LeakChaser: Helping Programmers Narrow Down Causes of Memory Leaks

Guoqing Xu Michael D. Bond Feng Qin Atanas Rountev

Department of Computer Science and Engineering
Ohio State University
{xug,mikebond,qin,rountev}@cse.ohio-state.edu

Abstract

In large programs written in managed languages such as Java and C#, holding unnecessary references often results in memory leaks and bloat, degrading significantly their run-time performance and scalability. Despite the existence of many leak detectors for such languages, these detectors often target low-level objects; as a result, their reports contain many false warnings and lack sufficient semantic information to help diagnose problems. This paper introduces a specification-based technique called LeakChaser that can not only capture precisely the unnecessary references leading to leaks, but also explain, with high-level semantics, why these references become unnecessary.

At the heart of LeakChaser is a three-tier approach that uses varying levels of abstraction to assist programmers with different skill levels and code familiarity to find leaks. At the highest tier of the approach, the programmer only needs to specify the boundaries of coarse-grained activities, referred to as transactions. The tool automatically infers liveness properties of these transactions, by monitoring the execution, in order to find unnecessary references. Diagnosis at this tier can be performed by any programmer after inspecting the APIs and basic modules of a program, without understanding of the detailed implementation of these APIs. At the middle tier, the programmer can introduce application-specific semantic information by specifying properties for the transactions. At the lowest tier of the approach is a liveness checker that does not rely on higher-level semantic information, but rather allows a programmer to assert lifetime relationships for pairs of objects. This task could only be performed by skillful programmers who have a clear understanding of data structures and algorithms in the program.

We have implemented LeakChaser in Jikes RVM and used it to help us diagnose several real-world leaks. The implementation incurs a reasonable overhead for debugging and tuning. Our case studies indicate that the implementation is powerful in guiding programmers with varying code familiarity to find the root causes of several memory leaks—even someone who had not studied a leaking program can quickly find the cause after using LeakChaser's iterative process that infers and checks properties with different levels of semantic information.

PLDI'11, June 4–8, 2011, San Jose, California, USA.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management, optimization, runtime environments; F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages—Program analysis; D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids

General Terms Language, Measurements, Performance

Keywords LeakChaser, memory leak detection, object lifetime assertions

1. Introduction

A memory leak in a managed language such as Java or C# occurs when references to some unused objects are unnecessarily held so that these objects cannot be garbage collected. Memory leaks crash programs when they exhaust the heap, and can frequently cause performance issues due to increased garbage collection (GC) runs and execution redundancies.

Problem: lack of semantics in leak detection Memory leak detection in Java software has recently gained much interest in the programming language and systems communities [8, 9, 10, 18, 20, 23, 29, 33, 36]. All existing detectors except one [36] track behaviors of arbitrary objects and report problems when tracked objects become *suspicious*. One major category of work [8, 9, 10, 18, 33] considers objects' staleness (i.e., time elapsed since the program last used these objects) as an indicator of problematic behavior, while another category [20, 23] treats objects as suspicious if instances of their types exhibit sustained growth. While both heuristics are reasonable in many cases, they are not *definitive evidence* of leaking objects because many normal objects can also exhibit such behaviors. For example, growing numbers of instances are not leaks if they are reclaimed later. And some GUI widgets may never be used after they are initialized, i.e., they become very stale—but they are not necessarily leaks. In addition, these leak detectors attempt to find root causes by starting from the suspicious objects (i.e., the leak symptom) and traversing the object graph. Due to the complexity of the graph, this attempt is often a heuristics-based process that ends up reporting a sea of likely problems with the true causes being buried among them. Because a memory leak often occurs due to the inappropriate handling of certain events and these tools profile the whole program execution without any focus, there is little hope that a completely automated tool can precisely pinpoint the problematic area(s) for large-scale Java applications. While prior work avoids heuristics by focusing on containers [36], the root causes of many memory leaks are not containers, but instead cached references that the programmer forgot to invalidate, as represented by the leak cases in SPECjbb2000 and Eclipse bug #115789 (Section 5).

For a more focused leak detector, it is necessary to take advantage of human insights and use programmer specifications to guide

leak detection. While prior work proposes heap assertions that can be checked during GC [3, 5, 30, 35], these low-level assertions may only be employed by programmers who have deep understanding of an application (e.g., algorithms, data structures, etc.). This requirement limits significantly their usefulness in leak detection for existing Java applications (i.e., because no prior assertions were written during development), as very few performance experts have sufficient program knowledge to use these assertions during post-mortem tuning.

Insight Almost all memory leaks we have studied are in regularly occurring program events: unnecessary references can quickly accumulate and cause the memory footprint to grow if each such event forgets to clean up a small number of references.

The techniques proposed in this paper are based on the following two observations about these frequently occurring code regions. First, in these regions there usually exist implicit invariants among lifetimes of objects (e.g., "objects a and b must die together," or "the old configuration object must die before the new configuration object is created"). When such invariants are violated, memory leaks result. It is often not easy to use reachability-based heap assertions from prior work to express these lifetime relationships. Second, for each such region, there often exist objects that are strongly correlated with the liveness of the entire region. For example, consider a (web or database) transaction in an enterprise Java application. A typical lifecycle consists of a sequence of events such as the creation of the transaction object, the creation of all other objects used in this transaction, the deallocation of these (other) objects, and the deallocation of the transaction object. For this lifecycle, the transaction object is the first one that is created and the last one that is reclaimed, and it is thus the object that controls the liveness of this entire transaction region.

Our proposal We propose a three-tier approach that exploits these insights to help a programmer quickly identify unnecessary references leading to a memory leak. This approach can be used by both novices and experts to diagnose memory problems. The key idea is to introduce high-level semantics by explicitly considering coarse-grained events where leaks are observed. We refer to these events as *transactions*. As extensions of—and inspired by—enterprise transaction models (e.g., EJB transactions), our transactions describe frequently executed code regions with user-defined boundaries. Objects associated with a transaction fall into three categories: (1) transaction identifier object, (2) transaction-local objects, and (3) objects shared among transactions.

The identifier object of a transaction controls the lifetime of the transaction: all transaction-local objects should be created after this object is created and should die before it dies. Only shared objects are allowed to live after the identifier object dies. Next, we introduce the three tiers of our approach in descending order of their levels of abstraction, which is often the order in which a programmer uses our tool to solve a real-world problem.

Tier H (the high-level approach): At this highest level of the approach, our framework attempts to automatically *infer* transaction-local and shared objects from the execution, while the programmer only needs to specify transaction boundaries and identifier objects. The tool starts to work in the inference mode, and once objects shared among transactions are identified, it switches to check whether the (inferred) shared objects can actually be used in other transactions. Violations are reported if these shared objects are not used for a certain period of time. The programmer's task at this level is the easiest to perform: we found that even when we had not studied a program before, we could quickly identify these (coarse-grained) events in order to perform the diagnosis.

Tier M (the medium-level approach): In this tier, the user needs to specify not only the boundary of a transaction and the transaction identifier object, but also the objects shared across transactions, which the user does not specify in tier H.

LeakChaser *checks* the given specifications, and reports violations if the specified transaction-local objects in one instance of the transaction escape to another instance. While employing this tier of the tool requires the user to have a deeper understanding of the program and the likely cause of the problem, it can generate a more precise report.

Tier L (the low-level approach): This lowest level is essentially an assertion framework that allows the user to specify lifetime invariants for focused memory leak detection. The framework contains binary assertions that directly express object lifetime relationships. For instance, one important assertion is to specify that object a must die before object b. This assertion fails only when LeakChaser observes definitive evidence, e.g., b is dead while a is still live.

