as soon as an object has been created. The first Java bytecode instructions executed after the object is created and accessible are in the constructor of `java.lang.Object`. Unfortunately instrumenting this method causes every JVM we have access to to crash. Therefore our strategy is to insert instrumentation right after every call to the `Object` constructor — either in constructors for direct subclasses of `Object`, or in normal code constructing a plain `Object`. Arrays are allocated with a different instruction sequence, and no constructor is called, so we instrument them separately.

To map Java objects to our records without necessarily causing space leaks, we maintain a hash table whose keys are weak references to the Java objects and whose values refer to our records.

We need a notification when objects are garbage collected. Java's "reference queue" mechanism notifies Partiqle whenever one of the weak references in the hash table loses its referent, at which point Partiqle executes the "end event" code for the object.

Creating one record for every single Java object is usually quite impractical. Fortunately, most queries do not refer to the information available only at allocation time (`allocThread` and `startTime`), and constrain the object to be some parameter or result of a method invocation. For these queries it suffices to allocate an object's record lazily, when the method invocation first constrains the object and makes it relevant to the query. This is particularly advantageous when optimizations severely reduce the number of methods instrumented, because most objects then never need records.

3.1.3 Partiqle Runtime Data Structures

Partiqle's runtime data structures must store event records until query evaluation. Partiqle keeps one *runtime table* per ⟨*relation*⟩ *identifier* pair in the `FROM` clause of the query. Each of these runtime tables is a collection of records that potentially satisfy the predicates associated with that slot in the query. For example the "Does `DB.doTransaction()` transitively call `sleep()`?" query from Section 1 has two runtime tables: one for invocations of `doTrans()` and one for invocations of `sleep()`.

In our current implementation, runtime tables of `MethodInvocation` records are indexed by the `receiver`, `param0`, and `result` fields. Runtime tables of `ObjectAllocation` records are not indexed. In future work, we plan to choose index fields based on predicates in the query.

The data gathering instrumentation creates records and adds them to suitable runtime tables. Note that there several instrumentation sites may generate records for a runtime table. A single instrumentation site may generate records for multiple runtime tables; in this case, a single record is allocated and shared among them.

Based on the discussion above, a runtime table must support the operations listed below. For completeness, we mention operations required by query evaluation as well as those required for optimizations:

- add a record
- update some fields of a record, e.g. `endTime`, `result`
- join a record into a partial query result (Section 3.1.4)
- check for existence of a record that satisfies some predicate (i.e., allow other runtime tables to do admission and retention checks – Section 3.5)
- delete a record (Section 3.5)

3.1.4 Query Evaluation

Query evaluation proceeds in a nested loop. At instrumentation time, Partiqle decides on the order in which to join the runtime tables. Thus, at query evaluation time, as records from each runtime table are considered in turn, Partiqle knows exactly which fields will be in the partial query result, which fields it will contribute, which of its indices it will use, and thus which predicates to evaluate – those that involve a record from the current runtime table and a record already in the partial result. Each time a new record is appended to the partial result, it knows which runtime table is next in the join order and loops through that runtime table, looking for records to join in.

3.2 Static Filtering

Predicates in the query that depend only on static properties of the code allow Partiqle to filter instrumentation sites. We refer to such predicates as *static predicates*. If an instrumentation site violates a static predicate, Partiqle need not insert instrumentation at that site. The predicates that Partiqle uses in this way are comparisons of the `methodName`, `declaringClass` and `implementingClass` fields in `MethodInvocation` records with constant strings, and comparisons of the `dynamicType` field in `ObjectAllocation` records with constant strings.

Consider for instance the example query from Section 1. One `MethodInvocation` record in the query is constrained to be named `doTransaction` (`doTrans.methodName = 'doTransaction'`) and the other `sleep` (`sleep.methodName = 'sleep'`). Thus, invocations of method `y` will never have any part in query results and Partiqle need not instrument the body of `y`.

This optimization is quite straightforward to implement at `MethodInvocation` instrumentation sites, because the method name, defining class and implementing class are all apparent from the method being instrumented. Static filtering on `dynamicType` is only possible at sites where enough is known about the static type of the object reference in question, and enough is known about the program's class hierarchy, to statically determine whether the object reference refers to an object of the desired class. To support these decisions, Partiqle builds a partial class hierarchy based on the code available at instrumentation time, making conservative (safe) approximations for unknown code.

3.3 Dynamic Filtering

Query predicates that involve only one record can be evaluated at the instrumentation site that sets the relevant fields of that record. We refer to these predicates as *simple dynamic predicates*. For instance consider the following query which lists all method invocations where the `this` pointer is the same as the first parameter:

```
SELECT  *
  FROM  MethodInvocation f
 WHERE  f.param0 = f.receiver
```

The instrumentation at the start of each method checks that the first parameter to the function is equal to the `this` pointer. If not, the record can never be part of a query result.

Sometimes the fields necessary to evaluate a simple dynamic predicate are not available when the record is generated. In this case the record is added to the runtime tables as usual. Later, when the missing fields become available, the predicate is evaluated. If it fails, the record is removed from the runtime tables. Consider the following example which lists all method invocations which return their `this` pointer:

```
SELECT  *
  FROM  MethodInvocation g
 WHERE  g.result = g.receiver
```

At the start of a method, a record will be added to the runtime table. Since `result` is not available until the end of the method, this predicate cannot be checked until then. If it fails, the record is removed from the table.

3.4 Timing Analysis

The optimizations to be described next require information about the ordering of the events in a query result. Partiqle performs *timing analysis* to compute this information and stores it as a *timing graph*. The timing graph is a directed acyclic graph with two nodes for each runtime table – one for the *start event* (the beginning of a method invocation or the allocation of an object) and one for the *end event* (the end of a method invocation or the garbage collection of an object). An edge from node x to node y indicates that event x must happen before event y for the events to satisfy the query. For example, if the query contains a term `a.startTime < b.startTime` then there will be an edge from a.START to b.START in the timing graph. Because timestamps are totally ordered, the graph is transitively closed.

