Relational Queries Over Program Traces*

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

Robert O'Callahan‡
Novell
rocallahan@novell.com

Alex Aiken
Stanford University
aiken@cs.stanford.edu

ABSTRACT

Instrumenting programs with code to monitor runtime behavior is a common technique for profiling and debugging. In practice, instrumentation is either inserted manually by programmers, or automatically by specialized tools that monitor particular properties. We propose Program Trace Query Language (PTQL), a language based on relational queries over program traces, in which programmers can write expressive, declarative queries about program behavior. We also describe our compiler, PARTIQLE. Given a PTQL query and a Java program, PARTIQLE instruments the program to execute the query on-line. We apply several PTQL queries to a set of benchmark programs, including the Apache Tomcat Web server. Our queries reveal significant performance bugs in the *jack* SpecJVM98 benchmark, in Tomcat, and in the IBM Java class library, as well as some correct though uncomfortably subtle code in the Xerces XML parser. We present performance measurements demonstrating that our prototype system has usable performance.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Monitors; D.3.2 [**Language Classifications**]: Specialized application languages

General Terms

Languages, Performance, Measurement

Keywords

Partiqle, PTQL, program trace query language, relational

*This research was supported in part by the National Science Foundation under grant numbers NSF CCR-0085949, CCR-0326577, CCR-0234689, NASA grant number NNA04CI57A, 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.

†Some of this research was conducted while Simon was an employee of IBM T.J. Watson Research Center.

‡This research was conducted while Robert was an employee of IBM T.J. Watson Research Center.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA'05, October 16–20, 2005, San Diego, California, USA.
Copyright 2005 ACM 1-59593-031-0/05/0010 ...$5.00.

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 simple example, understanding the who-calls-whom relation in a large, object-oriented program is a non-trivial task. A programmer might try to answer the question by instrumenting the code with the lines marked `*`:

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

However, this instrumentation is not even correct. If `doTransaction()` terminates with 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 more complex in a multi-threaded program. Each thread must track whether it is executing DB.doTransaction() and care must be taken to avoid data races.

We propose a system for answering such application-specific questions about program behavior. These questions are asked as queries in our *Program Trace Query Language* (PTQL), which we present in Section 2. We also describe our implementation (Section 3) and evaluation (Section 4) of PARTIQLE, a tool to compile a PTQL query into bytecode instrumentation that is injected into a Java program to execute the query as the program runs.

Using PARTIQLE avoids the problems of manual instrumentation. Separating queries from programs makes both of them easier to understand and maintain. Specifying queries in PTQL leaves details of gathering, storing, and maintaining data as well as issues such as thread safety and recursion to PARTIQLE. The following PTQL query asks *"Can method* DB.doTransaction() *transitively call method* sleep()*?"*:

```
SELECT doTrans.startTime, sleep.startTime
FROM MethodInvoc('DB.doTransaction') doTrans
   JOIN MethodInvoc('B.sleep') sleep
      ON doTrans.thread = sleep.thread
      AND doTrans.startTime < sleep.startTime
      AND sleep.endTime < doTrans.endTime
```

This PTQL query looks for two method invocations, doTrans and sleep, where doTrans is a method named doTransaction in class DB and sleep is a method named sleep in class B.[1] Furthermore, doTrans and sleep should happen in the same thread and sleep should happen during doTrans.

The contributions of this paper are:

- We introduce PTQL (Section 2), which is similar in spirit to SQL. With PTQL the user only specifies what data she wants and not how to gather it. The implementor is free to choose efficient data representations and query execution plans. PTQL supports timing constraints as well as data constraints among many program events.

- We describe the implementation of PARTIQLE (Section 3) and several optimizations that reduce the overhead imposed by PARTIQLE's instrumentation. We evaluate the effectiveness of these optimizations in Section 4.4.

- We experimentally demonstrate that the overhead PARTIQLE imposes is reasonable and that it can find interesting program behaviors (Section 4). We use PARTIQLE to run several queries on 11 real Java programs, including Apache Tomcat [2], and report the slowdown and memory footprint. Our queries reveal significant (and apparently unknown) performance bugs in the *jack* SpecJVM98 [26] benchmark, in Tomcat, and in the IBM Java class library, and some uncomfortably clever code in the Xerces XML parser.

[1]Note that this query uses syntactic sugar defined in Section 2.3.

2. Program Trace Query Language (PTQL)

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

- Program traces are naturally viewed as sets of timestamped 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 data and timing relationships among different events (i.e., relational joins).

- This view allows PTQL to be declarative, thus freeing the user from specifying how to gather and manage data and freeing the implementor to choose efficient data representations and query execution plans. There are many well-known and successful optimizations for SQL that can aid us in optimizing PTQL. Optimization is crucial since many natural queries produce enormous amounts of data.

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

2.1 Data Model: Tables and Fields

For the purposes of this paper, we consider a schema for program traces with two relations (sets):

- MethodInvoc contains a record[2] for each method invocation that occurs during program execution.

- ObjectAlloc contains a record for each object allocated during program execution.

The fields defined in MethodInvoc and ObjectAlloc are listed with their types in Figure 1. Fields of type ObjectId contain references to records in ObjectAlloc (i.e., anything that the Java type system could type as Object). Fields of type variant may contain values of any type. Fields like param1 have type variant because the type of values that they contain depends on the method invocation they match and thus cannot be determined until runtime.

Our data model is rich enough to express useful queries (see Section 4). Nonetheless, we designed it with extensibility in mind. Adding other relations allows PTQL to talk about different sorts of events like reads or writes to object fields, lock acquires and releases, and thread start and stop. Adding fields to existing relations allows 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

A query consists of three clauses (see Figure 2): a FROM clause, a WHERE clause, and a SELECT clause. The identifiers in the FROM clause give a unique name to each record that participates in a query. The join conditions, SELECT clause, and WHERE clause use these names to refer to specific records in a query result.

PTQL is a subset of SQL. The difference between PTQL and SQL is in the data model, in particular PTQL's relational schema

[2]Alternately, we could use the database term "tuple" instead of "record"; we use the terms interchangeably.

$$\left[\!\left[\begin{array}{ll} \texttt{SELECT} & x_1.d_1,\ldots,x_m.d_m \\ \texttt{FROM} & f \\ \texttt{WHERE} & w_1 \texttt{ AND} \ldots \texttt{AND } w_k \end{array} \right]\!\right]_\Sigma = \{\langle r(x_1)(d_1),\ldots,r(x_m)(d_m)\rangle \mid r \in [\![f]\!]_\Sigma \text{ and for all } j \in \{1,\ldots,k\}, r \models w_j\} \quad (1)$$

$$[x \mapsto \sigma](y) = \text{if } x = y \text{ then } \sigma \text{ else } \bot \quad (2)$$

$$(r_1 \oplus r_2)(x) = \text{if } r_1(x) \neq \bot \text{ then } r_1(x) \text{ else } r_2(x) \quad (3)$$

$$r \models x_1.d_1 < x_2.d_2 + n \text{ iff } r(x_1)(d_1) < r(x_2)(d_2) + n \quad (4)$$

$$[\![\texttt{MethodInvoc } x]\!]_\Sigma = \{[x \mapsto \sigma] \mid \sigma \in \Sigma \text{ and } \sigma \text{ a } \texttt{MethodInvoc} \text{ event}\} \quad (5)$$

$$[\![\texttt{ObjectAlloc } x]\!]_\Sigma = \{[x \mapsto \sigma] \mid \sigma \in \Sigma \text{ and } \sigma \text{ a } \texttt{ObjectAlloc} \text{ event}\} \quad (6)$$

$$[\![f_1 \texttt{ JOIN } f_2 \texttt{ ON } w]\!]_\Sigma = \{r_1 \oplus r_2 \mid r_1 \in [\![f_1]\!]_\Sigma \text{ and } r_2 \in [\![f_2]\!]_\Sigma \text{ and } r_1 \oplus r_2 \models w\} \quad (7)$$

$$[\![f_1 \texttt{ LEFT ANTIJOIN } f_2 \texttt{ ON } w]\!]_\Sigma = \{r_1 \mid r_1 \in [\![f_1]\!]_\Sigma \text{ and } \forall r_2 \in [\![f_2]\!]_\Sigma,\ r_1 \oplus r_2 \not\models w\} \quad (8)$$

Figure 3: Semantics of PTQL Query on Program Trace Σ.

Fields of `MethodInvoc`	
`startTime` : long	wall clock timestamp for the start
`endTime` : long	wall clock timestamp for the end
`mname` : string	name of the method
`declClass` : string	class that declares the method
`implClass` : string	class that implements the method
`receiver` : ObjectId	`this` parameter
`thread` : ObjectId	thread in which the method is invoked
`result` : variant	value returned by the method
`param1`, `param2`, ...: variant	values for the actual parameters

Fields of `ObjectAlloc`	
`startTime` : long	wall clock timestamp for allocation
`endTime` : long	wall clock timestamp for collection
`thread` : ObjectId	thread in which the object is allocated
`type` : string	class name of the object's runtime type
`obj` : ObjectId	the object

Figure 1: Fields of `MethodInvoc` and `ObjectAlloc`.