Compared to reachability-based assertions [3, 5, 30, 35], our framework has three advantages. First, passing/failing of our assertions does not depend on where GC runs are triggered and thus, assertion checking does not produce false positives. Second, program locations where our assertions are placed have no influence on the evaluation of these assertions. For example, we can assert that object a dies before object b immediately after they are created, while a reachability-based assertion such as *assertDead* in prior work has to be placed in a location where the asserted object is about to become unreachable [3]. Third, our framework can be used to assert arbitrary objects whose lifetimes are correlated due to some high-level semantics (e.g., events), while a reachability-based assertion can work only on objects that have low-level structural relationships (e.g., reachability in the object graph).

Of course, using this assertion framework requires one to understand considerable design and implementation details of the program such as how a data structure is constructed. However, performing this level of diagnosis can give the user a very precise report. Hence, these assertions can be added by developers during coding, which may ease significantly the diagnosis of memory problems when performance degradation is observed. In fact, the transaction-based properties described in the other two tiers are translated by our framework into these low-level assertions at run time. Section 3 presents details about this translation.

As the level of abstraction decreases (from tier H to tier L), the diagnosis becomes more focused. Figure 1 illustrates the process of memory leak detection using this framework. In our experience, LeakChaser is especially useful to a performance expert who is unfamiliar with the program code, since he or she can follow an iterative process that involves all these levels of diagnosis (i.e., from tier H down to tier L). The programmer starts with the tier H analysis with little program knowledge and insight. By repeating a higher-level analysis a few times (with refined specifications) and inspecting its reports, the programmer gains a deeper understanding of the program as well as more insight into the problem, and then will be able to move on to a lower-level analysis for a more focused diagnosis. Very often, this process ends up narrowing down the information to the exact cause of the leak.

For large-scale applications, LeakChaser allows programmers to specify transactions at *clients* of these applications, without digging into application implementation details. For example, to diagnose problems in a large database system, the programmer only needs to create transactions at client programs that perform database queries. We found that this feature of the tool is quite useful in simplifying the diagnostic task: all previous techniques require a programmer to understand a fair amount of low-level details of the system before she can start the diagnosis. This burden is reduced significantly by LeakChaser.

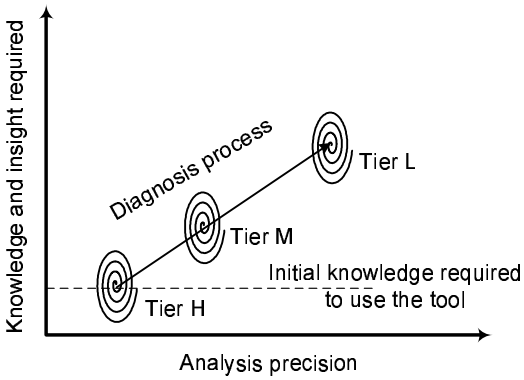

Figure 1. Illustration of the diagnosis process. Spirals at each level indicate that a user may need to run each tier multiple times with refined specifications (e.g., smaller transaction regions, more precise shared object specifications, etc.) to gain sufficient knowledge to proceed to a lower tier.

Implementation and experiments We implemented our approach in Jikes RVM 3.1.0 (`http://jikesrvm.org`), a high-performance Java-in-Java virtual machine, and successfully applied it to real leaks in large-scale applications such as Eclipse and MySQL. The implementation techniques are discussed in detail in Section 4. Section 6 evaluates analysis expenses. We add one extra word in the header of each run-time object and this space is used to store the assertion information for the object. The overall space overhead of the tool is less than 10%, including both extra header space and memory used to store the metadata of our analysis. Using the optimizing compiler and the Immix garbage collector [6], the current implementation imposes an average slowdown of 2.3× for GC only and 1.1× for overall executions for the framework infrastructure (i.e., no assertions added). Additional overhead is incurred for checking and inferring specifications. For example, after adding assertions to SPECjbb2000, 256,236 assertions were executed, and overall slowdown was 5.5×. While the overhead is probably high for production runs, we found it acceptable for performance tuning and memory leak diagnosis.

Section 5 presents six case studies on real-world memory leak problems. Among these problems, four are true leaks and for the remaining two, programmers experienced high memory footprints but were not sure whether or not there were leaks. Using our tool, we have quickly identified root causes for the true leaks, and found reasons that could explain the high memory consumption for the other two cases. In SPECjbb2000, in addition to the already-known leak, we found memory issues that have not been reported previously and actually cause more severe performance degradation than the already-known leak.

We observed significant performance improvements after fixing these problems. The experimental results strongly indicate that the proposed three-tier diagnosis methodology can be adopted in real-world development and tuning to find and prevent memory leaks, and LeakChaser is useful in helping a programmer quickly identify unnecessary references that lead to leaks and other memory issues.

The contributions of this work are:

- A three-layer methodology that introduces different levels of abstraction to help both experts and novices understand and diagnose memory leak problems.
- A new heap assertion framework that allows programmers to assert object liveness properties instead of using reachability to approximate liveness.
- An implementation of LeakChaser in Jikes RVM that piggybacks on garbage collection to check assertions.
- Six case studies demonstrating that LeakChaser can help a programmer unfamiliar with the program source code to quickly find root causes of memory leaks.

2. Overview

We illustrate our technique using a simplified version of a real-world memory leak (Eclipse bug #115789). Figure 2(a) shows the code that contains the leak. This bug can be easily reproduced on Eclipse 3.1.2 when comparing the structures of two large JAR files multiple times using an Eclipse built-in comparison option. Method `runCompare` implements a comparison operation (in plugin `org.eclipse.compare`). It is invoked every time the comparison option is chosen, and this information can be easily obtained from the Eclipse plugin APIs. The method takes a parameter of type `ISelection` that contains information about the two selected files to be compared. A `ResourceCompareInput` object is first created using this parameter (line 3). This object is fairly heavyweight as it caches the complete structures of the two files. The names of these files are then recorded in a list `visitedFiles` (lines 4 and 5), and this list may be used by the workspace GUI upon receiving a user request to view the history.

Next, `openCompareEditorOnPage` is invoked to open a compare editor in the workspace GUI that shows the differences between the two files. The (simplified) method body is shown at lines 11–17. In this method, the newly created `ResourceCompareInput` object is cached in a `NavigationHistoryInfo` object retrieved from the current workbench page for future save or restore operations (lines 14–16). Caching this heavyweight input object is actually the cause of the leak: the structures of the two files keep being created and referenced. As a result, the memory footprint grows quickly, and Eclipse runs out of memory. Note that in the real Eclipse code, this cache operation (at line 16) and the `runCompare` method are in two different plugins, which makes it particularly hard to diagnose the problem as these plugins are written by different groups of programmers. The two plugins communicate only through public interfaces and it is unclear to programmers of one plugin what side effects the other plugin can have. It is interesting to see that despite the fact that the developers of plugin `org.eclipse.compare` are aware that the input must be cleared after the comparison (e.g., in fact, the comment at line 8 is from the real code), these references are still unnecessarily kept somewhere out of their scope.