Figure 7 shows the timing graph for the example from Section 1. In addition to edges induced by explicit constraints in the query, Partiqle infers edges using axioms about the semantics of Java. In this example, the dotted edges from doTrans.START to doTrans.END and from sleep.START to sleep.END follow from the axiom that the start of a method invocation always precedes the end of that method invocation. The dashed edges from doTrans.START to sleep.END and from sleep.START to doTrans.END follow from transitivity.

The complete rules for building the timing graph are given in Figure 8. These rules are applied repeatedly until a fixpoint is reached.

For some optimizations, we also want an "unclosed" form of the graph. This is a minimal graph whose closure under the "transitive closure", "end follows start" and "overlapping method invocations" rules gives the basic timing graph. It can be obtained from the basic timing graph by repeatedly applying the rules:

- If $(a,b) \in E \wedge (b,c) \in E \wedge (a,c) \in E$, remove (a,c) from E
- For all query identifiers x, remove $(x.\text{START}, x.\text{END})$ from E.
- For method invocations x and y, if $(x.\text{START}, y.\text{START}) \in E \wedge (y.\text{START}, x.\text{END}) \in E \wedge$ "$x.\texttt{thread} = y.\texttt{thread}$" $\in Q \wedge (y.\text{END}, x.\text{END}) \in E$, remove $(y.\text{END}, x.\text{END})$ from E.

The minimal graph is not unique, but this is not a problem.

The idea is that if the timing relationships in the "unclosed" graph are dynamically verified for a set of events, then the rest of the timing relationships are guaranteed to hold for those events.

3.5 Admission Checks

We refer to query predicates that cannot be evaluated statically and that involve more than one record as *join predicates*. Armed with timing information, Partiqle adds instrumentation to check some join predicates when new records are created. We refer to these checks as *admission checks* because they deny a record admission to a runtime table if it cannot possibly satisfy a join predicate.

Before describing admission checks in detail, we return again to the example query from Section 1. Notice the join predicate "`doTrans.startTime < sleep.startTime`" and suppose the instrumentation at the start of `sleep()` is now executing (i.e., an invocation of `sleep()` is starting). If

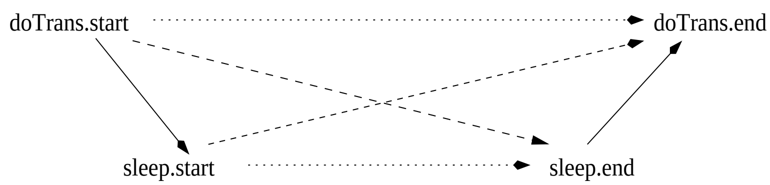


Figure 7: Timing graph for example query from Section 1

Let Q be the set of query whereitems. Let E be the set of timing graph edges.
Explicit timing constraints induce timing edges. For all identifiers x, y:

$$\begin{aligned}
\text{“}x.\texttt{startTime} < y.\texttt{startTime}'' \in Q &\implies (x.\text{START}, y.\text{START}) \in E \\
\text{“}x.\texttt{endTime} < y.\texttt{startTime}'' \in Q &\implies (x.\text{END}, y.\text{START}) \in E \\
\text{“}x.\texttt{startTime} < y.\texttt{endTime}'' \in Q &\implies (x.\text{START}, y.\text{END}) \in E \\
\text{“}x.\texttt{endTime} < y.\texttt{endTime}'' \in Q &\implies (x.\text{END}, y.\text{END}) \in E
\end{aligned}$$

The lifetime of the 'this' parameter of a method includes the method invocation. For all method invocations m and object instances o:

$$\text{“}m.\texttt{receiver} = o.\texttt{receiver}'' \in Q \implies$$
$$(o.\text{START}, m.\text{START}) \in E \wedge (m.\text{END}, o.\text{END}) \in E$$

The lifetime of an object mentioned as a parameter or result of a method must include the start (or end) of the method. For all method invocations m and object instances o, and all integers n:

$$\text{“}m.\texttt{paramn} = o.\texttt{receiver}'' \in Q \implies$$
$$(o.\text{START}, m.\text{START}) \in E \wedge (m.\text{START}, o.\text{END}) \in E$$
$$\text{“}m.\texttt{result} = o.\texttt{receiver}'' \in Q \implies$$
$$(o.\text{START}, m.\text{END}) \in E \wedge (m.\text{END}, o.\text{END}) \in E$$

The timing graph is transitively closed. For all nodes a, b, c:

$$(a, b) \in E \wedge (b, c) \in E \implies (a, c) \in E$$

End events follow start events. For query identifiers x:

$$(x.\text{START}, x.\text{END}) \in E$$

Overlapping method invocations on the same thread must actually be nested. For all identifiers x, y corresponding to method invocations:

$$(x.\text{START}, y.\text{START}) \in E \wedge (y.\text{START}, x.\text{END}) \in E$$
$$\wedge \text{“}x.\texttt{thread} = y.\texttt{thread}'' \in Q \implies (y.\text{END}, x.\text{END}) \in E$$

Figure 8: Timing Edge Inference

this `sleep` is to satisfy the join predicate above, then according to the timing graph any record for an invocation of `doTransaction()` that can match with this `sleep` must already have started. So, at the start of `sleep()` we check to see if any suitable "supporting" record `doTrans` has been stored in its table; if not, this `sleep` can never be part of a query result and it can be discarded.

