Dynamic Query-Based Debugging

Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh
Department of Computer Science
University of California, Santa Barbara, CA 93106
{raimisl,urs,ambuj}@cs.ucsb.edu
http://www.cs.ucsb.edu/{~raimisl,~urs,~ambuj}

Technical Report TRCS 98–34
December 1, 1998

Abstract. Program errors are hard to find because of the cause-effect gap between the time when an error occurs and the time when the error becomes apparent to the programmer. Although debugging techniques such as conditional and data breakpoints help to find error causes in simple cases, they fail to effectively bridge the cause-effect gap in many situations. Dynamic query-based debuggers offer programmers an effective tool that provides instant error alert by continuously checking inter-object relationships while the debugged program is running. To speed up dynamic query evaluation, our debugger (implemented in portable Java) uses a combination of program instrumentation, load-time code generation, query optimization, and incremental reevaluation. Experiments and a query cost model show that selection queries are efficient in most cases, while more costly join queries are practical when query evaluations are infrequent or query domains are small.

1. Introduction

Many program errors are hard to find because of a cause-effect gap between the time when the error occurs and the time when it becomes apparent to the programmer by terminating the program or by producing incorrect results [Eis97]. The situation is further complicated in modern object-oriented systems which use large class libraries and create complicated pointer-linked data structures. If one of these references is incorrect and violates an abstract relationship between objects, the resulting error may remain undiscovered until much later in the program's execution.

For example, consider the javac Java compiler, a part of Sun's JDK distribution. During a compilation, this compiler builds an abstract syntax tree (AST) of the compiled program. Assume that this AST is corrupted by an operation that assigns the same expression node to the field right of two different parent nodes (Figure 1). The parent nodes may be

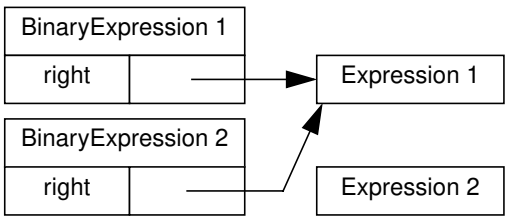

Figure 1. Error in javac AST

instances of any subclass of BinaryExpression; for example, the parent may be an AssignAddExpression object or a DivideExpression object, while the child could be an IdentifierExpression. The compiler traverses the AST many times, performing type checks and inlining transformations. During these traversals, the child expression will receive contradictory information from its two parents. These contradictions may eventually become apparent as the compiler indicates errors in correct Java programs or when it generates incorrect code. But even after discovering the existence of the error, the programmer still has to determine which part of the program originally caused the problem. How can we help programmers to find such errors as soon as they occur?

The programmer could try to use data breakpoints [WLG93], i.e., breakpoints that stop the program when the value of a particular field changes. However, data breakpoints (even if conditional) do not help to debug this error because they

are specific to a particular instance. With hundreds or even thousands of BinaryExpression instances, and in the presence of asynchronous events and garbage collection, the effectiveness of data breakpoints is greatly diminished. In addition, it is hard to express the above error as a simple boolean expression. The error occurs only if the expression is shared by another parent node—a relationship difficult to observe from the other parent or from the child itself. In other words, by looking just at the field right of some BinaryExpression object we cannot determine whether this object and its new field value are erroneous.

A programmer could also try to use another conventional tool, conditional breakpoints [Kes90]. Conditional breakpoints check a condition at a particular program location and stop the program if this condition is true. Conditional breakpoints fail to find our bug for the same reason: the condition cannot easily reference objects which are not reachable from the scope containing the breakpoint. Yet we must find exactly such an object—the BinaryExpression containing a duplicate reference to the child Expression object. To accomplish this task, the programmer could write custom testing code for use by conditional breakpoints. For example, the javac compiler could keep a list of all BinaryExpression objects and include methods that iterate over the list and check the correctness of the AST. However, writing such code is tedious, and the testing code may be used only once, so the effort of writing it is not easily recaptured. Finally, even with the test code at hand, the programmer still has to find all assignments to the field right and place a breakpoint there; in javac, there are dozens of such statements. In summary, the tool (conditional breakpoints) provides minimal support and the programmer ends up doing all the work "by hand".

A more effective way to check an inter-object constraint would be to combine conditional breakpoints with a query-based debugger [LHS97]. Similar to an SQL database query tool, a query-based debugger (QBD) finds all object tuples satisfying a given boolean constraint expression. For example, the query

```
BinaryExpression* e1, e2.    e1.right == e2.right && e1 != e2
```

would find the objects involved in the above javac error. The breakpoints would then carry the condition that the above query return a non-empty result. Unfortunately, even well-optimized QBD executions would be inefficient for this task. With hundreds or thousands of BinaryExpression objects, each query becomes quite expensive to evaluate, and since the query is reevaluated every time a conditional breakpoint is reached, the program being debugged may slow down by several orders of magnitude. (We will substantiate this claim in section 4.3.1.)

We propose a new solution, *dynamic* query-based debugging, which can overcome these problems. In addition to implementing the regular QBD query model, a dynamic query-based debugger continually updates the results of queries as the program runs, and can stop the program as soon as the query result changes. To provide this functionality, the debugger finds all places where the debugged program changes a field that could affect the result of the query and uses sophisticated algorithms to incrementally reevaluate the query. Therefore, a dynamic query-based debugger finds the javac AST bug as soon as the faulty assignment occurs, and it does so with minimal programmer effort and low program execution overhead.

We have implemented such a dynamic query-based debugger for Java. Our prototype is portable (written in 100% pure Java), and surprisingly efficient. Experiments with large programs from the SPECjvm98 suite [SPEC98] show that selection queries are very efficient for most programs, with a slowdown of less than a factor of two in most experiments. Through measurements, we determined that 95% of all fields in the SPECjvm98 applications are assigned less than 100,000 times per second. Using these numbers and individual evaluation times, our performance model predicts that selection queries will have less than 43% overhead for 95% of all fields in the SPECjvm98 applications. More complicated join queries are less efficient but still practical for small query domains or programs with infrequent queried field updates.

2. Query Model and Examples

Dynamic query-based debugging uses the query model proposed in QBD [LHS97]. The query syntax is as follows:

<Query>	::==	<DomainDeclaration> { **;** <DomainDeclaration> } **.** <ConditionalExpression>
<DomainDeclaration>	::==	<ClassName> [*] <DomainVariableName> { <DomainVariableName> }

The query has two parts: one or more DomainDeclarations that declare variables of class ClassName, and a ConditionalExpression. The first part is called the *domain part* and the second the *constraint part*. Consider another javac query:

```
FieldExpression fe; FieldDefinition fd.
fe.id == fd.name && fe.type == fd.type && fe.field != fd
```

The first part of the query defines the *search domain* of the query, using universal quantification. The domain part of the above example should be read as "for all FieldExpressions fe and all FieldDefinitions fd...". FieldExpression is a class name and its domain contains all instances of the class. If a "*" symbol in a domain declaration follows the class name (as in the javac query discussed in the introduction), the domain includes all objects of subclasses of the domain class, otherwise the domain contains only objects of the indicated class itself.

The second part of the query specifies the constraint expression to be evaluated for each tuple of the search domain. Constraints are arbitrary Java conditional expressions as defined in the Java specification §15.24 [GJS96] with certain syntactic restrictions. We disallow variable increments which have no semantic meaning in a query. We currently also disallow array accesses but plan to implement them in the future. Constraints can contain method invocations; we assume that these methods are side effect free.

Semantically, the expression will be evaluated for each tuple in the Cartesian product of the query's individual domains, and the query result will include all tuples for which the expression evaluates to true (similarly to an SQL select query). Conceptually, the dynamic debugger reevaluates a query after the execution of every bytecode, ensuring that no result changes are unnoticed. The debugger stops the program whenever the result changes. In reality, the debugger reevaluates the query as infrequently as possible without violating these semantics. In addition, the debugger will reevaluate only the part of the query that changed since the last evaluation. We describe the incremental reevaluation technique in detail in section 3.4.1.

We refer to queries with a single domain variable as *selection queries*; following common database terminology, we call the rest of the queries *join queries* because they involve a join (Cartesian product) of two or more domain variables. Join queries with equality constraints only (e.g., p1.x == p2.x) are *hash joins* because they can be evaluated more efficiently using a hash table [LHS97].

2.1 Examples

We now discuss examples of queries that illustrate the need for dynamic query debuggers.

2.1.1 Javac Compiler

What are examples of inter-object constraint violations that may be difficult to trace back to their origins? We have already discussed one possible error in the javac Java compiler in the introduction. Another error that could occur in javac involves the relationship between FieldExpression and FieldDefinition objects. Consider a situation where a FieldExpression object no longer refers to the FieldDefinition object that it should reference. Due to an error, the program may create two FieldDefinition objects such that the FieldExpression object refers to one of them, while other program objects reference the other FieldDefinition object (Figure 2). In other words, javac maintains a constraint that a FieldExpression object that shares the type and the identifier name with a FieldDefinition object must reference the latter through the field field. We can detect a violation of this constraint using the following query:

```
FieldExpression fe; FieldDefinition fd.
fe.id == fd.name && fe.type == fd.type && fe.field != fd
```

This complicated constraint can be specified and checked with a simple dynamic query, but it would be difficult to verify using conditional breakpoints.