While these plugins have been studied in previous work [8, 20], we start with the tier H approach to simulate what a programmer would do at the beginning of a diagnostic task. Hence, our experience with this case, to a large degree, reflects how a programmer unfamiliar with a program can use the tool to diagnose memory leaks. We first need to identify a transaction and let the tool infer unnecessary references for the transaction. This is easy: as the comparison is the regularly occurring event leading to the leak, it is a natural idea to let the transaction cross the entire body of method `runCompare`, as shown in Figure 2(a). A transaction creation (at line 2) takes two parameters: a transaction identifier object and a mode in which the transaction works. The identifier object must be unique per transaction and must be created before the transaction starts. Here we choose s to be the identifier object, because s refers to an `ISelection` object that is created per comparison operation before `runCompare` runs. Constant *INFER* informs the tool to run in the inference mode. We found that the identifier object is usually easy to find for a transaction: a transaction body often crosses a method that is invoked on an existing object. In many cases, either the receiver object or a parameter object of the method can be se-

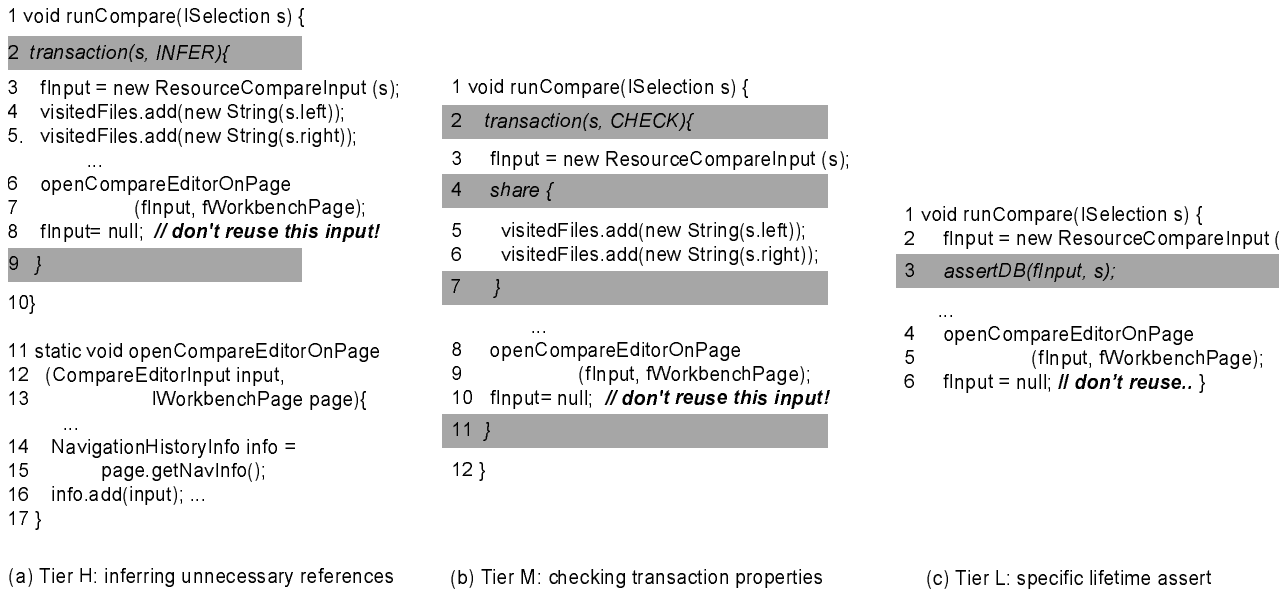

Figure 2. Overview of the technique, illustrated using a simplified version of Eclipse bug #115789.

lected as the transaction identifier object (as long as it is created per transaction and does not cross multiple transaction invocations).

For each transaction, its *spatial boundary* is specified by a pair of curly brackets (i.e., {. . . }) and its *temporal boundary* is defined by the lifetime of its identifier object. To find unnecessary references, we focus on objects created within its spatial boundary, and check their lifetimes against its temporal boundary. Informally, the semantics of a transaction is such that each *transaction-local* object must die[1] before the identifier object, and only *shared* objects can live after the identifier object dies. In the inference mode where a programmer does not explicitly specify local and shared objects, our tool infers shared objects automatically: objects created within the spatial boundary are treated as shared objects if they are still live at the time the identifier object dies. Once an object is marked as shared, the tool starts tracking its *staleness* and records a violation if it is not used for a given period of time, based on the intuition that objects that one transaction intends to share with other transactions should be used outside their creating transaction. Violations are aggregated and eventually reported in the order of their frequencies.

The following example shows a typical violation report that includes information about the transaction where the violating object is created, its creation site, violation type, the number of times this violation occurs, and the (heap) reference paths that lead to this object at the time of the violation. Each line in a reference path shows the information of an object on the path. While only one reference path is shown here (for illustration) for this violation, multiple paths were actually reported by the tool.

```
Transaction specified at:
   CompareAction:runCompare(ISelection), ln 2
Violating objects created at:
   CompareEditorInput:createOutlineContents
                (widgets.Composite), ln 439
Violation type:
   Objects shared among transactions are not used
Frequency: 4
Reference paths:
    Type: ArrayList, created at: NavigationHistory:
                                 <init>(WorkbenchPage), ln 44
--> Type: Object[], created at: ArrayList: <init>(I), ln 119
--> Type: NavigationHistoryEditorInfo, created at:
                NavigationHistory: createEntry(...), ln 553
--> Type: ResourceCompareInput, created at: CompareAction:
                                runCompare(ISelection), ln 3
```

This reference path makes it easy to explain why this `ResourceCompareInput` object is shared: the reference path exists because the object is cached (transitively) by a `NavigationHistory` object. However, this is not the only violation in the tier H report. The two strings created at lines 4 and 5 (together with many other objects) are also in the report as they are not used at all if there is no user request to perform history-related operations. It takes some time to inspect this report, as it contains a total of 36 violations. By ranking violations (based on frequencies and other factors), it is not hard for us to eliminate many of them that are lower on the list and that are obviously not leaks. For example, after inspecting the warnings, we determine that these small strings (and other similar objects) are not the major cause of the leak, because the growth of used memory is so significant that it is unlikely to be due to small objects (especially because their frequencies are not significantly higher).

As we obtain this knowledge (regarding these strings and other irrelevant violating objects), we move down to tier M for a more focused analysis. In Figure 2(b), we explicitly mark these strings (and others created by the two `add` calls) as shared (with a *share* region) and let our tool run in checking mode. Objects created in the transaction but not in the share region are marked (implicitly) as transaction-local objects. Our tool then ignores objects marked as shared and reports violations only when transaction-local objects are found live after the transaction identifier object dies. This gives us a much cleaner report (with 4 violations), and of course, the violation shown earlier appears in the report, indicating the `ResourceCompareInput` object is referenced from somewhere else.

After these two rounds of diagnosis, we have gained implementation knowledge (e.g., the general procedure of performing a comparison operation) and some insights into the problem (e.g., this leak might be caused by an unnecessary reference somewhere in `NavigationHistory`). During code inspection, we become interested in the comments in line 8 of the code: this statement has a

[1] In this paper, an object is considered to die immediately after it becomes unreachable in the object graph.

clear purpose of releasing the input object, but why did we see it is still reachable in both tier M and tier H reports? Having this question in mind, we decide to perform a detailed (tier L) diagnosis using a lifetime assertion (e.g., `assertDB` asserts a "dies-before" relationship), as shown in Figure 2(c). This single assertion fails, which confirms our suspicions.

After some code inspection (with the help of the reference path associated with the violation), we found that `NavigationHistory` allows a user to step backward and forward through browsed editor windows. It keeps a list of `NavigationHistoryEntry` objects, each of which points to an `EditorInfo` object that, in turn, points to a `CompareEditorInput` object, the root of a data structure that holds the diff results. `NavigationHistory` uses a count to control the number of `EditorInfo` objects it caches, and removes an `EditorInfo` if the count drops to zero. However, `NavigationHistory` does not correctly decrement this count in some cases, leading to unnecessary references.objects.