If the query includes additional constraints relating `doTrans` and `sleep`, for example "`doTrans.param0 = sleep.param0`", then these constraints are evaluated as part of the admission check; the check fails unless a matching `doTrans` record is found. Join predicates such as `doTrans.param0 = sleep.result`, which depend on information available at the end of `sleep`, cannot be evaluated by the admission check. However, we can defer such predicates to a "retention check"; at the end of the method invocation (or object lifetime), when the result is known, we check for supporting `doTrans` records and if none are found we discard the invocation `sleep`.

There are a wide range of strategies that can be used to exploit admission checks. Partiqle inserts admission checks at each instrumentation point (i.e., a start or end event). At each timing graph node e we perform admission checks against the tables corresponding to the nodes e' which are immediate predecessors of e in the *unclosed* timing graph. The admission check succeeds if there is a record in the runtime table for e' satisfying all join predicates between e and e' that depend only on fields available at e and e'. If any admission check fails, e's record is discarded from the table.

3.6 The Post-dominator

Efficient dynamic analysis often requires that results be output on-line. Otherwise intermediate data structures may become too large or even grow without bound. Our *post-dominator analysis* allows us to output results on the fly and prune intermediate data structures.

The post-dominator analysis identifies a node in the timing graph, the post-dominator node d, with the property that when an event e occurs at d, all record tuples that will satisfy the query and include the record for e can be computed from the records currently in tables. In other words, we guarantee that no future records will arrive which could combine with the record for e to produce a valid query result.

We ensure this by imposing the following conditions on d:

- There is a path from every start event to d in the timing graph.
- If the query selects a field for output that is only available at an end event, then there is a path from that end event's node to d.
- If the query contains a predicate depending on a field value which is only available at an end event, and the predicate is not a comparison of the event's end time with some other time, then there is a path from that end event's node n to d in the timing graph.
- If the *unclosed* timing graph has an edge from node p to node q, then there is a path from p to d in the timing graph.

The first condition ensures that all the records in a result tuple that can match a record at d will have at least started by the time the d event occurs — otherwise results will certainly be missed. The second condition ensures that the values output by the query are actually available by the time of the d event. The third condition ensures that when the query expression is evaluated on each tuple of records, the fields required by the predicates are available. The exception is when comparing the end time of an event e to the time of some other event; instead of doing a direct comparison, we can simply verify all the timing constraints in the unclosed timing graph. If they hold, we know that all the constraints in the full timing graph also hold (which will include all the time comparison constraints derived from the query predicates).

Note that if p and q are nodes, where p has a path to d in the timing graph but q does not, then when an event occurs at d we can dynamically check the relationship between the timestamps of all p events and q events relevant to the d event. For, we know that the timestamps for all relevant events at p must be in the past (because d dominates p) and are therefore available in the records in p's table. The timestamps for all relevant events at q must be either in the past, in which case they can be retrieved from the records of q's table and compared to the relevant p timestamps, or in the future, in which case we know that the timestamps on those events are strictly greater than the relevant p timestamps. (These two cases can be distinguished at runtime because until a record's end event happens, the `endTime` field holds a sentinel value.)

In the example from Section 1 and Figure 7, the node sleep.START is a post-dominator. The unclosed timing graph contains just the two edges (doTrans.START, sleep.START) and (sleep.START, doTrans.END). It is easy to verify that the required conditions hold. The remarkable thing is that although the query means to check sleep.`endTime` < doTrans.`endTime`, Partiqle infers that it can output results before either event has happened. This is an example of where starting with the temporal predicates from the query, closing the timing graph, and then unclosing it leads to a more minimal graph than the original.

If there is no post-dominator node, Partiqle currently reports an error and halts. In the future this could be relaxed, so that we switch to off-line analysis. If there is more than one post-dominator node, Partiqle chooses an "earliest post-dominator" — a post-dominator node d such that there is no path from any other post-dominator d' to d in the timing graph. If there are multiple earliest post-dominators, Partiqle chooses one arbitrarily.

When we execute an event e associated with a post-dominator node d and a record identifier x, we go ahead and evaluate the query, with x bound to the record for e and the other identifiers ranging over the current contents of their tables. Instead of checking explicit predicates on timestamps, which might require time values we do not yet have, we check the timing constraints of the unclosed timing graph. We output any query results found. We will have output all results that can ever involve the record for e. It is then safe to remove the record from x's table. if e is a start event, we never need to add the record to x's table; in fact, x's table will always be empty. Notice that this removal allows Partiqle to infer that some retention checks will always fail.

If a query has no post-dominator, a simple transformation can split it into several queries that do have post-dominators. The union of the results of these queries is the same as the results of the original query. The transformation on query q, generates one query, q_s, for each sink s in the timing graph. The query q_s is defined to be the query q with the additional constraint that s comes after all the other sink events in the timing graph.

3.7 Deletion Propagation

Suppose there is a predicate relating records in the runtime tables `xs` and `ys`. When Partiqle removes a record from `xs`, some records in `ys` may be unable to participate in further query results, because they can only match with the `xs` records that have been deleted. (In other words, `ys` records were subject to admission checks that only passed because of the presence of records in `xs` that have now been deleted.) Ideally, Partiqle would drop such records from `ys` immediately. We refer to this removal as *deletion propagation*.

In practice however, discovering opportunities for deletion propagation can require a large number of runtime checks. Partiqle takes a conservative approach and empties `ys` if `xs` has just been emptied and there was an admission check for `ys` records against `xs` records. This decision was made for ease of implementation; this is an important area of investigation in future work.

Record removal at the post-dominator (Section 3.6), retention checks (Section 3.5), and deletion propagation work together to remove records from the runtime tables once all results involving those records have been output.

3.8 Example of Optimized Instrumentation

In this section we show the instrumentation that Partiqle would add to the code from Section 1 to answer the example query from Section 1.