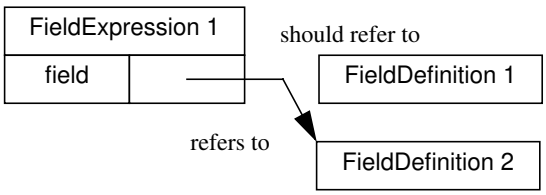

Figure 2. Another error in javac AST

2.1.2 Ideal Gas Tank Example

Another program we examined is an applet simulating a tank with ideal gas molecules. Though this applet is a simple simulation of gas molecules moving in the tank and colliding with the tank walls and each other, it has some interesting inter-object constraints. First, all molecules have to remain within the tank, a constraint that can be specified by a simple selection query:

```
Molecule* m.    m.x < 0 || m.x > X_RANGE || m.y < 0 || m.y > Y_RANGE
```

Another constraint requires that molecules not occupy the same position as other molecules, and the following query checks this constraint:

```
Molecule* m1 m2.    m1.x == m2.x && m1.y == m2.y && m1 != m2
```

This constraint is interesting because its violation is a transient failure. Transient failures disappear after some period of time, so even though the program behaves differently than the programmer expected, queries will not be able to detect failures if they are asked too late. The molecule collision error is such a transient failure—it will disappear as the molecules continue to move. However, the applet will behave erroneously: for example, molecules that should have collided with each other will pass through each other. Dynamic queries are necessary to find transient failures, as a delayed query reevaluation may fail to detect the error entirely.

3. Implementation

We have implemented a Java dynamic query-based debugger in pure Java. Java contains a number of features that simplified the implementation. We used the ability to write custom class loaders [LB98] to perform load-time code instrumentation. Java's bytecode class files proved simple to instrument. The debugger creates custom query evaluation code by using load-time code generation. The debugger can be ported to other languages (e.g. Smalltalk) that have an intermediate level format similar to bytecodes.

3.1 General Structure of the System

Figure 3 shows a data-flow diagram of the dynamic query-based debugger. To debug a program, the user runs a standard Java virtual machine with a custom class loader. The custom class loader loads the user program and instruments the bytecodes loaded, by adding debugger invocations for each domain object creation and relevant field assignment. The class loader also generates and compiles custom debugger code. After loading, the Java virtual machine executes the instrumented user program. Whenever the program reaches instrumentation points, it invokes the custom debugger code, which calls other debugger runtime libraries to reevaluate the query and to generate query results. The debugger currently does not handle multithreaded code.

The rest of this section discusses the most important parts of the debugger in more detail: how the debugger instruments a Java program, what parts it instruments, and how it evaluates a query.

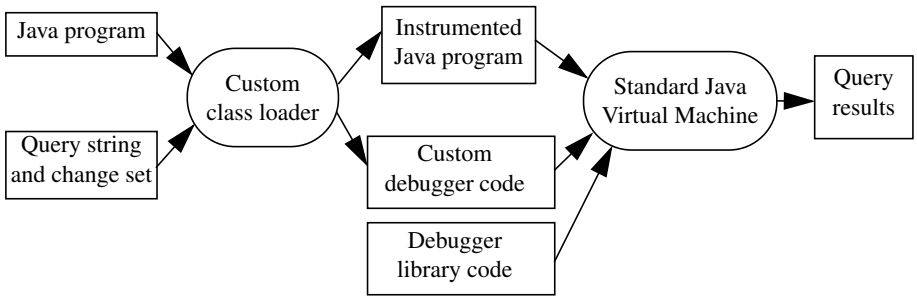

Figure 3. Data-flow diagram of dynamic query-based debugger

3.2 Java Program Instrumentation

To enable a dynamic query for a program, the user specifies a query string. The debugger then instruments class files to invoke the debugger after all events that may change the result of the query. The debugger finds assignments to the fields referenced in the query change set (section 3.3) and inserts debugger invocations after each one of them. The system also inserts debugger invocations after each call to a constructor of a domain object.

Figure 4 shows an example of the instrumentation process for a Java method. To instrument class files, the loader transforms them in memory into a malleable format using modified class file handling tools borrowed from the BCA class library [KH98]. Then the loader finds all putfield bytecodes that assign to the fields of interest—like field x in Figure 4—and replaces these putfield bytecodes with invokestatic bytecodes invoking debugger code. The system also inserts such debugger invocations after each call to a constructor of a domain object. When the debugger replaces a putfield bytecode with an invokestatic call, it also inserts the reference to the custom debug method of the DebuggingCode class into the constant pool of the instrumented class. The custom method takes two arguments: the object that the putfield would have updated—a Molecule object in the example—and the newValue value to be assigned to the object field. These objects are already on the stack before execution of the putfield, so they will be correctly passed as arguments to the debug method, and the debugger does no stack manipulation of the instrumented method. Since the original putfield has been replaced by the invokestatic bytecode, the custom debug method performs the assignment originally executed by the putfield. The debugger determines the name of the assigned field and the correct types of objects and values from the class file's constant pool. After instrumentation, the class loader transforms the code back into the class file format and passes the image to the default defineClass method.

The class loader instruments assignments and object constructors that influence the query result. The next section describes how the debugger determines which assignments and constructors to instrument.

3.3 Change Monitoring

The dynamic query debugger updates the query result every time the debugged program performs an operation that may affect the query result. Thus, the program being debugged has to invoke the debugger after every event that could change the query result. The query result may change because some object assigns a new value to one of its fields or because a new object is constructed. However, not all field assignments and object creations affect the query. We call the set of constructors and object field assignments affecting the results of a query the query's *change set*. Though we can use all assignments and all constructors as a conservative change set for any query, we are interested in a minimal change set for efficient query evaluation. Such a change set contains only constructors of domain objects and assignments to domain object fields referenced in a query.

Consider the Molecule query:

```
Molecule* m1 m2.    m1.x == m2.x && m1.y == m2.y && m1 != m2
```

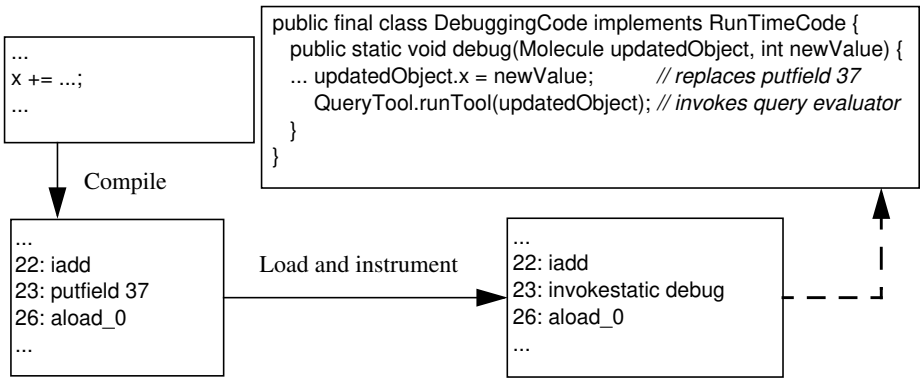

Figure 4. Java program instrumentation

The change set of this query consists of the constructors of the Molecule class and its subclasses as well as assignments to Molecule fields x and y. Assignments to other molecule fields such as color do not belong to the change set.

The change set of a query tells the class loader what assignments and constructors it should instrument. The debugger tracks all domain objects by maintaining domain object collections. Every time a domain object is created, the program invokes the debugger which places the new domain object into its domain collection. The debugger uses the domain collection in query evaluations to iterate through all domain objects. To maintain query correctness and to facilitate garbage collection, the debugger should allow the garbage collector to delete dead objects from domain collections. While such behavior can be implemented using weak pointers, we have not done so yet.

The change set of a query becomes complicated if constraints contain a chain of references. Consider a query for the SPECjvm98 ray tracing program:

```
IntersectPt ip.    ip.Intersection.z < 0
```

The Intersection field is a Point object, and the query result depends on its z value. The query result may change if the z value changes, or if a new value is assigned to the Intersection field. Furthermore, the Point object referenced by the Intersection field may be shared among multiple domain objects. In this case, a change in one Point object can affect multiple domain objects. A chain of references also occurs when a domain instance method invokes methods on objects referenced in its fields, and these methods in turn depend on the fields of the receiver. Tracking which objects accessed through a chain of field references influence which domain objects becomes a complicated task; for example, to do it efficiently, nested objects need to point back to the domain objects that reference them. To simplify the prototype implementation, we support only the explicit chains of references in the query, and we do not handle methods that access chains of references. Our debugger rewrites the query by splitting the chain into single-level accesses and by adding additional domains and constraints. For example, the ray tracing query above is rewritten as:

```
IntersectPt ip; Point* __Intersection.
ip.Intersection == __Intersection && __Intersection.z < 0
```

Chain reference splitting adds overhead by introducing additional joins into the query but it also allows users to ask more complex queries. The overhead can be an order of magnitude when a selection query is rewritten as a join query. We do not handle native methods, because their debugging is outside the scope of a Java debugger.