This example clearly shows the difficulty of diagnosing real-world memory problems. The root cause of this bug is that the `NavigationHistory` entries are cached and not getting removed due to reference-counting problems. This occurs entirely on the UI side. The developers of the `compare` plugin may have never thought that calling a general interface to open an UI editor can cause a big chunk of memory to be cached. The complexity of the large-scale code base and the limited knowledge of each individual developer strongly call for LeakChaser's step-by-step approach. Such tool support can help a programmer who starts without insights into the program or its leak, to systematically explore the leaky behavior in order to pinpoint its cause.

3. Assertions and Transactions

This section presents a formalism to describe our analysis of unnecessary references. The presentation proceeds in three steps. First, we define a simple garbage-collected language `assert` and its abstract syntax. We next give a semantics of this language by focusing on its *traces*. Each trace is a sequence of basic events (e.g., alloc, dealloc, and use) on objects, transaction events (e.g., start and end), and assertions. Finally, we formulate assertion checking and inference of unnecessary references as judgments on traces (i.e., trace validation). Note that in our implementation (discussed in Section 4), checking and inference are performed during GC runs. Here the trace collection phase and the trace validation phase are separated for ease of presentation and formal development.

Language `assert` The abstract syntax and the semantic domains for language `assert` are defined in Figure 3. The program has a fixed set of global reference-typed variables. An allocation site has the form a = new ref^o, where o stands for an allocation site ID defined at compile time.

There are two types of assertions: `assertDB` and `assertDBA`. `assertDB`(a, b) asserts that object (pointed to by) a must die before (or together with) object (pointed to by) b. object a must die before a new object is created by object b's allocation site (i.e., it is short for "Dies Before Allocation"). This assertion is useful to enforce a "replaces" relationship between two objects. For example, it can be used to enforce that an old (invalid) screen configuration is appropriately released before a new screen configuration is created upon repainting of an interface in a GUI program. As another example, in Figure 2(c), instead of using `assertDB`, we can also write `assertDBA(fInput, fInput)` to assert that the current `ResourceCompareInput` object must die before the next `ResourceCompareInput` object (created by the same allocation site) is allocated. While our framework includes a few other assertions, they are not discussed because they can be implemented using these two basic assertions.

Variables		a, b	$\in$ V
Allocation sites		o	$\in$ O
Instance fields		f	$\in$ F
Labels		l	$\in$ N
Assertions	e	::=	assertDB(a, b) | assertDBA(a, b)
Transactions	t	::=	transaction(a, m){ tb }
Trans bodies	tb	::=	s ; tb | share { s } tb | ϵ
Trans modes	m	::=	CHECK | INFER
Statements	s	::=	$a = b$ | a = new ref^o | a = null | while ... | $a = b.f$ | $a.f = b$ | e | gc | if ... | s ; s
Program	p	::=	s ; p | t ; p | ϵ

(a)

Labeled object	$\hat{o}$	::=	$o^l \in \Phi$
Environment	ρ	$\in$	$\mathsf{V} \rightarrow \Phi \cup \{\bot\}$
Heap	σ	$\in$	$\Phi \times \mathsf{F} \rightarrow \Phi \cup \{\bot\}$
Operation	τ	::=	$\langle \hat{o}, \mathrm{A} \mid \mathrm{D} \mid \mathrm{U} \rangle^o$ | $\langle \hat{o}, m, \mathrm{S} \mid \mathrm{E} \rangle^t$ | $\langle \mathrm{S} \mid \mathrm{E} \rangle^s$ | $\langle \hat{o}_a, \hat{o}_b, \mathrm{DBA} \mid \mathrm{DB} \rangle^a$
Traces	α	::=	τ , α | ϵ

(b)

Figure 3. A simple `assert` language: (a) abstract syntax (b) semantic domains.

To ease the formal development, a GC run can only be triggered by a `gc` statement, which traverses the object graph to reclaim unreachable objects. Note that we do not allow the nesting of transactions. While this is easy to implement in our framework, we have not found it helpful for pinpointing memory leak causes.

Each run-time object is labeled with its allocation site (e.g., o) and an integer (e.g., l), denoting the index of this object among all created by the allocation site. Environment and heap are defined in standard ways. A trace is a sequence of operations. There are four types of operations, each of which is a tuple annotated with a type symbol: o for object operation, t for transaction operation, s for share region operation, and a for assert operation. Each *object operation* is an object and event pair, where an event can be either A (i.e., Alloc), D (i.e., Dealloc), or U (i.e., Use). Each *transaction operation* is a triple containing its identifier object, the mode (i.e., CHECK or INFER), and whether this operation corresponds to the start of the transaction (S) or its end (E). A *share region operation* has only a type that indicates whether it is the start or the end of the region. Each *assert operation* contains two input objects and an assertion type (DBA for `assertDBA` and DB for `assertDB`). While the language does not explicitly consider threads, our analysis is thread-safe as each transaction creation is a thread-local event. Different transaction instances created by different threads can exist simultaneously for the same transaction declaration. Trace collection and validation are also performed on a per-thread basis. A special symbol $\bot$ is added to the heap and the environment to represent a `null` value.

Language semantics An operational semantics is given in Figure 4. A judgment of the form $s, \rho, \sigma, \alpha \Downarrow \rho', \sigma', \alpha'$ starts with a statement s, which is followed by environment ρ, heap σ, and trace α. The execution of s terminates with a final environment ρ', heap σ', and trace α'. Trace concatenation is denoted by $\circ$. Rules NEW, ASSERTDB, ASSERTDBA, and COMP are defined as expected. In rules LOAD and STORE, trace α is augmented with an object use event. In rule TRAN, for each transaction, the two transaction events (i.e., `S` and `E`) are added in the beginning and at the end of the trace for the execution of the sequence of statements in the transaction. The share region events are handled in a similar way (in rule SHAREREG). Rule GC removes unreachable objects from the heap, and records deallocation events for them in the trace α.

$$\frac{\alpha' = \alpha \circ \langle \hat{o}, \mathrm{A} \rangle^o \quad \hat{o}.allocsite = o \quad \hat{o}.l = \text{newObjIndex}(o)}{a = \text{new ref}^o, \rho, \sigma, \alpha \Downarrow \rho[a \mapsto \hat{o}], \sigma, \alpha'} \quad (\textsc{New})$$

$$a = b.f, \rho, \sigma, \alpha \Downarrow \rho[a \mapsto \sigma(\rho(b).f)], \sigma, \alpha \circ \langle \rho(b), \mathrm{U} \rangle^o \quad (\textsc{Load})$$

$$a.f = b, \rho, \sigma, \alpha \Downarrow \rho, \sigma[\rho(a).f \mapsto \rho(b)], \alpha \circ \langle \rho(a), \mathrm{U} \rangle^o \quad (\textsc{Store})$$

$$\frac{s, \rho, \sigma, \alpha \circ \langle \rho(a), m, \mathrm{S} \rangle^t \Downarrow \rho', \sigma', \alpha' \quad \alpha'' = \alpha' \circ \langle \rho(a), m, \mathrm{E} \rangle^t}{\text{transaction}(a, m)\{s\}, \rho, \sigma, \alpha \Downarrow \rho', \sigma', \alpha''} \quad (\textsc{Tran})$$

$$\frac{s, \rho, \sigma, \alpha \circ \langle \mathrm{S} \rangle^s \Downarrow \rho', \sigma', \alpha' \quad \alpha'' = \alpha' \circ \langle \mathrm{E} \rangle^s}{\text{share}\{s\}, \rho, \sigma, \alpha \Downarrow \rho', \sigma', \alpha''} \quad (\textsc{ShareReg})$$

