Automated Software Engineering, 10, 39–74, 2003

Dynamic Query-Based Debugging of Object-Oriented Programs

RAIMONDAS LENCEVICIUS Raimondas.Lencevicius@nokia.com (http://alumni.cs.ucsb.edu/∼raimisl)
Nokia Research Center, 5 Wayside Road, Burlington, MA 01803, USA

URS HÖLZLE urs@google.com (http://www.cs.ucsb.edu/∼urs)
Google Inc, 2400 Bayshore Parkway, Mountain View, CA 94043, USA

AMBUJ K. SINGH ambuj@cs.ucsb.edu (http://www.cs.ucsb.edu/∼ambuj)
Department of Computer Science, University of California, Santa Barbara, CA 93106, USA

Abstract. Program errors are hard to find because of the cause-effect gap between the instant when an error occurs and when the error becomes apparent to the programmer. Although debugging techniques such as conditional and data breakpoints help in finding errors in simple cases, they fail to effectively bridge the cause-effect gap in many situations. This paper proposes two debuggers that provide programmers with an instant error alert by continuously checking inter-object relationships while the debugged program is running. We call such tool a *dynamic query-based debugger*. 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. To enable query-based debugging in the middle of program execution in a portable way, our debugger performs efficient Java class file instrumentation. We call such debugger an *on-the-fly debugger*. Though the on-the-fly debugger has a higher overhead than a dynamic query-based debugger, it offers additional interactive power and flexibility while maintaining complete portability.

Keywords: object-oriented software development, debugging, query optimization, run-time tools

1. Introduction

Many program errors are hard to find because of a cause-effect gap between the instant when the error occurs and when it becomes apparent to the programmer by terminating the program or by producing incorrect results (Eisenstadt, 1997). 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.

Consider trying to debug 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

Substantial parts of this paper have previously appeared in Lencevicius et al. (1999, 2000a, b).

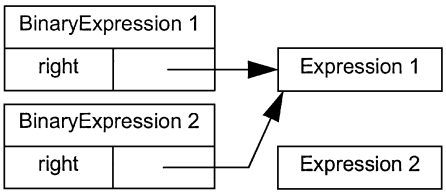

Figure 1. Possible error in javac AST.

the field right of two different parent nodes (figure 1). The parent nodes may be 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 debuggers help programmers to find such errors as soon as they occur?

The programmer could try to use data breakpoints (Wahbe et al., 1993), i.e., breakpoints that stop the program when the value of a particular field changes. However, even conditional data breakpoints 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 booleam 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 it is impossible to determine whether this object and its new field value are erroneous.

A programmer could also try to use another conventional tool, conditional breakpoints (Kessler, 1990). Conditional breakpoints check a condition at a particular program location and stop the program if this condition is true. Conditional breakpoints fail to find javac bug for the same reason: the condition cannot easily reference objects 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, conditional breakpoints provide 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 static query-based debugger (Lencevicius et al., 1997). Similar to an SQL database query tool, a static query-based debugger (QBD) allows user to stop a program at any instant and to find 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. However, the queries have to be fully reevaluated every time that the program is stopped. 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. This claim is substantiated in Section 5.3.1.

This paper proposes 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 reevaluate the query incrementally. 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.

The implementation of the dynamic query-based debugger requires users to specify queries before the program execution starts. Queries are enabled from the beginning of the program execution and remain active until its end. These requirements diminish the usefulness of the debugger because users can not restrict queries to parts of the program execution and can not ask new queries in the middle of a program run. In a long-running program, or in a hard-to-reproduce test case, the ability to add queries on the fly would save a substantial amount of debugging time. Conventional debuggers allow programmers to place simple breakpoints and check variable values on the fly but they do not support complex queries.

An *on-the-fly* query-based debugger implementation proposed in this paper makes query-based debuggers fully interactive while maintaining debugging portability for different Java virtual machines and operating systems. With the on-the-fly debugger, programmers can stop a program at any time using conventional breakpoints or another already enabled query, enter a query or change it later when more information about the error becomes available. The on-the-fly debugger adds a capability to stop the javac program just before the AST construction phase and enable the query. It also allows changing the query later. The system is portable without any changes across Java virtual machines, operating systems, and CPUs. The system was tested on Sun SPARC system running Solaris with Solaris Java 1.2 VM and Intel system running Windows NT with Sun Java 1.2 VM.

This paper discusses implementations and trade-offs of the dynamic and on-the-fly query-based debuggers. We have implemented debuggers for Java in 100% portable pure Java. The dynamic query-based debugger prototype is efficient; experiments with large programs from the SPECjvm98 suite (Standard Performance Evaluation Corporation, 1998) 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.

The on-the-fly debugging overhead measured for programs from the SPECjvm98 suite show that the instrumented programs with inactive debugger suffer an overhead of less than 70% for tested applications. In our experiments, selection query overheads range up to factor 9.5. Join queries are less efficient but may be practical for short debugging runs or small query domains.

2. Query model

Query-based debugging uses the query model proposed in Lencevicius et al. (1997). The query syntax is as follow:

```
<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. We have not implemented existential quantifiers or uniqueness operator in the search domain because of time constraints. The possible use of existential quantifiers is discussed in Lencevicius (2000a).

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 (Gosling et al., 1996) with certain syntactic restrictions. They should not contain variable increments which have no semantic meaning in a query. Constraints can contain method invocations; we assume that these methods are side effect free, i.e. they do not modify any existing program objects. We have not implemented array handling because of time constraints, so we currently disallow array accesses. However, arrays introduce no new implementation difficulties.

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 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 practice, the debugger reevaluates the query as infrequently as we could allow without violating these semantics. In addition, the debugger reevaluates the query only for the part of the query domain that has changed since the last evaluation. Section 4.4.1 describes the incremental reevaluation technique in detail.

Dynamic queries should be evaluated only when the queried objects are in a consistent state. In other words, the expression evaluation should succeed and provide meaningful results. At some program execution points the query evaluation may be unsafe. For example, during an insertion of an element into the list, the list may have an inconsistent state. Therefore, query results can be updated only when all abstractions involved are in a consistent state. However, excluding inconsistent regions is problematic since some of these regions may hide genuine errors. Prohibiting query execution from a part of the runtime makes it more difficult for programmers to understand the query and the program. Consequently, such region exclusion should be left to the direct supervision of programmers, especially since automatic determination of inconsistent states is probably impossible. One way to give users control over excluding inconsistent regions is to use guarded queries. With each query users would provide a "when" or "while" clause that could be consistently evaluated at any time. The rest of the query is evaluated only when (or while) the guard is true. Guards allow to declaratively specify program regions where the query reevaluation is safe. Even though programmers have to spend time writing guards, such programmer assistance seems to be reasonable when a guard is already available or easy to construct. Introduction of guards does not complicate the query model.

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 with selection) of two or more domain variables. Join queries with equality constraints only (e.g., `p1.x == p2.x`) are called *hash joins*. They can be evaluated more efficiently for most domains using a hash table (Lencevicius et al., 1997).

3. Examples

This section presents examples of debugging situations handled by dynamic query debuggers.

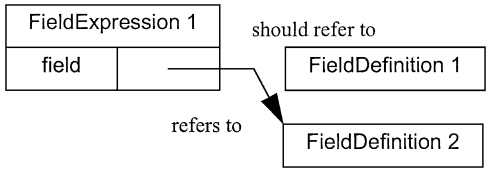

Figure 2. Another possible error in javac AST.

3.1. 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. Programmers 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 specify using conditional breakpoints.

3.2. Ideal gas tank example

Another program we examined is an applet simulating a tank with ideal gas molecules. Although 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 no two molecules occupy the same position. Even this simple application may violate the constraint in different places: in the regular molecule move method, in a method that handles molecule bounces from the walls, and so on. The following query discovers the constraint violation:

```
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 delayed query reevaluation may fail to detect the error entirely.

