DEAL: Rich Heap Assertions (Almost) For Free

Christoph Reichenbach
University of Massachusetts
creichen@gmail.com

Neil Immerman
University of Massachusetts
neil.immerman@gmail.com

Yannis Smaragdakis
University of Massachusetts
yannis@cs.umass.edu

Edward Aftandilian
Tufts University
eaftan@cs.tufts.edu

Samuel Z. Guyer
Tufts University
sguyer@cs.tufts.edu

Abstract

We present the DEAL language for heap assertions that are efficiently evaluated during garbage collection time. DEAL is a rich, declarative, logic-based language whose programs are guaranteed to be executable with good whole-heap locality, i.e., within a single traversal over every live object on the heap and a finite neighborhood around each object. As a result, evaluating DEAL programs incurs negligible cost: for simple assertion checking at each garbage collection, the end-to-end execution slowdown is below 2%. DEAL is integrated into Java as a VM extension and we demonstrate its efficiency and expressiveness with several applications and properties from the past literature.

Compared to past systems for heap assertions, DEAL is distinguished by its very attractive expressiveness/efficiency tradeoff: it offers a significantly richer class of assertions than what past systems could check with a single traversal. Conversely, past systems that can express the same (or more) complex assertions as DEAL do so only by suffering orders-of-magnitude higher costs.

1. Introduction

Garbage collection (GC) is a widely popular mechanism in modern programming languages, employed in representatives of virtually all language classes, from mainstream managed languages (e.g., Java, C#), to dynamic languages (e.g., Perl, Python), and to the language avant-garde (e.g., Haskell, OCaml). The traditional view on GC considers it a necessary cost. Our work is based on the observation that GC also represents a unique opportunity. GC necessitates an occasional traversal of all live program objects. If all objects are going to be traversed anyway, it is natural to consider using this traversal for more than computing liveness. With minimal machinery that piggybacks on the GC traversal, the system can perform computation over a program's heap. Such computation should be side-effect free, but also similar in structure to a garbage collection traversal, in order to exploit GC well without undue overhead. Furthermore, this computation should not compute results essential to the main program's control flow, or it would cause much more frequent garbage collection than would be otherwise necessary.

Assertion checking is a good fit for these requirements. Ensuring program properties by way of programmer-defined assertions is a crucial technique in software development and forms the cornerstone of methodologies such as Design by Contract [15]. Virtually every programmer with modern training has been well-exposed to the idea that using assertions to trigger program failures is an excellent way to reveal faults. Since assertion checking only queries program data but does not update them, it requires no computations with side effects. Additionally assertions should not alter the program control flow or compute results that the program will later use.

Performing assertion checking during GC time is not new. The QVM Java virtual machine [3] allows *heap probes*, which check reachability and dominance properties at GC time. The *GC Assertions* system of Aftandilian and Guyer [1] describes a suite of assertions that can be efficiently checked at GC time. The limitation of these past systems, however, is that either they do not offer a general language for writing assertions or they require multiple GC traversals to evaluate a single assertion. In this way, they sacrifice either efficiency or expressiveness. For instance, the Aftandilian and Guyer system includes very specific kinds of assertions, such as `assert-unshared(obj)`. Recent versions of the QVM system allow full JML assertions but cannot evaluate those in a single GC traversal, often resulting in numerous heap traversals in the course of evaluating an asser-

OOPSLA 2010, you've got to accept this paper

tion. As a consequence, efficiency for QVM assertions relies on reducing the cost of every GC via parallelism. Still, performing multiple GCs, even in parallel, is much more costly than piggybacking on a single GC (which would need to occur anyway).

In contrast, we present a Declarative Assertion Language (DeAL), which fulfills the requirements of computing during GC time: it only allows side-effect-free computations and it can only express programs whose object access patterns are a good fit for a GC traversal and incur constant overhead per object. Our approach is informed by Descriptive Complexity, where the important complexity classes (LOGSPACE, PTIME, NP, PSPACE, etc.) have been captured by logic-based languages [12]. Although the performance property we are interested in does not correspond to a complexity class, one can view it as a limited case of linear-time computation, where we have the extra guarantee that the program is evaluated in a single traversal of the heap. This is not a property that is well-expressed in complexity theory, but it is an often-cited property of good locality for garbage collectors. For instance, Wilson's GC survey [25] distinguishes algorithms based on their locality. A GC algorithm is considered to have good locality when it only needs to visit each object once. Algorithms that need to perform multiple passes over the heap have bad temporal locality, since they always access the data that have not been accessed the longest. This behavior is a bad fit for LRU-based cache mechanisms, especially for traversals that revisit structures slightly larger than the cache.

DeAL is a logic-based extension of Java and guarantees that any assertion expressible in it runs by traversing each live heap object (and a constant-size neighborhood around it) exactly once. From a logic standpoint, DeAL is a single-variable first order logic with predicates of the form $\mathbf{R}c/d(x)$, meaning that object x is reachable from object c without going through object d. For instance, we can express the well-formedness property of a doubly-linked list d via a logical assertion:

$$\forall x \in \texttt{Node} : \mathbf{R}d(x) \Rightarrow (x.\texttt{next.prev} = x)$$

The concrete DeAL syntax uses a textual representation of logical operators. For instance, the above property is written:

```
forall Node x: reach[d](x) -> x.next.prev == x
```

DeAL is an expressive language, as we will demonstrate with many examples. Its generality allows us to write complex properties (which past systems needed to hard-code) as straightforward logical expressions. For instance, we can write the property "all `Node` objects are dominated by object d", i.e., no path to them exists that does not pass through d:

$$\forall x \in \texttt{Node} : \neg\mathbf{R}r_0/d(x)$$

The special constant r_0 refers to the root set of the GC traversal. Therefore the property means that no `Node` object is reachable without going through d. We may write this more simply as $\forall x \in \texttt{Node} : \neg\mathbf{R}/d(x)$. If there is nothing before the slash sign, the source is assumed to be r_0.

Using standard boolean connectives we can write interesting combinations of properties—e.g., the DeAL assertion:

$$(\forall x \in \texttt{Node} : (x.\texttt{data} > \texttt{threshold}) \Rightarrow \mathbf{R}\texttt{hotcache}(x)) \wedge$$
$$(\exists y \in \texttt{Node} : \neg\mathbf{R}\texttt{hotcache}(y) \wedge y.\texttt{next} = \texttt{null})$$

(In words, "every `Node` object with an above-threshold `data` value is in the `hotcache` data structure, and some `Node` object that is not in `hotcache` has no `next` link.") Although non-trivial, this assertion (like any other legal DeAL program, as defined in Section 2) is evaluated in a single heap traversal, per DeAL's locality guarantees.

Exploiting the GC allows DeAL assertions to be evaluated with very low cost. We implemented DeAL by extending the Jikes RVM and Eclipse Java Compiler and evaluated its efficiency extensively. The overhead of evaluating DeAL assertions of course varies with the complexity of the assertion: if the assertion consists of comparing k field dereference expressions that depend on the current object, it is inevitable to incur $\Omega(k)$ cost per object. For typical small properties, however, DeAL imposes at most a few percent overhead over the cost of garbage collection, resulting in near-zero (sub-2%) overhead over the end-to-end execution time of applications.

In summary our paper makes the following contributions:

- We define a declarative language for linear time, single-traversal computations, which results in good locality for whole-heap traversals. To our knowledge, no past language has ever offered this guarantee. The property that all legal programs visit each object exactly once is a general program property, not limited to the domain of computing during GC. As such, our language design and overall approach may find even more applications in the future.
- We designed our language, DeAL, as a natural Java extension for assertion checking and implemented it in a modern high-performance JVM. We demonstrate the expressiveness of our design with numerous examples, mostly from the recent research literature.
- We evaluate the practical effectiveness and efficiency of DeAL over a suite of assertions and applications. Our results demonstrate that the idea of an expressive language for assertion checking at GC time has excellent properties and promise for wide adoption.