To summarize, we use the change set of the query to instrument the Java program. The instrumented program calls the debugger after every event that could change the result of the query, and the debugger reevaluates the query during each call.

3.4 Overview of Query Execution

In this section we describe what happens after an instrumented event occurs in the debugged program. Whenever the program invokes the debugger, it passes the object involved in the event. If the event is a field assignment, the

program also passes the new value to be assigned to the field. Figure 5 shows the control flow of the query execution. First, the debugger checks whether the changed object is a domain object. Consider a query that finds Id objects with a negative type code:

```
Id x.    x.type < 0
```

Here, Id is a subclass of the Expression class, and the type field is defined in Expression. Thus, the program may invoke the debugger when the type field inherited from the Expression class is assigned in an object of another Expression subclass. For example, the program invokes the debugger after assigning the type field in an ArithmeticExpression object. This object shares the type field with the domain class objects, but it does not belong to the query domain, so the debugger immediately returns to the execution of the user program without reevaluating the query.

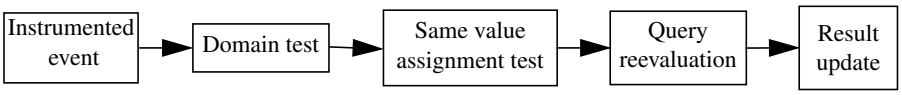

Figure 5. Control flow of query execution

If the object passes the domain test, the debugger checks whether the value being assigned to the object field is equal to the value previously held by the field. For example, some molecules do not move in the ideal gas simulation, yet their coordinates are updated at each simulation step. Such assignments do not change the result of the query and can be ignored by the debugger[1]. The debugger does not perform this test if the invoking event is an object creation.

After these two tests, the debugger starts reevaluating the query. Our previous work on non-incremental query-based debuggers [LHS97] contained a query evaluation algorithm similar to the evaluation of a relational database join coupled with a selection. The dynamic query-based debugger improves upon the previous algorithm by using incremental reevaluation as discussed below.

3.4.1 Incremental Reevaluation

When the program invokes the debugger, it passes the changed object to the debugger. From the properties of our change sets, we know that this object is the only object that changed since the last query evaluation. Consequently, a full reevaluation of the query for all domain objects is unnecessary. We use incremental reevaluation techniques developed for updates of materialized views in databases [BC79, BLT86] to speed up query execution. Consider a query, a join of three domains A * B * C, e.g.,

```
A a; B b; C c.    a.x == b.y && b.z < c.w
```

The "*" symbol denotes a Cartesian product with some selection constraint; the "+" symbol below denotes set union. If an object of domain B changes, the new result of the query is

$$A * (B + \Delta B) * C = (A * B * C) + (A * \Delta B * C)$$

The first part of the result is the result of the previous query evaluation. The debugger stores this result—usually empty for assertion queries—and does not need to reevaluate it. The second part of the result contains only the changed object (ΔB) of domain B combined with objects of the other domains. The debugger evaluates the changed part in the same way as it would evaluate the whole query. Figure 6 shows an incremental evaluation of changes in the query result. The execution starts with the changed object ΔB passed from the user program. Because this is the only object for which the debugger evaluates the first constraint, the intermediate result is likely to be empty. In general, the size of intermediate results is much smaller in the incremental evaluation, speeding up the query evaluation. If intermediate results are not empty, the debugger continues the evaluation in the usual manner and produces an incremental result (A * ΔB * C). The system then merges the result with the previous result to form the complete query result.

[1] This test is just one example of tests that quickly verify whether the query result changed due to the assignment. We are currently investigating more sophisticated tests that detect more query-invariant assignments.

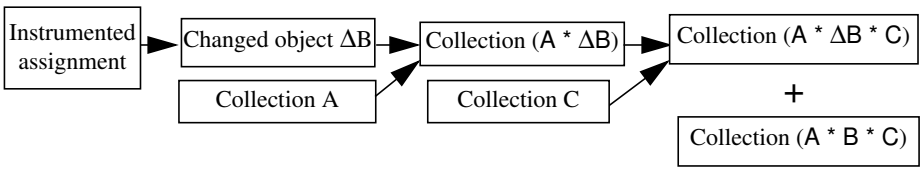

Figure 6. Incremental query evaluation

The query evaluation is further optimized by finding efficient join orders and by using hash joins as described in [LHS97]. Because sizes of domains change during program runtime and we cannot efficiently determine the selectivities of constraints, we use simple heuristics for join ordering: execute selections first, equality joins next, and inequality constraints last.

3.4.2 Custom Code Generation for Selection Queries

Constraints of selection queries are usually very simple and can be evaluated very fast. Instead of performing the general query execution algorithm described in section 3.4.1, which goes through numerous general steps and calls a number of methods, the debugger can evaluate just the few tests necessary to check the selection constraints. Because these tests depend on the query asked, the code for their evaluation has to be generated at program load time. During the loading of the user program, the debugger generates a Java class with a debug method. We show such a method in Figure 7 for the query

```
Molecule1 m.    m.x > 350
```

The first three statements of the method contain the code common for both unoptimized and optimized versions. This code performs the domain test and the same value assignment test described in section 3.4. The optimized code that follows evaluates the selection constraint on the changed object and calls the debugger runtime only if the query has a

```
public final class DebuggingCode implements RunTimeCode {
    public static void debug (Molecule updatedObject, int newValue) {
        // Code common for both general and optimized versions
        if (! (updatedObject instanceof Molecule1))
            { updatedObject.x = newValue; return; }
        if (updatedObject.x == newValue)  return;
        updatedObject.x = newValue;
        // Instead of calling general query evaluation method,
        // evaluate constraint here
        if (updatedObject.x > 350) QueryTool.outputResult(updatedObject);
    }
```

Figure 7. Selection evaluation using custom code

non-empty result. The debugger uses the debug method as an entry point that the user program calls when it reaches instrumentation points. With custom code generated, the debug method contains all code needed to evaluate a selection, so the reevaluation costs only one static method call. Furthermore, the debug method—a member of a final class—may even be inlined into the instrumentation points by a JIT compiler. We could also inline the bytecodes into the instrumented method.

4. Experimental Results

Ideally, a test of the efficiency of a dynamic query-based debugger would use real debugging queries asked by programmers using the tool for their daily work. Though we tried to predict what queries programmers will use, each debugging situation is unique and requires different queries. To perform a realistic test of the query-based debugger without writing hundreds of possible queries, we selected a number of queries that in complexity and overhead cover the range of queries asked in debugging situations. The selected queries contain selection queries with low and high cost constraints. The test also includes hash-join and nested-join queries with different domain sizes. The queries

Query	Slowdown	Invocation frequency (events / s)
Molecule1 z. z.x > 350	1.02	15,000
Id x. x.type < 0	1.11	16,000
spec.benchmarks._202_jess.jess.Token z. z.sortcode == -1	1.25	169,000
spec.benchmarks._201_compress.Output_Buffer z. z.OutCnt < 0	1.18	1,900,000
spec.benchmarks._201_compress.Output_Buffer z. z.count() < 0	1.27	
spec.benchmarks._201_compress.Output_Buffer z. z.lessOutCnt(0)	1.37	
spec.benchmarks._201_compress.Output_Buffer z. z.complexMathOutCnt(0)	5.83	
spec.benchmarks._201_compress.Compressor z. z.in_count < 0	1.18	933,000
spec.benchmarks._201_compress.Compressor z. z.out_count < 0	1.10	196,000
spec.benchmarks._201_compress.Compressor z. z.complexMathOutCount(0)	1.83	
spec.benchmarks._205_raytrace.Point p. p.x == 1	1.23	787,000
spec.benchmarks._205_raytrace.Point p. p.farther(100000000)	1.98	2,300,000
Molecule1 z; Molecule2 z1. z.x == z1.x && z.y == z1.y && z.dir == z1.dir && z.radius == z1.radius (33x33 hash join)	2.13	54,000
Lexer l; Token t. l.token == t && t.type == 27 (120,000x600 hash join)	3.43	25,000
spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace.IntersectPt ip. p.z == ip.t && p.z < 0 (85,000x8,000 hash join)	229	350,000
spec.benchmarks._201_compress.Input_Buffer z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt == z.InCnt && z1.OutCnt < 100 && z.InCnt > 0 (1x1 hash join)	157	1,500,000
spec.benchmarks._201_compress.Compressor z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt < 100 && z.out_count > 1 && z1.OutCnt / 10 > z.out_count (1x1 join)	77	2,600,000
Test5 z. z.x < 0	6.4	42,000,000
TestHash5 th; TestHash1 th1. th.i == th1.i (1x20 hash join)	228	40,000,000
TestHash5 th; TestHash1 th1. th.i < th1.i (1x20 join)	930	

Table 1. Benchmark queries

check programs that range from small applets to large applications and (for stress-tests) microbenchmarks. These applications invoke the debugger with frequencies ranging from low to very high, where a query has to be evaluated at every iteration of a tight loop. Consequently, the experimental results obtained for the test set should indicate the range of performance to be expected in real debugging situations.