4. Implementation

This section presents the implementation of the dynamic query-based debugger and the modification of this implementation for the on-the-fly debugging. We have implemented query-based debuggers in pure Java. We used Java's custom class loaders (Liang and Bracha, 1998) 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. The prototype implementation only demonstrates the query-based debugging and on-the-fly capability; for production use, the system should be integrated into a full-fledged debugging suite.

4.1. General structure of the system

Figure 3 shows a data-flow diagram of the on-the-fly debugger. The dynamic query-based debugger structure is similar, only queries have to be specified before program execution. To debug a program using an on-the-fly query-based debugger, 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 object creation and field assignment. The class loader also generates and compiles custom debugger code. After loading, the Java virtual machine executes the instrumented user

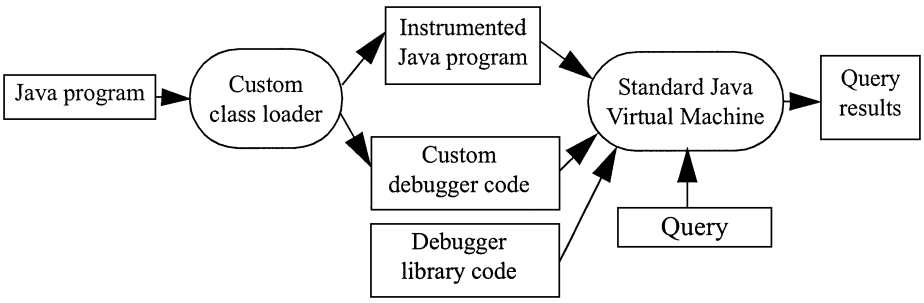

Figure 3. Data-flow diagram of on-the-fly debugger.

program. Whenever the program reaches instrumentation points, it checks whether the debugger is active and if so, invokes the custom debugger code, which calls other debugger runtime libraries to reevaluate the query and to generate query results. Query results are shown as a collection of tuples containing object instances that satisfy the query. Our implementation displays a simple serialized text view of objects; a production debugger could display a manipulative graphical view of objects.

The debugger currently does not handle multithreaded code since operations that affect the query change set would need to be disallowed in threads concurrent to the one evaluating the query. This is a difficult problem and we have not addressed it in our implementation. The debugger does not handle native methods, because their debugging is outside the scope of a Java debugger.

4.2. *Change monitoring*

Debuggers update 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*. All assignments and all constructors form a conservative change set for all queries, however, we are interested in finding a reduced change set that is as small as possible. In our implementation, this change set contains only constructors of domain objects and assignments to domain object fields referenced in a query. Static analysis could decrease the size of the reduced change set even more.

Consider the Molecule query:

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

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 are not included in the reduced change set.

The change set of a query indicates to the debugger which assignments and constructors are relevant for query evaluation. The class loader of a dynamic debugger instruments exactly this set. The on-the-fly debugger instruments all assignments and all constructors at load time, but when a query is defined, the on-the-fly debugger uses the reduced change set to decide when a query should be reevaluated. Before definition of any queries the on-the-fly debugger does not execute evaluation code at all.

To support on-the-fly debugging, the on-the-fly debugger keeps collections of objects belonging to all classes. A dynamic debugger maintains only collections of objects in domain classes. These collections are necessary to evaluate join queries. Since the standard Java debugging API does not allow debuggers to retrieve all objects of a class, debuggers have to track creation of all program objects to have access to all query domain objects. Every time an object is created, the program invokes the debugger which places the new

object into a collection according to its class. During join query evaluation a debugger uses object collections to iterate through all domain objects. To maintain query correctness and to facilitate garbage collection, the debugger allows the garbage collector to delete dead objects from domain collections. Object tracking, although inexpensive by itself, becomes costly because of the excessive memory use—for each object created by a program, the debugger has to maintain a WeakReference object and space in a domain collection. Referring to domain objects through weak references allows the Java virtual machine garbage collector to collect all objects that are referenced only by the debugger. However, even though domain objects are garbage collected, the weak references themselves remain in the collection, so the collection grows as the program runs. Some programs like the gas tank simulation (Section 3.2) create so many temporary objects that weak references fill all available memory. To prevent such internal garbage, a more sophisticated implementation uses an internal "garbage collector" to recycle the weak references no longer pointing to the reachable objects. Unfortunately, the internal garbage collection of weak references adds an additional speed overhead so we have not enabled it by default.

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 explicit chains of references in a 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.

To summarize, debuggers use the change set of the query to reevaluate the query after interesting events. 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 if the change affects query domains.

4.3. Java program instrumentation

In a dynamic query-based debugger, the user specifies a query string at the program start-up. The debugger then instruments Java class files during loading 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 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. Such an implementation cannot support on-the-fly debugging because the debugger has to know a query and its change set to instrument class files at load time. Class files cannot be instrumented after loading without changing the Java virtual machine. On the other hand, changing the JVM would compromise the portability of the debugger across different virtual machines.

If queries are unknown before the program execution, the program has to invoke the debugger after all events that may change the result of any possible query. The on-the-fly debugger instruments class files by inserting debugger invocations after each assignment to an object field. The system also inserts debugger invocations after each call to a constructor of an object. Figure 4 shows an example of such 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 from the Binary Component Adaptation library (Keller and Hölzle, 1998). Then the loader finds all putfield bytecodes and adds invokestatic bytecodes invoking debugger code after the putfield bytecodes. Figure 5 shows the exact bytecodes that replace a single putfield. The system also inserts such debugger invocations after each call to a constructor. The debugger adds a reference to the run method of the Debugger class into the constant pool of the instrumented class. The method takes as an argument the object updated by the putfield—a Molecule object in the example. This object is on the stack before execution of the putfield, so a copy of it can be obtained by stack manipulation (figure 5) that duplicates the top two values on the stack (bytecode #22 in

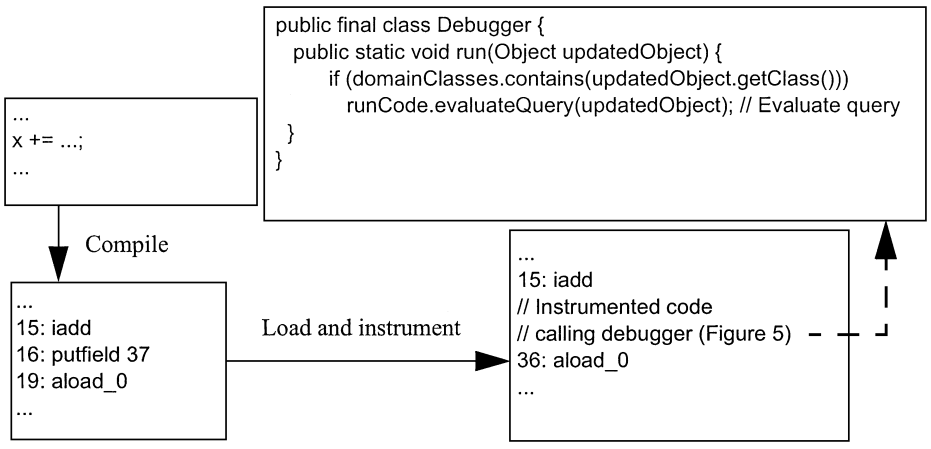

Figure 4. Java program instrumentation.

```
        16: getstatic 133 // Get debugger activation flag
        19: ifeq 14 (33) // If debugger disabled go to bytecode 33
// If debugger enabled get debugger parameters,
// perform putfield, and invoke debugger
        22: dup2 // Duplicate modified object and assigned value
                    on the stack
        23: putfield 35 // Perform original putfield
        26: pop // Pop the assigned value
        27: invokestatic 127 // Invoke debugger
        30: goto 6 (36) // Go to bytecode 36
        33: putfield 35 // Perform original putfield
        36: // end of instrumented block
```

Figure 5. On-the-fly debugging instrumentation.

figure 5) and then discards the top one (the value assigned by the putfield—bytecode #26 in figure 5). The debugger determines the correct types of objects 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.