⟨*query*⟩ ::= `SELECT` ⟨*select*⟩ [, ⟨*select*⟩]*
`FROM` ⟨*from*⟩
`WHERE` ⟨*conj*⟩
⟨*select*⟩ ::= *identifier.field*
⟨*from*⟩ ::= ⟨*relation*⟩ *identifier*
| ⟨*from*⟩ `JOIN` ⟨*from*⟩ `ON` ⟨*conj*⟩
| ⟨*from*⟩ `LEFT ANTIJOIN` ⟨*from*⟩ `ON` ⟨*conj*⟩
⟨*conj*⟩ ::= ⟨*pred*⟩ [`AND` ⟨*pred*⟩]*
⟨*pred*⟩ ::= *identifier.field* ⟨*op*⟩ *identifier.field*
| *identifier.field* ⟨*op*⟩ *identifier.field+long*
| *identifier.field* = '*string*'
| *identifier.field* `IN` {'*string*' [, '*string*']+}
| *identifier.field* `instanceof` '*string*'
| *identifier.field* `notinstanceof` '*string*'
⟨*relation*⟩ ::= `MethodInvoc` | `ObjectAlloc`
⟨*op*⟩ ::= < | = | != | >

Figure 2: Syntax of Query Language.

over program traces. Join and antijoin in PTQL have the same semantics as in SQL.[3] Informally, the join of sets A and B on predicate p is the set of records in $A \times B$ for which p holds. The result of A left antijoined with B on predicate p is the set of records $a \in A$ for which there is no record $b \in B$ such that $\langle a,b\rangle$ satisfies p. Adding semijoin and outer join to PTQL is easy in the sense that SQL already defines their semantics. SQL style aggregation primitives (`GROUP BY`, `DISTINCT`) are also a natural extension.

Figure 3 gives the semantics of a PTQL query on a program trace Σ. A program trace is a set of events; each event $\sigma \in \Sigma$ corresponds to either a record from `MethodInvoc` or `ObjectAlloc`. For convenience in defining the semantics, we treat events from the program trace as mappings from field names (as given in Figure 1) to values for the fields.

In Equation 1 of Figure 3, the set of records specified by the `FROM` clause f is $[\![f]\!]_\Sigma$. Each record r in this set is a map from identifiers in the `FROM` clause to events in the program trace; Equation 2 defines these maps. Equation 3 shows how join combines two such mappings. For instance, by Equation 5 the clause `FROM MethodInvoc x` specifies a set of mappings each of which maps x to a different `MethodInvoc` record $\sigma \in \Sigma$. Each candidate record r must satisfy the predicates w_j in the `WHERE` clause, that is $r \models w_j$. Each such record is transformed into a query result by projecting the fields requested by the `SELECT` clause, as in $r(x_i)(d_i)$.

Equation 4 shows the resolution of one predicate; the rule resolves position and field names to values and checks the predicate. Rules for the other predicates are similar but not shown. We define the comparison operators so that fields with incompatible types are not equal, greater, nor less than each other and fields of type `ObjectId` are neither less than nor greater than each other. If the field `x.paramI` of "`MethodInvoc` x" is used in a query, only invocations of methods with at least `I` 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`. The `instanceof` operator allows comparisons behaves like `instanceof` in Java. The `notinstanceof` operator is the negation of `instanceof`. The `instanceof` and `notinstanceof` operators are defined only on fields of type `ObjectId`.

Equation 7 defines joins. The mappings for joined sets combine (via Equation 3) to form a new mapping that resolves names from both sets. Only records that satisfy the join condition are kept. Consider for instance a map r in $[\![\texttt{MethodInvoc } x \texttt{ JOIN ObjectAlloc } y \texttt{ ON } x.\texttt{receiver} = y.\texttt{obj}]\!]_\Sigma$.

[3] Some variants of SQL define antijoin, left antijoin, and right antijoin. PTQL only defines left antijoins.

Each such r is a combination of maps (analogous to a concatenation of records) $r = r_1 \oplus r_2$ such that $r_1 \in [\![\texttt{MethodInvoc}\ x]\!]_\Sigma$ and $r_2 \in [\![\texttt{ObjectAlloc}\ y]\!]_\Sigma$. Then $r(x) = r_1(x)$ is a `MethodInvoc` event, $r(y) = r_2(y)$ is an `ObjectAlloc` event, and $r(x)$(`receiver`) must be equal to $r(y)$(`obj`).

Similarly, equation 8 computes the set of records specified by a left antijoin; mappings r_1 from the set on the left are removed if they satisfy the predicate w when joined with any mapping r_2 from the set on the right.

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.param1, Y.param2
  FROM  MethodInvoc('Foo.y') Y
```

Calls to methods named `y` that are declared in class `Foo` match this query. The query result contains the first two actual parameters of the call.

This query uses syntactic sugar. As many queries specify `mname` and `declClass` fields, we use shorthand to specifying them in the `FROM` clause. The desugared query is:

```
SELECT  Y.param1, Y.param2
  FROM  MethodInvoc Y
 WHERE  Y.mname = 'y' AND Y.declClass = 'Foo'
```

Notice that the query constrains the `declClass` field. Suppose class `Foo` has a subclass `OSubFoo` that overrides the implementation of `y` and another subclass `ISubFoo` that inherits (but does not override) the implementation of `y`. Calls to `ISubFoo.y` and `OSubFoo.y` match the query. If instead of the predicate `Y.declClass = 'Foo'`, the query contained the predicate `Y.implClass = 'Foo'`, then calls to `ISubFoo.y` would match, but calls to `OSubFoo.y` would not match. Since `OSubFoo` overrides `y`, the `implClass` field for invocations of `OSubFoo.y` is 'OSubFoo'.

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

The documentation for `java.lang.Object.hashCode` [13] requires implementations of `hashCode()` to agree with `equals()`. In particular, if `x.equals(y)` returns `true`, then `x.hashCode() == y.hashCode()` should hold. This query checks that calls to `hashCode()` and `equals()` follow this specification.