2. The DeAL Language

We next present DeAL's syntax, semantics, and usage examples.

2.1 Language Design

Syntax. The full abstract syntax of the DeAL language is shown in Figure 1. Since DeAL's expressions are first-

$$
\begin{aligned}
\text{DealAssertion} &::= \texttt{assertion}(f) \mid \\
&\quad\ \texttt{assertDisjoint}(f) \mid \\
&\quad\ \texttt{unsafeAssert}(f) \mid \\
f \in \text{Formula} &::= q \mid f \vee f \mid f \wedge f \mid \neg f \mid f \Rightarrow f \mid f \Leftrightarrow f \\
q \in \text{QFormula} &::= \exists x \in T : b \mid \forall x \in T : b \\
b \in \text{BoolFormula} &::= r \mid b \vee b \mid b \wedge b \mid \neg b \mid b \Rightarrow b \mid b \Leftrightarrow b \\
r \in \text{RelExpr} &::= o \mid o \in T \mid o = o \mid o < o \mid o > o \mid \\
&\quad\ o \leq o \mid o \geq o \mid o \neq o \mid \mathbf{R}t(x) \\
t \in \text{TraversalSrc} &::= j/\overrightarrow{j} \mid /\overrightarrow{j} \mid t \vee t \\
o \in \text{Obj} &::= v \mid c \mid x \mid o.o \\
j \in \text{JavaObj} &::= v \mid c \mid j.j \\
x \in \text{LVar} &\text{ is a set of logic variable names} \\
v \in \text{JVar} &\text{ is a set of Java variable names} \\
T \in \text{Typ} &::= \text{ Java types} \\
c \in \text{Const} &::= \text{ Java constants}
\end{aligned}
$$

Figure 1. DEAL syntax. Note the distinction between expressions that do and do not contain logic variables (sets Obj and JavaObj, respectively). Their only difference is that the former cannot appear as traversal sources in an **R** predicate.

order logic formulas, we will freely alternate between the logical and the concrete DEAL syntax, as convenient for presentation purposes. The reader should treat the logical notation as a shorthand for the text DEAL/Java syntax, i.e., $\forall x \in T$, $\exists x \in T$, $\mathbf{R}t$, $\wedge$, $\vee$, $\neg$, $\in$, $=$, $\Rightarrow$, $\Leftrightarrow$, $\neq$, $\leq$, $\geq$ are synonyms of the actual DEAL operators `forall T x`, `exists T x`, `reach[t]`, `&&`, `||`, `!`, `instanceof`, `==`, `->`, `<->`, `!=`, `<=`, `>=`, respectively.

The language has three primitives: `assertion(`f`)`, `assertDisjoint(`f`)`, and `unsafeAssert(`f`)`, where f is a sentence in first order logic. All three primitives attempt to check whether the formula is true, but they have different requirements and soundness guarantees, discussed later. All the logical connectives have their usual meaning, with forall/exists quantification implicitly applying over all live heap objects. Objects can be compared for equality and primitive-valued expressions can be compared with the standard primitive relational operators $<$, $\leq$, etc. There is a single kind of unary predicate, **R**, described below.

There are some design decisions visible in the syntax that are key for establishing the property that each heap object is traversed once. The first important element is that our first order logic has only one variable. This does not mean that we cannot use multiple logic variable names in a single formula for clarity (as we did in our example in Section 1 with a "$(\forall x \ldots) \wedge (\exists y \ldots)$"), but that quantifiers cannot be nested, hence all logic variables could be renamed to have the same name without changing the meaning of the formula.

The second important element is that the unary predicates of the DEAL logic have the form $\mathbf{R}c_1/\{o_1, \ldots, o_i\} \vee c_2/\{o_{i+1}, \ldots, o_j\} \vee \ldots \vee c_n/\{o_{l+1}, \ldots, o_m\}(x)$. Note that the $\vee$s in the above are part of the predicate name and should not be confused with the usual logical "or" connective. The meaning of the above relation is that x is reachable from c_1 (called a *source*) via a path that does not include any of the objects $o_1, \ldots, o_i$ or x is reachable from c_2 via a path that does include any of the objects $o_1, \ldots, o_j$ or $\ldots$ or x is reachable from c_n via a path that does not include any of the objects $o_1, \ldots, o_m$. (Note that the set of excluded objects keeps growing monotonically: the objects specified to be excluded for every new source c_i are in addition to all previously excluded objects.) An empty source c_i is shorthand for the root set of the garbage collection.

As we will see, the construction of the **R** predicate allows us to evaluate it in a single pass through the heap. Intuitively, the key feature of using a single **R** predicate to link reachability properties only with an *or* connective is that the predicate value is determined regardless of which path in the object graph was used to reach an object. That is, if we reach an object via one path, we do not need to later try to also reach it via other paths.

We should point out precisely how the language of DEAL formulas, f, formally fits the mathematical definition of a first order logic—e.g., see [12]. From the logic perspective, Java object expressions are treated as logical constants—e.g., a Java variable `v` is a constant from the perspective of the logic, since its meaning does not change for different values of the logical variables. Java fields are treated as functions—for instance, the expression `x.next` is equivalent to a logical function "`next`" applied to logical object `x`.

Type Checking and Assertion Semantics. DEAL assertions can appear as Java statements at any legal point in a Java+DEAL program. Each assertion (a member of DealAssertion in the syntax of Figure 1) is a separate DEAL program, evaluated independently. The DEAL guarantee is that each assertion requires a single traversal over objects, and thus can be evaluated during a single garbage collection. (In Section 2.2 we discuss in more detail the various options on how often assertions are evaluated and how they are integrated in our current implementation.)

For a syntactically well-formed DEAL program to also be valid, it first needs to satisfy Java type constraints on its expressions. The types of field reference expressions (either j or o in Figure 1) have to be compatible with their uses with operators $=$, $\leq$, etc. That is, a relational expression (r in the syntax) should be either a legal Java expression with type boolean or an instance of predicate **R**.

More interestingly, however, DEAL imposes restrictions on how the **R** predicates are used in different kinds of asser-

tions. These restrictions are closely tied to the meaning of `assertion` vs. `assertDisjoint` vs. `unsafeAssert`.

- In instances of `assertion(`f`)`, at most one reachability relation, $\mathbf{R}c_1/\{o_1,\ldots,o_i\} \vee c_2/\{o_{i+1},\ldots,o_j\} \vee \ldots \vee c_n/\{o_{l+1},\ldots,o_m\}(x)$, may occur in f, although the same relation may occur multiple times. An `assertion` primitive straightforwardly checks the asserted condition in a single traversal over all live objects.
- In instances of `assertDisjoint(`f`)`, multiple reachability relations may occur, but, if two such relations occur, then in addition to asserting f we are asserting that the domains of the two relations are disjoint. If there is a node that satisfies both relations then we will detect this and report failure of the assertion. (If the domains are all disjoint, we can evaluate f in a single run through the heap during garbage collection, and if the domains are not all disjoint then we will detect this during the single traversal of the heap.)
- In instances of `unsafeAssert(`f`)`, multiple **R** relations may occur without a claim of disjointness. In this mode, the DeAL system will behave as if the domains of the **R** relations mentioned are all disjoint, without checking this property. If the relations are not disjoint, then we are not guaranteed to detect this, nor to actually check f. Specifically, DeAL will not correctly check formula f if the truth value of f depends on objects that simultaneously satisfy multiple **R** relations. The reason is that DeAL will only traverse such objects once, while processing *one* of the **R** relations and without knowing whether the object satisfies other **R**s. Yet formula f may express different requirements for an object depending on which **R** it satisfies.