To limit the overhead if the on-the-fly debugger is not enabled, the instrumentation inserts a test around each putfield code. It is possible to tie the enabling flag to a class, so debugger would be "enabled" per class, however we have not implemented such a scheme. If the debugger is not enabled, the program executes only two additional bytecodes per each putfield bytecode: a load of a debugger flag (getstatic—bytecode #16 in figure 5) and a conditional jump (ifeq—bytecode #19 in figure 5) to the original putfield. Figure 5 shows the instrumentation performed on a single putfield bytecode with a "fast path" of only two extra bytecodes. However, if the debugger is enabled, the overhead is higher. In this case, the debugger has to replicate the reference to the updated object, pass it to the debugger's run method and invoke that method. Then the debugger determines if modified object belongs to one of the query's domain classes, and if so proceeds with further evaluation. The instrumentation increases class file sizes less than 7%.

In a dynamic debugger, the query is known beforehand and only putfield bytecodes belonging to a change set are instrumented. Since the debugger is enabled from the beginning of the execution, no bytecodes checking the debugger activation are inserted.

The instrumentations performed on the bytecode are correct with respect to JVM verifiers. We do not introduce any unverifiable modifications and we preserve the type safety that is checked by the verifier. We have executed the applications on standard JVMs that check the bytecode integrity.

The next section describes how the debugger determines which assignments and constructors influence the query result and when the query is reevaluated.

4.4. Query execution

This section describes what happens in a dynamic debugger after an instrumented event occurs in the debugged program. Whenever the program invokes the debugger, it passes the

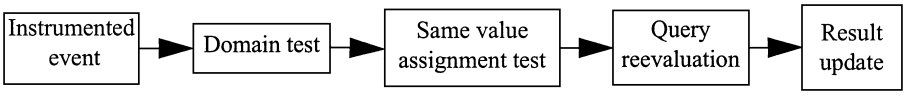

Figure 6. Control flow of query execution.

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 6 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.

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. 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 static query-based debuggers (Lencevicius et al., 1997) 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.

4.4.1. 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 in the query domain 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 (Buneman and Clemons, 1979; Blakeley et al., 1986) to speed up query execution. Consider a query, a join of three domains $A \times B \times C$, e.g.,

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

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

$$A \times B_{changed} \times C = (A \times (B_{original} - \Delta B) \times C) \cup (A \times \Delta B \times C)$$

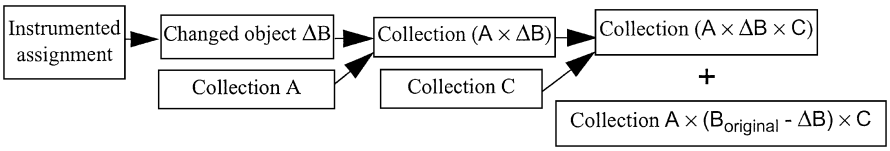

Figure 7. Incremental query evaluation.

The first part of the result is the result of the previous query evaluation after removing all tuples that contain the changed object ΔB. The debugger stores this result—empty for queries in which non-empty result denotes a program error—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 7 shows an incremental evaluation of the query. 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.

The query evaluation is further optimized by finding efficient join orders and by using hash joins as described in Lencevicius et al. (1997). Because sizes of domains change during program runtime and debuggers cannot efficiently determine the selectivities of constraints, we use simple heuristics for join ordering: execute selections first, equality joins next, and inequality constraints last.