$$\text{assertDB}(a, b), \rho, \sigma, \alpha \Downarrow \rho, \sigma, \alpha \circ \langle \rho(a), \rho(b), \mathrm{DB} \rangle^a \quad (\textsc{AssertDB})$$

$$\text{assertDBA}(a, b), \rho, \sigma, \alpha \Downarrow \rho, \sigma, \alpha \circ \langle \rho(a), \rho(b), \mathrm{DBA} \rangle^a \quad (\textsc{AssertDBA})$$

$$\frac{tr = \circ\{\langle \hat{o}, \mathrm{D} \rangle^o | \hat{o} \in \sigma \wedge \hat{o} \notin \text{reachable}(\rho, \sigma)\} \quad \forall \langle \hat{o}, \mathrm{D} \rangle^o \in tr : \hat{o} \notin \sigma' \quad \alpha' = \alpha \circ tr}{\text{gc}, \rho, \sigma, \alpha \Downarrow \rho, \sigma', \alpha'} \quad (\textsc{GC})$$

$$\frac{s_1, \rho, \sigma, \alpha \Downarrow \rho', \sigma', \alpha' \quad s_2, \rho', \sigma', \alpha' \Downarrow \rho'', \sigma'', \alpha''}{s_1; s_2, \rho, \sigma, \alpha \Downarrow \rho'', \sigma'', \alpha''} \quad (\textsc{Comp})$$

Figure 4. Operational semantics.

Trace validation Checking and inference of unnecessary references are formulated as judgments $\omega, \gamma, \pi, \iota \vdash \alpha \rightsquigarrow \omega', \gamma', \pi', \iota'$ on traces. Here α is an execution trace, ω is a stack of transactions and share regions, γ is a "diesBefore" map in which each pair ($\hat{o}_a$, $\hat{o}_b$) has been asserted to have a "diesBefore" relationship (i.e., $\hat{o}_a$ must die before $\hat{o}_b$), π is a "diesBeforeAlloc" map which contains pairs ($\hat{o}_a$, o) where object $\hat{o}_a$ has been asserted to die before allocation site o creates a new object, and ι maps each object that has been marked shared (i.e., in inference mode) to its staleness value (measured in terms of the number of transactions). As we currently do not support nested transactions, ω can contain at most one transaction identifier object (and one $\perp$ symbol indicating the execution is in a share region). Transaction nesting can be implemented easily by allowing ω to contain multiple identifier objects.

The validation rules are given in Figure 5. Validity checks are underlined, and the remaining clauses (without underlines) are for environment updates. Rule VALLOC first ensures that if an object $\hat{o}_1$ has been asserted to die before this allocation site creates a new object (i.e., due to an `assertDBA` assertion), $\hat{o}_1$ is not live anymore. A violation is recorded if there is such an object. If ω is not empty (meaning the execution is currently in a transaction) and $\texttt{top}(\omega)$ is not $\perp$ (meaning we are not in a share region), a pair ($\hat{o}$, top(ω)) is added to map γ because, as mentioned earlier, all transaction-local objects are asserted to die before the transaction identifier object, and top(ω) returns the identifier object of the current transaction.

VDEALLOC removes all entries $\langle \hat{o}, * \rangle$ in the three maps upon the deallocation of $\hat{o}$. If there is no running transaction, or the running transaction is in CHECK mode, this rule checks if there exists $\langle \hat{o}_1, \hat{o} \rangle \in \gamma$. If this is the case, a violation is reported, because $\hat{o}_1$ is still live at the time $\hat{o}$ dies. If the running transaction is in INFER mode, such $\hat{o}_1$ is marked as a shared object and the tool starts to track its staleness: a pair $(\hat{o}_1, 0)$ is added to map ι.

Rule VTRANS first pushes the transaction identifier object onto stack ω. It then adds a "diesBeforeAlloc" assertion on the identifier object itself: adding a pair ($\hat{o}, \hat{o}.allocsite$) indicates that this current instance $\hat{o}$ is asserted to die before allocation site $\hat{o}.allocsite$ creates a new object. As the lifetimes of transaction identifier objects

$$\frac{\underline{\nexists \hat{o}_1 : (\hat{o}_1, \hat{o}.allocsite) \in \pi} \quad \gamma' = \gamma \cup (\hat{o}, \text{top}(\omega)), \text{if } \omega \neq \emptyset \wedge \text{top}(\omega) \neq \perp \quad \gamma' = \gamma, \text{otherwise}}{\omega, \gamma, \pi, \iota \vdash \langle \hat{o}, \mathrm{A} \rangle^o \rightsquigarrow \omega, \gamma', \pi, \iota} \quad (\textsc{VAlloc})$$

$$\frac{\gamma' = \gamma \backslash \{(\hat{o}, *)\} \quad \pi' = \pi \backslash \{(\hat{o}, *)\} \quad \underline{\nexists \hat{o}_1 : (\hat{o}_1, \hat{o}) \in \gamma \wedge \iota' = \iota \backslash \{(\hat{o}, *)\}}, \text{if } \omega = \emptyset \vee \text{MODE} = \text{CHECK} \quad \iota' = (\iota \backslash \{(\hat{o}, *)\}) \cup \{(\hat{o}_1, 0) \mid (\hat{o}_1, \hat{o}) \in \gamma\}, \text{otherwise}}{\omega, \gamma, \pi, \iota \vdash \langle \hat{o}, \mathrm{D} \rangle^o \rightsquigarrow \omega, \gamma', \pi', \iota'} \quad (\textsc{VDealloc})$$

$$\frac{\omega' = \omega \circ \hat{o} \quad \text{MODE} = m \quad \pi' = \pi \cup (\hat{o}, \hat{o}.allocsite) \quad \iota' = \iota[\forall \hat{p} : \hat{p} \mapsto \iota(\hat{p}) + 1]}{\omega, \gamma, \pi, \iota \vdash \langle \hat{o}, m, \mathrm{S} \rangle^t \rightsquigarrow \omega', \gamma, \pi, \iota'} \quad (\textsc{VTranS})$$

$$\frac{\omega = \omega' \circ \hat{o} \quad \underline{\forall \hat{p} \in \text{dom}(\iota) : \iota(\hat{p}) < \mathrm{T}}}{\omega, \gamma, \pi, \iota \vdash \langle \hat{o}, m, \mathrm{E} \rangle^t \rightsquigarrow \omega', \gamma, \pi, \iota} \quad (\textsc{VTranE})$$

$$\frac{\iota' = \iota \backslash \{(\hat{o}, *)\}}{\omega, \gamma, \pi, \iota \vdash \langle \hat{o}, \mathrm{U} \rangle^o \rightsquigarrow \omega, \gamma, \pi, \iota'} \quad (\textsc{VUse})$$

$$\omega, \gamma, \pi, \iota \vdash \langle \mathrm{S} \rangle^s \rightsquigarrow \omega \circ \perp, \gamma, \pi, \iota \quad (\textsc{VShareS})$$

$$\frac{\omega = \omega' \circ \perp}{\omega, \gamma, \pi, \iota \vdash \langle \mathrm{E} \rangle^s \rightsquigarrow \omega', \gamma, \pi, \iota} \quad (\textsc{VShareE})$$

$$\omega, \gamma, \pi, \iota \vdash \langle \hat{o}_1, \hat{o}_2, \mathrm{DB} \rangle^a \rightsquigarrow \omega, \gamma \cup \{(\hat{o}_1, \hat{o}_2)\}, \pi, \iota \quad (\textsc{VDB})$$