2.2 DeAL Language Usage

We next discuss how the user interacts with the DeAL language and offer specific usage scenarios. We begin with specifics on the current DeAL integration in Java. Although these are largely engineering choices (which could change without altering the essence of the language) they are useful in order for the reader to establish a concrete model of usage.

Integration. We have integrated DeAL to Java as a conservative extension. The DeAL language is hidden behind a library and the user invokes assertions by calling static methods `assertion`, `assertDisjoint`, and `unsafeAssert` in the main DeAL class. The method calls accept DeAL formulas as regular strings. For instance, the doubly-linked list consistency invariant is asserted as:

```
assertion(
 "forall Node x: reach[d](x) -> x.next.prev == x");
```

This allows unsuspecting Java compilers to process DeAL programs by linking to a simple stub library. The DeAL compiler, however, fully parses and statically type checks the assertion string. (As indicated earlier, most of the examples in this paper use the equivalent DeAL syntax of Figure 1 instead of the concrete textual syntax. Accordingly, we elide the quotes that delimit the assertion string.)

DeAL assertions are evaluated during a garbage collection traversal of the heap. The default evaluation model for DeAL executes assertion statements at the point they are encountered, rather than delaying until the time when garbage collection would occur anyway. Compared to past systems, this design decision resembles that of the QVM system [3], as opposed to the option followed by Aftandilian and Guyer [1], which puts off assertion checking until GC time. The latter has the disadvantage that the shape of the heap may have changed between the point of assertion statement and the point of its evaluation, resulting in incorrect evaluation (both unsound and incomplete).

Naturally, the issue with evaluating assertions at the point of their statement is that if every assertion causes a garbage collection then GC will occur too frequently, with possibly high overhead. We address this by only evaluating assertions if sufficiently long time has elapsed since the last GC. In this way, assertions can be ignored, resulting in incompleteness.[1] By default, we tie our decision of whether an assertion is to be evaluated or not to the same internal metrics that the Jikes RVM GC uses to decide whether to perform a collection. Thus, we evaluate an assertion roughly around the time a GC would be due to happen anyway. This is what we view as the most likely mode of use of DeAL: assertion checking happens at low frequency but without imposing heavy overhead. We offer the user the option of "throttling" the evaluation frequency of assertions, up to executing every single assertion encountered and suffering the resulting overhead.

Examples. There are several possibilities for using DeAL in the course of regular programming. These include local data structure invariants, ownership and dominance properties, disjointness properties, reachability properties, and combinations of those. We show below several DeAL assertions, most of them taken from the recent literature [7–9, 13, 18, 19, 21].

A practically common kind of condition concerns type constraints. For instance, all objects in a data structure may need to be `Serializable` and no instance of a subtype of `Thread` may be stored in the structure:

$$\texttt{assertion}(\forall x \in \texttt{Node} : \mathbf{R}\texttt{HttpSession}(x) \Rightarrow x\ \texttt{instanceof Serializable} \wedge \neg(x\ \texttt{instanceof Thread}));$$

(Note that although class `Thread` is not `Serializable`, a user-defined subclass of it may be.)

Local invariants for data structures are quite common and useful. We already saw a doubly-linked list invariant, and a

[1] We choose to view DeAL as a fault detector, hence the meanings of "sound" and "complete" are exactly inverse from what they would be for a correctness verifier.

sortedness invariant is equally easy to express:

$$\texttt{assertion}(\forall x \in \texttt{Node} : \mathbf{R}\texttt{c}(x) \Rightarrow x.\texttt{data} \leq x.\texttt{next.data});$$

(Note that we can ignore the case of x.next being null. If a subexpression of the field dereference expression is null DEAL ignores the corresponding value of the logical variable.[2] This, however, only applies to field dereference expressions that start with a logical (quantified) variable. For field dereferences that are regular Java expressions, it is the responsibility of the programmer to ensure that the references are not null. For instance, if an assertion contains the expression "x.next $\neq$ list.first", where list is a regular Java variable, then DEAL evaluation will cause an exception in case the value of list is null.)

It is similarly easy to specify that a data structure rooted at c should have a cycle:

$$\texttt{assertion}(\forall x \in \texttt{Node} : \mathbf{R}\texttt{c}(x) \Rightarrow x.\texttt{next} \neq \texttt{null});$$

In the same way, we can state data structure invariants for binary and n-ary trees, heaps, etc.

More interesting properties are those that require the reachability of all objects that satisfy a condition:

$$\texttt{assertion}(\forall x \in \texttt{Node} : (x.\texttt{next} = \texttt{null}) \Rightarrow \mathbf{R}\texttt{leaves}(x));$$

Such properties are not easily expressible in plain Java, because they refer to any possible live object (with a given type, in this case) and not just to objects in a specific structure that could be traversed with a plain for loop.

With an existential quantifier, we can easily assert that a value satisfying certain properties is found, independently of which data structure contains it:

$$\texttt{assertion}(\exists x \in \texttt{Node} : x.\texttt{data} > 3.141 \wedge x.\texttt{data} < 3.142);$$

As mentioned briefly in the Introduction, we can express dominance properties, as non-reachability if the dominator is excluded. For instance, we can evaluate a property over all dominated nodes:

$$\texttt{assertion}(\forall x \in \texttt{Node} : \neg\mathbf{R}/\texttt{c}(x) \Rightarrow x.\texttt{data} > 0);$$

Dominance by a single object is an object ownership property. Combinations of dominance properties allow more complex conditions. For instance, we can specify that every object of type Person has to be dominated by the combination of data structures males and females. That is, no path in the object graph should be able to reach a Person object without going through the object called males or the object called females in the program:

$$\texttt{assertion}(\forall x \in \texttt{Person} : \neg\mathbf{R}/\texttt{males},\texttt{females}(x));$$

[2] For example, if in the quantified expression, "$Qx : \varphi$," x.next.data occurs in φ, then the quantifier is restricted to the case that x.next $\neq$ null, i.e, (Qx such that x.next $\neq$ null) : φ.

Note that dominance by a set of sources is not equivalent to any boolean combination of individual dominance properties. In particular, "x is dominated by the combined objects males and females" is equivalent to neither "x is dominated by males AND x is dominated by females" nor "x is dominated by males OR x is dominated by females". (Consider objects reachable from both data structures, which are dominated by neither one.)

If we want to state that the two structures are disjoint, we can do so with an assertDisjoint statement, which will also allow us to combine other conditions on the objects. For instance, we can have:

$$\texttt{assertDisjoint}(\forall x \in \texttt{Person} : \mathbf{R}\texttt{males}(x) \vee \mathbf{R}\texttt{females}(x));$$

This states that any object of type Person is reachable from one of the two data structures, though there may also be other paths to it. Since an assertDisjoint allows multiple disjoint **R** predicates in the same assertion, we can easily strengthen the condition to express "every Person is dominated by either males or females but not both", by combining the last two example formulas:

$$\begin{aligned}\texttt{assertDisjoint}(\forall x \in \texttt{Person} :\\ (\mathbf{R}\texttt{males}(x) \vee \mathbf{R}\texttt{females}(x)) \wedge\\ \neg\mathbf{R}/\texttt{males},\texttt{females}(x));\end{aligned}$$

Disjointness conditions have a variety of applications, including leak detection (data may be reachable by unexpected data structures) and detection of improper sharing among threads. For instance, we can state that trees from distinct compilation phases do not share structure:

$$\texttt{assertDisjoint}(\forall x : \neg(\mathbf{R}\texttt{parseTree}(x) \wedge \mathbf{R}\texttt{syntaxTree}(x)));$$

The above examples are representative of simple but powerful DEAL assertions. The examples do not include uses of unsafeAssert which allows more complex assertions but with few guarantees, as we describe next.