```
SELECT xhc.implClass, yhc.implClass, eq.implClass
FROM MethodInvoc('Object.equals') eq
    JOIN MethodInvoc('Object.hashCode') xhc
        ON eq.receiver = xhc.receiver
    JOIN MethodInvoc('Object.hashCode') yhc
        ON eq.param1 = yhc.receiver
        AND xhc.result != yhc.result
WHERE eq.result = true
```

In this query `eq` matches calls to `equals()`, and `xhc` and `yhc` match calls to `hashCode()`. The query is interested in a group 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

This section discusses PARTIQLE, our compiler for PTQL queries. We outline our instrumentation strategy for recording event records, describe runtime support structures needed to execute queries online, give our algorithm for query execution, 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 postmortem) and online analysis. Offline execution of the query allows for a constant size memory footprint as events are gathered during program execution. However, in practice, the postmortem processing of large traces usually requires similar resources to simply doing the analysis online; in particular, because random accesses to disk are very slow, efficient analyses must read traces sequentially. The main advantages of offline execution are that the analysis need not compete with the application for space, and clever offline analyses exploit the ability to read the trace sequentially multiple times [23].

We prefer online processing whenever reasonable performance can be obtained. Even though storage is cheap, managing large volumes of trace data imposes considerable overhead. Online query execution presents a simpler model to the user by eliminating post-processing steps, and provides the quickest feedback, keeping the code-debug cycle short. Another advantage is the ability to stop the program when certain behaviors are detected and start a debugger or dump a stack trace. For these reasons, plus the fact that there are opportunities to optimize and reduce query overhead to less than the minimum overhead of offline analysis, we chose to implement PARTIQLE as an online analysis engine. However, one can easily imagine implementing PTQL queries using offline analysis.

To ease development and deployment of PARTIQLE, we instrument the program at the level of Java bytecodes. The instrumentation's use of Java library classes creates re-entrancy issues, which we resolve by avoiding the use of most Java library classes and not instrumenting those few library classes that 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.

Compilation of a PTQL query to program instrumentation has four phases:

- selection of data structures for runtime tables
- timing analysis
- generation of instrumentation to create and store event records
- generation of query evaluation code

Section 3.1 describes the process of generating custom record and table types to store event records at runtime. Section 3.2 discusses *timing analysis*, which computes a partial order of events in a query. Timing analysis is crucial to PARTIQLE's instrumentation and optimization. Section 3.3 discusses the instrumentation of user code to generate event records. This instrumentation is also responsible for adding and removing these records from their collections and triggering query evaluation. Section 3.4 discusses the code PARTIQLE generates to perform query evaluation. Section 3.5 instantiates PARTIQLE's instrumentation, runtime data structures, and optimizations for an example query.

3.1 Runtime Data Structures

The data gathering instrumentation creates and adds records to PARTIQLE's runtime data structures. PARTIQLE keeps two *runtime tables* per identifier in the query's `FROM` clause — one table

for start events and one table for end events. Each of these run-time tables is a collection of records that may satisfy the predicates associated with that slot in the query. For example the "Does `DB.doTransaction()` transitively call `sleep()`?" query from Section 1 has four runtime tables: one each for the start and end of invocations of `doTrans()`, and one each for the start and end of invocations of `sleep()`.

Operations on large run-time tables are expensive and it is important that both the tables and queries be optimized for performance. In designing the tables, we can exploit our knowledge of the query; for example, if a query does not require a certain data field, then there is no need to store that field in the table in the first place. Because such opportunities for optimization are common and important, we have developed a TABLEEMITTER library that generates data structure implementations given a specification of the operations on the table.

3.1.1 Table Specifications

A *table specification* consists of two parts: a list of the types of the fields (currently we support Java types `int`, `long` and `Object`) and a list of operations that the table must support. There are five kinds of operations, the last four of which are parameterized on a given predicate P:

- *add* a record to the table (this operation is included in every specification);
- *delete* all records satisfying P;
- *check* whether there is a record satisfying P;
- *count* the number of records satisfying P;
- create an *iterator* to iterate through all the records that satisfy P, returning each record projected onto some set of fields F and allowing the client to optionally delete each record after it has been returned.

Performing an *add* or *delete* operation invalidates all extant iterators. This condition is not checked at run time.

The predicates that may be used to specify operations are conjunctions of terms, each term constraining one field of the record. We allow the following constraints on a scalar field value v, where a and b are parameters in the predicate that are supplied when the operation is called at run time:

- $v = a$
- $v \leq a$
- $a \leq v$
- $a \leq v \leq b$

Thus each field requires only one constraint. For object types, only equality constraints are allowed, but PARTIQLE supports both equality under object identity and equality based on the Java `equals` and `hashCode` methods.

Our TABLEEMITTER table code generator takes a table specification and emits a set of Java classes in bytecode form. It also outputs one method name for each of the operations, one class name for each type of iterator, one class name for the records returned by iterators, and the class name of the `main` class. These classes and methods stand alone and implement the desired specification.

3.1.2 Index Trees

Our basic approach stores each record in a Java object and builds indexes so that each required operation (other than *add*) can find all the relevant records efficiently. An operation using a predicate P requires an index on the record fields that appear in P. Indexes over multiple fields are constructed as nested indexes; an index on field `A` and `B` would be represented as a map from an `A`-value to a map from `B`-values to sets of records.

We must choose a field order for each operation. We can consolidate outer indexes, so for example indexes on `A`, `B` and `A`, `C` can be represented as a map from an `A`-value to a pair of maps, from `B`-values to records and from `C`-values to records. In general we build a tree where the edges are labeled with field names such that for every query predicate, there is an *index path* in the tree from the root whose labels are exactly the set of field names mentioned in the predicate. We assign a simple cost function to trees (currently, the number of nodes in the tree) and use brute force with some search pruning to find the best tree.

Each edge in the index tree corresponds to a map data structure in the instrumentation. The map for an edge is implemented as a hash table if all predicates whose index path includes the edge require only equality constraints. Otherwise the map is implemented as a trie, which provides efficient iteration over all key values in a given range. (The trie depth is bounded by a constant, since our keys are fixed-bit-width scalar values.) The code for hash tables and tries is duplicated and specialized to the field type.

The map chains terminate at sets of records, generally stored as doubly-linked lists, but sometimes optimized (see below).

3.1.3 Record Counts

Count and operations that check the existence of records do not need access to actual record objects. Existence check operations simply traverse maps to see if there is any mapping for the field values matching the predicate. *Count* operations are optimized by maintaining in each map and linked list the number of records ultimately contained by the map or linked list.

Only *iterator* and in some cases *delete* operations actually require record field values (see below). Thus, in some cases we can prove that a map data structure implementation's record lists are never used. In that case we eliminate the linked lists and keep just the record count. (The count must be used by some existence check or *count* operation, or else the corresponding index tree leaf could have been pruned to reduce the tree cost metric.)

3.1.4 Deletion

Delete operations iterate through all records matching the deletion predicate. Each record that satisfies the deletion predicate is removed from every list in which it appears. In cases where the list is statically optimized away, the record count is decremented. Finding the lists in which each record appears requires lookup along the map chains corresponding to other branches of the index tree.

If a runtime table is represented as a doubly-linked list and requires no *delete*s, PARTIQLE instead represents it as a singly-linked list. This optimization also applies to lists where PARTIQLE can prove that *delete* always deletes either all records in a list or none of them. For example, when there is a single *delete* the lists at the end of the map chains for that *delete* satisfy this property.

3.1.5 Record Representation

PARTIQLE analyzes the operations and index tree to determine the minimal set of fields that each event record must store to answer the query. Even if a field appears in a predicate, it does not need to be stored in the event record if the field's value is implicit in the

keys of the map chain leading to the record. Of course PARTIQLE stores in the event record all fields that must be returned during iteration. Deletion operations may require that an event record store a field's value so that when that record is deleted, PARTIQLE can look up and remove corresponding records in other maps.

3.1.6 Object Inlining

Whenever we know that there will always be exactly one reference to an object (for example, the objects representing maps), and the reference is not from an array, PARTIQLE inlines the object fields into the site of the reference. This optimization sometimes requires duplication and specialization of the referencing object type. This optimization folds all linked-list links into the record objects.

3.1.7 Examples

These techniques together allow efficient representation of the runtime tables as many common (and not so common) data structures.

- *Linked List* The client specifies just an iterator with no predicate, returning some fields. We generate just a singly-linked list.
- *Hash Set* The client specifies an existence check testing a single field for equality and a deletion operation testing that field for equality. We generate a hash table mapping field values to a count of the number of records matching the field value.
- *Hash Table* The client specifies a deletion operation testing a single field for equality, and an iterator testing that field for equality and returning a number of fields. We generate a hash table mapping field values to records.
- *Trie* The client specifies an iterator testing a field for containment in some range, returning a number of fields. We generate a trie. Additional requests for deletion and existence check on that field would give the same structure.
- *Double-Indexed Hash Table*. The client specifies an iterator with no predicate that returns some fields and two deletion operations, one deleting all records with field $\texttt{F}_1 = \texttt{v}_1$ and another deleting all records with $\texttt{F}_2 = \texttt{v}_2$. We generate a pair of hash tables, one mapping $\texttt{F}_1$ values to a doubly-linked list of records and the other mapping $\texttt{F}_2$ values to a doubly-linked list of records. $\texttt{F}_1$ and $\texttt{F}_2$ are in the record object even if they aren't required by the iterator. The links of the doubly-linked lists are inlined into the record object. The *delete* $F_1 = v_1$ operation looks up v_1 in the $\texttt{F}_1$ table to get all the records that match $\texttt{F}_1 = \texttt{v}_1$, and removes them from both of the doubly-linked lists (which may require deleting entries from the F_2 hash table).

3.2 Timing Analysis

PARTIQLE's instrumentation and optimizations require information about possible event orderings in query results. PARTIQLE performs *timing analysis* to compute a partial order of events in a query result and stores this partial order as a *timing graph*, a directed acyclic graph with two nodes for each identifier in the FROM clause of the query — 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 is an edge from `a.start` to `b.start` in the timing graph.

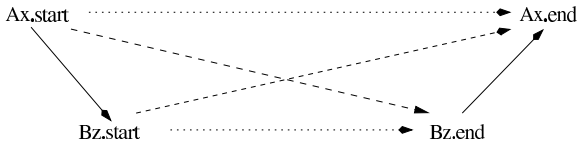

Figure 5: Timing graph for example query from Section 1.

For reference, the places where PARTIQLE uses timing information are:

- Query evaluation must be triggered after any event which may be the last in a query result (see Section 3.3).
- If event B must always happen after a related event A, PARTIQLE may do an admission check when B happens (see Section 3.3.3).
- If a query has an event that must happen after all other events in the query, some additional optimizations are possible (see Section 3.3.4).

The complete rules for building the timing graph are given in Figure 4. These rules are applied repeatedly until a fixed point is reached. We call this fixed point the *closed timing graph*. In the figure, Q is the set of predicates in the join conditions and `WHERE` clause of the query and E is the set of edges in the timing graph. Furthermore, m_1, m_2, o, x, y are query identifiers (m_1, m_2 for a `MethodInvoc` invocation, o for an `ObjectAlloc`), a, b, c are nodes in the timing graph, and n is an integer. Start nodes precede end nodes (Rule 1). Rules 2 – 5 show how explicit ordering constraints induce timing edges. The lifetime of the `this` parameter of a method includes the method invocation (Rule 6). The lifetime of an object mentioned as a parameter of a method must include the start of the method (Rule 7). Similarly a method result's lifetime must include the end of the method (Rule 8). The timing graph is transitively closed (Rule 9). Overlapping method invocations on the same thread must actually be nested (Rule 10).

Figure 5 shows the timing graph for the example from Section 1. In addition to edges induced by explicit constraints in the query, PARTIQLE infers edges from properties of Java's semantics. 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 (Rule 1). The dashed edges from `doTrans.start` to `sleep.end` and from `sleep.start` to `doTrans.end` follow from transitivity (Rule 9).

For some optimizations, we identify a subset of the ordering constraints guaranteeing that if the subset is observed to hold dynamically, then all constraints in the timing graph are satisfied. A *reduced* timing graph is a graph whose closure under the rules in Figure 4 gives the closed timing graph. Note that reduced timing graphs are not unique. Starting with the closed timing graph, we build a reduced graph by repeatedly applying the rules:

- If $\{(a,b),(b,c),(a,c)\} \subseteq E$, remove (a,c) from E
- Remove $(\texttt{x.start}, \texttt{x.end})$ from E.
- For method invocations x and y, if $\{(\texttt{x.start}, \texttt{y.start}), (\texttt{y.start}, \texttt{x.end}), (\texttt{y.end}, \texttt{x.end})\} \subseteq E$ and $(x.\texttt{thread} = y.\texttt{thread}) \in Q$, remove $(\texttt{y.end}, \texttt{x.end})$ from E.

$$
\begin{array}{rcll}
\texttt{true} & \Rightarrow & (\texttt{x.start},\texttt{x.end}) \in E & (1)\\
(x.\texttt{startTime} < y.\texttt{startTime}) \in Q & \Rightarrow & (\texttt{x.start},\texttt{y.start}) \in E & (2)\\
(x.\texttt{endTime} < y.\texttt{startTime}) \in Q & \Rightarrow & (\texttt{x.end},\texttt{y.start}) \in E & (3)\\
(x.\texttt{startTime} < y.\texttt{endTime}) \in Q & \Rightarrow & (\texttt{x.start},\texttt{y.end}) \in E & (4)\\
(x.\texttt{endTime} < y.\texttt{endTime}) \in Q & \Rightarrow & (\texttt{x.end},\texttt{y.end}) \in E & (5)\\
(m.\texttt{receiver} = o.\texttt{obj}) \in Q & \Rightarrow & (\texttt{o.start},\texttt{m.start}) \in E & \\
 & & \wedge\,(\texttt{m.end},\texttt{o.end}) \in E & (6)\\
(m.\texttt{paramn} = o.\texttt{obj}) \in Q & \Rightarrow & (\texttt{o.start},\texttt{m.start}) \in E & \\
 & & \wedge\,(\texttt{m.start},\texttt{o.end}) \in E & (7)\\
(m.\texttt{result} = o.\texttt{obj}) \in Q & \Rightarrow & (\texttt{o.start},\texttt{m.end}) \in E & \\
 & & \wedge\,(\texttt{m.end},\texttt{o.end}) \in E & (8)\\
(a,b) \in E \wedge (b,c) \in E & \Rightarrow & (a,c) \in E & (9)\\
(m_1.\texttt{thread} = m_2.\texttt{thread}) \in Q & & & \\
\wedge\,(\texttt{m}_2\texttt{.start},\texttt{m}_1\texttt{.end}) \in E & & & \\
\wedge\,(\texttt{m}_1\texttt{.start},\texttt{m}_2\texttt{.start}) \in E & \Rightarrow & (\texttt{m}_2\texttt{.end},\texttt{m}_1\texttt{.end}) \in E & (10)
\end{array}
$$

Figure 4: Timing Edge Inference.

The reduced timing graph represents a set of timing constraints that are sufficient to guarantee that all timing constraints in the original query are met. The rules above ensure that a reduced timing graph contains constraints that can be checked relatively early in the execution of a query (see Section 3.3.3).

3.3 Instrumentation to Create Event Records

PARTIQLE inserts instrumentation at the start of each method so that each invocation generates a new `MethodInvoc` record (containing values for `startTime`, `thread`, `receiver`, `param1`, etc.) and adds the record to PARTIQLE's runtime data structures. Similarly, instrumentation before method return stores the return value (`result`) and end time (`endTime`). Should the method throw an exception, PARTIQLE catches the exception, records `endTime` and an exceptional result for `result`, and re-throws the exception. A global lock protects accesses to the shared data structures. Figure 6 contains pseudocode to generate this instrumentation.

Java dictates that in a constructor, the `this` reference is not accessible until after the superclass constructor has been called. This condition 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 situation complicates some of the analyses described below; the details are tedious and beyond the scope of this paper.

Gathering information about object lifetimes is harder than for method invocations because more code locations are involved. PARTIQLE inserts instrumentation to record object allocations after calls to the `Object` constructor.[4] By tracking objects with instances of `java.lang.ref.WeakReference`, PARTIQLE is notified by the garbage collector when tracked objects are collected.

Creating a record for every allocated Java object is impractical. Fortunately, most queries do not refer to information available only at allocation time (`thread` and `startTime`), and constrain objects to be a parameter or result of a method invocation. For such queries it suffices to allocate an object's record lazily, when a method invocation first makes the object relevant to the query.

[4]Instrumenting the `java.lang.Object` constructor at all causes many JVMs to crash.

PARTIQLE outputs each query result exactly once and as early as possible. The occurrence of an event which may, according to timing analysis, generate the last record in a (single) query result is a *candidate*. A candidate triggers a search through the runtime tables for records that together with the candidate satisfy the query predicate. We refer to this search as *query evaluation*. Section 3.4 describes PARTIQLE's query evaluation algorithm.

The following subsections discuss several optimizations that apply to data gathering instrumentation.

3.3.1 Static Filtering

Static predicates in a query depend only on static properties of the code. If an instrumentation site violates a static predicate, PARTIQLE need not insert instrumentation at that site. PARTIQLE exploits several static predicates: comparisons of the `mname`, `declClass` and `implClass` fields in `MethodInvoc` records with constant strings and comparisons of the `type` field in `ObjectAlloc` records with constant strings. Static filtering on `type` is only possible at sites where enough is known about both the static type of the object reference in question and the program's class hierarchy to statically determine whether the object reference refers to an object of a desired class.

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

3.3.2 Dynamic Filtering

Query predicates involving fields from only one record can be checked at the instrumentation site generating the relevant fields. We refer to these predicates as *simple dynamic predicates*. For example, consider a query that lists all method invocations where the `this` pointer is the same as the first parameter:

```
SELECT  f.mname, f.implClass
  FROM  MethodInvoc f
 WHERE  f.param1 = 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.

PARTIQLE handles comparisons of the `result` of a method invocation to fields from the start of the event similarly. Suppose we replace `param1` with `result` in the example. At the start of method invocations, a record is added to the start event table for f. When `result` becomes available at the end of the invocation, the predicate is checked. The record is dropped if the predicate is false.

3.3.3 Admission Checks

Query predicates that cannot be checked statically and that involve more than one record are *join predicates*. Using the timing graph, PARTIQLE adds instrumentation to check some join predicates when new records are created. These *admission checks* deny a record admission to a runtime table if it cannot satisfy a join predicate.

Before describing admission checks in detail, we return to the example from Section 1. Notice the join predicate (`doTrans.startTime < sleep.startTime`) and suppose the instrumentation at the start of an invocation of `sleep()` is now executing (i.e., an invocation of `sleep()` is starting). If this `sleep` is to satisfy the join predicate above, then according to the timing graph (Figure 5) an invocation of `doTransaction()` that matches this `sleep` must already have started. So, at the start of `sleep()` we check if a *supporting* record `doTrans` is stored in the `doTrans` table; if not, this `sleep` cannot be part of a query result and is discarded.

If the query includes additional constraints relating `doTrans` and `sleep`, for example (`doTrans.param1 = sleep.param1`), then these constraints are checked as part of the admission check; the check fails unless a supporting `doTrans` record is found. Join predicates such as `doTrans.param1 = sleep.result`, which depend on information available at the end of `sleep`, cannot be checked by the admission check. We 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; if none are found we discard the record for the invocation of `sleep()`.

PARTIQLE inserts admission or retention checks for each event e whose node in the *reduced timing graph* has a predecessor for which join predicates with e are available.

3.3.4 The Post-dominator

Some queries have a *post-dominator*: an event that must happen after all other events that contribute to query results. A post-dominator's node in the timing graph is reachable from (i.e., after) all nodes that generate either fields in the `SELECT` clause, or fields that appear in `WHERE` or join predicates. Additional optimizations apply to queries with post-dominators.

Again we consider the query from Section 1. Its timing graph is in Figure 5. This query requires fields available at `doTrans.start` and `sleep.start`. The reduced timing graph contains just the two edges (`doTrans.start`, `sleep.start`) and (`sleep.start`, `doTrans.end`). Notice that at `sleep.start`, even though `doTrans` has not yet finished, we know that its `endTime` must be greater than any time which has happened. Thus, PARTIQLE may discharge the ordering constraint `sleep.startTime < doTrans.endTime` at `sleep.start` and thus the node `sleep.start` is a post-dominator. Although the query means to check `sleep.endTime < doTrans.endTime`, timing analysis allows PARTIQLE to infer that it can output results before either event in this predicate has happened. Performing query evaluation earlier reduces the need to hold records in the runtime tables.

Consider a query with a post-dominator event p. Since p is the last event in a query result, p triggers query evaluation; because it is the post-dominator, it must be the only event that does so. When p happens, since any events which join with it must have already happened, no record for p needs to be stored. If p is an end event, the start record for p may be deleted. If p is a start event, no runtime table is necessary for p. Notice that the absence of p in its runtime table may cause subsequent retention checks to fail, thus allowing deletion of other records.

3.4 Query Evaluation

Query evaluation is the process of searching runtime tables for query results. Any algorithm for evaluating SQL queries can be adapted to evaluate PARTIQLE queries. In PARTIQLE query evaluation is triggered when PARTIQLE observes an event that may be the last in a query. At that point, PARTIQLE performs query evaluation with respect to that event record — that is, it searches the runtime tables for records that match.