4.4.2. 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 4.4.1, 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. Figure 8 shows such a method 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 4.4. The optimized code that follows evaluates the selection constraint on the changed object and calls the debugger runtime only if the query has a 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

```
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 8. Selection evaluation using custom code.

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.

4.4.3. Query execution in an on-the-fly debugger. The query execution in the on-the-fly debugger is the same as in the dynamic debugger. However, custom code generation for selection queries and same value assignment test cannot be performed. It becomes more important to quickly check whether an object that caused a debugger invocation belongs to a relevant domain. If the object does not belong to the query domain, the debugger immediately returns to the execution of the user program without reevaluating the query.

4.5. Alternative on-the-fly implementations

On-the-fly debugging could be implemented using alternative techniques that may increase the debugger efficiency. One approach would be to change the Java virtual machine. Even though we did not pursue this approach because of its lack of portability, JVM changes may lead to the most efficient implementations. These changes could be simple or sophisticated. A simple JVM change would allow the debugger to retrieve all objects of a class.[1] Such capability would remove the necessity to track all objects of all classes and would reduce both the direct object tracking overhead and excessive memory use by weak references. More sophisticated JVM changes would allow to instrument already loaded classes and avoid the overhead of extra bytecodes surrounding each putfield bytecode.

JVM changes are not portable. An alternative technique to speed up a debugger would be to use *shadow* classes. In other words, while the debugger is not enabled, the program would execute the code that is instrumented to check the debugger activation only at the beginning of the methods and possibly at the back branches of the loops. When the debugger is enabled, it would execute fully instrumented versions of the classes. Such fully instrumented shadow methods would be invoked through the redirection at the beginning of the regular methods. This method reduces the overhead of the instrumented putfield execution but does not solve the problem of object tracking. Also, the debugger activation would be delayed

until the instrumentation point is reached. Due to this delay, a debugger could miss some errors.

5. Experimental results

Ideally, a test of the efficiency of a 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 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)) (Java™ 2, 1999). Execution times are elapsed times and were measured with millisecond accuracy using the System currentTimeMillis( ) method.

5.1. *Benchmark queries*

To test query-based debuggers, 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 (Standard Performance Evaluation Corporation, 1998).
- 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.

Table 1. Benchmark queries.

Application		Query	Slowdown	Invocation frequency (events/s)
Ideal gas tank	1.	Molecule1 z. z.x > 350	1.02	15,000
Decaf	2.	Idx. x.type < 0	1.11	16,000
Jess	3.	spec.benchmarks._202_jess.jess.Token z. z.sortcode == −1	1.25	169,000
Compress	4.	spec.benchmarks._201_compress.Output_Buffer z. z.OutCnt < 0	1.18	1,900,00
	5.	spec.benchmarks._201_compress.Output_Buffer z. z.count() < 0	1.27	
	6.	spec.benchmarks._201_compress.Output_Buffer z. z.lessOutCnt(0)	1.37	
	7.	spec.benchmarks._201_compress.Output_Buffer z. z.complexMathOutCnt(0)	5.83	
	8.	spec.benchmarks._201_compress.Compressor z. z.in_count < 0	1.18	933,000
	9.	spec.benchmarks._201_compress.Compressor z. z.out_count < 0	1.10	196,000
	10.	spec.benchmarks._201_compress.Compressor z. z.complexMathOutCount(0)	1.83	
Ray tracer	11.	spec.benchmarks._205_raytrace.Point p. p.x == 1	1.23	787,000
	12.	spec.benchmarks._205_raytrace.Point p. p.farther(100000000)	1.98	2,300,000
Ideal gas tank	13.	Molecule1z; Molecule2 z1. z.x == z1.x && z.y == z1.y && z.dir == z1.dir && z.radius == z1.radius (33 × 33 hash join)	2.13	54,000
Decaf	14.	Lexer 1; Token t. 1.token == t && t.type == 27 (120, 000 × 600 hash join)	3.43	25,000
Ray tracer	15.	spec.benchmarks._205_raytrace. Point p; spec.benchmarks._205_raytrace.IntersectPt ip. p.z==ip.t && p.z < 0 (85,000 × 8,000 hash join)	229 229	350,000 350, 000
Compress	16.	spec.benchmarks._201_compress.Input_Buffer z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt == z.InCnt && z1.OutCnt < 100 && z.InCnt > 0 (1 × 1 hash join)	157	1,500,000
	17.	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 (1 × 1 join)	77	2,600,000
Microbenchmark	18.	Test5 z. z.x < 0	6.4	42,000,000
	19.	TestHash5 th; TestHash1 th1. th.i == th1.i (1 × 20 hash join)	228	40,000,000
	20.	TestHash5 th; TestHash1 th1. th.i < th1.i (1 × 20 join)	930	

Table 2. Application sizes and execution times.

Application	Original size (Kilobytes)	Execution time (s)
1. Compress	17.4	50
2. Jess	387.2	22
3. Ray tracer	55.7	17
4. Decaf	55	15
5. Ideal gas tank	14.3	57

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 dynamic debugger invocation, i.e., the invocation frequency for a dynamic debugger that would perform no operations and would have 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.

The next section discusses the performance of a dynamic query-based debugger for these queries. Section 5.3 then discusses the efficiency benefits of incremental evaluation, custom selection code, and unnecessary assignment detection. Section 5.4 provides experiment results for an on-the-fly debugger.

5.2. Execution time

Figure 9 shows the program execution slowdown for application programs when dynamic 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. We expect overheads of common practical selection queries to be in

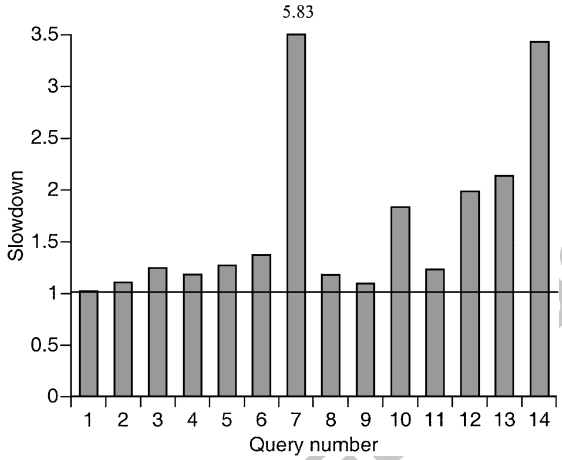

Figure 9. Program slowdown (queries 15–20 not shown). The slowdown is the ratio of the running time with the query active to the running time without any queries. For example, the slowdown of query 3 indicates that the Jess expert system ran 25% slower when the query was enabled.

the same range as our experimental queries; the performance model discussed in Section 6 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 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. Section 6 discusses the likelihood of high update frequencies.

Figure 10 shows the components of the overhead:

- *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.

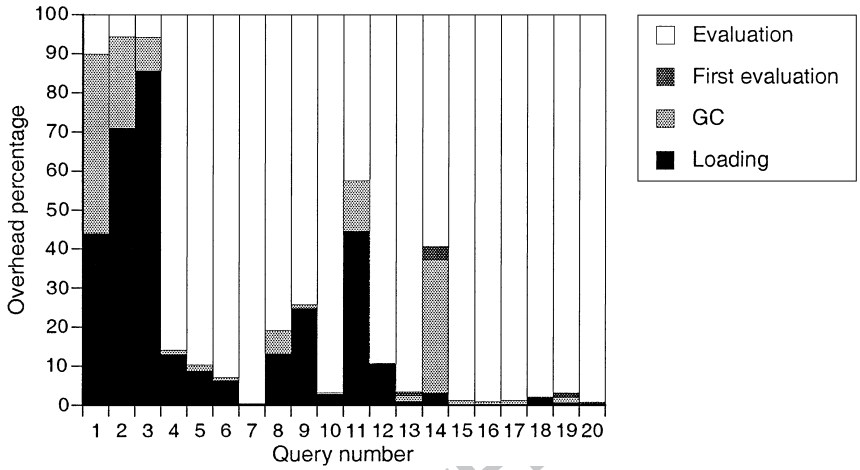

Figure 10. Breakdown of query overhead as a percentage of total overhead. For example, 3% of query 14 overhead is spent on instrumentation, 34% on garbage collection, 3% in the first evaluation, and 60% in subsequent reevaluations.

- *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.
- *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 10 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.[2] 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. The next section discusses some optimizations already reflected in our results.

5.3. 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. Optimizations in the Sections 5.3.1–5.3.3 are disabled independently.

5.3.1. 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 numbers in the table show the ratio of the program running time with a non-incremental query to the program running time with a fully optimized incremental query. For example, a program with query 2 ran for 2.5 hours, which was 554 times slower than the optimized version using the incremental reevaluation that finished in 16.4 seconds. Query 1 was the only query that the non-incremental debugger was evaluated 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.