Unsafe Assertions. The unsafeAssert primitive gives the user extra flexibility but at the cost of not guaranteeing correctness. We do not consider unsafeAssert to be a core part of the DEAL language, and we have not used it in practice. Nevertheless, it represents an escape clause for the user to specify richer properties than those that DEAL can statically guarantee to check in a single traversal. It is the responsibility of the user to ensure that such properties make sense under DEAL's single-traversal execution. Therefore, it is worth elucidating possible usage patterns, since correctness depends on understanding them. There are four possibilities for the correctness of evaluating an unsafeAssert:

- The formula is evaluated soundly and completely, i.e., DEAL assertion failures correctly indicate that the formula is false, and if the DEAL assertion evaluation does not fail,

the formula is true. This is the case when the shape of the properties or of the heap ensures that the truth value of the formula does not depend on which objects satisfy multiple **R**s. For a simple example, consider a formula that treats all objects reachable under two different sets of conditions the same way: $\forall x : (\mathbf{R}c_1/o_1(x) \Rightarrow x.\texttt{data} > 0) \wedge (\mathbf{R}c_2/o_2(x) \Rightarrow x.\texttt{data} > 0)$.

- The formula is evaluated soundly, i.e., DEAL assertion failures correctly indicate that the formula is false. This means that some DEAL warnings may be missed, but this is likely practically acceptable. For example, the formula may contain two distinct **R** predicates with one of them implying stronger conditions than the other (e.g., $\forall x : (\mathbf{R}c_1(x) \Rightarrow x.\texttt{data} > 0) \wedge (\mathbf{R}c_2(x) \Rightarrow x.\texttt{data} \geq 0)$). DEAL offers no guarantee as to which condition it will check for objects that satisfy both **R**s.
- The formula is evaluated completely: there may be false assertion failures, but if no assertion fails then the formula was true at the point of evaluation. An example would be the complement of our previous "unsound" formula: $\exists x : \mathbf{R}c_1(x) \wedge x.\texttt{data} \leq 0 \vee \mathbf{R}c_2(x) \wedge x.\texttt{data} < 0$.
- The formula is evaluated neither soundly nor completely—e.g., it combines an incomplete and an unsound subformula. This scenario may still be practically valuable, if it turns out to often detect failures while issuing false reports acceptably rarely. Practical automatic bug detectors are often unsound and incomplete—e.g., the authors of ESC/Java have argued forcefully that soundness and completeness are overrated properties compared to practical usefulness in catching bugs [11].

3. Single Traversal Property and Language Implementation

We next state and prove DEAL's single traversal property. This discussion also serves to introduce the way the DEAL compiler processes programs and the changes required to the runtime system so that assertions are evaluated as part of a GC traversal.

3.1 Single Traversal Evaluation

DEAL assertions are guaranteed to be evaluated in a single traversal of live heap objects. Although in the rest of the paper we informally describe this property as "each object is traversed once", this assumes an implicit understanding that an object may be re-examined briefly, only to discover that it was visited earlier. Below we state the desired property more precisely.

Theorem 1. *A valid, safe* DEAL *assertion (i.e., a single* `assertion`*, or* `assertDisjoint` *statement but* not *an* `unsafeAssert` *statement) can be evaluated correctly in a single traversal that crosses each edge in the object graph exactly once and performs a constant amount of work per object. (This constant depends on the size of the assertion.)*

Proof. Consider first the case of an `assertion` statement. According to Figure 1, an asserted formula is a boolean combination of quantified formulas (each starting with $\forall$ or $\exists$), with each quantified formula being a boolean combination of relational expressions, r. Since all quantification is over live objects, it is clear that we only need to evaluate every relational expression r once per object. All relational expressions of a form other than $\mathbf{R}t(x)$ can be evaluated correctly in constant time for any object. Thus, we can evaluate all such expressions (from the entire asserted formula) at whatever time an object is visited. The difficulty lies in proving that we can define a traversal (i.e., a loop over live objects) for which expressions of the form $\mathbf{R}t(x)$ are also correctly evaluated in constant time. If we also show this, then our traversal only needs to keep current truth values for all quantified formulas and at the end of the entire traversal perform their boolean combination for the final result. (An optimization consists of short-circuiting the evaluation of quantified formulas. I.e., if an $\exists$ or $\forall$ formula has already had its truth value determined before the end of the traversal—this can only be to true for $\exists$ and false for $\forall$—there is no reason to evaluate for further objects the relational expressions r in the body of this quantified formula.)

Recall that, per the restrictions of Section 2.1, a valid assertion statement can only contain instances of a single $\mathbf{R}t(x)$ predicate. The truth value of such a predicate can be computed (in constant time) for a traversal visiting every live object once. The full form of the predicate is $\mathbf{R}c_1/\{o_1, \ldots, o_i\} \vee c_2/\{o_{i+1}, \ldots, o_j\} \vee \ldots \vee c_n/\{o_{l+1}, \ldots, o_m\}(x)$. We define the traversal to satisfy the required property as follows:

- Mark objects $o_1, \ldots, o_i$ `ignored`.

  For example, for query $\mathbf{R}a/b \vee c/d, e(x)$ in Figure 2, we mark b as ignored.
- Perform a standard reachability traversal starting at object c_1. The traversal processes objects that are neither `ignored` nor `visited` and marks objects `visited` after processing them. Predicate $\mathbf{R}t(x)$ is true for each processed object, since it is reachable from object c_1 without going through $o_1, \ldots, o_i$. Thus, when processing an object, the traversal can evaluate (for the current object) all relational expressions r in the entire formula.

  For our example in Figure 2, we execute traversal ❶, starting from a, ignoring b.
- Proceed in the same manner for other sources and excluded objects. That is, mark objects $o_{i+1}, \ldots, o_j$ `ignored`, perform a standard reachability traversal starting at object c_2, etc. In total there are n such steps (one for each object $c_1, \ldots, c_n$).

  In our running example (Figure 2) we now ignore objects d and e and then perform traversal ❷.
- Remove the `ignored` flag from all objects $o_1, \ldots, o_m$.

  In our running example we now remove this flag from b, d, and e.

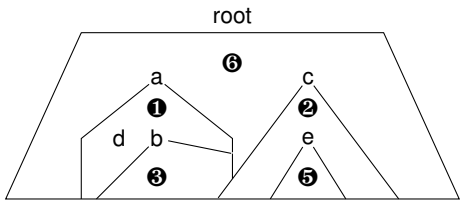

Figure 2. Traversals for $\mathbf{R}a/b \vee c/d, e(x)$, in order.

- Perform m reachability traversals, each starting at object o_k for $1 \leq k \leq m$, followed by a reachability traversal starting at the root set (i.e., the default live objects of the program's execution, such as the stack and the global region). Every traversal processes all non-`visited` objects and marks objects `visited` after processing them. During each of these $m + 1$ traversals, predicate $\mathbf{R}t(x)$ is false for each processed object, by nature of the traversal: the object was not reachable by any c_i without going through the corresponding excluded objects $o_1, \ldots, o_m$. When processing an object, the traversal can evaluate (for the current object) all relational expressions r in the entire formula.

  In our running example, we first run traversals ❸ through ❺, though ❹ would be a no-op (since we already visited d in ❶ before ignoring it). Then we conclude with the root traversal ❻.

Note that no edge in the object graph (i.e., field of an object) is processed twice: the standard reachability traversal follows edges out of an object exactly once, after marking the object `visited`, and the only further operation on a `visited` object is to check whether it is `visited`. We invert the meaning (i.e., mapping to a 0 or 1 bit) of `visited` after every collection. (This is a standard technique in Mark/Sweep garbage collection, avoiding a second visit of nodes just to reset them.) After collection no object is `ignored` (since no object is ignored initially.)