This query requires two runtime tables: `xs` for `MethodInvocation doTrans` and `zs` for `MethodInvocation sleep`. Based on the static predicates

```
     doTrans.methodName = 'doTransaction'
AND  doTrans.declaringClass = 'DB'
AND  sleep.methodName = 'sleep'
AND  sleep.declaringClass = 'B'
```

only `DB.doTransaction()` needs to be instrumented to add records to `xs` and only `sleep()` needs to be instrumented to add records to `zs`.

As discussed in Section 3.6, the start event for `sleep()` is the post-dominator for this query. Therefore, the instrumentation at the start of `sleep()` creates a record, Z, and then computes and outputs query results involving Z; that is, for each record X in `xs` with X.`thread` = Z.`thread`, output (X.`startTime`, Z.`startTime`). The timing constraints need not be checked at query evaluation since they are always satisfied (records in `xs` are for calls to `DB.doTransaction()` that have started, but not completed). Since all query results involving Z are output at the start of `sleep()`, Z need not be recorded. In fact, no table `zs` is actually necessary.

Since `zs` is always empty, the retention check at the end of `DB.doTransaction()` will always fail. The instrumentation at the end of `DB.doTransaction()` removes the record from `xs`.

4. RESULTS

4.1 Benchmarks

We present several examples of the use of Partiqle on real programs. Our suite of benchmark programs is shown in Table 1. They include the SpecJVM98 benchmark suite [17], a variety of other benchmarks, and the Apache Tomcat [4] version 4 Web server and servlet container (which includes the Xerces XML parser and other components). The code size is reported as the number of methods in the application. However, we also instrument the Java class library, so the actual code subject to instrumentation is very much larger than reported here (although hard to directly measure).

Except for Tomcat, we ran the programs on inputs provided; we used the largest input size available for the SpecJVM98 benchmarks. For Tomcat, we gathered a list of all URLs to pages under Tomcat's "examples" directory and wrote a harness that loads these pages sequentially, running through the complete list twice. This exercises a number of JSPs and servlets.

4.2 Queries

We constructed several queries aimed at finding correctness or performance bugs in Java code.

- *HashCodeConsistent* checks that `hashCode` called on the same object always returns the same value. Violations of this rule would cause problems if the object was stored as a key in some data structure.

  ```
  SELECT  first.result, second.result
    FROM  MethodInvocation first,
          MethodInvocation second,
          ObjectAllocation obj
   WHERE  first.methodName = 'hashCode'
     AND  second.methodName = 'hashCode'
     AND  first.receiver = obj.receiver
     AND  second.receiver = obj.receiver
     AND  first.result != second.result
  ```

  Introducing an explicit query variable for the object allows the end event for the object to be a post-dominator for the query. This simple transformation would be easy to automate.

- *EqualObjectsButInequalHashCodes* checks that if two objects were deemed equal by `equals`, then they have the same `hashCode`. This is an important invariant of these methods.

Example	Methods	Description
db	35	small database management program (SpecJVM98)
compress	44	Java port of LZW (de)compression (SpecJVM98)
lisp	104	Lisp interpreter
jscheme	110	Scheme interpreter
MipsSimulator	112	architectural simulator
raytrace	180	ray-tracing program (SpecJVM98)
mtrt	184	multi-threaded ray-tracing program (SpecJVM98)
mpegaudio	280	MP3 decoder (SpecJVM98)
jack	313	Java parser generator (SpecJVM98)
jess	673	expert shell system (SpecJVM98)
javac	1179	JDK 1.0.2 Java compiler (SpecJVM98)
tomcat	16940	Apache Web application server (v4.0.4)

Table 1: Benchmark programs

- *InequalObjectsButEqualHashCodes* checks that if two objects are not deemed equal by `equals`, then their hash codes should be different. Violation of this rule is not strictly speaking a bug, but could lead to performance problems due to hash collisions. The query is very similar to the previous query.

- *StringConcats* searches for the anti-pattern

```
String s = ...;
for (...)  { s = s + ...; }
```

This code can induce $O(n^2)$ performance where n is the length of the final string. Avoiding this problem is listed as "Best Practice 11" in an IBM white paper [18]. We look for the result of a call to `StringBuffer.toString` being passed to the constructor of another `StringBuffer`, then the result of that `StringBuffer`'s `toString` being passed to construct another `StringBuffer`, and so on. Our actual query looks for a chain of five such constructions. For the sake of brevity, here is the code for a chain of two constructions:

```
SELECT  c2.param0
  FROM  MethodInvocation ts1,
        MethodInvocation c1,
        MethodInvocation ts2,
        MethodInvocation c2
 WHERE  ts1.methodName = 'toString'
   AND  ts1.implementingClass = 'StringBuffer'
   AND  c1.methodName = '<init>'
   AND  c1.implementingClass = 'StringBuffer'
   AND  ts2.methodName = 'toString'
   AND  ts2.implementingClass = 'StringBuffer'
   AND  c2.methodName = '<init>'
   AND  c2.implementingClass = 'StringBuffer'
   AND  ts1.result = c1.param0
   AND  ts1.endTime < c1.startTime
   AND  c1.receiver = ts2.receiver
   AND  c1.endTime < ts2.startTime
   AND  ts2.result = c2.param0
   AND  ts2.endTime < c2.startTime
```

Here the timing constraints are necessary to prevent ambiguities, as well as helping to ensure a post-dominator exists (c2.START, in this case). For example, the intuition that the result of `ts1` is passed as a parameter to `c1` means not only that the value is the same but `c1` starts after `ts1` returns. In general it might be possible for `ts1` to return a `String` that previously existed and had been used to construct a `StringBuffer`, so it is important to stipulate that `ts1.endTime < c1.startTime`. It is easy to accidentally under-constrain a query this way. Fortunately this often leads to no post-dominator node being found and Partiqle issuing a warning.