$$\omega, \gamma, \pi, \iota \vdash \langle \hat{o}_1, \hat{o}_2, \mathrm{DBA} \rangle^a \rightsquigarrow \omega, \gamma, \pi \cup \{(\hat{o}_1, \hat{o}_2.allocsite)\}, \iota \quad (\textsc{VDBA})$$

$$\frac{\omega, \gamma, \pi, \iota \vdash \tau \rightsquigarrow \omega', \gamma', \pi', \iota' \quad \omega', \gamma', \pi', \iota' \vdash \alpha \rightsquigarrow \omega'', \gamma'', \pi'', \iota''}{\omega, \gamma, \pi, \iota \vdash \tau, \alpha \rightsquigarrow \omega'', \gamma'', \pi'', \iota''} \quad (\textsc{VTrace})$$

where

ω	$\in$ *Transaction stack*:	$\mathsf{N} \rightarrow \Phi \cup \{\perp\}$
γ	$\in$ *"diesBefore" map*:	$\Phi \rightarrow \Phi$
π	$\in$ *"diesBeforeAlloc" map*:	$\Phi \rightarrow \mathsf{O}$
ι	$\in$ *Staleness map for shared objects*:	$\Phi \rightarrow \mathsf{N}$
MODE:	Mode of transaction (CHECK or INFER)	
T:	Threshold staleness value	

Figure 5. Checking and inferring of unnecessary references.

are used to specify temporal boundaries of transactions, they are not allowed to overlap. Lifetime overlapping can lead to ambiguity of transaction behaviors. For each object inferred to be a shared object (i.e., $\hat{p} \in \text{dom}(\iota)$), its staleness value is incremented. An object's staleness is defined as the number of transactions since it has been marked as shared. At the end of each transaction (rule VTRANE), this staleness value is checked against a user-defined threshold T. A violation is reported if a shared object's staleness exceeds this threshold. This rule also pops stack ω. This stack becomes empty after this rule, indicating that no transaction is currently running.

VUSE removes from ι an object marked as shared: its staleness is no longer tracked because the object is used. Rules VSHARES/VSHAREE push/pop $\perp$ onto/from stack ω. Having $\perp$ on top of ω means the execution is in a share region of a transaction. Rule VTRACE specifies the composition of two different traces.

Our system allows one to specify transactions and assertions simultaneously in a program. Transaction properties are translated into basic assertions, which are checked together with assertions specified by programmers. For example, upon the allocation of each object $\hat{o}$ in a transaction (shown in VALLOC), $\hat{o}$ and the transaction identifier object (i.e., top(ω)) are added into map γ, and

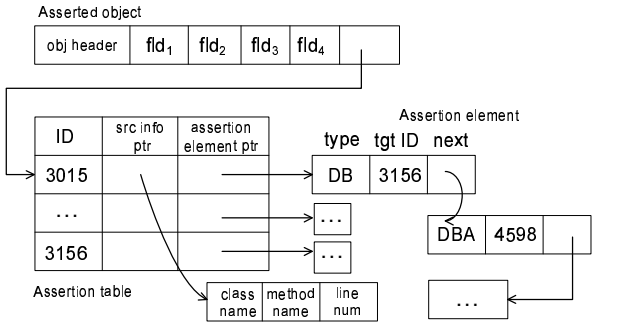

Figure 6. Assertion table implementation that allows quick assertion checks.

this relationship is checked in exactly the same way as a normal `assertDB` assertion (shown in VDB).

4. Implementation

We have implemented LeakChaser in Jikes RVM 3.1.0, a high-performance Java virtual machine [1]. LeakChaser is publicly available on the Jikes RVM Research Archive.[2]

Metadata and instrumentation LeakChaser adds one word to the header of each object that tracks the allocation site information of the object. There are two dynamic compilers in Jikes that transform Java bytecode into native code. The baseline compiler compiles each method when it first executes. When a method becomes hot (i.e., executed frequently), the optimizing compiler recompiles it at increasing levels of optimizations. LeakChaser adds instrumentation to both compilers. LeakChaser adds instrumentation at each allocation site that stores an identifier for the source code location (class, method, and line number) into the allocated object's extra header word.

Garbage collection LeakChaser performs assertion checks during garbage collection runs. It uses a table structure (discussed shortly) to represent the assertions, and scans this table to perform checks at the end of each GC run.

To implement quick assertion checks, we create an assertion table, shown in Figure 6, to record asserted "diesBefore" and "diesBeforeAlloc" relationships during the execution. Once a pair of objects is asserted, we create an assertion table entry for each of them, and then update the LeakChaser-reserved header word in each object with the address of the object's corresponding assertion table entry. The pointer to the source code information of the object initially stored in this space is moved to a field of this table entry.

For each table entry, we let its "assertion element pointer" field point to a linked list of assertion elements, each of which represents an object that has been asserted together with this object. For example, for an assertion `assertDB`(a, b), we create an assertion element (`DB`, id_b, null) and store its address in a's assertion table entry. Once a new assertion `assertDBA`(a, c) is executed, a new assertion element (`DBA`, id_c, null) is created and appended to the list (i.e., now the previous element's next field points to this new element). In this way, all objects that have been asserted through `assert...`(a,...) are in an assertion chain that is going to be checked when a's entry is traversed. We do not need to create an assertion element representing a and associate it with b and c's table entries, as this unnecessarily duplicates information.

The current implementation of LeakChaser supports all non-generational garbage collectors (e.g., MarkSweep, MarkCompact, and Immix). At the end of each GC run, the assertion table is scanned twice: the first scan marks entries that correspond to objects that are unreachable in this GC (i.e., dead objects), and the second scan performs violation detection. Hence, in the second scan, all table entries represent live objects. In order to slow down the growth of the assertion table, table entries corresponding to unreachable objects are reclaimed and reassigned later to newly asserted objects. Our current implementation does not work correctly in a generational garbage collector, as a nursery GC scans only part of the heap and thus may cause LeakChaser to report either false positives or false negatives.

For each entry, its assertion element chain is traversed. For an element whose type is `DB`, if the object represented by this element has been reclaimed, we report a violation. It is much more difficult to check a `DBA` assertion. We create a global array (per thread) and each entry in this array records the status of an allocation site. Once an `assertDBA`(a, b) assertion is executed, the array entry corresponding to b's allocation site is marked as "ASSERTED". When this allocation site creates a new object, the status of its array entry is changed to "ALLOCATED". During the scanning of the assertion table, for an assertion element (`DBA`, i, *), a violation is reported if the entry of this global array corresponding to the allocation site of the object whose assertion table entry id is i is "ALLOCATED", because it creates a new object before the asserted object dies. No false information can result from assertion failures reported by the tool because the assertions directly specify the liveness of objects (instead of reachability on the object graph) and report violations only when definitive evidence is observed.

In this paper, we focus on the real-world utility of LeakChaser, rather than its performance. Hence, this assertion table is scanned during every GC. Future work coul incorporate sampling to reduce overhead (i.e., scan the assertion table less frequently). To report the reference paths that lead to a violating object, we modify the worklist-based algorithm used by the tracing collector in a way so that when a reachable object is added into the worklist, its reference path (from which it is reached) is also added (to another worklist). The length of this path can be determined by the user. We develop a technique that aggregates violations of objects that are created by the same allocation sites and whose reference paths match, so that a violation is reported only after its (aggregated) frequency exceeds a user-defined threshold value. LeakChaser filters out violations that are associated with VM objects. Future work could consider more powerful aggregation and ranking functions, such as a combination of frequency, staleness, and the amount of memory leaked. In order to make LeakChaser work for a generational GC, future work could perform the two assertion table scans separately: the first scan (that marks dead objects) would be performed at the end of every GC, while the second scan (that checks assertions and detects violations) would be performed only at the end of each full-heap GC.