For our tests we used an otherwise idle Sun Ultra 2/2300 machine (with two 300 MHz UltraSPARC II processors) running Solaris 2.6 and Solaris Java 1.2 with JIT compiler (Solaris VM (build Solaris_JDK_1.2_01, native threads, sunwjit)) [Sun99]. Execution times are elapsed times and were measured with millisecond accuracy using the System.currentTimeMillis() method.

4.1 Benchmark Queries

To test the dynamic query-based debugger, we selected a number of structurally different queries (Table 1) for a number of different programs (Table 2):

- Queries 1 and 13 check a small ideal gas tank simulation applet that spends most of the time calculating molecule positions and assigns object fields very infrequently. It has 100 molecules divided among Molecule1, Molecule2 and Molecule3 classes. The application performs 8,000 simulation steps.
- Queries 2 and 14 check the Decaf Java subset compiler, a medium size program developed for a compiler course at UCSB. The Token domain contains up to 120,000 objects.
- Query 3 checks the Jess expert system, program from the SPECjvm98 suite [SPEC98].

- Queries 4–10, and 16–17 check the compress program from the SPECjvm98 suite. Our queries reference frequently updated fields of compress.
- Queries 11–12 and 15 check the ray tracing program from the SPECjvm98 suite. The Point domain contains up to 85,000 objects; the IntersectPt domain has up to 8,000 objects.
- Queries 18–20 check artificial microbenchmarks. These microbenchmarks stress test debugger performance by executing tight loops that continuously update object fields.

Application	Size (Kbytes)	Execution time (s)
Compress	17.4	50
Jess	387.2	22
Ray tracer	55.7	17
Decaf	55	15
Ideal gas tank	14.3	57

Table 2. Application sizes and execution times

Structurally, queries can be divided into the following classes:

- Queries 1–12 and 18 are simple one-constraint selection queries with a wide range of constraint complexities. For example, query 4 has a very simple low-cost constraint that compares an object field to an integer. The more costly constraint in query 5 invokes a method to retrieve an object field. Another costly alternative constraint (query 6) invokes a comparison method that takes a value as a parameter. Finally, the most costly constraint in query 7 performs expensive mathematical operations before performing a comparison. Queries 8 and 9 have very similar constraints, but differ 4.8 times in debugger invocation frequency. In this paper, by "debugger invocation frequency" we mean the frequency of events in the original program that would trigger a debugger invocation, i.e., the invocation frequency for a debugger with no overhead. Query 12 compares the parameter of the method to the distance of a point to the origin. This query combines costly mathematical operations with increased debugger invocation frequency, because its result depends on all three coordinates of Point objects.
- Queries 13–17 and 19–20 are join queries. Queries 13–16 and 19 can be evaluated using hash joins. The evaluation of queries 17 and 20 has to use nested-loop joins. For join queries, the slowdown depends both on the debugger invocation frequency and sizes of the domains. Queries 13–14 have low invocation frequencies; queries 15–17, 19–20 have high invocation frequencies. Queries 14 and 15 have large domains.

In the next section, we discuss the performance of these queries. Section 4.3 then discusses the efficiency benefits of incremental evaluation, custom selection code, and unnecessary assignment detection.

4.2 Execution Time

Figure 8 shows the program execution slowdown for application programs when queries are enabled. The slowdown is the ratio of the running time with the query active to the running time without any queries. For example, the slowdown of query 3 indicates that the Jess expert system ran 25% slower when the query was enabled.

Overall the results are encouraging. All selection queries except query 7 have overheads of less than a factor of 2. The median slowdown is 1.24. We expect overheads of common practical selection queries to be in the same range as our experimental queries; the performance model discussed in section 5 supports this prediction.

Join queries have overheads ranging from 2.13 to 229 for applications. Hash queries (which can be used for equality joins) are efficient for queries 13–14, and other joins are practical for query 13 in which the domains contain only 33 objects each. Queries 15–17 have large overheads because of frequent invocations (e.g., 2.6 million times per second for query 16) and large domains. Join query performance is acceptable if join domains are small, and the program invokes the debugger infrequently. For large domains and frequently invoked queries, the overhead is significant.

Microbenchmark stress-test queries 18–20 show the limits of the dynamic query-based debugger. The benchmark updates a single field in a loop 40 million times per second. When queries depend on this field, the program

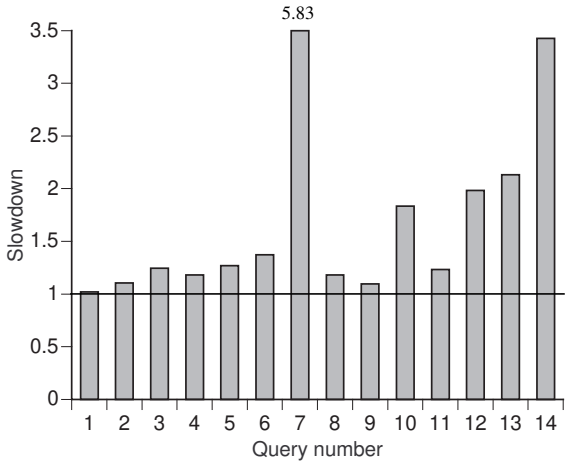

Figure 8. Program slowdown (queries 15—20 not shown)

The slowdown is the ratio of the running time with the query active to the running time without any queries. For example, the slowdown of query 3 indicates that the Jess expert system ran 25% slower when the query was enabled.

slowdown is significant. Selection query 18 has a slowdown factor of 6.4, the hash-join evaluation has a slowdown of 228 times, and the slower nested-loop join that checks twenty object combinations in each evaluation has a slowdown of 930 times.

Though the microbenchmark results indicate that in the worst case the debugger can incur a large slowdown, these programs represent a hypothetical case. Such frequent field updates are possible only with a single assignment in a loop. Adding a few additional operations inside the loop drops the field update frequency to 3 million times per second which is more in line with the highest update frequencies in real programs. For such update frequencies, the slowdown is much lower as indicated by query 4. We discuss the likelihood of high update frequencies in section 5.

Figure 9 shows the components of the overhead:

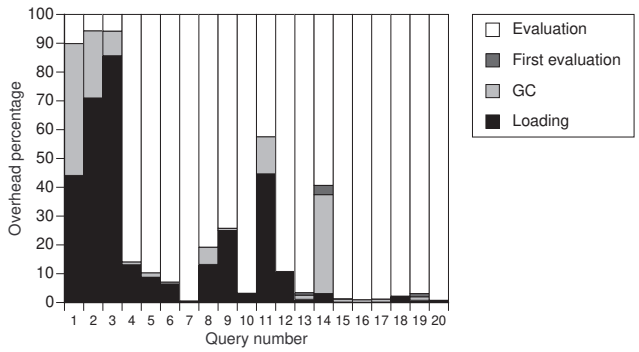

Figure 9. Breakdown of query overhead as a percentage of total overhead

For example, 3% of query 14 overhead is spent on instrumentation, 34% on garbage collection, 3% in the first evaluation, and 60% in subsequent reevaluations.

- *Loading time*, the difference between the time it takes to load and instrument classes using a custom class loader, and the time it takes to load a program during normal execution.
- *Garbage collection time*, the difference between the time spent for garbage collection in the queried program and the GC time in the original program.
- *First evaluation time,* the time it takes to evaluate the query for the first time. For join queries, the first query is the most expensive, because it sets up data structures needed for future query reevaluations. We separate this time from the rest of the query evaluation time, because it is a fixed overhead incurred only once.