In practice the order the tables are searched, the *join order* in the jargon of relational databases, has a major effect on performance. It is cheaper to apply joins with tables having a small number of matching records before applying joins with tables having many matching records. For each event E that triggers query evaluation, all other records in E's table have already triggered query evaluation, so we start the join with the single record for event E. The next table in the join order is the one whose join predicates with E's table are most selective. We heuristically order the selectivity of operations as follows: equals on object fields is most selective, then equals on other fields, then any of $<$, $>$, $\geq$, $\leq$, and finally $\neq$ is least selective. The remaining tables are ordered by repeatedly choosing the table whose join predicates with the already joined tables are most selective and making it next in the join order.

PARTIQLE uses a nested-loop join that exploits indexes in our runtime tables. Pseudocode for generating query evaluation code is shown in Figure 7. Each recursive call to `generateQueryCode` generates another nested loop. Each loop iterates through the records of a runtime table. For a join, if a record satisfies all checkable join predicates with the result so far, the record is added to the result and query evaluation enters the next nested loop. Left antijoin proceeds to the next nested loop if no records match the result so far. In the case of left antijoin, no record is saved (because of the scoping of names in queries, no other predicates can refer to the antijoined table). The base case of `generateQueryCode` generates code to print query results. Recursion starts at `2` because the first two slots in result (and the first table or two in the join order) correspond to the event which triggered query evaluation. That event's runtime (start and end) tables are not searched — query evaluation is with respect to the newly generated record that triggered query evaluation.

Our current implementation does not handle left antijoins where the right side is other than a `ObjectAlloc` or `MethodInvoc`. We have yet to encounter a query where this shortcoming matters.

3.5 Example of Optimized Instrumentation

In this section we show the PARTIQLE instrumentation to answer the example query from Section 1.

This query requires only one runtime table: `tbl_doTrans` for `MethodInvoc doTrans`. As discussed in Section 3.3.4, the start event for `sleep()` is the post-dominator for this query, and thus requires no table. No fields are required from end events, so no tables are required to store them. The example also has a number

```
// data gathering -- Method Invocations

// Code enclosed in {}s is generated code.
// Literal generated instructions are denoted in italics.
// For an object x of type Code, Paste(x) pastes x's contents into a {} block.

for each method M in user code
    Code methodBody = code for M;
    Code startCode = { skip };
    Code endCode = { skip };
    for each MethodInvoc E in query
        if (M satisfies static predicates) then //Section 3.3.1
            startCode = {
                Paste(startCode);
                synchronized(partiqleLock)
                    startRec = new startRecord_E(getTime(), getThread(), ...);
                    if (dynamicPreds(startRec)) then //Section 3.3.2
                        if (admissionChecks(startRec)) then //Section 3.3.3
                            startTable_E.add(startRec); //Section 3.1
            };
            Code queryEvalCode_E = (
                if (E may be last event in query) then //Section 3.2
                    { query evaluation with respect to startRec,endRec; } //Section 3.4
                else
                    { skip; }
            );
            endCode = {
                Paste(endCode);
                synchronized(partiqleLock)
                    endRec = new endRecord_E(getTime(), return value);
                    if (!dynamicPreds(startRec, endRec)) then //Section 3.3.2
                        startTable_E.remove(startRec); //Section 3.1
                    else if (!retentionChecks(startRec,endRec)) then //Section 3.3.3
                        startTable_E.remove(startRec); //Section 3.1
                    else
                        endTable_E.add(endRec); //Section 3.1
                        Paste(queryEvalCode_E);
            };
    end for;
    methodBody = methodBody with endCode inserted before returns;
    methodBody = {
        Paste(startCode);
        try
            Paste(methodBody);
        catch (Throwable exn)
            Paste(endCode);
            throw exn;
    };
    replace body of M with methodBody;
end for;
```

Figure 6: Pseudocode for generating instrumentation to create and store event records for `MethodInvoc` events.

```
//simplified query evaluation

// Code enclosed in {}s is generated code.
// Literal generated instructions are denoted in italics.
// For an object x of type Code, Paste(x) pastes x's contents into a {} block.

Code[] queryEvalCode;
for each event E that could occur last
   EventRecord[] joinOrder = computeJoinOrder(E);
   queryEvalCode[E] = generateQueryCode(joinOrder, 2);

// The inductive assumption is that result[0..i-1] has event records
// for the events in joinOrder[0..i-1].
// The recursion starts at 2 because query evaluation is always with
// respect to a particular event record.

Code generateQueryCode(EventRecord[] joinOrder, int i)
   if (i>joinOrder.length) then
      return { output query results; };
   Code restOfQuery = generateQueryCode(joinOrder, i+1);
   E = joinOrder[i];

   // For joins the generated code looks through the table for joinOrder[i]
   // and finds any records which match the result so far.  If a record matches,
   // it is added to the result and the next table in the join order is considered.
   if (E requires a join) then
      return {
         for each r in runtimeTable_E
            if (joinPreds(result,r)) then
               result[i] = r;
               Paste(restOfQuery);
            //continue looping through runtimeTable_E
      };
   // For left antijoins the generated code looks through the table for joinOrder[i]
   // and finds any records which match the result so far.  If no records match,
   // the next table in the join order is considered.
   else if (E requires a left antijoin) then
      return {
         boolean match = false;
         for each r in runtimeTable_E
            match = match || antiJoinPreds(result,r);
         if (!match)
            Paste(restOfQuery);
      };
```

Figure 7: Pseudocode for generating query evaluation code.

```
public class DB {
  // ...
  B b;
  void doTransaction() {
    StartRecord_doTrans r;
    synchronized(partiqleLock) {
      r = tbl_doTrans.add(getTime(), getThread());
    }
    try {
      b.y(); /* method body */
    } catch (Throwable e) {
      synchronized(partiqleLock) {
        tbl_doTrans.remove(r);
      }
      throw e;
    }
    synchronized(partiqleLock) {
        tbl_doTrans.remove(r);
} } }

public class B {
  // ...
  void y() { /* method y is unchanged */
    sleep();
  }
  void sleep() {
    synchronized(partiqleLock) {
      if (tbl_doTrans.check(getThread())) {
        //query evaluation
        print(...);
} } } }
```

Figure 8: Optimized instrumented code for Section 1 example.

of static predicates (expanded here from syntactic sugar):

```
        doTrans.mname = 'doTransaction'
  AND   doTrans.declClass = 'DB'
  AND   sleep.mname = 'sleep'
  AND   sleep.declClass = 'B'
```

Only `DB.doTransaction()` needs to be instrumented to add records to `tbl_doTrans` and only `sleep()` needs to be instrumented to perform query evaluation.

Since the (optimized away) table for `sleep.start` is always empty, the retention check at the end of `DB.doTransaction()` always fails. Thus, the instrumentation at the end of `DB.doTransaction()` unconditionally removes the record from `tbl_doTrans`, and `tbl_doTrans` contains only records for method invocations that have begun but not completed.

Query evaluation at the start of `sleep()` gets the current time and thread and finds records in `tbl_doTrans` with the same thread and earlier `startTime`. For each such record X it outputs $\langle X.\texttt{startTime}, \texttt{startTime}\rangle$. PARTIQLE proves that the timing constraint need not be checked during query evaluation since every X we find in `tbl_doTrans` must have started before now, i.e. `sleep.start`. Thus we get the code shown in Figure 8. We delete a particular record by deleting all records with the record's start time (start times are unique).

In Figure 8, the only required operations are add a record, delete a record with a given start time, and check to see if there are any records matching a particular thread. TABLEEMITTER implements this structure with two hash tables: one mapping a start time to a doubly-linked list of records with that start time and one mapping a thread to a doubly-linked list of records for that thread. Thus *add*, *remove* and *check* are all constant-time operations. The *remove* operation is constant time because start times are unique. While we have not implemented it, it is possible to do even better in this example. Uniqueness of the start times could allow TABLEEMITTER to reduce the data structure to a single hash table mapping a thread to the count of records for that thread, which can then be specialized to a thread-local integer that does not require synchronization.

Example	Methods	Description
`db`	35	database management (SpecJVM98)
`compress`	44	LZW (de)compression (SpecJVM98)
`lisp`	104	Lisp interpreter
`jscheme`	110	Scheme interpreter
`mips`	112	architectural simulator
`mtrt`	184	multi-threaded ray-tracer (SpecJVM98)
`mpeg`	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)

Figure 9: Benchmark Programs.

4. EXPERIMENTS

4.1 Benchmarks

Our benchmark programs are listed in Figure 9. They include programs from the SpecJVM98 suite [26] and the Apache Tomcat [2] version 4 Web server and servlet container (which includes the Xerces XML parser and other components). We report code size as the number of methods in the application. However, we also instrument the Java class library, so the actual code subject to instrumentation is much larger than reported here (although hard to measure directly).

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 fifty times. This test exercises a number of JSPs and servlets.

4.2 Queries

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

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

2. `EqualObjectsButInequalHashCodes` checks that if two objects are deemed equal by `equals`, then they have the same `hashCode`.

3. `InequalObjectsButEqualHashCodes` checks that if two objects are not deemed equal by `equals`, then their hash codes are 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.

4. `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 [28]. 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.