5.3.2. Custom generated selection code. To estimate the benefit of generating custom code as discussed in Section 4.4.2, we ran all selection queries with the optimization disabled. The results of the experiment are shown in Table 4. The numbers show the slowdown of the unoptimized version compared to the optimized version. For example, a program with query 4 ran 58 times slower than the the program with an 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.

5.3.3. 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.

Table 3. Overhead of non-incremental evaluation.

Application		Query	Slowdown versus optimized
Ideal gas tank	1.	Molecule1 z. z.x > 350	1.16
Decaf	2.	Id x. x.type < 0	554
Jess	3.	spec.benchmarks._202_jess.jess.Token z. z.sortcode == −1	5,725
Compress	4.	spec.benchmarks._201_compress.Output_Buffer z. z.OutCnt < 0	402
	5.	spec.benchmarks._201_compress.Output_Buffer z. z.count() < 0	373
	6.	spec.benchmarks._201_compress.Output_Buffer z. z.lessOutCnt(0)	428
	7.	spec.benchmarks._201_compress.Output_Buffer z. z.complexMathOutCnt(0)	88
	8.	spec.benchmarks._201_compress. Compressor z. zin_count < 0	233
	9.	spec.benchmarks._201_compress.Compressor z. z.out_count < 0	33.8
	10.	spec.benchmarks._201_compress.Compressor z. z.complexMathOutCount(0)	21.8
Ray tracer	11.	spec.benchmarks._205_raytrace.Point p. p.x == 1	8,496
	12.	spec.benchmarks._205_raytrace.Point p. p.farther (100000000)	8,972
Ideal gas tank	13.	Molecule1 z; Molecule2 z1. z.x == z1.x && z.y == z1.y && z.dir == z1.dir && z.radius == z1.radius (33 × 33 hash join)	10.3
Decaf	14.	Lexer l; Token t. 1.token == t && t.type == 27 (120,000 × 600 hash join)	576
Ray tracer	15.	spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace. IntersectPt ip. p.z == ip.t && p.z < 0 (85,000 × 8,000 hash join)	54
Compress	16.	spec.benchmarks._201_compress.Input_Buffer z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt == z.InCnt && z1.OutCnt < 100 && z.InCnt > 0 (1 × 1 hash join)	11
	17.	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 (1 × 1 join)	9
Microbenchmark	18.	Test5 z. z.x < 0	821
	19.	TestHash5 th; TestHash1 th1. th.i == th1.i (1 × 20 hash join)	6.6
	20.	TestHash5 th; TestHash1 th1. th.i < th1.i (1 × 20 join)	6.02

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%

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

Application	Query	Slowdown versus optimized
Ideal gas tank	1. Molecule1 z. z.x > 350	1.03
Decaf	2. Id x. x.type < 0	1.34
Jess	3. spec.benchmarks._202_jess.jess.Token z. z.sortcode == −1	9.26
	4. spec.benchmarks._201_compress.Output_Buffer z. z.OutCnt < 0	58
	5. spec.benchmarks._201_compress.Output_Buffer z. z.count() < 0	51
	6. spec.benchmarks._201_compress.Output_Buffer z. z.lessOutCnt(0)	47
Compress	7. spec.benchmarks._201_compress.Output_Buffer z. z.complexMathOutCnt(0)	12
	8. spec.benchmarks._201_compress.Compressor z. z.in_count < 0	37
	9. spec.benchmarks._201_compress.Compressor z. z.out_count < 0	9.6
	10. spec.benchmarks._201_compress.Compressor z. z.complexMathOutCount(0)	6
Ray tracer	11. spec.benchmarks._205_raytrace.Point p. p.x == 1	15
	12. spec.benchmarks._205_raytrace.Point p. p.farther(100000000)	31
Microbenchmark	13. Test5 z. z.x. < 0	307

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

Application	Query	Slowdown versus optimized	% unnecessary assignments
Ideal gas tank	1. Molecule1 z. z.x > 350	0.99	95
Jess	2. spec.benchmarks._202_jess.jess.Token z. z.sortcode == −1	0.997	7
	3. spec.benchmarks._205_raytrace.Point p. p.x == 1	0.988	15
Ray tracer	4. spec.benchmarks._205_raytrace.Point p. p.farther (100000000)	1.16	40
Ideal gas tank	5. Molecule1 z; Molecule2 z1. z.x == z1.x && z.y == z1.y && z.dir == z1.dir && z.radius == z1.radius (33 × 33 hash join)	1.61	54
Ray tracer	6. spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace.IntersectPt ip. p.z == ip.t && p.z < 0 (85,000 × 8,000 hash join)	1.02	15

unnecessasry 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.

Table 6. On-the-fly debugging overhead.

Application	Size (Kilobytes)	Total number of field assignments	Total assignment frequency (field assignments per second)	Original program execution time (s)	Disabled debugger slowdown	Enabled debugger slowdown
1. Compress	17.4	392,000,000	7,800,000	50.4	1.70	3.14
2. Jess	387.2	25,000,000	1,100,000	22.45	1.30	1.54
3. Db	12	67,000	897	72	1.0	1.0
4. Javac	548	100,000,000	2,600,000	38	1.27	1.62
5. Mpegaudio	117	148,000,000	2,600,000	49.5	1.25	1.96
6. Jack	127	5,700,000	214,000	26	1.15	1.19
7. Ray tracer	55.7	44,000,000	2,200,000	17	1.12	1.62
8. Decaf	55	7,900,000	528,000	15	1.15	1.40
9. Ideal gas tank	14.3	4,000,000	70,000	57	1.27	2.0
10. Microbenchmark	1	100,000,000	40,000,000	2.4	3.28	11.14

Column one gives application, column two—size of its class files, column three—total number of field assignments during the program's execution, column four—field assignment frequency, column five—original program execution time, column six—the slowdown with instrumentation but without debugger invocations, column seven—slowdown with debugger invocations, but with no query evaluations. For example, compress has the size of 17.4 Kilobytes, has 392 million field assignments performed 7.8 million times per second. The program executes in 50.4 seconds. Instrumented compress has a slowdown of 70% and with debugger enabled 3.14 times.

5.4. *On-the-fly debugger overhead*

To evaluate the on-the-fly debugger, we performed the following measurements. First of all, since programs instrumented by the debugger suffer a slowdown even when the debugger is not enabled, we measured this slowdown. Table 6 shows slowdowns together with the total field assignment frequencies for SPECjvm98 programs as well as microbenchmarks. This table indicates that adding two bytecodes (getstatic-ifeq) before each putfield costs less than 70% for applications with an overhead of 3.3 times for a microbenchmark.

If the debugger is enabled, but the query is never evaluated, for example, because domains contain only non-instantiated classes, programs have a larger slowdown. In this case, the instrumented byte code invokes the debugger run method. This method at the very least checks whether the changed object is a domain object. With the debugger enabled, but no query ever evaluated, the applications have a slowdown ranging up to 3.14. The microbenchmark slowdown is 11.14, a number increased by the fact that the microbenchmark assigns to a long integer field that needs more complicated instrumentation with a higher overhead. Both experiments above do not include the object tracking overhead.

Finally, if a query needs to be reevaluated, the additional slowdown to reevaluate the query depends on the query. A large part of the query reevaluation time is consumed by the domain collection maintenance and by extra garbage collection. For example, in selection query 11, 36% of the query evaluation time was due to the object collection and additional GC overhead, 17% of the time was consumed by the domain class check. Overheads for all queries are given in Table 7. Selection overhead ranges up to factor

Table 7. On-the-fly query overhead.

Application		Query	On-the-fly slowdown	DQBD slowdown	Invocation frequency (events/s)
Ideal gas tank	1.	Molecule1 z. z.x > 350	3.23	1.02	15,000
Decaf	2.	Id x. x.type < 0	1.83	1.11	16,000
Jess	3.	spec.benchmarks._202_jess.jess.Token z. z.sortcode == −1	4.05	1.25	169,000
Compress	4.	spec.benchmarks._201_compress.Output_Buffer z. z.OutCnt < 0	6.3	1.18	1,900,000
	5.	spec.benchmarks._201_compress.Output_Buffer z. z.count() < 0	5.48	1.27	
	6.	spec.benchmarks._201_compress.Output_Buffer z. z.lessOutCnt(0)	5.72	1.37	
	7.	spec.benchmarks._201_compress.Output_Buffer z. z.complexMathOutCnt(0)	9.36	5.83	
	8.	spec.benchmarks._201_compress.Compressor z. z.in_count < 0	5.58	1.18	933,000
	9.	spec.benchmarks._201_compress.Compressor z. z.out_count < 0	5.54	1.10	196,000
	10.	spec.benchmarks._201_compress.Compressor z. z.complexMathOutCount(0)	9.54	1.83	