The above traversal satisfies all requirements of the proof. It also requires space of two bits per object (one for the `visited` flag and one for `ignored`).[3] From the perspective of the `visited` flag, the above traversal is a full replacement of a GC reachability traversal with only the object visiting order modified.

The handling of statement `assertDisjoint` is quite similar. The difference is that, in addition to the `visited` and `ignored` flags we maintain per object, we need $\lceil \log(u) \rceil$ bits per object, where u is the number of distinct predicates $\mathbf{R}t(x)$. These bits form a `traversalNum` counter. The traversal proceeds by repeating all but the last step of the `assertion` traversal for every $\mathbf{R}t(x)$ predicate, with some small differences. First, while marking an object `visited` after processing, the traversal sets `traversalNum` to the current traversal number, which is initially 0 and incremented for every $\mathbf{R}t(x)$ predicate examined. Second, for each previously visited object encountered, the traversal checks whether the visited object's `traversalNum` is equal to the current traversal number (indicating a previously visited object during the current traversal) or lower (indicating a disjointness error and causing an immediate assertion failure, unless the object is also `ignored`). The final traversal (from the root set and from excluded objects) uses a traversal number of 0, thereby avoiding disjointedness errors.

The above `assertDisjoint` traversal maintains the properties required for the proof. First, the truth value of every **R** predicate is evident at the time of visit of each object, since the disjointness property guarantees that the only predicate **R** that is true for the object is the **R** predicate whose traversal reaches the object. Second, the `visited` flag ensures no edge is crossed more than once, by the same argument as for the `assertion` traversal. □

Although the above proof is not hard, this is largely due to the careful design of the DEAL language, which maintains the properties required for a single traversal evaluation of assertions. We note that it is quite easy to be misled with respect to predicates that can be evaluated in a single traversal. For instance, a common mistake is to attempt to integrate the inverse of predicate $\mathbf{R}c(x)$ in a single traversal language: It is not possible to add to DEAL a predicate $\mathbf{P}c(x)$ with the meaning "x can reach c" while maintaining the same single traversal property for arbitrary graphs.

3.2 DEAL Implementation

The proof of the single traversal theorem essentially lays out the principles of the DEAL implementation. DEAL instructs a garbage collector to perform the traversals described in the proof for `assertion` and `assertDisjoint`. Statement `unsafeAssert` is handled identically to `assertion` with the only difference being that all but the last step of the `assertion` traversal are repeated for every $\mathbf{R}t(x)$ predicate in the formula. Therefore, `unsafeAssert` visits every object exactly once but without guaranteeing correct evaluation of the assertion, since each object is visited only as part of a traversal corresponding to a single **R** predicate.

DEAL is implemented on top of the Jikes RVM [2], modifying the Jikes MarkSweep garbage collector. The bulk of the implementation consists of changes to the runtime system, and especially interfacing with the collector. In order to produce a clean and general interface, we introduced a small "trace-and-test" language inside the VM. A trace-and-test program instructs the garbage collector what to mark recursively, what to test for, and what individual objects to expressly exclude or include in the traversal. The trace-and-test language has primitives such as `exclude-node(o)` (to exclude an object from the traversal), `include-node(o)`

[3] This requirement can be reduced to just one bit per object, by using the `visited` bit to also mean `ignored`, plus a second bit only for each of the m ignored objects. The reason for the second bit is that, during the final traversal from the root set, it is necessary to remember whether an ignored object has been visited or not.

(to unmark an object, so that it is processed when next encountered), `mark-traversal-id(x)` (to increase the traversal number), `set-expressions`$(k_1, \ldots, k_n)$ (to set up the sets of relational expressions that need to be evaluated for every node, together with information, per set, on whether the expressions are in an existential or universal context), and `trace(o)` (to start a reachability traversal at object `o`).

The DeAL compiler translates an assertion into trace-and-test language instructions, following closely the traversal structure described in the proof of Section 3.1. The relational expressions that need to be evaluated per-object are translated into plain Java methods that the runtime compiles with the maximum optimization level allowed by the Jikes just-in-time compiler. We altered the garbage collection logic in the virtual machine to execute the trace-and-set instructions. For marking nodes, we employ 4 bits per object that are currently unused by the runtime, resulting in a maximum of 8 **R** predicates per `assertDisjoint` formula (since one bit is occupied by the `ignored` flag); in addition we obtain the `visited` flag from a region in the object header reserved for the existing MarkSweep garbage collector. In the future, the implementation could allocate extra space per object on a side structure, allowing unlimited **R** predicates.

4. Discussion: Expressiveness Limitations, Programming Patterns

DeAL is an expressive language but has by design some expressiveness limitations, in order to support its single traversal guarantee. Clearly, DeAL cannot express computations that are inherently non-linear, or non-single-traversal. Furthermore, DeAL occasionally cannot support computations that happen to be computable in a single traversal but their structure obscures this fact. For instance, consider the property "all `Node` objects directly referenced by array `arr` are dominated by `arr`". This is a property that can be evaluated in a single traversal, yet it is not expressible in DeAL. One can attempt to write the property straightforwardly as:

$$\texttt{assertDisjoint}(\forall x \in \texttt{Node} : \mathbf{R}\texttt{arr}(x) \Rightarrow \mathbf{R}/\texttt{arr}(x));$$

Yet this is not the desired property: the `Node` objects directly referenced by `arr` are only a subset of all the objects reachable by `arr`. For instance, the `Node` objects referenced by `arr` may themselves refer to other objects that are not dominated by `arr`, causing a spurious disjointness error. Intuitively, the desired property should be expressible in DeAL but is not because of a technicality: Although the property is local (it requires traversing just one link to tell that an object is directly referenced by `arr`), it is not local from the perspective of the referenced object: when visiting a random object it is not possible to know in constant time whether it is directly referenced by `arr`.

These (as well as many other) expressiveness limitations of DeAL can be overcome by combining Java computation with DeAL assertion checking. The general programming pattern consists of having Java code compute a property of objects, summarize the results as a local property (e.g., by marking a field for all objects satisfying the desired property) and invoke a DeAL assertion that checks whether the marked objects have the expected structure.

In our above example, if we add an otherwise unused field `mark` in class `Node`, we can easily express the desired assertion by combining Java and DeAL evaluation:

```
Object dummy = new Object();
for (Node n : arr) n.mark = dummy;
```
$$\texttt{assertion}(\forall x \in \texttt{Node} : (x.\texttt{mark} = \texttt{dummy}) \Rightarrow \mathbf{R}/\texttt{arr}(x));$$

This programming pattern is quite powerful. It allows nearly arbitrary extension of the properties that can be checked, at the expense of performing some of the work outside the DeAL traversal. Given the generality of the pattern, we are considering adding language support for it in future versions of DeAL. Namely, we can add a `labelObject` primitive to DeAL and allow the Java program to mark objects with a finite set of labels, at any point in the execution. Then, when a DeAL assertion is checked it can refer to whether an object has a certain label or not (i.e., labels are unary predicates). The markings will be cleared after evaluation of the assertion. This simple addition to the language is just a convenience feature that obviates the need for extra fields, such as `mark` in our earlier example. The markings can be kept on a side structure that the DeAL traversal consults. We have not added such convenience features in our first version of DeAL, preferring to concentrate on the core language instead of on elements that can be easily emulated.