Query	Slowdown versus non-instrumented	Slowdown versus optimized
Molecule1 z. z.x > 350	1.19	1.16
Id x. x.type < 0	613	554
spec.benchmarks._202_jess.jess.Token z. z.sortcode == -1	7135	5,725
spec.benchmarks._201_compress.Output_Buffer z. z.OutCnt < 0	475	402
spec.benchmarks._201_compress.Output_Buffer z. z.count() < 0	474	373
spec.benchmarks._201_compress.Output_Buffer z. z.lessOutCnt(0)	587	428
spec.benchmarks._201_compress.Output_Buffer z. z.complexMathOutCnt(0)	513	88
spec.benchmarks._201_compress.Compressor z. z.in_count < 0	275	233
spec.benchmarks._201_compress.Compressor z. z.out_count < 0	37	33.8
spec.benchmarks._201_compress.Compressor z. z.complexMathOutCount(0)	40	21.8
spec.benchmarks._205_raytrace.Point p. p.x == 1	10,500	8,496
spec.benchmarks._205_raytrace.Point p. p.farther(100000000)	17,800	8,972
Molecule1 z; Molecule2 z1. z.x == z1.x && z.y == z1.y && z.dir == z1.dir && z.radius == z1.radius (33x33 hash join)	21.96	10.3
Lexer l; Token t. l.token == t && t.type == 27 (120,000x600 hash join)	1,973	576
spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace.IntersectPt ip. p.z == ip.t && p.z < 0 (85,000x8,000 hash join)	12,400	54
spec.benchmarks._201_compress.Input_Buffer z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt == z.InCnt && z1.OutCnt < 100 && z.InCnt > 0 (1x1 hash join)	1,708	11
spec.benchmarks._201_compress.Compressor z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt < 100 && z.out_count > 1 && z1.OutCnt / 10 > z.out_count (1x1 join)	697	9
Test5 z. z.x < 0	5,213	821
TestHash5 th; TestHash1 th1. th.i == th1.i (1x20 hash join)	1,491	6.6
TestHash5 th; TestHash1 th1. th.i < th1.i (1x20 join)	5,602	6.02

Table 3. Overhead of non-incremental evaluation

- *Evaluation time*, the time spent evaluating the query. This component does not include the first evaluation time. The first evaluation time and the evaluation time together compose the *total evaluation time*.

Figure 9 shows the components of the overhead. For example, 3% of the overhead of query 14 is spent on instrumentation, and 34% on garbage collection. The total evaluation time is 63% of the overhead, with 3% spent in the first evaluation, and 60% spent in subsequent reevaluations. On average, the largest part of the overhead is the evaluation time (75.5%), while loading takes only 17% and garbage collection has a negligible overhead (less than 7%) in most cases[1]. The loading overhead becomes a significant factor when the loaded class hierarchy is large, as in query 3 on the Jess system. The loading overhead also takes a larger proportion of time when query reevaluations are infrequent or fast as in queries 1, 2, 9, and 11. Garbage collection was not a significant factor except in query 14 which creates 120,000 token objects, and in query 1 which has such a small absolute overhead that even a slight increase in GC and loading time becomes a large part of the overhead. Since the evaluation component dominates the overhead, especially in high-overhead, long-running queries, evaluation optimizations are very important for good performance. We discuss some optimizations already reflected in this graph in the next section.

4.3 Optimizations

To evaluate the benefit of optimizations implemented in the dynamic query-based debugger, we performed a number of experiments by turning off selected optimizations.

[1] Experiments were run with 128M heap, a factor that decreased the GC overhead.

4.3.1 Incremental Reevaluation

The dynamic query debugger benefits considerably from the incremental evaluation of queries. We disabled incremental query evaluation and reran all queries. Table 3 shows the results of this experiment. The first column of numbers in the table shows the ratio of non-incremental query running time to the running time of the original program. The second column shows the ratio of non-incremental query running time to the running time of fully optimized incremental query evaluation. For example, query 2 had a factor of 613 overhead and ran for 2.5 hours. In contrast, the same query ran 554 times faster using the incremental reevaluation, had only 11% overhead and finished in 16.4 seconds. Query 1 was the only query that the non-incremental debugger could evaluate in a reasonable time. The overheads of all other queries were enormous; some programs would have run for more than a day. (For queries 3–12 and 14–17, we stopped query reevaluation after the first 100,000 evaluations and estimated the total overhead.) Despite the large overall overhead, the individual non-incremental query evaluations are reasonably fast. For example, even for large join queries 14 and 15, a single query evaluation only took about 50 ms.

The join queries on compress have an overhead of only 9–11 compared to the incremental optimized version. These joins did not benefit much from incremental evaluation and its optimizations because the domains of these joins contain only a single object.

Overall, the experiments with non-incremental evaluation of queries show that incremental evaluation is imperative, greatly reducing the overhead and making a much larger class of dynamic queries practical for debugging.

Query	Slowdown versus non-instrumented	Slowdown versus optimized
Molecule1 z. z.x > 350	1.05	1.03
Id x. x.type < 0	1.46	1.34
spec.benchmarks._202_jess.jess.Token z. z.sortcode == -1	11.70	9.26
spec.benchmarks._201_compress.Output_Buffer z. z.OutCnt < 0	68.5	58
spec.benchmarks._201_compress.Output_Buffer z. z.count() < 0	64	51
spec.benchmarks._201_compress.Output_Buffer z. z.lessOutCnt(0)	65	47
spec.benchmarks._201_compress.Output_Buffer z. z.complexMathOutCnt(0)	69.6	12
spec.benchmarks._201_compress.Compressor z. z.in_count < 0	43.6	37
spec.benchmarks._201_compress.Compressor z. z.out_count < 0	10.5	9.6
spec.benchmarks._201_compress.Compressor z. z.complexMathOutCount(0)	11	6
spec.benchmarks._205_raytrace.Point p. p.x == 1	21	15
spec.benchmarks._205_raytrace.Point p. p.farther(100000000)	61	31
Test5 z. z.x < 0	1,952	307

Table 4. Benefit of custom selection code (selection queries only)

4.3.2 Custom Generated Selection Code

To estimate the benefit of generating custom code as discussed in section 3.4.2, we ran all selection queries with the optimization disabled. The results of the experiment are shown in Table 4. The first column of numbers shows the slowdown of the unoptimized version compared to the original program. The second column indicates the slowdown of the unoptimized version compared to the optimized version. For example, query 4 ran 68.5 times slower than the original program and 58 times slower than the optimized query.

The ideal gas tank applet and Decaf compiler queries did not benefit from this optimization, because these programs reevaluate the query infrequently, and the optimization benefit is masked by variations in start-up overhead. All other queries show significant speedups with the optimization enabled. The benefit of the optimization increases with the frequency of debugger invocations; overall, custom generated selection code produces a median speedup of 15.

4.3.3 Same Value Assignment Test

Before evaluating a query after a field assignment, the debugger checks whether the value being assigned to the object field is equal to the value previously held by the field. Such assignments do not change the result of the query and can be ignored by the debugger.

Table 5 shows that the number of unnecessary assignments differs highly depending on the programs or fields. While some programs and fields do not have them at all, others have from 7% to 95% of such assignments. Only the ideal gas tank simulation, the Jess expert system, and the ray tracing application have unnecessary assignments to the queried fields.

Query	Slowdown versus optimized	% unnecessary assignments
Molecule1 z. z.x > 350	0.99	95%
spec.benchmarks._202_jess.jess.Token z. z.sortcode == -1	0.997	7%
spec.benchmarks._205_raytrace.Point p. p.x == 1	0.988	15%
spec.benchmarks._205_raytrace.Point p. p.farther(100000000)	1.16	40%
Molecule1 z; Molecule2 z1. z.x == z1.x && z.y == z1.y && z.dir == z1.dir && z.radius == z1.radius (33x33 hash join)	1.61	54%
spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace.IntersectPt ip. p.z == ip.t && p.z < 0 (85,000x8,000 hash join)	1.02	15%

Table 5. Unnecessary assignment test optimization
(excluding queries with no unnecessary assignments)

To check the efficiency of the same-value test, we disabled it while leaving all other optimizations enabled. The results show that the test does not make much of a difference in query evaluation for most queries. For selections that can be evaluated fast, the cost of the same-value test is similar to the cost of the full selection evaluation. Only when the selection constraint is costly (as in query 4), does the same-value test reduce the overhead. For joins, the cost reduction is significant for the ideal gas tank query that contains 54% unnecessary assignments. For other joins, the percentage of unnecessary assignments is too low to make a difference.

To summarize, the test whether an assignment changes a value of a field costs only one extra comparison per debugger invocation. It does not change the overhead for most programs, but saves time when the number of unnecessary assignments is large or the query expression is expensive.