5. `DelayedClose` searches for `close()` operations on stream objects that have not been read or written to for a certain length of time — one second in these tests. 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 the same IBM white paper [28].

   We constrain the search to the Apache stream classes used by Tomcat because instrumenting basic Java streams causes reentrancy problems for the PARTIQLE runtime support library. We only report results for this query on Tomcat since none of the other benchmarks uses Apache stream classes.

6. `CompareToReflexive` searches for `Comparable` objects o which return nonzero from a call to o.`compareTo`(o).

7. `CompareToAntisymmetric` searches for objects x and y such that the sign of x.`compareTo`(y) is not the opposite of the sign of y.`compareTo`(x). Because PARTIQLE currently lacks a "sign" function, we construct three queries covering the cases where x.`compareTo`(y) < 0, $= 0$, and > 0. We report results for the first case.

8. The queries `CompareToNonZeroButEqualsTrue` and `CompareToZeroButEqualsFalse` check that for all `Comparable` objects x and y, x.`equals`(y) is true if and only if x.`compareTo`(y) returns zero.

4.3 Overhead of Instrumentation

We measured the baseline performance of our benchmarks without instrumentation and compared them to the instrumented performance for each of the queries. We recorded the wall-clock running time of each run in seconds and the heap memory high-water mark in megabytes (measured by sampling Java's `System.totalMemory() - System.freeMemory()` every 500 milliseconds). The experiments were carried out on an unloaded dual-processor 550MHz Pentium III with 1.5 GB of memory, running IBM's JDK 1.4.2 on Fedora Core 1 Linux.

Figure 10 shows the runtime overhead as a ratio of the runtime with instrumentation to the runtime without instrumentation; the second line shows running time in seconds for the uninstrumented program. Figure 11 shows the ratio of maximum memory consumed by the instrumented program during its run to that of the uninstrumented program; the second line shows maximum memory usage for the uninstrumented program.

These results show that the overhead of PARTIQLE is quite reasonable for use in a testing environment. 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 behavior, which can be sensitive to small changes in program behavior, especially for short-lived benchmarks.

4.4 Evaluation of Optimizations

In this section we report on the effect of the various query optimizations: static predicates, dynamic predicates, admissions checks, exploiting post-dominators, and join ordering. Not surprisingly, different optimizations are important in different circumstances. Overall, we find that using static predicates and join ordering are particularly important for achieving reasonable performance.

Checking static predicates at instrumentation time is very important and only grows in importance with the size of the program. For example, if a query only focuses on events that happen in a small number of methods, instrumenting every method in a large application is extremely wasteful. While we did not evaluate this claim quantitatively (it is difficult to turn off use of static predicates in our system) our experience with early versions of PARTIQLE was that having every method invocation or object allocation run instrumentation can make the program run hundreds of times more slowly.

To measure the performance impact of checking simple dynamic predicates early, we modified PARTIQLE to check these predicates later, at query evaluation time instead of when the relevant event record is generated, and re-ran our queries. This delay causes event records that never participate in query results (because they do not satisfy a simple dynamic predicate) to be stored and traversed during admission checks and query evaluation. Figure 12 shows the run time and maximum memory usage in this situation as a fraction of the run time and maximum memory usage reported in Figures 10 and 11. As shown in the figure, without early filtering on simple dynamic predicates run time and memory usage increase; in several situations overhead is much worse than double what it is without this optimization.

To measure the performance impact of admission checks, we ran the `StringConcats` query, the only query which makes admission checks, with all admission checks turned off. Admission checks trade time for space (thus improving scalability). That is, we normally expect admissions checks to make the program run more slowly with the benefit of conserving space. Figure 13 shows the results, again as a ratio of time or memory usage with optimization disabled to time or memory usage with optimization enabled. As expected, some queries run faster with admission checks disabled. Admission checks substantially help `jack`, the benchmark with the highest overhead. On most of the other benchmarks, however, their effect is lost in the noise.

To measure the performance impact of join order we ran our queries with alternate join orders. In most cases there was only one other join order and we chose it. For `StringConcats`, we reversed the order. Figure 14 shows the results. To save space, we omit queries with only one join order and queries for which changing the join order made no difference. As expected, join order has no effect on memory consumption, so we report only slowdown. Cells with ∞ indicate that the query took longer than an hour and was killed.

One notable cell is `CompareToZeroButEqualsFalse` applied to `db`. The alternate join order actually shows a substantial (roughly 10x) speed up. PARTIQLE's static selectivity heuristic for determining join order ranks the two join orders equally and arbitrarily chooses between them. Choosing a join order dynamically, based on the sizes of the tables and the selectivity of the join predicates, is standard in relational databases and would benefit PARTIQLE in cases such as this one.

Our implementation is not well suited to measuring the benefit of our post-dominator optimizations. Since these optimizations elide storage of records in runtime tables and remove checks to delete records, exploiting a post-dominator can only improve both time and space. Post-dominator optimizations apply to seven of our nine example queries.

	db	compress	lisp	jscheme	MipsSimulator	mtrt	mpegaudio	jack	jess	javac	tomcat
baseline time (seconds)	37	21	5	9	10	12	17	22	19	34	54
CompareToAntisymmetric	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
CompareToReflexive	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
HashCodeConsistent	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.8	1.2
DelayedClose											1.1
CompareToNonzeroButEqualsTrue	1.4	1.0	1.0	1.0	1.0	1.0	1.0	1.0	10.0	2.8	1.1
CompareToZeroButEqualsFalse	17.6	1.0	1.0	1.1	1.0	1.0	1.0	1.0	9.9	2.5	1.1
EqualObjectsButInequalHashCodes	1.1	0.9	1.0	1.0	1.0	1.0	1.0	1.0	12.2	4.9	2.0
InequalObjectsButEqualHashCodes	8.7	1.0	1.0	1.1	1.0	1.0	1.0	1.0	14.0	3.2	1.3
StringConcats	1.2	1.2	1.4	1.9	2.7	1.3	1.2	9.7	2.0	5.8	3.9

Figure 10: Runtime Overhead (instrumented time / uninstrumented time).

	db	compress	lisp	jscheme	MipsSimulator	mtrt	mpegaudio	jack	jess	javac	tomcat
baseline memory (MB)	18	10	3	3	16	11	2	4	8	28	13
CompareToAntisymmetric	1.0	1.2	1.1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
CompareToReflexive	1.0	1.0	1.2	1.0	1.0	1.3	1.0	1.1	1.0	1.0	1.1
HashCodeConsistent	1.0	1.2	0.9	1.0	1.0	1.0	1.0	1.1	0.8	1.4	1.4
DelayedClose											5.8
CompareToNonzeroButEqualsTrue	1.0	1.2	0.9	1.2	0.9	1.1	1.0	1.1	94.2	2.3	1.5
CompareToZeroButEqualsFalse	5.6	1.0	0.9	1.2	0.9	1.2	1.0	1.0	94.7	2.0	2.2
EqualObjectsButInequalHashCodes	1.2	1.0	1.0	1.2	1.0	1.0	1.0	1.1	110.4	2.9	1.7
InequalObjectsButEqualHashCodes	5.5	1.2	0.9	1.2	1.0	1.0	1.0	1.1	101.9	2.5	2.0
StringConcats	1.2	1.0	1.2	1.5	1.4	1.7	1.5	22.9	2.1	4.7	13.4

Figure 11: Memory Overhead (instrumented max memory / uninstrumented max memory).

	db	compress	lisp	jscheme	MipsSimulator	mtrt	mpegaudio	jack	jess	javac	tomcat
CompareToReflexive	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
HashCodeConsistent	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
DelayedClose											1.0
CompareToNonzeroButEqualsTrue	12.6	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.2	1.1	1.0
CompareToZeroButEqualsFalse	0.9	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.5	1.4	1.1
EqualObjectsButInequalHashCodes	7.4	1.1	1.0	1.0	1.0	1.0	1.0	1.0	2.9	1.1	1.1
InequalObjectsButEqualHashCodes	0.8	1.1	1.0	1.0	1.0	1.0	1.0	1.0	1.2	1.2	1.0
StringConcats	1.0	1.1	1.0	1.0	1.0	1.0	1.0	1.0	1.1	1.0	1.0

	db	compress	lisp	jscheme	MipsSimulator	mtrt	mpegaudio	jack	jess	javac	tomcat
CompareToReflexive	1.0	1.0	0.7	1.0	0.9	0.8	1.0	0.9	1.0	1.1	1.0
HashCodeConsistent	1.0	1.0	1.4	1.0	1.0	1.0	1.0	0.9	1.2	1.0	0.9
DelayedClose											1.0
CompareToNonzeroButEqualsTrue	7.7	1.0	1.3	1.0	1.0	0.9	1.0	1.1	2.0	1.4	1.7
CompareToZeroButEqualsFalse	1.0	1.2	1.0	1.0	1.1	1.0	1.0	0.9	1.6	1.4	1.2
EqualObjectsButInequalHashCodes	5.8	1.2	0.9	1.0	1.1	1.5	1.0	0.9	1.7	1.2	1.5
InequalObjectsButEqualHashCodes	1.0	0.8	1.3	1.0	1.0	1.0	1.0	0.8	1.8	1.5	1.2
StringConcats	1.0	1.0	0.8	0.9	0.9	1.0	1.0	1.0	1.0	1.0	1.0