5. Experiments

We evaluate the implementation of DeAL with several experiments:

- We evaluate the assertions of Section 2.2 in a full heap of objects with no other computation performed in the application. This is a controlled microbenchmark for the given assertions, intended to check how much the overhead varies with the complexity of typical assertions and with the percentage of relevant objects.
- We run a wide spectrum of actual applications (including the DaCapo Benchmarks, SPEC JVM98, and pseudojbb—a version of SPEC JBB2000). We measure the overhead of simple (generic) DeAL assertion checking, when assertions are evaluated with the same frequency as that of normal garbage collection in the course of the application. This is an end-to-end experiment establishing cost under conditions where the user intends to achieve near-zero-overhead execution.

- We analyze in more depth the overhead of a custom assertion that reveals a (known) bug in pseudojbb. This is a close approximation of a "real use" scenario.

All experiments confirm that the overhead of DEAL is small and becomes negligible for assertion evaluation at roughly normal GC frequencies.

5.1 Methodology and Setup

Our system is implemented on top of Jikes RVM 3.1.0, so we compare our results to those of the unmodified Jikes RVM 3.1.0, both using the full-heap MarkSweep collector.

We use the DaCapo benchmarks versions 2006-10-MR2 and 9.12-bach. For SPEC JVM98, we use the large input size (-s100); for DaCapo 2006, DaCapo 9.12, and pseudojbb, we use the default input size. Many of the Dacapo 9.12 benchmarks do not run on the unmodified Jikes RVM 3.1.0, so we provide results from only those that do. We perform our experiments on a Core 2 Duo E6750 machine with 2 GB of RAM, running Ubuntu Linux 9.04.

We use the adaptive configuration of Jikes RVM, which dynamically identifies frequently executed methods and recompiles them at higher optimization levels. We iterate each benchmark k times and record the results from the last iteration. For microbenchmarks (Section 5.2) $k = 10$, and for larger programs $k = 4$. We repeat this twenty times for each benchmark. All numbers reported are means of the 20 runs, and our graphs also include 90% confidence intervals. We execute each benchmark with a heap size fixed at two times the minimum possible for that benchmark using the MarkSweep collector. We vary the UNIX environment size to avoid measurement bias [16].

5.2 Controlled Microbenchmarks

The overhead of DEAL evaluation depends strongly on the complexity of assertions. For instance, if the user asserts a property that is a function of the values of k neighbors of every live object, then the runtime cost will necessarily be $\Omega(k)$. Therefore, the interesting question concerns the overhead for typical assertions. We used as our evaluation set the example assertions of Section 2.2 (which mostly emulate assertions from the recent research literature [7–9, 13, 18, 19, 21]). The assertions are applied to specially designed skeletal programs, with appropriate data structures for each assertion.

The intention of these microbenchmarks is to show what can be expected as a worst-case execution time for representative assertions. These programs *do nothing aside from building the data structure and running an assertion on it*. We used three different input sizes, determining the number of objects on the heap (counting only objects with types relevant to the assertion): small (10,000 objects), medium (100,000 objects), and large (1 million objects).[4] The comparison is to an unmodified Jikes RVM that runs a regular `System.gc()` every time DEAL would check an assertion. Figure 3 shows the GC overhead results for all three input sizes.

The large input size expectedly incurs the most overhead. The geometric mean of the GC overhead is 1.25. That is, assertion checking slows down garbage collection by just 25%. The absolute worst-case GC slowdowns rise to 60-70%. Generally, assertions that access fields perform worse than those that just check disjointedness or reachability properties. Note that the object type provided in the assertion does not serve as an efficient filter, since the program is specially designed to only create objects matching the type examined in the assertion. Thus, the fields of virtually every program-generated object need to be examined. (This is unlikely to be the case in a realistic setting: object types will be used to disqualify most objects without needing to examine their fields.) The three worst performers are assertions "cyclic", "doubly_linked_list", and "sorted", reproduced here:

$$\texttt{assertion}(\forall x \in \texttt{Node} : \mathbf{R}\texttt{c}(x) \Rightarrow x.\texttt{next} \neq \texttt{null});$$

$$\texttt{assertion}(\forall x \in \texttt{Node} : \mathbf{R}d(x) \Rightarrow (x.\texttt{next.prev} = x));$$

$$\texttt{assertion}(\forall x \in \texttt{Node} : \mathbf{R}\texttt{c}(x) \Rightarrow x.\texttt{data} \leq x.\texttt{next.data});$$

As can be seen from Figure 3, even the worst case slowdown is quite low, considering that GC time is a small part of total program execution time. Indeed, even though these skeletal programs do very little other than assertion checking, the total execution time overhead (not shown in Figure 3) has a geometric mean of 1.07 for the small configuration, 1.08 for the medium, and 1.13 for the large—i.e., assertion checking causes slowdowns of just 7, 8, and 13%, respectively.

5.3 End-to-end Simple Assertion Cost

We ran a large suite of end-to-end benchmarks to quantify the overall overhead of our system. Our suite of applications includes the DaCapo benchmarks [5], SPEC JVM98 [23], and a fixed-workload version of SPEC JBB2000 called pseudojbb [24]. Clearly, the challenge of evaluating overheads over a wide range of applications is that real assertions are application-specific. To overcome this difficulty, we added a "generic" assertion in our runtime that runs at the same frequency as normally-triggered GC. This assertion checks that there are no objects on the heap that are an instance of a certain unused class. Thus, the assertion should always return true, but it will cause an `instanceof` predicate to be tested on every live object. In addition, we allow this dummy assertion to be combined with a traversal disjointness test, which incurs the cost of also writing and checking a traversal ID in the header of every live object.

[4] The choice of sizes follows the sizes of the DaCapo benchmarks. Across all DaCapo benchmarks, the geometric mean of the maximum number of live objects is roughly 100,000. The minimum is about 3,000 and the maximum is 3.2 million [5]. Therefore, our three setups have roughly the same spread of sizes as the DaCapo benchmarks.