Ray tracer	11.	spec.benchmarks._205_raytrace.Point p. p.x == 1	4.82	1.23	787,000
	12.	spec.benchmarks._205_raytrace.Point p. p.farther(100000000)	4.82	1.98	2,300,000
Ideal gas tank	13.	Molecule1 z; Molecule2 z1. z.x == z1.x && z.y == z1.y && z.dir == z1.dir && z.radius == z1.radius (33 × 33 hash join)	21.82	2.13	54,000
Decaf	14.	Lexer l; Token t. l.token == t && t.type == 27 (120,000 × 600 hash join)	6.4	3.43	25,000
Ray tracer	15.	spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace.IntersectPt ip. p.z == ip.t && p.z < 0 (85,000 × 8,000 hash join)	Inf	229	350,000
Compress	16.	spec.benchmarks._201_compress.Input_Buffer z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt == z.InCnt && z1.OutCnt < 100 && z.InCnt > 0 (1 × 1 hash join)	384	157	1,500,000
	17.	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 (1 × 1 join)	263	77	2,600,000
Microbenchmark	18.	Test5 z. z.x < 0	28	6.4	42,000,000
	19.	TestHash5 th; TestHash1 th1. th.i == th1.i (1 × 20 hash join)	935	228	
	20.	TestHash5 th; TestHash1 th1. th.i < th1.i (1 × 20 join)	935	930	40,000,000

This table gives program slowdown for different queries. First column gives the query, second column—the program slowdown with this query enabled in the on-the-fly debugger, third column—the slowdown for the same query in dynamic query-based debugger, fourth column—debugger invocation frequency if there was no overhead. For example, query 3 has on-the-fly slowdown of 4.05 times, dynamic query slowdown of 25% and invocation frequency of 169000 times per second.

9.5. Selection query overheads almost totally depend on the program executed and neither on the query itself, nor on the query reevaluation frequency. The low cost of selection reevaluation seems to be overshadowed by the overheads of on-the-fly instrumentation, domain collection maintenance, and garbage collection. These areas could yield substantial optimization benefits. Alternatively, user could opt to use a special version of on-the-fly debugger that allows only selections and does not need to maintain domain collections.

Join query overheads are very high. Query 15 was aborted after running for more than a day. However, on-the-fly debugging may be used when programmers only need to check query results during a part of prgram execution. We compared the overheads of the on-the-fly debugger with the dynamic query debugger which instruments only the relevant field assignments but has to run for the whole program execution time. The on-the-fly debugger overhead is on average four times higher than the overhead of the dynamic debugger. The performance is much closer for expensive queries. If programmer wants to inspect only small part of the program runtime, the on-the-fly method is more useful than the dynamic debugging. Also the convenience of asking a query at a breakpoint may be more important than extra overhead.

6. Performance model

To better predict the performance of a dynamic query-based debugger 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 $T_{evaluate}$ 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 131 ns. $T_{evaluate}$ 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.

6.1. *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 8 shows results of these measurements.

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 11 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

Table 8. Maximum field assignment frequencies.

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

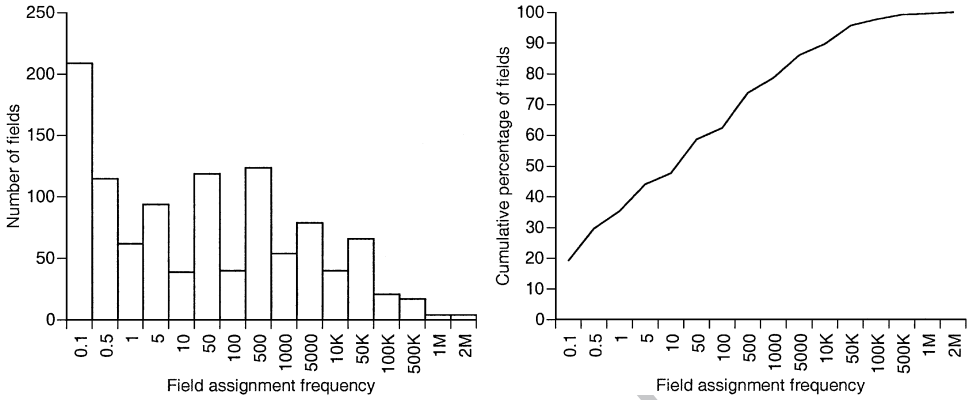

Figure 11. Field assignment frequency in SPECjvm98.

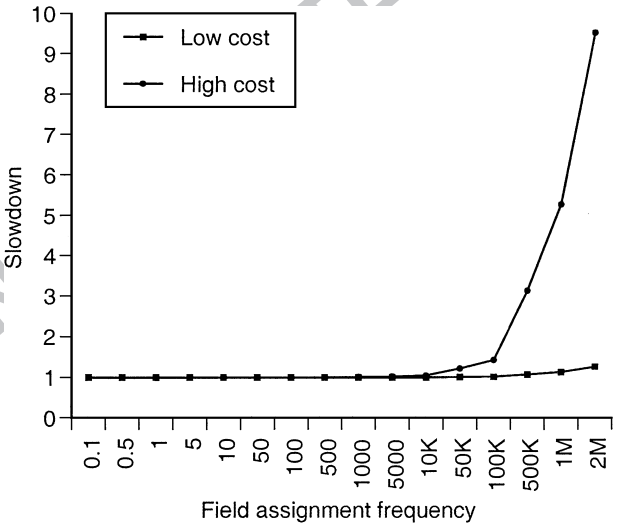

Figure 12. Predicted slowdown. The graph shows the predicted overhead as a function of update frequency. For example, the predicted overhead of a low-cost selection query on a field updated 500,000 times per second is 6.5%; the predicted overhead of a high-cost query with the same frequency is a factor of 3.13.

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 12 uses the minimum (130 ns) and maximum (4.26 μs) values of $\mathsf{T}_{\mathsf{evaluate}}$ from Table 9. 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. It is possible that

Table 9. Frequencies and individual evaluation times.

Application	Query	$F_{evaluate}$ (assignments per second)	$T_{evaluate}$ (μs)
Ideal gas tank	1. Molecule1 z. z.x > 350	N/A	N/A
Decaf	2. Id x. x.type < 0	16,000	3.73
Jess	3. spec.benchmarks._202_jess.jess.Token z. z.sortcode == −1	169,000	3
Compress	4. spec.benchmarks._201_compress.Output_Buffer z. z.OutCat < 0	1,900,000	0.140
	5. spec.benchmarks._201_compress.Output_Buffer z. z.count() < 0		0.208
	6. spec.benchmarks._201_compress.Output_Buffer z. z.lessOutCnt(0)		0.286
	7. spec.benchmarks._201_compress.Output_Buffer z. z.complexMathOutCnt(0)		3.7
	8. spec.benchmarks._201_compress.Compressor z. z.in_count < 0	933,000	0.193
	9. spec.benchmarks._201_compress.Compressor z. z.out_count < 0	196,000	0.488
	10. spec.benchmarks._201_compress.Compressor z. z.complexMathOutCount(0)		4.26
Ray tracer	11. spec.benchmarks._205_raytrace.Point p. p.x == 1	787,000	0.486
	12. spec.benchmarks._205_raytrace.Point p. p.farther(100000000)	2,300,000	0.461
Ideal gas tank	13. Molecule1 z; Molecule2 z1. z.x == z1.x && z.y == z1.y && z.dir == z1.dir && z.radius == z1.radius (33×33 hash join)	N/A	N/A
Decaf	14. Lexer1; Token t. 1.token == t && t.type == 27 (120,000 × 600 hash join)	25,000	56.8
Ray tracer	15. spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace.IntersectPt ip. p.z == ip.t&&p.z < 0 (85,000×8,000 hash join)	350,000	546
Compress	16. spec.benchmarks._201_compress.Input_Buffer z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt == z.InCnt && z1.OutCnt < 100 && z.InCnt > 0 (1×1 hash join)	1,500,000	60
	17. 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 (1×1 join)	2,600,000	51
Microbenchmark	18. Test5 z. z.x < 0	42,000,000	0.131
	19. TestHash5 th; TestHash1 th1. th.i == th1.i (1×20 hash join)	40,000,000	5.7
	20. TestHash5 th; TestHash1 th1. th.i < th1.i (1×20 join)		23

some selection queries will have even larger reevaluation times and consequently higher overheads. 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.

In summary, the performance model predicts that most selection queries in a dynamic debugger will have less than 43% overhead. The model can be used as a framework for concrete overhead predictions and future model refinements. In the future, it is possible to create a more detailed query performance model that takes into account query domain sizes, operations used in the query constraints, and so on.

7. 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?

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 4 remain the same. The only new component of the debugger is a module that maintains the current query result. As discussed in Section 4.4.1, 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.

Table 10. Benchmark queries with non-empty results.

Application	Query	Slowdown
Ideal gas tank	1. Molecule1 z. z.x < 200	1.05
Decaf	2. Id x. x.type == 0	1.23
Jess	3. spec.benchmarks._202_jess.jess.Token z. z.sortcode == 0	1.3
Compress	4. spec.benchmarks._201_compress.Compressor z. z.OutCnt == 0	1.19
	5. spec.benchmarks._201_compress.Compressor z. z.out_count == 0	1.09
Ideal gas tank	6. Molecule1 z; Molecule2 z1. z.x < z1.x && z.y < z1.y (33 × 33 join)	1.47
Decaf	7. Lexer1; Token t. 1.token == t && t.type == 0 (120,000 × 600 hash join)	4.09
Ray tracer	8. spec.benchmarks._205_raytrace.Point p; spec.benchmarks._205_raytrace. IntersectPt ip. (p.z == ip.t) && (p.z > 100) (85,000 × 8,000 hash join)	212.4
Compress	9. spec.benchmarks._201_compress.Compressor z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt == z. out_count (1 × 1 hash join)	9.07
	10. spec.benchmarks._201_compress.Input_Buffer z; spec.benchmarks._201_compress.Output_Buffer z1. z1.OutCnt < z1.InCnt (1 × 1 join)	127
Microbenchmark	11. Test5 z. z.x%2 == 0	45

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 10). 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.