5. Performance Model

To better predict debugger performance for a wide class of queries, we constructed a query performance model. The slowdown depends on the frequency of debugger invocations and on the individual query reevaluation time. This relationship can be expressed as follows:

$$T = T_{original} (1 + T_{nochange} * F_{nochange} + T_{evaluate} * F_{evaluate})$$

This formula relates the total execution time of the program being debugged T and the execution time of the original program $T_{original}$ using frequencies of field assignments in the program and individual reevaluation times. The model divides field assignments into two classes:

- Assignments that do not change the value of a field. These assignments do not change the result of the query. The debugger has to perform only two comparisons in this case—a domain test and the value equality test, so it spends a fixed amount of time ($T_{nochange}$) in such invocations independent of the query. We calculated $T_{nochange}$ by running a query on a program that repeatedly assigned the same value to the queried field; for the machine/JVM combination we used, $T_{nochange}$ = 66 ns.
- Assignments that lead to the reevaluation of a query. The time to reevaluate a query $T_{evaluate}$ for such an assignment depends on the query structure and on the cost of the query constraint expression. For each query, we calculate

Query	$F_{evaluate}$ (assignments per second)	$T_{evaluate}$ (µs)
Molecule1 z. z.x > 350	N/A	N/A
Id x. x.type < 0	16,000	3.73
spec.benchmarks._202_jess.jess.Token z. z.sortcode == -1	169,000	3
spec.benchmarks._201_compress.Output_Buffer z. z.OutCnt < 0	1,900,000	0.140
spec.benchmarks._201_compress.Output_Buffer z. z.count() < 0		0.208
spec.benchmarks._201_compress.Output_Buffer z. z.lessOutCnt(0)		0.286
spec.benchmarks._201_compress.Output_Buffer z. z.complexMathOutCnt(0)		3.7
spec.benchmarks._201_compress.Compressor z. z.in_count < 0	933,000	0.193
spec.benchmarks._201_compress.Compressor z. z.out_count < 0	196,000	0.488
spec.benchmarks._201_compress.Compressor z. z.complexMathOutCount(0)		4.26
spec.benchmarks._205_raytrace.Point p. p.x == 1	787,000	0.486
spec.benchmarks._205_raytrace.Point p. p.farther(100000000)	2,300,000	0.461
Molecule1 z; Molecule2 z1. z.x == z1.x && z.y == z1.y && z.dir == z1.dir && z.radius == z1.radius (33x33 hash join)	N/A	N/A
Lexer l; Token t. l.token == t && t.type == 27 (120,000x600 hash join)	25,000	56.8
spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace.IntersectPt ip. p.z == ip.t && p.z < 0 (85,000x8,000 hash join)	350,000	546
spec.benchmarks._201_compress.Input_Buffer z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt == z.InCnt && z1.OutCnt < 100 && z.InCnt > 0 (1x1 hash join)	1,500,000	60
spec.benchmarks._201_compress.Compressor z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt < 100 && z.out_count > 1 && z1.OutCnt / 10 > z.out_count (1x1 join)	2,600,000	51
Test5 z. z.x < 0	42,000,000	0.131
TestHash5 th; TestHash1 th1. th.i == th1.i (1x20 hash join)	40,000,000	5.7
TestHash5 th; TestHash1 th1. th.i < th1.i (1x20 join)		23

Table 6. Frequencies and individual evaluation times

$T_{evaluate}$ by dividing the additional time it takes to run a program with a query into the number of debugger invocations. This calculation gives an exact result for programs that have no unnecessary assignments ($F_{nochange} = 0$). For example, for query 18 $T_{evaluate}$ is 131ns. $T_{evaluate}$ for query 4 is 140 ns, which is close to the time to evaluate a similar query in a microbenchmark. When constraints are more costly, $T_{evaluate}$ increases; for example, for the highest cost selection query (query 10) it is 4.26 µs. It is even higher for join queries where it depends on the size of domains in joins; for example, for query 16 it is 60 µs, and for query 15 which has large domains, it is 546 µs.

Using the values of reevaluation times and the frequency of assignments to the fields of the change set, we can estimate the debugging overhead. First, we determine the typical field assignment frequency.

5.1 Debugger Invocation Frequency

Debugger invocation frequency is an important factor in the slowdown of programs during debugging. The program invokes the debugger after object creation and after field assignments. For most queries, the field assignment component dominates the debugger invocation frequency. To find the range of field assignment frequencies in programs, we examined the microbenchmarks and the SPECjvm98 application suite. We instrumented the applications to record every assignment to a field. Table 7 shows results of these measurements.

Application	Maximum frequency (field assignments per second)	Original program execution time (s)
Compress	1,900,000	50.4
Jess	169,000	22.45
Db	254	75
Javac	217,000	38
Mpegaudio	495,000	57.4
Jack	27,000	27
Ray tracer	787,000	17
Decaf	56,000	15
Ideal gas tank	23,150	57
Microbenchmark	40,000,000	2.4

Table 7. Maximum field assignment frequencies

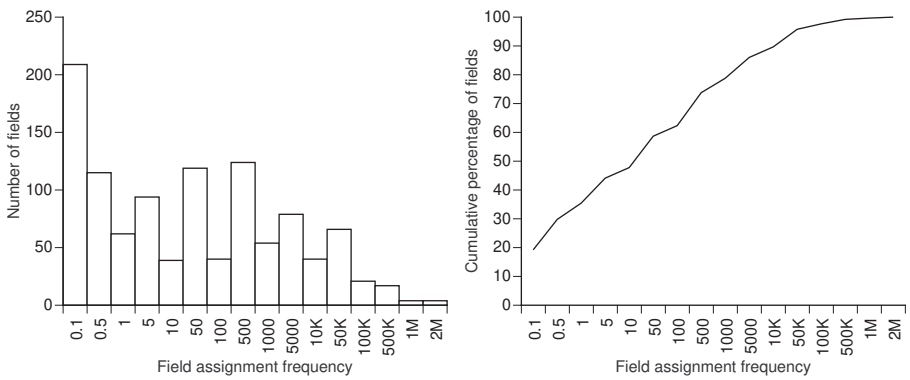

Figure 10. Field assignment frequency in SPECjvm98

The maximum field assignment frequency in microbenchmarks is 40 million assignments per second, but that would be difficult to reach in an application because the microbenchmarks contain a single assignment inside a loop. The compress program has the highest field assignment frequency in the SPECjvm98 application suite, 1.9 million assignments per second. Other SPEC applications, as well as the Decaf compiler and the ideal gas tank applet, have much lower maximum field assignment frequencies.

Figure 10 shows the frequency distribution of field assignments in the SPECjvm98 applications. The left graph indicates how many fields have an assignment frequency in the range indicated on the x axis. For example, only four fields are assigned between one million and two million times per second. The right graph shows the cumulative percentage of fields that have assignment frequencies lower than indicated on the x axis; 95% of all fields have fewer than 100,000 assignments per second.

To predict the overhead of a typical selection query, we can now calculate the overhead as a function of invocation frequency. Figure 11 uses the minimum (130 ns) and maximum (4.26 µs) values of $T_{evaluate}$ from Table 6 to plot the estimated selection query overhead for a range of invocation frequencies. For example, a selection query on a field updated 500,000 times per second would have an overhead of 6.5% if its reevaluation time was 130 ns. If the reevaluation time was 4.26 µs, the overhead will be a factor of 3.13. The graph reveals that selection queries on fields assigned less than 100,000 times a second—95% of fields—have a predicted overhead of less than 43% even for the most costly selection constraint. For less costly selections, the query overhead is acceptable for all fields.

In the current model, the evaluation time $T_{evaluate}$ models all sources of query overhead. This time includes the actual reevaluation time as well as the additional garbage collection time, the class instrumentation cost, and the first evaluation cost. It would be more exact to model each of these overheads separately. However, for long running programs the evaluation time dominates the total cost, so the values of $T_{evaluate}$ are likely to fall in the range we have covered.

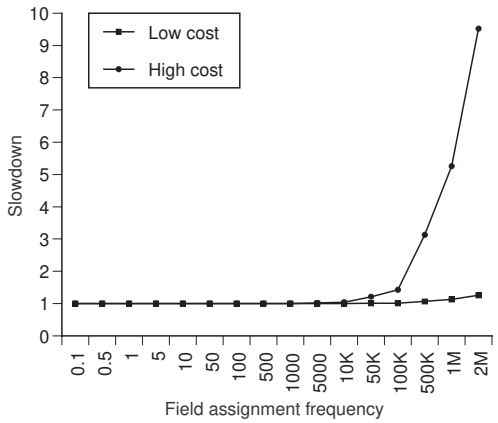

Figure 11. Predicted slowdown

The graph shows the predicted overhead as a function of update frequency. For example, the predicted overhead of a low-cost selection query on a field updated 500,000 times per second is 6.5%; the predicted overhead of a high-cost query with the same frequency is a factor of 3.13.

In summary, the performance model predicts that most selection queries will have less than 43% overhead. The model can be used as a framework for concrete overhead predictions and future model refinements.

6. Queries with Changing Results

So far we discussed using dynamic queries for debugging, where the program stops as soon as the query returns a non-empty result. However, programmers can also use queries to monitor program behavior. For example, in the ideal gas tank simulation, users may want to monitor all molecule near-collisions with a query:

```
Molecule* m1 m2.    m1.closeTo(m2) && m1 != m2
```

Programmers may use this information to check the frequency of near-collisions, to find out if near-collisions are handled in a special way by the program, or to check the correspondence of program objects with the visual display of the simulation. In this case, the debugger should not stop after the result becomes non-empty, but instead should continue executing the program and updating the query result as it changes. Such monitoring, perhaps coupled with visualization of the changing result, can help users understand abstract object relationships in large programs written by other people. How can a debugger support continuous updating of query results while the program executes?