- *DelayedClose* searches for `close` operations on stream objects that have not been read or written to for a certain length of time. Such streams could be considered resource leaks; they should be closed as soon as the application has finished using them. This query was inspired by "Best Practice 8" in an IBM white paper [18]. This query requires some extensions to PTQL not described in detail in this paper:

  - "startRealTime" and "endRealTime" fields that record actual wall clock timestamps for events, in milliseconds
  - Simple arithmetic expressions (+)
  - String pattern matching expressions (=˜)
  - Simple negation: the ability to specify that a query variable have no matches in order for the rest of the tuple of records to satisfy the query (`NEGATIVE`)

```
SELECT  close.receiver0
  FROM  MethodInvocation rw,
        NEGATIVE MethodInvocation norw,
        MethodInvocation close
 WHERE  close.methodName = 'close'
   AND  close.implementingClass =˜ 'org.apache.*'
   AND  rw.methodName =˜ 'read|write'
   AND  rw.implementingClass =˜ 'org.apache.*'
   AND  norw.methodName =˜ 'read|write'
   AND  norw.implementingClass =˜ 'org.apache.*'
   AND  rw.receiver = norw.receiver
   AND  rw.receiver = close.receiver
   AND  rw.endTime < norw.endTime
   AND  norw.endTime < close.startTime
   AND  close.realStartTime > rw.realEndTime + 10000
```

We constrain the search to Apache stream classes because applying it to basic Java streams causes

```
public class A {
   B b;
   //...
   void doTransaction() {
      get global lock;
      MethodDescriptor X = new MethodDescriptor(
         this,
         null,  /* no arguments */
         1  /* method id for doTransaction */
      );
      xs.add(X);
      release global lock;

      try {
         b.y();
      } catch (Throwable e) {
         get global lock;
         xs.delete(X);
         release global lock;
         throw e;
      }
      get global lock;
      xs.delete(X);
      release global lock;
   }
}

public class B {
   //...
   void y() { //method y is unchanged
      sleep();
   }
   void sleep() {
      get global lock;
      MethodDescriptor Z = new MethodDescriptor(
         this,
         null,  /* no arguments */
         2      /* method id for sleep */
      );
      output query results for Z;
      release global lock;

   }
}
```

Figure 9: Optimized instrumented code for example from Section 1

reentrancy problems for the Partiqle runtime support library.

- `CompareToReflexive` searches for `Comparable` objects o which return nonzero from a call to o.`compareTo`(o).
- `CompareToAntisymmetric` searches objects x and y such that the sign of x.`compareTo`(y) $\neq$ minus the sign of y.`compareTo`(x). Because Partiqle currently lacks a "sign" function, we map this to three queries covering the cases where x.`compareTo`(y) < 0, $= 0$, or > 0. The first case is the one we report results for here.
- `CompareToTransitive` searches for objects x, y and z whose `compareTo` methods violate transitivity. Currently PTQL requires this query to be split into two queries: one where all of x.`compareTo`(y), y.`compareTo`(z) and z.`compareTo`(x) are all non-negative and at least one is positive (without loss of generality, we take x.`compareTo`(y) to be positive), and one were they are all non-positive and at least one (x.(compareTo)(y)) is negative. Furthermore, to ensure there is a post-dominator node each of those queries needs to be split into three cases, depending on which `compareTo` method call is constrained to be last. We report results for the query covering the case where x.`compareTo` is positive and called last.

4.3 Overhead of Instrumentation

We measured the baseline performance of our benchmarks without instrumentation and compared them to the performance when instrumented for each of the queries. We recorded the wall-clock running time of each run, and also the heap memory high-water mark (measured by sampling Java's `System.totalMemory() - System.freeMemory()` every 500 milliseconds). The experiments were carried out on an unloaded 2.4 GHz Pentium IV with 1.5GB of memory, running IBM's JDK 1.4.0 on Red Hat Linux 7.3.

Table 2 shows the runtime overhead as a ratio of the runtime with instrumentation to the runtime without instrumentation. Table 3 shows the memory overhead as a ratio of the memory high-water-mark with instrumentation to the high-water-mark without instrumentation. `jess` and `javac` took too long on a couple of the queries and had to be terminated. More work is required on these benchmarks, in particular, to optimize the speed of query processing.

These results show that, with the exception of a few outliers, the overheads are acceptable. Jitter in the results — especially where the instrumented code runs faster or in less space than the uninstrumented code — seems to be due to changes in the garbage collection or JIT behaviour, which can be sensitive to small changes in program behaviour (especially for small and short-lived benchmarks as most of ours are).

4.4 Effects of Optimizations

Figure 10 shows the time overheads on Tomcat with full optimization on, with admission checks turned off (i.e., we assume that the admission checks always succeed without executing them), and with the post-dominator node turned off (so records are queued up and all processed at the end of

Query	db	compress	lisp	jscheme	Mips	raytrace	mtrt	mpeg	jack	jess	javac	tomcat
HashCodeConsistent	0.97	1.00	1.00	1.00	1.00	0.91	1.00	1.00	1.00	1.06	2.23	5.35
EqualButInequalHash	1.81	0.97	1.00	1.12	1.01	1.00	0.92	1.00	3.86	—	—	4.65
InequalButEqualHash	3.40	0.97	1.00	1.08	1.00	0.91	0.92	1.00	28.44	—	68.14	1.70
StringConcats	1.00	0.97	1.00	1.17	1.15	1.00	1.00	1.00	1.97	1.12	160.40	46.91
DelayedClose	0.97	0.97	1.00	1.08	1.00	1.00	1.00	1.00	0.95	1.05	0.97	1.17
CompareToReflexive	1.01	1.00	1.00	1.00	1.00	0.92	1.00	1.00	1.00	1.06	0.97	0.96
CompareToAnti	1.03	0.97	1.00	1.08	1.00	0.92	1.00	1.00	0.95	1.05	1.03	0.97
CompareToTransitive	0.98	0.97	1.00	1.03	1.00	0.92	0.92	1.00	1.00	1.00	0.97	1.00