5. Case Studies

We have evaluated LeakChaser on six real, reported memory leaks. While our ultimate goal is to evaluate with enterprise-level applications such as application servers and programs running on top of them, Jikes RVM fails to run some larger server applications (such as `trade` in the latest DaCapo benchmark set [7]).

For these six cases, we also applied Sleigh [8], a publicly available research memory leak detector for Java. Sleigh finds leaks by tracking the staleness of arbitrary objects and reporting allocation and last-use sites for stale objects.

For each case, we compared our report with the report generated by Sleigh, and found that first, for the same amount of running time, information reported by our (even tier H) approach is much more relevant than that reported by Sleigh. Sleigh requires much more time to collect information and generate relatively precise re-

[2] `http://www.jikesrvm.org/Research+Archive`

Case	*#Tran*	*#TO*	*#LO*	*#SO*	*#V*	*#F*
Diff	8	2048148	1707244	340904	36	14
Jbb	4346	256236	186752	69484	14	4
Editor	12	512471	506620	5851	2	13
WTP	20	2774556	2767237	7319	27	39
MySQL	10000	319529	170902	148627	11	0
Mckoi	100	2689366	2243888	445478	10	193

Table 1. Tier H statistics for transactions for the case studies. Shown are the number of transaction runs (#Tran), the total number of objects tracked (#TO), the number of transaction-local objects (#LO), the number of shared objects inferred (#SO), the number of violations reported after filtering (#V), and the number of violations filtered by our filtering system (#F).

ports because it is designed for production runs and uses only one bit per object to encode information statistically, whereas LeakChaser tracks much more information for debugging and tuning (at a higher cost). We did not run Sleigh for as much time as in the prior work [8], and thus Sleigh results are different from those reported earlier. Second, our iterative technique produces more precise information at each tier, which eventually guides us to the root causes of leaks.

Experience summary We found that it is quite easy to specify transactions in large programs even for users who have never studied the programs before. From our experience using LeakChaser, we generalize an approach that can be employed by performance experts to select transactions. All large-scale applications we have studied can be classified into two major categories: event-based systems (e.g., web servers and GUI applications) and transaction-based systems (e.g., enterprise Java applications and databases). An event-based system often uses a loop to deal with different events received and dispatch them to their corresponding handlers. For such a system, our transaction can cross the body of the loop; that is, the handling of each event is treated as a transaction. Each event object can be used as the identifier object for the transaction. It is much easier to determine a transaction for a transaction-based system, as each "system transaction" in the original program can be naturally treated as a transaction in LeakChaser.

A particularly useful feature of LeakChaser is that it allows programmers to quickly specify transactions at a *client* that interacts with a complex system, without digging into the system to understand its implementation details. For example, we diagnose Eclipse framework bugs by specifying transactions in plugins (i.e., clients) that trigger these bugs. Likewise, for databases such as `MySQL` and `Mckoi`, we only need to create transactions at client programs that perform database queries. This feature allows performance experts like us (who are unfamiliar with these databases) to quickly get started with LeakChaser. If we could not put transactions around clients, it would be hard to even find a starting point in these systems that have thousands of classes and millions of lines of code.

Table 1 shows transaction and assertion statistics for all six cases. Despite the large number of assertions checked for each benchmark, LeakChaser reports a small number of warnings. We show statistics for tier H only, as it is the coarsest-grained approach and thus runs the most assertions and reports the most violations.

Diff *(Eclipse bug #115789)* We quickly found this bug using our three-tier approach and the detailed diagnosis process discussed in Section 2.

We have tried Sleigh on this program and the top four last-use sites (i.e., where objects are last used) reported by Sleigh are in class `java.util.HashMap` (e.g., in methods `put` and `addEntry`).

In this case, the sites leading to these `HashMap` operations point back to method `createContainer` in class `org.eclipse.compare.ZipFileStructureCreator`, which indicates that the structures of input files are cached but not used. At this point, it is completely up to the programmer to find out why these structures become stale and which objects reference them. Recall that even a violation reported by our tier H approach shows the violating object is referenced by a `NavigationHistory` object, which is actually the root cause of the leak (shown in in Section 2). According to [8], Sleigh could have reported much more precise information if the program was run for significantly more time. For example, LeakChaser reported the root cause after only 8 structure diffs were performed, while Sleigh may need more than 1000 diffs to report relatively precise information regarding where the stale objects are created. In addition, as we observed in our experiments, reporting reference paths can be more helpful for finding leak root causes than reporting program locations, which are usually far from where the true problem occurs.

Jbb SPECjbb2000 simulates an online trading system. It contains a known memory leak that has been studied many times [3, 8, 20, 36]. This bug is caused by unnecessary caching of `Order` objects. To investigate the usefulness of LeakChaser in helping programmers unfamiliar with the code, we attempt to simulate what these programmers would do. First, we need to understand the basic modules and functions of the program so that we can identify regularly occurring events. This is not hard at all, as SPECjbb is a transaction-based system where a `TransactionManager` class runs different transactions types, and the transaction-creating method is only a few calls away from method `main`. This method contains a loop that retrieves a command from an input map per iteration, and creates and runs a transaction whose type corresponds to the command received. This was the knowledge we obtained quickly before trying the tier H approach.

Problem 1: unused objects We added a transaction that encloses the main Jbb transaction-creating method. Each Jbb transaction object is used as the transaction identifier object of its corresponding transaction. We ran the program on our modified VM and let LeakChaser infer unnecessary references. The tool reports 14 violations (detailed statistics are shown in Table 1). Most of the violating objects are `String` and `History` objects cached in orders, and the rest are `Order` objects transitively referenced by `District` objects. We inspected the source code and found that these `String` objects are created to represent district information or names of order lines, and the `History` objects are used to represent histories of orders made by each company. These objects are indeed never used in the program. We modified the program to eliminate these unused strings and `History` objects. The throughput improvements are shown in Figure 7(a). We did not expect to find this problem, which had not been reported before.

Problem 2: mutual references It was unclear to us what to do about the `Order` objects reported above: while many are not used again, some are retrieved by subsequent transactions. We further inspect the code in order to run a more focused diagnosis. The reference paths in our tier H report show that the unused `Order` objects are referenced by `Customer` objects and then by `Company` objects. Following this clue, we inspect all classes related to `Company` and `Customer`. One clear piece of information that we can take advantage of is the ownership relationships among `Company` and `Customer` objects and objects referenced by them. For example, each customer has an order array that stores orders made by that customer. It is clearly problematic if a customer object dies while an order made from this customer is still alive. Using this information, we wrote 72 `diesBefore` assertions to assert such relationships.