Figure 12: Runtime (top) and Memory (bottom) overhead for delaying dynamic predicates until query evaluation (non-optimized / regular optimized). Higher numbers mean checking simple dynamic predicates early is a win.

	db	compress	lisp	jscheme	MipsSimulator	mtrt	mpegaudio	jack	jess	javac	tomcat
StringConcats	1.0	0.9	0.8	0.7	0.6	0.9	0.9	3.6	0.8	0.6	0.8

	db	compress	lisp	jscheme	MipsSimulator	mtrt	mpegaudio	jack	jess	javac	tomcat
StringConcats	1.0	1.2	0.9	1.0	1.0	1.0	0.9	1.2	1.1	1.0	1.0

Figure 13: Runtime (top) and Memory (bottom) overhead for not doing any admission checks (non-optimized / regular optimized). Higher numbers mean admission checks are a win.

	db	compress	lisp	jscheme	MipsSimulator	mtrt	mpegaudio	jack	jess	javac	tomcat
CompareToNonzeroButEqualsTrue	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.1	1.0
CompareToZeroButEqualsFalse	0.1	0.9	1.0	1.0	1.0	1.0	1.0	1.0	0.9	1.2	1.0
EqualObjectsButInequalHashCodes	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.8	∞	15.0
InequalObjectsButEqualHashCodes	1.7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.6	∞	1.2
StringConcats	1.1	1.1	1.2	1.1	1.4	1.2	1.1	∞	∞	∞	∞

Figure 14: Runtime Overhead for Alternate Join Order (alternate join time / regular time). Higher numbers mean PARTIQLE's join order is better.

4.5 Query Results

Our queries discovered several interesting program behaviors. When PARTIQLE outputs a query result, 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 online query execution. These issues could have been found with custom dynamic analysis or even in some cases with simple static analysis; however, writing PTQL queries is an extremely quick way to look for new kinds of behaviors.

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 code is $O(n^2)$ in the length of the tokens. Although `jack` is a very well-studied benchmark, we believe this bug was previously unknown.

Applying `HashCodeConsistent` to `tomcat` found a situation in the `org.apache.xerces.validators.common` package of the Xerces XML parser where `CMStateSet` objects return different `hashCode`s at different times. A code fragment in `DFAContentModel` in the same package looks like:

```
CMStateSet newSet = null;
HashMap states = new HashMap();
for (...) {
    if (newSet == null) {
        newSet = new CMStateSet();
    } else {
        newSet.clear();
    }
    ...
    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 `DelayedClose` to `tomcat` detected examples of HTTP response streams being closed a few seconds after the last write to the stream. The closure delays are associated with HTTP requests for non-existent documents, but they appear only intermittently, and only when our test harness uses the HTTP/1.1-aware `URLConnection` Java class to issue the requests; simple HTTP requests issued over a plain socket do not cause the problem to manifest. We have not isolated the exact issue, but it appears to be a problem in the `tomcat` HTTP stack.

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 code 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 could slow JVM startup, 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 IBM JDK 1.4.2.

4.6 Matching Method Pairs in Eclipse

In this section we compare PARTIQLE to PQL (see Section 5.2.1), a related system for analyzing dynamic program behavior, by reproducing one of the experiments from [24].

In many APIs there are methods which must be invoked in a certain order. A requirement is of the form "a call to method A must always be followed by a call to method B." We instrumented Eclipse [6] version 3.0.0, a Java integrated development environment, with queries to check 8 such properties. For example, `o.createWidget()` must always be followed by a call to `o.destroyWidget()`. To look for query results, we ran the instrumented Eclipse and exercised various pieces of its functionality (for example creating new projects and classes, refactoring code, opening and closing projects and files). The query below is representative of the 8 we ran.