Table 2: Runtime Overhead

Query	db	compress	lisp	jscheme	Mips	raytrace	mtrt	mpeg	jack	jess	javac	tomcat
HashCodeConsistent	1.02	1.09	1.05	0.88	1.00	0.93	0.75	1.01	0.86	1.08	5.51	2.41
EqualButInequalHash	1.96	1.09	1.43	1.07	1.00	0.94	0.86	1.01	2.52	—	—	2.06
InequalButEqualHash	46.15	1.09	1.04	1.05	1.00	1.01	0.93	1.01	202.49	—	29.83	9.35
StringConcats	1.10	1.09	1.37	1.32	1.50	0.92	0.72	1.22	4.35	2.02	8.98	16.97
DelayedClose	1.18	1.06	1.18	0.93	1.01	1.00	0.97	1.00	0.97	0.76	1.01	2.78
CompareReflexive	1.20	1.02	1.15	1.00	1.02	1.03	0.87	1.00	0.85	1.01	1.01	1.07
CompareAnti	1.00	1.00	1.01	0.98	1.00	0.97	0.73	1.00	0.95	1.08	0.99	0.89
CompareTransitive	1.16	1.06	1.02	0.86	1.01	0.84	0.92	1.00	0.94	1.09	1.01	0.89

Table 3: Memory Overhead

the program run) but admission checks on. Figure 11 shows the corresponding memory overheads.

These results show that most of the time the post-dominator makes little difference in time, that it is essential for one query, but really hurts `StringConcats`! We also see that surprisingly, using the post-dominator increases measured memory usage — surprising because the post-dominator is supposed to allow us to discard records from tables. Profiles show that this increased memory usage is due to our query evaluator's intensive traversals of Java collections, which require the allocation of a very large number of iterators. (For example, before Tomcat has served a single page, Partiqle has already allocated over 1GB of iterators.) When partial query results are being computed frequently at the post-dominator's program point, enormous amounts of memory are being allocated — and quickly collected — but the virtual machine's GC heuristics are allowing the heap to grow quite large before collection. Without the post-dominator, at the end of the program we need to traverse more records, but only once, so the cost of the iterators is greatly amortized.

We believe that the high time overhead for `StringConcats` induced by the post-dominator is related to this problem. The massive and frequent allocation of short-lived iterators seems to interfere with the operation of the VM and slow down the application severely. Obviously a high priority for future work is to overhaul the query evaluator for high speed and minimal allocation.

The results also show that admission checks generally make a small improvement in time, but in some cases (`StringConcats`) they lead to a large improvement in space.

4.5 Query Results Found

Our queries discovered several interesting program behaviours. When Partiqle detects a query result being produced at a post-dominator node, it produces a stack trace for the current event to aid diagnosis. (The usefulness of these stack traces in diagnosing faults is one advantage of using post-dominators to output results incrementally instead of post-mortem.) These issues could have been found with custom dynamic analysis or even in some cases simple static analysis; however, writing PTQL queries is an extremely quick way to look for new kinds of behaviours.

- Applying `StringConcats` to the `jack` benchmark found a classic poorly-performing `String` concatenation loop. Unfortunately the loop is in the heart of the `jack` lexer: `jack` builds tokens by appending one character at a time to a `String`! This is $O(n^2)$ in the length of the tokens. Despite `jack` being an extremely well-studied benchmark, we are not aware of anyone previously having reported this bug.

- Applying `HashCodeConsistent` to `tomcat` found a situation in the Xerces XML parser where `org.apache.xerces.validators.common.CMStateSet` objects were returning different `hashCodes` at different times. It turns out that `org.apache.xerces.validators.common.DFAContentModel` has an algorithm that looks like this:

```
CMStateSet newSet = null;
HashMap states = new HashMap();
for (...) {
    if (newSet == null) {
        newSet = new CMStateSet();
    } else {
        newSet.clear();
    }
```