Query	Slowdown
Molecule1 z. z.x < 200	1.05
Id x. x.type == 0	1.23
spec.benchmarks._202_jess.jess.Token z. z.sortcode == 0	1.3
spec.benchmarks._201_compress.Compressor z. z.OutCnt == 0	1.19
spec.benchmarks._201_compress.Compressor z. z.out_count == 0	1.09
Molecule1 z; Molecule2 z1. z.x < z1.x && z.y < z1.y (33x33 join)	1.47
Lexer l; Token t. l.token == t && t.type == 0 (120,000x600 hash join)	4.09
spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace.IntersectPt ip. (p.z == ip.t) && (p.z > 100) (85,000x8,000 hash join)	212.4
spec.benchmarks._201_compress.Compressor z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt == z.out_count (1x1 hash join)	9.07
spec.benchmarks._201_compress.Input_Buffer z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt < z.InCnt (1x1 join)	127
Test5 z. z.x % 2 == 0	45

Table 8. Benchmark queries with non-empty results

The dynamic query-based debugger described above needs only a few changes to support monitoring queries. The basic scheme and the implementation of the dynamic query-based debugger discussed in section 3 remain the same. The only new component of the debugger is a module that maintains the current query result. As discussed in section 3.4.1, the debugger reevaluates only the changed part of the query. Consequently, the result handling module must store the query result from the previous evaluation and then merge it with the new partial result. To achieve that, after query execution the debugger deletes all tuples from the previous result that contain the changed domain object and inserts the new tuples generated by the incremental reevaluation.

Experiments with queries similar to the ones in Table 1 show that adding the query result update functionality does not significantly change the query evaluation overhead (Table 8). The only exception is the microbenchmark selection query 11 which updates the query result during each reevaluation. Consequently, the overhead of the selection increases from 6.4 times to 45 times, although part of this increase can be attributed to the more costly selection constraint. However, such frequent result updates are unlikely for most monitoring queries: programmers can only absorb infrequent result changes, so, if results change rapidly, the display will be unintelligible unless it is artificially slowed down or used off-line.

To summarize, monitoring queries are useful for understanding and visualizing program behavior. With slight modifications our debugger supports monitoring queries. Unless the result changes very rapidly, the additional overhead of monitoring query execution is insignificant when compared to similar debugging queries.

7. Related Work

We are unaware of other work that directly corresponds to dynamic query-based debugging. The query-based debugging model and its non-dynamic implementation are presented in a previous paper [LHS97].

Extensions of object-oriented languages with rules as in R++ [LMP97] provide a framework that allows users to execute code when a given condition is true. However, R++ rules can only reference objects reachable from the root object, so R++ would not help to find the javac error we discussed. Due to restrictions on objects in the rule, R++ also does not handle join queries.

Sefika et al. [SSC96] implemented a system allowing limited, unoptimized selection queries about high-level objects in the Choices operating system. The system dynamically shows program state and run-time statistics at various levels of abstraction. Unlike our dynamic query-based debugger, the tool uses instrumentation specific to the application (Choices).

While no one has investigated the query-based debugging specifically, various researchers have proposed a variety of enhancements to conventional debugging [And95, Cop94, DHKV93, GH93, GWM89, KRR94, Laf97, LM94, LN97, WG94]. The debuggers most closely related to dynamic query-based debugging visualize object relationships—usually references or an object call graph. Duel [GH93] displays data structures by using user script code. HotWire [LM94] allows users to specify custom object visualizations in constraint language. Look! [And95], Object Visualizer [DHKV93], PV [KRR94], and Program Explorer [LN97] provide numerous graphical and statistical run-time views with class-dependent filtering but do not allow general queries. Our debugger can gather statistical data through queries with non-empty results ("How many lists of size greater than 500 exist in the program?") but does not display animated statistical views.

Visualizing debuggers gather information by either instrumenting the source code [DHKV93, LM94] or by using program traces [KRR94, LN97]. A port of our debugger to C++ would have to use one of these techniques. Laffra [Laf97] discusses visual debugging in Java using source code instrumentation or JVM changes. We opted for the third method—class file instrumentation at load time. Consens et al. [CHM94, CMR92] use the Hy^+ visualization system to find errors using post-mortem event traces. De Pauw et al. [DLVW98] and Walker et al. [WM+98] use program event traces to visualize program execution patterns and event-based object relationships, such as method invocations and object creation. This work is complementary to ours because it focuses on querying and visualizing run-time events while we query object relationships.

Dynamic query-based debugging extends work on data breakpoints [WLG93]—breakpoints that stop a program whenever an object field is assigned a certain value. Pre-/postconditions and class invariants as provided in Eiffel [Mey88] can be thought of as language-supported dynamic queries that are checked at the beginning or end of methods. Unlike dynamic queries, they are not continuously checked, they cannot access objects unreachable by references from the checked class, nor can they invoke arbitrary methods. Dynamic queries could be used to implement class assertions for languages that do not provide them. The current implementation of dynamic queries cannot use the "old" value of a variable, as can be done in postconditions. We view the two mechanisms as complementary, with queries being more suitable for program exploration as well as specific debugging problems.

Software visualization systems such as BALSA [Bro88], Zeus [Bro91], TANGO/XTANGO/POLKA [Sta90], Pavane [Rom92], and others [HKWJ95, Mos97, RC93] offer high-level views of algorithms and associated data structures. Software visualization systems aim to explain or illustrate the algorithm, so their view creation process emphasizes vivid representation. Hart et al. [HKR97] use Pavane for query-based visualization of distributed programs. However, their system only displays selected attributes of different processes and does not allow more complicated queries.

Dynamic queries are related to incremental join result recalculation in databases [BC79, BLT86]. We use the basic insights of this work to implement the incremental query evaluation scheme. Coping with inter-object constraints in the extended ODMG model [BG98] may require methods similar to dynamic query-based debugging.

Slicing [Wei81, Tip95] determines the program statements that affect a certain program point. It could be modified to determine the change sets of queries.

8. Conclusions

The cause-effect gap between the time when a program error occurs and the time when it becomes apparent to the programmer makes many program errors hard to find. The situation is further complicated by the increasing use of large class libraries and complicated pointer-linked data structures in modern object-oriented systems. A misdirected reference that violates an abstract relationship between objects may remain undiscovered until much later in the program's execution. Conventional debugging methods offer only limited help in finding such errors. Data breakpoints and conditional breakpoints cannot check constraints that use objects unreachable from the statement containing the breakpoint.

We have described a dynamic query-based debugger that allows programmers to ask queries about the program state and updates query results whenever the program changes an object relevant to the query, helping programmers to discover object relationship failures as soon as they happen. This system combines the following novel features:

- An extension of query-based debugging to include dynamic queries. Not only does the debugger check object relationships, but it determines exactly when these relationships fail while the program is running. This technique closes the cause-effect gap between the error's occurrence and its discovery.
- Implementation of monitoring queries. The debugger helps users to watch the changes in object configurations through the program's lifetime. This functionality can be used to better understand program behavior.

The implementation of the query based debugger has good performance. Selection queries are efficient with less than a factor of two slowdown for most queries measured. We also measured field assignment frequencies in the SPECjvm98 suite, and showed that 95% of all fields in these applications are assigned less than 100,000 times per second. Using these numbers and individual evaluation time estimates, our debugger performance model predicts that selection queries will have less than 43% overhead for 95% of all fields in the SPECjvm98 applications. Join queries are practical when domain sizes are small and queried field changes are infrequent.

Good performance is achieved through a combination of two optimizations:

- Incremental query evaluation decreases query evaluation overhead by a median factor of 160, greatly expanding the class of dynamic queries that are practical for everyday debugging.
- Custom code generation for selection queries produces a median speedup of 15, further improving efficiency for commonly occurring selection queries.

We believe that dynamic query-based debugging adds another powerful tool to the programmer's tool chest for tackling the complex task of debugging. Our implementation of the dynamic query-based debugger demonstrates that dynamic queries can be expressed simply and evaluated efficiently. We hope that future mainstream debuggers will integrate a similar functionality, simplifying the difficult task of debugging and facilitating the development of more robust object-oriented systems.

9. Acknowledgments

We thank the anonymous reviewers, Amer Diwan, Karel Driesen, Sylvie Dieckmann, Andrew Duncan, and Jeff Bogda for valuable comments on earlier versions of this paper. This work was funded in part by Sun Microsystems, the State of California MICRO program, and by the National Science Foundation under CAREER grant CCR96–24458 and grants CCR92–21657 and CCR95–05807.

10. References

[And95] Anderson E., Dynamic Visualization of Object Programs Written in C++, *Objective Software Technology Ltd.,* http://www.objectivesoft.com/, 1995.