```
SELECT A.implClass
FROM MethodInvoc('*.createWidget') A
    JOIN ObjectAlloc o
        ON A.receiver = o.obj
    LEFT ANTIJOIN MethodInvoc('*.destroyWidget') B
        ON B.receiver = o.obj
```

The results of running these 8 queries are shown in Figure 4.6. As in the PQL paper [24], we report the number of concrete types for `o` that match the query. The results do not exactly match those in [24]. It's likely that our experiments involved different sets of plug-ins and different workloads. Regardless, both PARTIQLE and PQL find a substantial number of bugs in Eclipse.

5. RELATED WORK

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

5.1 Program Monitors

Given a specification of acceptable runtime behavior, a program monitor observes program execution (perhaps as a stream of events or state transitions) and signals an error when the execution violates the specification. Program monitoring systems vary in the nature and expressive power of their languages for specifying monitors, in their notion of an event, and in the efficiency of their implementations.

Bates proposes [4] an Event-Based Behavioral Abstraction (EBBA) to monitor and debug distributed systems. He presents a regular expression-like specification language in which user-defined events in the execution of the system correspond to characters in the alphabet. The user chooses places in source code to emit events and the system adds instrumentation. EBBA's specification language cannot express data constraints among events.

Rapide [22] is a language for constructing executable models of distributed systems. Rapide abstracts the system into a network of processes that communicate via events. An execution of the model results in a set of partially ordered events. Rapide modules may specify patterns of events that must not occur in the partial order.

Method *A*	Method *B*	Instrumentation points	Types matching query
`createWidget`	`destroyWidget`	17	14
`register`	`deregister`	63	18
`acquireXMLParsing`	`releaseXMLParsing`	4	0
`acquireDocument`	`releaseDocument`	8	0
`install`	`uninstall`	113	8
`installBundle`	`uninstallBundle`	10	3
`start`	`stop`	323	7
`startup`	`shutdown`	150	6
Total		688	56

Figure 15: Results of running matching pairs query on Eclipse

Some processes in the model may be replaced by a concrete implementation of that process together with code to map the behavior of the implementation to events. Thus can one test the conformance of actual systems to a model. Rapide's event patterns do support data and time constraints, but lack any mechanism (like PTQL's antijoin) for specifying events that do not happen.

The monitoring and checking (MaC) framework [16, 15] allows the user to define events based on changes to values of variables and calls to methods. The user tracks state and specifies undesirable behavior via a reactive system. Thus, in MaC, it is the user's burden to define events and track system state.

Eagle [3] is a framework for defining finite trace program monitoring logics, specifying formulas to monitor, and augmenting Java programs with these monitors. Eagle can define logics with past and future time temporal operators, interval logics, real time constraints, data constraints, and other idioms. The runtime component of Eagle rewrites monitoring formulas on the fly as the program executes and reports whether they have been satisfied at program termination. We are unaware of an evaluation of Eagle's efficiency or how it could be optimized.

5.2 Trace Query Engines

Closely related to program monitors are trace query systems. Like PARTIQLE, these systems answer queries about the sequence of events during program execution.

Finkelbeiner et al. [9] extend unquantified LTL to collect statistics over program runs. Example statistics include counts of a particular event and average number of retransmissions for a packet. Because the query language is based on LTL without quantification, there is no support for data constraints between events. Expressing partially ordered events in LTL is also burdensome.

Goldszmidt et al. [10] propose a system for debugging concurrent programs. Their system records a trace of events in an offline database. The user can then query this database with LTL formulas or replay sequences of events.

More similar to PARTIQLE is the proposal for a program monitoring and measuring system (PMMS) by Liao and Cohen [19, 18]. Like PARTIQLE, PMMS compiles a high level, relational query language (similar to the domain calculus) over program traces to program instrumentation. Our contributions over PMMS are in the sophistication and completeness of our implementation and optimizations, the application to Java (including handling of threads and objects, not addressed by PMMS), and a more thorough evaluation, including demonstration of queries that yield insightful results on real programs. The experimental evaluation of PMMS consists of instrumenting one program and reporting the number of checks saved by using static and dynamic filtering versus the naive approach of instrumenting everything. In the vocabulary of Section 3, PMMS has a timing graph but does not use inference rules to infer additional edges for the graph. PMMS only supports incremental output of query results on queries for which the events in the query are enclosed by a single *interval event* (e.g., a method invocation) – the output comes at the closing event in the interval; PARTIQLE is more general. PMMS also does not create custom data structures for its runtime tables.

The Hyades project [12] (now apparently subsumed by the Eclipse Test and Performance Tools Platform) is developing a data model for traces of Java programs. The data model is expressed in the Eclipse Modeling Framework [7] 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. We found, however, that for our purposes the navigational nature of OCL queries (as opposed to PTQL's relational queries) gives them unnecessary structure and complexity.

Our contribution over this prior work is in our combination of a relational data model for program traces, declarative query language, handling of real-time and data constraints, automatic instrumentation of user code, and online query execution. Also significant is our demonstration that PARTIQLE has acceptable performance for a variety of programs and queries, and that it is possible to create useful and concise queries.

5.2.1 Program Query Language (PQL)

PQL [24] is another proposal for instrumenting Java programs to answer declarative queries about program behavior, developed concurrently and independently of our efforts. The goals of PQL are the same as those of PARTIQLE, but our query languages differ. PQL does more sophisticated static analysis of the Java source than PARTIQLE, including a static pointer analysis. PQL queries can recursively call subqueries, which allows queries about properties like transitive data flow. PQL, however, lacks any notion of real time.

Of the four experiments reported in [24], PARTIQLE can express three in their full generality: serialization errors, mismatched method pairs (see Section 4.6), and lapsed listeners. PARTIQLE can express a less powerful version of the SQL injection attack query that does not consider transitive data flow.

Of our queries, PQL can express `StringConcats`. With a few straightforward extensions, however, it could express all of them. PQL cannot express `DelayedClose` because of the time constraint. PQL cannot express our other queries because the queries reason about integer and boolean return values of methods.

The dynamic query execution strategies of PQL and PARTIQLE make different tradeoffs. PQL uses a state machine approach, essentially constructing partial query results eagerly. PARTIQLE

keeps tables of each event kind and joins them only when a full query result may be available. If there are k kinds of events and a program trace contains E_i events of kind i, PQL risks using space proportional to the product $(\prod_{i=1}^{k} E_i)$ in cases where there are many partial query results. With PARTIQLE, the memory requirement is proportional to the sum $(\sum_{i=1}^{k} E_i)$. PARTIQLE on the other hand risks storing records that will never participate in query results (a risk that admission and retention checks mitigate). It is clear from the results in [24] that PQL's implementation choice works well in practice for many useful queries. PARTIQLE's implementation choice allows for reasonable performance even when there are many events, few ordering constraints, and many partial results.

5.3 Predicate-Guessing Systems

DIDUCE [11] instruments Java programs to track invariants at various program sites. The violation of an apparent invariant, especially one that had been true many times, yields a warning and a relaxation of the monitored property. Deviations from the norm often indicate bugs or interesting facts about program execution. Liblit et al. [20, 21] instrument programs to use random sampling of program points to gather small amounts of data from each of many executions. Statistical analysis correlates certain observations with program failure, giving the developer insight into what situations cause bugs. Daikon [8] intensively instruments programs to discover likely invariants.

These systems are complementary to PARTIQLE. While PARTIQLE provides sparse instrumentation to answer specific questions, these systems use general instrumentation to find interesting facts about a program.

5.4 Aspect-Oriented Programming

In traditional object-oriented code, certain concerns, such as what data to record in a log file, must be spread across many classes. Aspect-oriented programming seeks to concentrate the code for these *crosscutting* concerns in a single module instead of spreading it across many modules. AspectJ [14], an aspect-oriented extension to Java, accomplishes this concentration via *aspects*, *join points*, *pointcuts*, and *advice*. A *join point* is a point in the program's execution, such as a method entry or return, an object allocation, or a field write. A *pointcut* is a set of join points. *Advice* is code to be executed either before, after, or instead of the code at a join point. Advice code has access to data about the join point that triggered its execution. An *aspect* is a module that contains pointcuts and advice. For instance, a logging aspect might contain a pointcut that identifies the points in a program where data is to be output to a log file and advice to actually do the logging.

An important feature of an aspect-oriented programming language is its language for defining pointcuts. The pointcut language of AspectJ is somewhat limited; other proposals [25, 27, 1] are more expressive. In particular, these proposals allow advice to execute based on the history of program execution.

The goals of PTQL and PARTIQLE are similar, but not identical, to the goals of languages for specifying pointcuts (or tracecuts or tracematches). The common concern is with identifying points in program execution.[5] Indeed, with a sufficiently expressive pointcut language, one could implement a PTQL query with a pointcut to specify the interesting patterns of events and advice to print out desired query results. Conversely, any sufficiently expressive pointcut language has to solve many of the same problems PARTIQLE does to obtain reasonable performance.

[5]The particular notion of "event" (or join point in aspect-oriented programming terms) is less important; any sufficiently rich data model can accommodate many kinds of primitive events.

A pointcut in ALPHA [25] is the set of join points that satisfy a logic query (in Prolog) against models of a program's abstract syntax tree, execution trace, heap, and static type assignments. ALPHA queries over the execution trace have similar expressive power to PARTIQLE queries.

Tracecuts [27] are a generalization of pointcuts that take a program's execution history into account. A tracecut is specified as a context free grammar over events in the program trace. When a sequence of events occurs that form a string in the tracecut's (context free) language, the advice for the tracecut executes. Specifying patterns of events as a context-free language allows, as in PQL (Section 5.2.1), a kind of recursion in subqueries, but makes it awkward to express queries in which events may happen in any order.

Neither the work on ALPHA nor the work on tracecuts address the problem of efficiently implementing their query languages. Neither language is expressive enough to reason about real time constraints.[6]

5.4.1 *Tracematches*

Like tracecuts, tracematches [1] generalize pointcuts. A tracematch is specified as a regular expression over events in the program trace. Whenever a string of events that matches a tracematch's regular expression occurs, that tracematch's advice is executed. In order to allow data constraints among events, each event may bind variables; the events in a match must agree on the value of all variables. Further data constraints can be checked in the advice (in many cases, PARTIQLE can check data constraints earlier).

The differing goals of PARTIQLE and tracematches lead to different notions of matching a program trace. A PTQL query matches all subsequences of a program trace that satisfy the query; all events that are not explicitly forbidden (with an antijoin) are allowed to occur between these matching events. PARTIQLE's notion of a match makes sense for its goal of examining program behavior. A tracematch regular expression matches all subsequences of the program trace where events in the match are separated by either events that are not in the tracematch's alphabet (of events) or that contradict a variable binding from a previous event in the match. In other words, all events in the alphabet (that do not contradict previous variable bindings) that are not explicitly allowed are forbidden. The tracematch notion of a match makes sense for its goal of executing advice when a particular series of events occurs.

Consider for instance a tracematch that defines three events corresponding to calls to `a`, `b`, and `c` respectively and that binds the receiver object to x:

```
tracematch(Object x) {
   // events
   // bind receiver object to x
   sym A after: call(* a()) && target(x);
   sym B after: call(* b()) && target(x);
   sym C after: call(* c()) && target(x);

   // pattern
   A B C

   // advice
   { System.out.prinln("a b c"); }
}
```

[6]We offer this observation by way of comparison; determining whether real time constraints are desirable for a pointcut language is beyond the scope of this paper.

Consider also a seemingly identical PTQL query:

```
SELECT *
FROM MethodInvoc('*.a') A
    JOIN MethodInvoc('*.b') B
        ON A.receiver = B.receiver
    JOIN MethodInvoc('*.c') C
        ON B.receiver = C.receiver
```

On the following program, the PTQL query returns a query result, but the tracematch does not execute its advice.

```
o.a(); o.c(); o.b(); o.c();
```

On the following program, the PTQL query returns two results and the tracematch executes its advice twice.

```
o.a(); q.a(); o.b(); q.b(); q.c(); o.c();
```

Unlike PARTIQLE, the implementation of tracematches eagerly constructs partial matches. Thus, the implementation of tracematches resembles that of PQL and the discussion in Section 5.2.1 applies.

5.5 Other Related Work

Lencevicius et al. [17] propose a query-based debugger. While the program is stopped at a breakpoint, the user may query the objects in the heap. A query consists of two parts: a search domain, specified as a tuple of types, and a constraint expression, specified as an arbitrary expression in the source language, over tuples of objects in the search domain. PARTIQLE's focus on relationships among events that occur during various points in program execution is quite different.

6. 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 PTQL can express queries that find interesting program behaviors and that PARTIQLE is efficient enough to run these queries on real Java programs.

7. REFERENCES

[1] C. Allen, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

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

[3] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of the 4th International Workshop on Runtime Verification (RV'04)*, 2004.

[4] P. C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*, 13(1), 1995.

[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] Eclipse. Project page at `http://www.eclipse.org/`.

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

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

[9] B. Finkbeiner, S. Sankaranarayanan, and H. Sipma. Collecting statistics over runtime executions. In *Proceedings of the 2nd International Workshop on Runtime Verification (RV'02), Electronic Notes in Theoretical Computer Science, Elsevier Science*, volume 70, 2002.

[10] G. S. Goldszmidt, S. Yemini, and S. Katz. High-level language debugging for concurrent programs. *ACM Transactions on Computer Systems*, 8(1), 1990.

[11] 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.

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

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

[14] 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.

[15] M. Kim. *Information Extraction for Run-time Formal Analysis*. Ph.D. thesis, CIS Dept., University of Pennsylvania, 2001.

[16] 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.

[17] R. Lencevicius, U. Hölzle, and A. K. Singh. Query-based debugging of object-oriented programs. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 304–317, New York, NY, USA, 1997. ACM Press.

[18] Y. Liao. *An Automatic Programming Approach to High Level Program Monitoring and Measuring*. Ph.D. thesis, Dept. Computer Science, University of Southern California, Los Angeles, CA., 1992.

[19] 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.

[20] 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.

[21] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, Illinois, June 12–15 2005.

[22] D. C. Luckham. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. In *POMIV '96: Proceedings of the DIMACS workshop on Partial order methods in verification*, pages 329–357, New York, NY, USA, 1997. AMS Press, Inc.

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

[24] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[25] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *European Conference on Object-Oriented Programming (ECOOP), Springer LNCS, 2005*, 2005.

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

[27] R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 159–169, New York, NY, USA, 2004. ACM Press.

[28] 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.