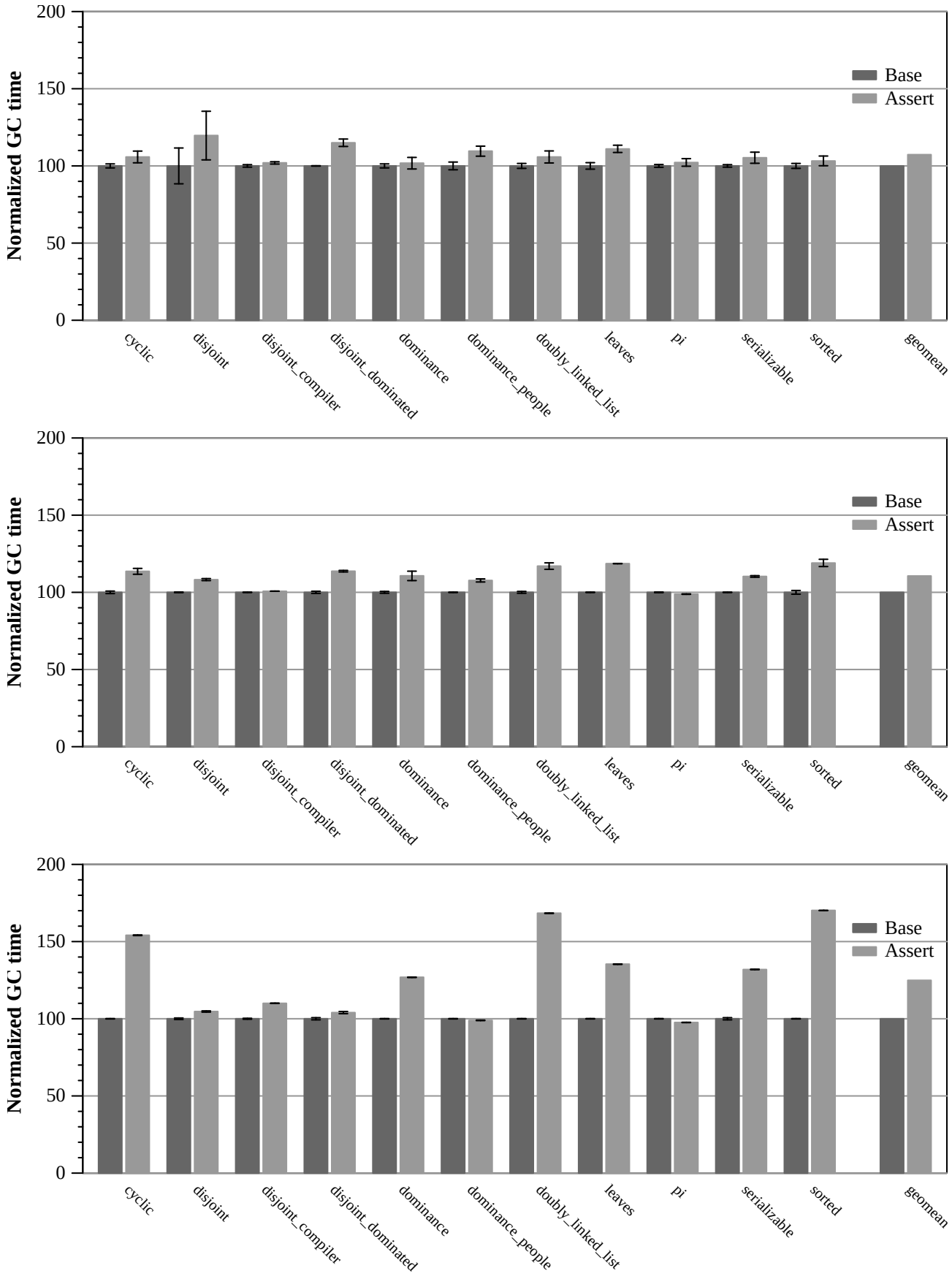

Figure 3. GC time overhead for representative assertions and skeletal programs, for small (top), medium (middle), and large (bottom) input sizes.

Figure 4 shows the overhead of DeAL on GC time for the benchmark applications. The figure shows three configurations: Base (unmodified Jikes RVM), Assert (`instanceof` assertion), and AssertDisjoint (`instanceof` assertion + disjointness check).

Overall, the geometric mean of the slowdown in GC time for Assert is 9.10% and for AssertDisjoint is 11.96%. These overheads are quite low and become *vanishingly small when computed as a percentage of total execution time, instead of just GC time*: the geometric mean of the total execution slowdown for both kinds of assertions is *below 1.9%*. (The mean number of garbage collections for these benchmarks is 20.3, with a standard deviation of 25.9.)

5.4 Representative Example

To evaluate the overhead of a real use of DeAL, we simulated, with pessimistic assumptions, the process by which DeAL would be used to detect a bug in practice. We picked the most complex of the bugs previously identified in pseudojbb using heap assertions [1]. The bug consists of removing `Order` objects from the `orderTable` data structure. These orders should then become unreachable and be garbage collected. However, the objects are also erroneously referenced from `Customer` objects, causing a memory leak. (The bug fix consists of unlinking `Order` objects from `Customers` when they are removed from the `orderTable`.) Our assertion checks whether the objects removed from `orderTable` can be reached. This can be effected with a simple variant of the programming pattern of Section 4. Namely, once an object is removed from the `orderTable` we mark it by assigning a special value to one of its fields. The DeAL assertion then becomes just $\forall x \in \texttt{Order} : x.\texttt{field} \neq \texttt{specialValue}$. Since the $\forall$ iteration is over live objects and we expect our removed objects to be unreachable, no live object should have the special value.

The bug is hard to detect because `Order` objects are removed from the `orderTable` only at rare program instances. Specifically, the `orderTable` is occasionally completely emptied and then destroyed. Since the programmer might not know how often this happens, we measured the cost of employing DeAL at high frequency: the assertion is evaluated once every 5 times it is encountered. The high frequency of assertion checking causes much more frequent GCs than normal program execution without DeAL: the mean of the number of GCs (over 20 executions) rises from 2.85 to 12.7, and the total GC cost is 3.37 times higher for the version employing DeAL. However, even this high GC overhead is hardly noticeable in terms of total program execution time. The slowdown is merely 6.1%: the assertion-checking program runs in 3.87s vs. 3.65s for the original program running on an unmodified VM.

6. Related work

Our work is related to a wide range of techniques, both static and dynamic, for checking properties of data structures.

Research in static analysis has yielded a significant body of sophisticated techniques for modeling the heap and analyzing data structures at compile-time. The strength of static analysis is that it explores all possible paths through the program: a sound analysis algorithm can prove the absence of errors, or even verify the full correctness of a data structure implementation. The weakness of static analysis is that many properties, particular those involving pointers and heap-based structures, are undecidable in principle [17], and extremely difficult to approximate precisely in practice [6, 14, 20, 26]. These problems are particular severe for languages such as Java, which include many hard-to-analyze features such as reflection, bytecode rewriting, and dynamic class loading.

Our work is most closely related to several recent techniques for dynamically analyzing heap data structures, including two that check properties at garbage collection time. GC Assertions provides a set of heap assertions that, like DeAL, are checked by the garbage collector at runtime [1]. This work differs from DeAL in two critical ways. First, it does not provide a full language for heap properties, but rather a fixed set of assertions that cannot be composed. Second, it does not check assertions at the time they are encountered, but instead defers checking until the next regularly scheduled garbage collection. While this choice improves performance, it limits the kinds of heap properties to those that can be checked at any time, since they might, e.g., be deferred to a point at which they are knowingly but temporarily invalidated.

The QVM system provides a richer set of heap checks, called *heap probes*, which QVM performs using garbage collector machinery at the point they are requested [4]. These properties, however, can require multiple heap traversals to check, significantly impacting performance. The authors explore both sampling (at a user chosen rate) and parallelization to control the cost. We believe that DeAL represents a sweet spot between GC Assertions and QVM: unlike QVM, DeAL controls overhead by guaranteeing that all properties can be checked in a single heap traversal, and by only checking these properties at or near a regularly scheduled collection; but unlike GC Assertions, DeAL offers a full assertion language, and a clearer semantics regarding the timing of checks.

The Ditto system reduces the cost of data structure checking by incrementalizing the checking code itself [21]. This system provides a significantly different tradeoff than DeAL: it supports arbitrary checks written procedurally in Java itself, but even with incrementalization incurs a significant overhead (in the range of 3X to 10X slowdown).

A number of systems developed for data structure *repair* include a method for detecting data structure errors. Al-