[BC79] Buneman, O.P.; Clemons E.K., Efficiently Monitoring Relational Databases. *ACM Transactions on Database Systems,* 4(3), pp. 368-382, September 1979.

[BG98] Bertino, E., Guerrini, G., Extending the ODMG Object Model with Composite Objects, *Proceedings of OOPSLA'98*, pp. 259-270, Vancouver, October 1998. Published as *SIGPLAN Notices* 33(10), October 1998.

[BLT86] Blakeley, J.A.; Larson P.-A.; Tompa F. Wm.; Efficiently Updating Materialized Views. *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp. 61-71, Washington, D.C., USA, May 1986. Published as *SIGMOD Record* 15(2), June 1986.

[Bro88] Brown, M.H., Exploring Algorithms Using Balsa-II, *IEEE Computer* 21(5), pp. 14-36, May 1988.

[Bro91] Brown, M.H., Zeus: A System for Algorithm Animation and Multi-View Editing, *Proceedings of IEEE Workshop Visual Languages*, pp. 4-9, IEEE CS Press, Los Alamitos, CA., 1991.

[CHM94] Consens, M. P., Hasan M.Z., Mendelzon A.O., Debugging Distributed Programs by Visualizing and Querying Event Traces, *Applications of Databases, First International Conference*, ADB-94, Vadstena, Sweden, June 21-23, 1994, Proceedings in Lecture Notes in Computer Science, Vol. 819, Springer, 1994.

[CMR92] Consens, M.; Mendelzon, A.; Ryman, A., Visualizing and Querying Software Structures, *International Conference on Software Engineering*, Melbourne, Australia, May 11-15, 1992, ACM Press, IEEE Computer Science, p. 138-156, 1992.

[Cop94] Coplien, J.O., Supporting Truly Object-Oriented Debugging of C++ Programs., In: Proceedings of the 1994 USENIX C++ Conference, Cambridge, MA, USA, 11-14 April 1994. pp. 99-108, Berkley, CA, USA: USENIX Assoc, 1994.

[DHKV93] De Pauw, W.; Helm, R.; Kimelman, D.; Vlissides, J. Visualizing the Behavior of Object-Oriented Systems. In *Proceedings of the 8th Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 1993*, Washington, DC, USA, 26 Sept.-1 Oct. 1993. SIGPLAN Notices, Oct. 1993, vol.28, (no.10):326-37.

[DLVW98]de Pauw, W.; Lorenz, D.; Vlissides, J.; Wegman, M. Execution Patterns in Object-Oriented Visualization. *Proceedings of the Fourth USENIX Conference on Object-Oriented Technologies and Systems*, Sante Fe, NM, USA, 27-30 April 1998, USENIX Association, 1998. pp. 219-34.

[Eis97] Eisenstadt, M., My Hairiest Bug War Stories, *Communications of the ACM*, Vol. 40., No. 4, pp. 30–38, April 1997.

[GH93] Golan, M.; Hanson, D.R. Duel-A Very High-Level Debugging Language. In: USENIX Association. *Proceedings of the Winter 1993 USENIX Conference*. San Diego, CA, 25-29 Jan. 1993. Berkley, CA, USA: USENIX Assoc, 1993. p. 107-17.

[GJS96] Gosling, J., Joy, B., Steele, G., *The Java Language Specification*, Addison-Wesley 1996.

[GWM89] Gamma E., Weinand A., Marty R., Integration of a Programming Environment into ET++ - a Case Study, *Proceedings ECOOP'89* (Nottingham, UK, July 10-14), pp. 283-297, S. Cook, ed. Cambridge University Press, Cambridge, 1989.

[HKR97] Hart D., Kraemer E., Roman G.-C., Interactive Visual Exploration of Distributed Computations. *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, pp.121-127, April 1997.

[HKWJ95]Hao, M.C.; Karp, A.H.; Waheed, A.; Jazayeri, M., VIZIR: An Integrated Environment for Distributed Program Visualization. *Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS '95*, pp.288–92, Durham, NC, USA, January 1995.

[Kes90] Kessler, P., Fast Breakpoints: Design and Implementation. *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation* 1990, Published as *SIGPLAN Notices* 25(6), pp. 78–84, ACM Press, June 1990.

[KH98] Keller, R., Hölzle, U.; Binary Component Adaptation, *Proceedings ECOOP'98*, Springer Verlag Lecture Notes on Computer Science, Brussels, Belgium, July 1998.

[KRR94] Kimelman D., Rosenburg B., Roth T., Strata-Various: Multi-Layer Visualization of Dynamics in Software System Behavior, *Proceedings of Visualization'94*, pp. 172-178, IEEE 1994.

[Laf97] Laffra C., *Advanced Java: Idioms, Pitfalls, Styles and Programming Tips*, pp. 229-252, Prentice Hall 1997.

[LB98] Liang, S., Bracha, G.; Dynamic Class Loading in the Java™ Virtual Machine, *Proceedings of OOPSLA'98*, pp. 36-44, Vancouver, October 1998. Published as *SIGPLAN Notices* 33(10), October 1998.

[LHS97] Lencevicius, R.; Hölzle, U.; Singh, A.K., Query-Based Debugging of Object-Oriented Programs, *Proceedings of OOPSLA'97*, pp. 304-317, Atlanta, GA, October 1997. Published as *SIGPLAN Notices* 32(10), October 1997.

[LM94] Laffra C., Malhotra A., HotWire: A Visual Debugger for C++, *Proceedings of the USENIX C++ Conference*, pp. 109-122, Usenix Association 1994.

[LMP97] Litman D.; Mishra A.; Patel-Schneider P.F., Modeling Dynamic Collections of Interdependent Objects Using Path-Based Rules, *Proceedings of OOPSLA'97*, pp. 77-92, Atlanta, GA, October 1997. Published as *SIGPLAN Notices* 32(10), October 1997.

[LN97] Lange, D.B., Nakamura Y. Object-Oriented Program Tracing and Visualization, *IEEE Computer*, vol. 30, no. 5, pp. 63–70, May 1997.

[Mey88] Meyer B., *Object-Oriented Software Construction*, pp. 111 - 163, Prentice-Hall, 1988.

[Mos97] Mössenböck, H., Films as Graphical Comments in the Source Code of Programs. *Proceedings of the International Conference on Technology of Object Oriented Systems and Languages, TOOLS-23*, pp. 89-98, Santa Barbara, CA, USA, July-August 1997.

[RC93] Roman G.-C., Cox K.C., A Taxonomy of Program Visualization Systems, *IEEE Computer* 26(12), pp. 11-24, December 1993.

[Rom92] Roman, G.-C. et al., Pavane: A System for Declarative Visualization of Concurrent Computations, *Journal of Visual Languages and Computing*, Vol. 3, No. 2, pp. 161-193, June 1992.

[SPEC98] Standard Performance Evaluation Corporation, SPEC JVM98 Benchmarks, http://www.spec.org/osg/jvm98/, 1998.

[SSC96] Sefika M., Sane A., Campbell R.H., Architecture-Oriented Visualization, In *Proceedings of OOPSLA'96*, pp. 389-405, San Jose, CA, October 1996. Published as SIGPLAN Notices 31(10), October 1996.

[Sun99] Java™ 2 SDK Production Release, http://www.sun.com/solaris/, 1999.

[Sta90] Stasko, J., TANGO: A Framework and System for Algorithm Animation, *IEEE Computer* 23(9), pp. 27-39.

[Tip95] Tip, F., A Survey of Program Slicing Techniques. *Journal of Programming Languages*, vol.3, (no.3) pp. 121-89, Sept. 1995.

[Wei81] Weiser, M., Program Slicing. In: *5th International Conference on Software Engineering*, San Diego, CA, USA, 9-12 March 1981. New York, NY, USA, pp. 439-49, IEEE, 1981.

[WLG93] Wahbe R., Lucco S., Graham S.L., Practical Data Breakpoints: Design and Implementation. *Proceedings of ACM SIGPLAN conference on Programming Language Design and Implementation* 1993, Albuquerque, June 1993. ACM Press 1993.

[WG94] Weinand, A.; Gamma, E. ET++-a portable, homogenous class library and application framework. In: *Computer Science Research at UBILAB, Strategy and Projects. Proceedings of the UBILAB Conference '94*, Zurich, Switzerland, 1994. pp. 66-92. Edited by: Bischofberger, W.R.; Frei, H.-P. Konstanz, Switzerland: Universitätsverlag Konstanz, 1994.

[WM+98] Walker, R.J., Murphy, G.C., Freeman-Benson, B., Wright, D., Swanson, D., Isaak, J., Visualizing Dynamic Software System Information through High-level Models, *Proceedings of OOPSLA'98*, pp. 271-283, Vancouver, October 1998. Published as *SIGPLAN Notices* 33(10), October 1998.