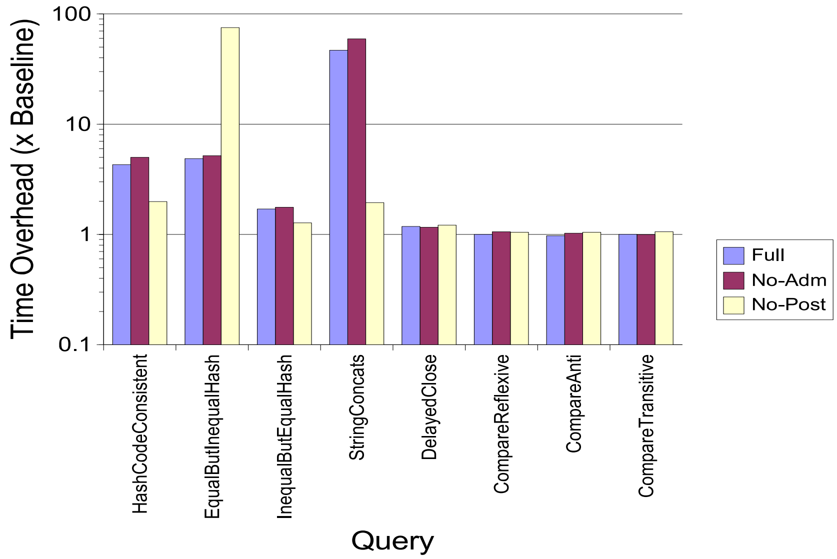


Figure 10: Runtime Overheads For Tomcat

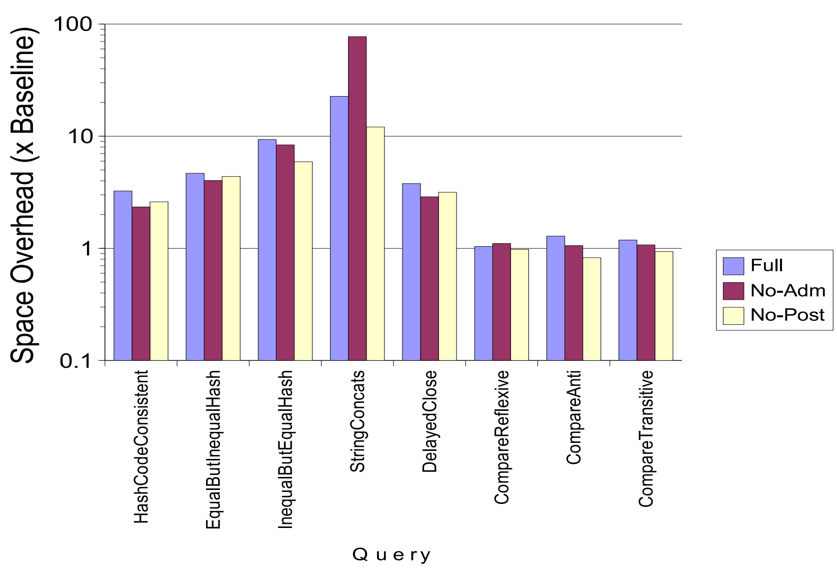


Figure 11: Memory Overheads For Tomcat

```
    ...
    if (...) {
        ... = states.get(newSet);
    }
    if (...) {
        states.put(newSet, ...);
        newSet = null;
    }
}
```

So objects referenced by `newSet` are used for lookups interleaved with mutations, but once the objects are put into the `states` as keys, the objects are no longer mutated. This code is correct, but very subtle.

- Applying `StringConcats` to `tomcat` found performance bugs in classes `org.apache.catalina.util.xml.XmlMapper` and `com.ibm.security.util.ObjectIdentifier`.

  `XmlMapper` handles SAX XML parsing events. It has a `String` field `body`. The SAX parser calls `XmlMapper.characters` repeatedly to signal that new body characters have been parsed. `XmlMapper.characters` appends them to the body using

  ```
  body = body + new String(buf, offset, len);
  ```

  This can lead to parsing taking time $O(n^2)$ in the length of the body text. This bug persisted in the `XMLMapper` source until the whole package was obsoleted.

  The method `ObjectIdentifier.toString` builds a string representation for an `ObjectIdentifier` by concatenating the string representation of each member of an array of components; the string is accumulated in a `String` object. This bug is potentially serious since `ObjectIdentifier.toString` appears to be called when security certificates are parsed, which happens when classes are loaded from signed JAR files. The bug is still present in the latest available version of the IBM JDK.

5. RELATED WORK

There are four main branches of related work: program monitors, systems that "guess" large numbers of predicates and return those that were true during program execution, aspect oriented programming systems, and other instrumentation and trace query engines.

5.1 Program Monitors

The monitoring and checking (MaC) framework [12, 11] monitors a running program and looks for violations of a formal specification. It automatically inserts instrumentation based on a description of interesting events (in PEDL) and a high level specification of undesirable concordances of events (in MEDL). Should the running program violate its specification, MaC raises an alarm.

In a similar vein, Roşu and Havelund [9] describe a system to check the conformance of a program's execution to a specification in linear temporal logic (LTL). The atomic propositions of their logic are events; formulae are interpreted over finite sequences of events (i.e. program traces). Examples of events seem to include function calls, reads and writes to variables, and lock acquires and releases.

Whereas these systems are concerned with decision problems with boolean answers, Partiqle is concerned more generally with data collection and thus operates on and returns sets of data. Counting the number of times an event occurs or inspecting variations in method arguments or durations are natural with PTQL. Furthermore, PTQL supports constraints on data values, which do not fit naturally into any framework which compiles to finite automata. Queries that look for groups of method calls on the same object are natural when examining object-oriented programs. Others have argued that even with their limited expressiveness, temporal logic formulas are hard to understand [6].

5.2 Predicate-Guessing Systems

DIDUCE [8] instruments Java programs to track invariants at various program sites. The violation of an invariant, especially one that had been true many times, yields a warning and a relaxation of the monitored invariant. Deviations from the norm often indicate bugs or interesting facts about program execution.

Liblit et al. [15] instrument programs to use random sampling of program points to gather small parcels of data from a large user base. Statistical analysis correlates certain observations with program failure, giving the developer insight into what situations elicit bugs.

Daikon [7] intensively instruments programs to discover likely invariants.

We view these systems as complementary to Partiqle. While Partiqle provides sparse instrumentation to answer specific questions, these systems monitor entire programs and look for interesting invariants or gather information about program failures.

5.3 Aspect-Oriented Programming Systems

AspectJ [10] is an aspect-oriented extension to Java. By defining pointcuts and advice, one can add functionality to a Java program that cross-cuts the class hierarchy. PTQL and Partiqle solve the more specific problem of instrumenting a Java program to execute a query over its program trace. An aspect to implement a PTQL query would have to contain a point cut (with advice) for each item in the `FROM` clause of the query. It would be up to the programmer to choose suitable runtime data structures, manage them, and optimize. PTQL's declarativeness allows it to choose efficient data structures and perform optimizations.

5.4 Trace Query Engines

Most similar to Partiqle is a the program monitoring and measuring system (PMMS) by Liao and Cohen [14, 13]. Like Partiqle, PMMS compiles a high level query language over program traces to program instrumentation. Our contributions over PMMS are in the sophistication of our implementation and optimizations, the application to Java (including handling of threads and objects, not addressed by PMMS), a more complete implementation, and much more thorough evaluation. In terms of our vocabulary, PMMS has a timing graph but does not use inference rules to infer additional edges for the graph. PMMS has a form of post-dominator but it is restricted to an "interval event" (method invocations) that entirely enclose all other relevant events.