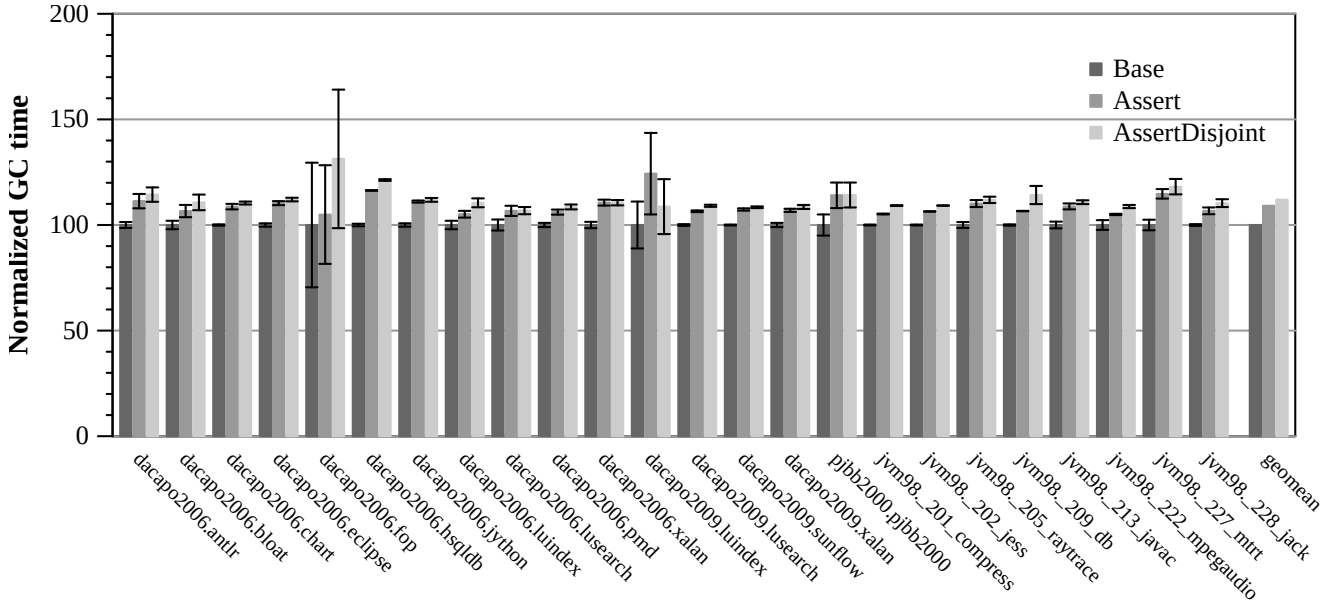

Figure 4. GC time overhead of our system for end-to-end benchmarks.

though these checking subsystems are not the focus of the work, they highlight the difficulty of detecting such errors. The STARC system uses programmer-written `repOk` methods to detect errors, which incur a very substantial overhead (30X) when run frequently, and even a 20% overhead when run only at the point of failure [10]. Demsky et al. present a technique based on Daikon to detect heap invariants and repair broken data structures [7]. While the checking overhead is difficult to evaluate (since it is presented in milliseconds), the paper describes it as "a rather large overhead".

7. Conclusions and Future Work

We presented the DEAL language for assertions that are evaluated in a single traversal of live objects, implemented as a variation of a garbage collection traversal. DEAL is a rich language that can express many useful properties as logical combinations of reachability conditions and local comparisons. At the same time, the overhead of DEAL on a program's total execution time remains very low—typically well below the 5% level.

Although the current language forms a nicely closed set of features, there are clear directions of interesting future extensions. One possibility is to extend DEAL from an assertion language to a general query language that can produce new data structures. The typical way to define a declarative query language is by allowing unbound logical variables. For instance, one can imagine a `query` primitive, used with first-order logic formulas. The matching values for unbound variables become the tuples returned as the result of the query. For example, one might produce a new Java set containing all positive elements of a list by writing:

```
Set posval =
    query(x.val, (x ∈ Node ∧ Rd(x)) ⇒ (x.val > 0));
```

The first part, `x.val`, of the above query determines the form of the result as a projection of the matched values, while the second contains the formula defining the desired x values.

Such a query language is easy to define and can perform useful computation during garbage collection. The challenge, however, is that querying to produce results that are subsequently used in the Java program means that the query evaluation can no longer be skipped. Thus, garbage collection overhead may need to be incurred quite often, making the approach inefficient. A specific danger is that the user will employ the query mechanism for properties that can be evaluated much more efficiently without a traversal of all live objects. Therefore, adding querying capabilities to a GC traversal requires careful thought. Either somehow the queries need to be limited to expensive "whole-heap" properties, or the querying needs to be combined with some mechanism for speculative execution, so that the Java program can proceed correctly regardless of whether or not the query was evaluated.

Another interesting possibility is that of allowing arbitrary Java methods in assertions, as long as the methods always terminate and do not have externally observable side-effects. This property is hard to enforce statically (through language design or static analysis) but can be enforced dynamically via resource limits (e.g., sandboxing and timeouts). Such methods will be invoked on every visited object during a GC traversal and can use the values of DEAL's

R predicates. At an extreme, one can consider a variant of DEAL where all logical operations (i.e., ∃, ∀, boolean connectives) are not part of the language. Instead, the language just offers the ability to call Java methods and have Java methods reference **R** predicates, in a co-routine fashion. In this way, the main ideas of DEAL can be reduced to a modified GC traversal that guarantees every object is visited exactly once and at visit time the value of reachability predicates is fully determined. This is an alternative design that maximizes low-level control and expressiveness of the language for local computations, yet loses the disciplined elegance of the declarative DEAL design.

In all, we consider DEAL to be an excellent ambassador of several interesting ideas and directions that are likely to be important in future research. These include, of course, DEAL's hallmark property of single-traversal computations, but also the smooth integration of declarative, logic-based, resource bounded languages in a general-purpose programming language. We believe that the DEAL design will be multiply influential in such endeavours.

References

[1] E. E. Aftandilian and S. Z. Guyer. Gc assertions: using the garbage collector to check heap properties. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 235–244, New York, NY, USA, 2009. ACM.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalape no jvm. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–65, New York, NY, USA, 2000. ACM.

[3] M. Arnold, M. Vechev, and E. Yahav. Qvm: an efficient runtime for detecting defects in deployed systems. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 143–162, New York, NY, USA, 2008. ACM.

[4] M. Arnold, M. Vechev, and E. Yahav. QVM: an efficient runtime for detecting defects in deployed systems. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 143–162, 2008.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, Oct. 2006. ACM Press.

[6] P. T. Darga and C. Boyapati. Efficient software model checking of data structure properties. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 363–382, 2006.

[7] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard. Inference and enforcement of data structure consistency specifications. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 233–244, New York, NY, USA, 2006. ACM.

[8] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid. Assertion-based repair of complex data structures. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 64–73, New York, NY, USA, 2007. ACM.

[9] B. Elkarablieh, S. Khurshid, D. Vu, and K. S. McKinley. STARC: Static analysis for efficient repair of complex data. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 387–404, 2007.

[10] B. Elkarablieh, S. Khurshid, D. Vu, and K. S. McKinley. STARC: Static analysis for efficient repair of complex data. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 387–404, 2007.

[11] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245. ACM, June 2002.

[12] N. Immerman. *Descriptive Complexity*. Springer, 1998.

[13] D. Jackson. Object models as heap invariants. *Programming methodology*, pages 247–268, 2003.

[14] S. McPeak and G. Necula. Data structure specifications via local equality axioms. In *Computer Aided Verification*, pages 476–490, 2005.

[15] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 2nd edition, 1997.

[16] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 265–276, New York, NY, USA, 2009. ACM.

[17] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, 1994.

[18] D. Rayside and L. Mendel. Object ownership profiling: a technique for finding and fixing memory leaks. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 194–203, New York, NY, USA, 2007. ACM.

[19] D. Reimer, E. Schonberg, K. Srinivas, H. Srinivasan, J. Dolby, A. Kershenbaum, and L. Koved. Validating structural properties of nested objects. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 294–304, New York, NY, USA, 2004. ACM.

[20] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *ACM Symposium on the Principles of Programming Languages*, pages 105–118, 1999.

[21] A. Shankar and R. Bodík. Ditto: automatic incrementalization of data structure invariant checks (in java). In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 310–319, New York, NY, USA, 2007. ACM.

[22] A. Shankar and R. Bodík. Ditto: automatic incrementalization of data structure invariant checks (in java). In *ACM Conference on Programming Languages Design and Implementation*, pages 310–319, 2007.

[23] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, 1999.

[24] Standard Performance Evaluation Corporation. *SPECjbb2000 Documentation*, release 1.01 edition, 2001.

[25] P. R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *ACM International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, France, Sept. 1992. Springer-Verlag.

[26] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *ACM Conference on Programming Languages Design and Implementation*, pages 349–361, 2008.