8. Related work

We are unware of other work the directly corresponds to dynamic query-based debugging and its on-the-fly extension. The query-based debugging model and its non-dynamic implementation are presented in a previous paper (Lencevicius et al., 1997). Query-based debugging is discussed in detail in R. Lencevicius' book "Advanced debugging methods" (Lencevicius, 2000a). This book also lists and discusses over 90 different queries and classifies them according to different parameters.

Extensions of object-oriented languages with rules as in R++ (Litman et al., 1997) 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. (1996) 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). Sefika's system allows on-the-fly queries because the underlying system is instrumented for information gathering.

On-the-fly debugging idea is based on the design of the commercial debuggers (e.g., gdb) that allow programmers to stop the program and to add breakpoints before further execution (Kessler, 1990) and to check the values of different variables at a breakpoint. Such capabilities are also available in data breakpoint debuggers (Wahbe et al., 1993). Dynamic query-based debugging extends work on data breakpoints (Wahbe et al., 1993)—breakpoints that stop a program whenever an object field is assigned a certain value. Debuggers that instrument source code programs have also been proposed (Ferguson and Berner, 1963). However, the problem of allowing a portable implementation of on-the-fly debugging through automatic instrumentation of Java class files is new and not addressed in classical debuggers.

While no one has investigated the query-based debugging specifically, various researchers have proposed a variety of enhancements to conventional debugging (Anderson, 1995; Coplien, 1994; De Pauw et al., 1993; Golan and Hanson, 1993; Gamma et al., 1989; Kimelman et al., 1994; Laffra, 1997; Laffra and Malhotra, 1994; Lange and Nakamura, 1997; Weinand and Gamma, 1994). The debuggers most closely related to dynamic query-based debugging visualize object relationships—usually references or an object call graph. Duel (Golan and Hanson, 1993) builds on gdb facilities to display data structures by using user script code at a breakpoint. HotWire (Laffra and Malhotra, 1994) allows users to specify custorn object visualizations in constraint language. Look! (Anderson, 1995) supports adding breakpoints, filters and watch windows at runtime for C++ debugging. Object visualizer (De Pauw et al., 1993). PV (Kimelman et al., 1994), and Program Explorer (Lange and Nakamura, 1997) provide numerous graphical and statistical runtime views with class-dependent filtering but do not allow general queries. Our debugger can gather statistical data through queries with nonempty results ("How many lists of size greater than 500 exist in the program?") but does not display animated statistical views.

Debuggers that gather information by either instrumenting the source code (De Pauw et al., 1993; Laffra and Malhotra, 1994) or by using program traces (Kimelman et al., 1994; Lange and Nakamura, 1997) usually require program recompilation for each different view and do not allow on-the-fly modifications. Gamma et al. (1989) allow different on-the-fly views of debugged programs based on ET++ framework. Laffra (1997) discusses visual debugging in Java using source code instrumentation or JVM changes. As mentioned above, JVM modifications would support on-the-fly debugging, but would make the tool dependent on the modified JVM. We opted for a portable method—class file instrumentation

at load time. The instrumentation is done by providing a custome class loader. Other load-time instrumentation alternatives were comprehensively explored by Duncan and Hölzle (1999).

Consens et al. (1992, 1994) use the Hy$^+$ visualization system to find errors using post-mortem event traces. Ducassé (1999) proposes Coca debugger that allows to users to ask Prolog queries on events reported by a program. Coca selects program's control flow events according to observed data characteristics. De Pauw et al. (1998) and Walker et al. (1998) use program event traces to visualize program execution patterns and event-based object relationships, such as method invocations and object creation. Similarly, Bruegge and Hibbard (1983) use generalized path expressions that refer to program events and variables to check program execution path correctness. The path expressions are powerful tool allowing to identify program statements and variables in different procedure invocations. Path rules can be enabled on-the-fly. However, path expressions do not allow optimized join query evaluations or selections on large groups of objects. The work on trace analysis is complementary to ours because it focuses on querying and visualizing runtime events while we query object relationships.

Au: Pls. verify the year of publication in the ref. citation and list.

Software visualization systems such as BALSA (Brown, 1988), Zeus (Brown, 1991), TANGO/XTANGO/POLKA (Stasko, 1990) Pavane (Roman et al., 1992), and others (Hao et al., 1995; Roman and Cox, 1993) 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. (1997) 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.

Pre-/postconditions and class invariants as provided in Eiffel (Meyer, 1988) 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 arbitary methods. Assertions and invariants cannot be invoked on-the-fly. Dynamic queries could be used to implement class assertions for languages that do not provide them and to support assertions active only during part of the execution. 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.

Defining the model of the debugger independent of the underlying language is a topic of future research. This could be achieved through approaches such as small step semantics (Winskel, 1993).

Dynamic queries are related to incremental join result recalculation in databases (Buneman and Clemons, 1979; Blakeley et al., 1986). We use the basic insights of this work to implement the incremental query evaluation scheme. Database queries and views automatically support "on-the-fly" functionality. There is currently no lightweight Java-based database library that could be used in QBD to provide query evaluation optimizations. Using a stand-alone database for QBD query processing would be nearly impossible because the overhead of data shipping between Java and the database would negate any potential benefits. Future optimizations to the dynamic query-based debugger could include

static program analysis allowing to decrease the necessary instrumentation points and the instrumentation overhead.

9. Conclusions

The cause-effect gap between the instant when a program error occurs and 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.

Our work describes 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. The debugger also 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 dynamic 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, greatly expanding the class of dynamic queries that are practical for everyday debugging.
- Custom code generation speeds up selection queries, further improving efficiency for commonly occurring selection queries.

This paper also presents the on-the-fly query-based debugger that achieves the following goals:

- Interactivity—allows programmers to ask queries in the middle of the program execution to exclude runtime periods when the query is not satisfied or to increase the query's efficiency by enabling it only in a small part of the execution.
- Portability—provides on-the-fly functionality in a portable way for different operating systems and Java virtual machines at a reasonable cost.

These features of the on-the-fly debugger makes it attractive for object-oriented debugging needs.