The Hyades project [2] is developing a data model for traces of Java programs. The data model is expressed in the Eclipse Modelling Framework [1] and therefore one can write queries in terms of this data model using the Object Constraint Language [5]. We initially tried to use OCL over the Hyades model as the query language for our work. However, the Hyades model is oriented towards "implicit time": for example, one specifies directly that a method invocation called another method invocation, rather than specifying temporal relationships between start and end events. We discovered that for complex queries, it was much easier for both the query engine and us as query writers to deal directly with explicit timestamps as much as possible. Furthermore OCL is a rich language, for example including functions, and we would have had to extend it with transitive closure, making it a good deal more complex than PTQL. It should be possible to translate some subset OCL queries into PTQL, however.

6. FUTURE WORK

PTQL and Partiqle are just a first step. There are many areas for improvement in the performance and expressiveness of the system.

As mentioned in Section 4.2, we have already begun extending the query language. Obvious candidates for extensions include:

- New fields, such as real-time timestamps.
- New event types, such as field read and write events, lock acquire and release events, thread start and stop events, and breakpoint events (execution of particular program instructions), with new fields associated with these events (e.g., for breakpoint events, access to local variables).
- New table types, such as a table that gives access to heap contents.
- Arithmetic.
- The ability to execute arbitrary expressions, possibly including even calls to (side effect free) methods in the program.
- Negation and, more generally, subqueries (i.e., queries with quantifiers not at the top level).
- Aggregation, e.g., `SELECT SUM(x) WHERE ....` This would provide more opportunities for optimizations.

We can also improve the implementation significantly. One priority is to replace the current query evaluator, which is a form of interpreter, with a compiler generating specialized evaluation code for each query. This also involves replacing the generic record structures we use for all method invocations and objects with customized records containing exactly the fields needed by the query. We also need to generate exactly the indexes needed to evaluate the query efficiently. This work is already under way.

Another obvious improvement would be to add the ability to evaluate multiple queries at once in a single program run. This would be easy to do naively, but optimizations are possible where queries can share data.

We would like to extend the optimizations so that there is no need or benefit for query authors to add constraints to break symmetry or ensure the existence of a post-dominator node. Our system should automatically turn a single general query into the union of several more constrained queries. Similarly, our system should automatically introduce tracking of object lifetimes when that can be used to prune intermediate tables.

With profiling, we can gather data about the selectivity of predicates and other useful information for improving query evaluation. Many techniques from the world of databases can probably be imported and profitably applied.

There seem to be many opportunities to employ static analysis to further reduce overhead. For example, in the example of "can `doTransaction` call `sleep`?", a context-sensitive static call graph could be used to deduce that a particular call site of `doTransaction` can never in turn invoke `sleep`, and therefore we can safely call a specialized instrumentation-free copy of `doTransaction`. In general, any analysis that can statically determine the value of some part of the query with respect to a given program could be useful for reducing overhead.

Another rich area for exploration is adding support for approximate query evaluation via sampling. Sampling is widely used to make expensive dynamic analyses tractable, but it requires considerable care to use correctly. A general framework for computing approximate answers to PTQL queries would be extremely powerful and useful. Again, techniques from databases seem to be applicable.

7. CONCLUSION

We have described PTQL, a language for writing expressive, declarative queries about program behavior, and Partiqle, a system for compiling PTQL queries into light-weight instrumentation on Java programs. Using PTQL and Partiqle avoids the complexity and code maintenance problems of manual instrumentation.

We demonstrate that Partiqle is efficient enough to run interesting queries on real world Java programs and that our optimizations are crucial to achieving this performance.

8. REFERENCES

[1] Eclipse technology - emf project. Project page at `http://www.eclipse.org/emf/`.

[2] Eclipse technology - hyades project. Project page at `http://www.eclipse.org/hyades/`.

[3] *JavaTM 2 Platform, Standard Edition, v 1.4.2 API Specification.* `http://java.sun.com/j2se/1.4.2/docs/api/`.

[4] Apache tomcat. project home page at `http://jakarta.apache.org/tomcat/`.

[5] T. Clark, J. Warmer, and B. Schmidt. *Object Modeling With the Ocl: The Rationale Behind the Object Constraint Language*. LNCS 2263. Spring-Verlag, 2002.

[6] J. Corbett, M. Dwyer, J. Hatcliff, and Robby. Expressing checkable properties of dynamic systems: The bandera specification language. *International Journal on Software Tools for Technology Transfer*, Sept. 2002.

[7] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.d., University of Washington Department of Computer Science and Engineering, Seattle, Washington, Aug. 2000.

[8] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, May 2002.

[9] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[11] M. Kim. *Information Extraction for Run-time Formal Analysis*. PhD thesis, CIS Dept., Univ. of Pennsylvania, 2001.

[12] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[13] Y. Liao. *An Automatic Programming Approach to High Level Program Monitoring and Measuring*. PhD thesis, Dept. Computer Science, Univ. Southern California, Los Angeles, CA., 1992.

[14] Y. Liao and D. Cohen. A specificational approach to high level program monitoring and measuring. *IEEE Transactions On Software Engineering*, 18(11):969–978, November 1992.

[15] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, California, June 9–11 2003.

[16] D. Marinov and R. O'Callahan. Object equality profiling. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003)*, Anaheim, CA, Oct. 2003.

[17] Specjvm98 benchmarks. Information available at `http://www.specbench.org/osg/jvm98/`.

[18] WebSphere application server development best practices for performance and scalability. White paper available from `http://www-3.ibm.com/software/webservers/appserv/ws_bestpractices.pdf`, Sept. 2000.