On-the-fly debugging still has a relatively high overhead. Programs in our experiments with inactive debugger suffer an overhead of up to 70%. An enabled debugger that never evaluates a query—for example, because the query references only non-instantiated classes—has an overhead of up to factor 3. Selection slowdowns for tested queries range up to factor 9.5. The on-the-fly debugger has four times higher overhead than the dynamic query-based debugger. Further optimizations could reduce this overhead. The tool is practical for short program runs and infrequently evaluated queries.

We believe that query-based debugging adds another powerful tool to the programmer's tool chest for tackling the complex task of debugging. 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.

Acknowledgments

We thank Karel Driesen, Edu Metz, and anonymous reviewers for valuable comments on this paper. This work was funded in part by Nokia Research Center, 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.

Notes

1. We implemented it for JDK 1.1.5 during the initial design of a query-based debugger. Such collection may contain dead objects, but so can current domain collections.
2. Experiments were run with 128 M heap, a factor that decreased the GC overhead.

References

Anderson, E. 1995. Dynamic visualization of object programs written in C++. *Objective Software Technology Ltd.*, http://www.objectivesoft.com.

Blakeley, J.A., Larson, P.-A., and Tompa, F. Wm. 1986. Efficiently updating materialized views. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Washington, DC, USA, May 1986. Published as *SIGMOD Record* 15(2):61–71.

Brown, M.H. 1988. Exploring algorithms using Balsa-II. *IEEE Computer*, 21(5):14–36.

Brown, M.H. 1991. Zeus: A system for algorithm animation and multi-view editing. In *Proceedings of IEEE Workshop Visual Languages*, IEEE CS Press, Los Alamitos, CA, pp. 4–9.

Bruegge, B. and Hibbard, P. 1983. Generalized path expressions: A high level debugging mechanism. *Journal of Systems and Software*, 3:265–276.

Buneman, O.P. and Clemons, E.K. 1979. Efficiently monitoring relational databases. *ACM Transactions on Database Systems*, 4(3):368–382.

Consens, M.P., Hasan, M.Z., and Mendelzon, A.O. 1994. Debugging distributed programs by visualizing and querying event traces. *Applications of Databases, First International Conference*, ADB-94, Vadstena, Sweden, In *Proceedings in Lecture Notes in Computer Science*, vol. 819, Springer.

Consens, M., Mendelzon, A., and Ryman, A. 1992. Visualizing and querying software structures. In *International Conference on Software Engineering*, Melbourne, Australia, ACM Press, IEEE Computer Science, pp. 138–156.

Coplien, J.O. 1994. Supporting truly object-oriented debugging of C++ programs. In *Proceedings of the 1994 USENIX C++ Conference*, Cambridge, MA, USA, Berkley, CA, USA: USENIX Assoc, pp. 99–108.

De Pauw, W., Helm, R., Kimelman, D., and Vlissides, J. 1993. 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, *SIGPLAN Notices*, 28(10):326–337.

De Pauw, W., Lorenz, D., Vlissides, J., and Wegman, M. 1998. Execution patterns in object-oriented visualization. In *Proceedings of the Fourth USENIX Conference on Object-Oriented Technologies and Systems*, Sante Fe, NM, USA, USENIX Association, pp. 219–234.

Ducassé, M. 1999. Coca: An automated debugger for C. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, Los Angeles, CA, pp. 504–513.

Duncan, A. and Hölzle, U. 1999. *Load-Time Adaptation: Efficient and Non-Intrusive Language Extension for Virtual Machines*, Technical Report TRCS99–09.

Eisenstadt, M. 1997. My hairiest bug war stories. *Communications of the ACM*, 40(4): 30–38.

Ferguson, H.E. and Berner, E. 1963. Debugging systems at the source language level. *Communications of the ACM*, 6(8):430–432.

Gamma, E., Weinand, A., and Marty, R. 1989. Integration of a programming environment into ET++—A case study. In *Proceedings ECOOP'89* (Nottingham, UK), pp. 283–297, S. Cook, editor, Cambridge University Press: Cambridge.

Golan, M. and Hanson, D.R. 1993. Duel—A very high-level debugging language. In USENIX Association. *Proceedings of the Winter 1993 USENIX Conference*, San Diego, CA, Berkley, CA, USA: USENIX Assoc, pp. 107–117.

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

Hao, M.C., Karp, A.H., Waheed, A., and Jazayeri, M. 1995. VIZIR: An integrated environment for distributed program visualization. In *Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS'95*, Durham, NC, USA, pp. 288–292.

Hart, D., Kraemer, E., and Roman, G.-C. 1997. Interactive visual exploration of distributed computations. In *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, pp. 121–127.

Java™ 2 SDK Production Release. 1999. http://www.sun.com/solaris/.

Keller, R. and Hölzle, U. 1998. Binary component adaptation. In *Proceedings ECOOP'98*, Springer Verlag, Lecture Notes on Computer Science, Brussels, Belgium.

Kessler, P. 1990. Fast breakpoints: Design and implementation. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation* 1990, Published as *SIGPLAN Notices*, 25(6):78–84, ACM Press.

Kimelman, D., Rosenburg, B., and Roth, T. 1994. Strata-various: Multi-layer visualization of dynamics in software system behavior. In *Proceedings of Visualization'94*, IEEE, pp. 172–178.

Laffra, C. and Malhotra, A. 1994. HotWire: A visual debugger for C++. In *Proceedings of the USENIX C++ Conference*, Usenix Association, pp. 109–122.

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

Lange, D.B. and Nakamura, Y. 1997. Object-oriented program tracing and visualization. *IEEE Computer*, vol. 30, no. 5, pp. 63–70.

Lencevicius, R. 2000a. *Advanced Debugging Methods*, Kluwer Academic Publishers.

Lencevicius, R. 2000b. On-the-fly query-based debugging with examples. In *Proceedings of the Fourth International Workshop on Automated Debugging* (AADEBUG 2000), pp. 55–68.

Lencevicius, R., Hölzle, U., and Singh, A.K. 1997. Query-based debugging of object-oriented programs. In *Proceedings of OOPSLA'97*, Atlanta, GA, Published as *SIGPLAN Notices*, 32(10):304–317.

Lencevicius, R., Hölzle, U., and Singh, A.K. 1999. Dynamic query-based debugging. In *Proceedings of the 13th European Conference on Object-Oriented Programming'99*, (ECOOP'99), Lisbon, Portugal, Published as *Lecture Notes on Computer Science 1628*, Springer-Verlag.

Liang, S. and Bracha, G. 1998. Dynamic class loading in the Java™ virtual machine. In *Proceedings of OOPSLA'98*, Vancouver, Published as *SIGPLAN Notices*, 33(10):36–44.

Litman, D., Mishra, A., and Patel-Schneider, P.F. 1997. Modeling dynamic collections of interdependent objects using path-based rules. In *Proceedings of OOPSLA'97*, Atlanta, GA, Published as *SIGPLAN Notices*, 32(10):77–92.

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

Roman, G.-C. et al. 1992. Pavane: A system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing*, 3(2):161–193.

Roman, G.-C. and Cox, K.C. 1993. A taxonomy of program visualization systems. *IEEE Computer*, 26(12): 11–24.

Sefika, M., Sane, A., and Campbell, R.H. 1996. Architecture-oriented visualization. In *Proceedings of OOPSLA'96*, San Jose, CA. Published as *SIGPLAN Notices*, 31(10):389–405.

Au: Pls. verify the year of publication for stasko, J.

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

Stasko, J. 1990. TANGO: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39.

Wahbe, R., Lucco, S., and Graham, S.L. 1993. Practical data breakpoints: Design and implementation. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Albuquerque, ACM Press.

Walker, R.J., Murphy, G.C., Freeman-Benson, B., Wright, D., Swanson, D., and Isaak, J. 1998. Visualizing dynamic software system information through high-level models. In *Proceedings of OOPSLA'98*, Vancouver, Published as *SIGPLAN Notices*, 33(10):271–283.

Weinand, A. and Gamma, E. 1994. 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, W.R. Bischofberger and H.-P. Frei, editors, Konstanz, Switzerland: Universitätsverlag Konstanz, pp. 66–92.

Winskel, G. 1993. *The Formal Semantics of Programming Languages: An Introduction*, MIT Press.