We easily added these assertions in constructors, wherever an owned object is assigned to an owner object. Running this version of the program resulted in 4 violations, all of which are related to orders. By inspecting reference paths associated with these viola-

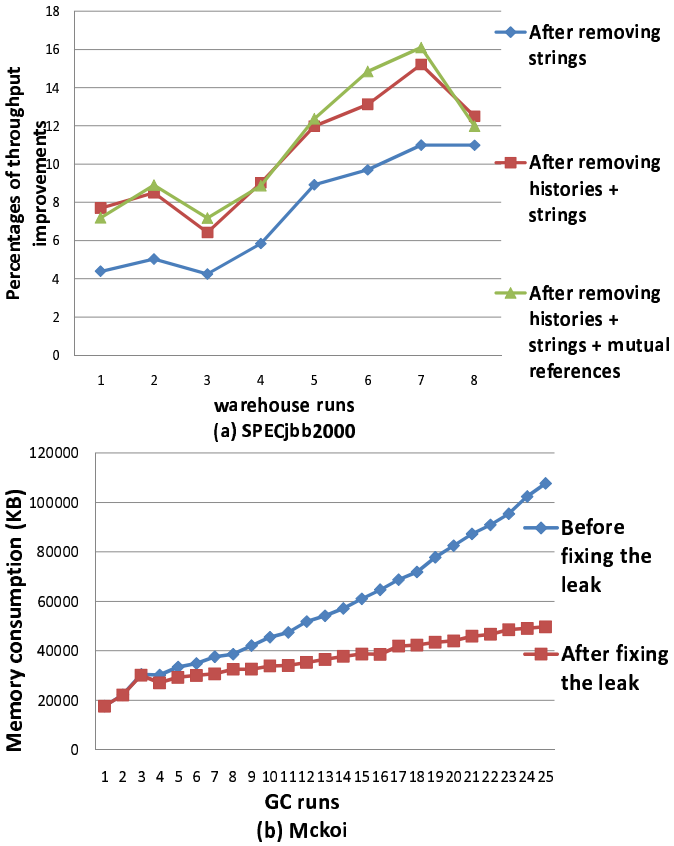

Figure 7. (a) Throughput improvements after fixing the memory problems detected in SPECjbb; (b) Comparison between the memory footprints of Mckoi before and after fixing the leak.

tions, we found the following three important problems. (1) `Order` and `Customer` are mutually referenced, which explains the persistent increase in `Order` objects, even though customers frequently drop orders. (2) `Customer` objects are not released even after their container arrays in `Company` die. They are not released because `Company` holds unnecessary references to these objects. (3) `Order` and `Heap` are mutually referenced, which prevents `Heap` objects from being garbage collected. Fixing all of these problems (including the unused `String` and `History` objects) led to both the elimination of the quick growth of memory consumption and an overall 12.7% throughput improvement (from 13,644 to 15,372 ops/sec). Previous work has not reported all of these problems that we found using LeakChaser.

An important last-use site reported by Sleigh is in method `getObject` of a BTree data structure that is used to retrieve `Order` objects from customers. The associated calling contexts tell us that this call is made transitively by `processLine`. While this is indeed a method that needs to be fixed, the key to understanding the problem is learning about the mutual references among `Order` and other classes. Without such reference information, it remains a daunting task to identify the root cause and develop a fix.

Editor *(Eclipse bug #139465)* This reported memory leak in Eclipse 3.2 occurs when opening a `.outline` file using a customized editor and making selections on the "outline" page. Objects created while displaying the file keep accumulating, and Eclipse eventually crashes. This bug is still open because users can rarely reproduce the leak. While we could not observe the reported symptoms, we still ran the test case with LeakChaser. We wrote a plugin that repeats multiple times the process of opening the editor, making selections, and closing the editor. We added a transaction in this plugin that crosses each such process (which manipulates the editor), and used the editor object as the transaction identifier.

The tier H approach reported only two violations even when we set a very small number (2) as the staleness threshold (see T in Figure 5). For each violation, we could not find any object on its associated reference paths that is related to the editor of interest. Hence, we quickly concluded that there were no unnecessary references for this case in Eclipse 3.2 on the Linux platform where we ran the experiment: had there existed a problem with respect to a transaction, it should have been reported by the tier H approach, as the approach captures references to all objects that are shared but not used. For this case, Sleigh reported four last-use sites, and we found it difficult to verify whether or not these sites are relevant. The key is to understand how these objects are reachable in the heap, instead of where they are used in the program.

WTP *(Eclipse bug #155898)* According to the bug report for this memory leak in the Eclipse Web Tool Platform (WTP), the memory footprint grows quickly when copying and pasting large text blocks in the JSP editor associated with the WTP framework. This bug is still open because it cannot always be reproduced. One developer suspected that "the bug might have already been fixed by the addition of other features in a later version of WTP" one year after it was reported. We reproduced the problem and saw the memory footprint growth by writing a plugin that automatically copies and pastes texts several times. Similarly to the previous case, we added a transaction in the plugin and let it cross each iteration that involves a pair of copy-and-paste operations. Our tier H approach reported a total of 26 violations, among which 11 seemed clearly irrelevant to the problem (e.g., regarding Eclipse's Java Development Tools and other plugins).

To confirm this observation, we use this plugin to perform the same copy-and-paste operations in a regular text editor in Eclipse, which does not have memory problems. These 11 violations also appeared in the generated report. Thus, they were safely discarded. As this bug is caused by copying and pasting text, we focus on string-related violations. Only one violation among the remaining 15 is about `String` objects, shown as follows:

```
Transaction specified at:
  Quick_copy_pastePlugin:mouseUp(MouseEvent), ln 258
Violating objects created at:
  StructuredTextUndoManager:createNewTextCommand(String, String),
                                                          ln 302
Violation type:
  objects shared among transactions are not used
Frequency: 28
Reference paths:
    Type: StructuredTextViewerUndoManager, created at:
              StructuredTextViewerConfiguration
              : getUndoManager(ISourceViewer), ln 469
--> Type: StructuredTextUndoManager, created at:
     BasicStructuredDocument: getUndoManager(), ln 1823
--> Type: BasicCommandStack, created at:
          StructuredTextUndoManager : <init>(), ln 160
--> Type: ArrayList, created at: BasicCommandStack: <init>(),
                                                        ln 67
--> Type: Object[], created at: ArrayList: ensureCapacity(I),
                                                       ln 176
```

This violation clearly shows that the strings are (transitively) referenced by a text undo manager. This information quickly directed us to classes responsible for undo operations. The cause of the problem was clear to us almost immediately after inspecting how an undo operation is performed. In this JSP editor, all commands (and their related data) are cached in a command stack, in case an undo is requested in the future. There is a tradeoff between "undoability" (i.e., how many commands are cached) and performance, especially when there is a large amount of data being cached with each com-

mand. In this version of WTP (WTP 1.5 with Eclipse 3.2.0), this command stack can grow to be very deep (it is not cleared even when a save operation is performed), and thus, many strings can be cached, leading to significant performance degradation. A later version has limited the depth of this stack (for other purposes), implicitly fixing this bug. For this case, we understood the problem even without moving to tier M. We did not manage to run Sleigh for this case, as Sleigh was developed on top of an older version of Jikes RVM (2.4.2), which cannot run Eclipse 3.2.

MySQL leak This case is a simplified version of a JDBC application that exhausts memory if the application keeps using the same connection but different SQL statements to access the database. Prior work reproduced this leak to evaluate tolerating leaks [9], but no prior work reports the leak cause. The leak occurs because the JDBC library caches already-executed SQL statements in a container unless the statements are explicitly closed. We create a transaction that crosses the creation and execution of each `PreparedStatement` (in a loop). We executed 100 iterations, and the tier H approach reported 10 violations, all of which were related to `PreparedStatement` (i.e., regarding either the statement object itself, or objects reachable from the statement).

From the associated reference paths, it took only a few minutes for us to identify the container that caches the statements: a `HashMap` created at line 1486 (in the constructor) of class `com.mysql.jdbc.Connection`. For this case, Sleigh reported warnings after the program ran for 303 iterations, significantly longer than for LeakChaser. Sleigh reported a few last-use sites where `PrepareStatement` objects are last touched. We did not find the `HashMap` information in Sleigh's report, which is the key to tracking down this leak bug.

Mckoi leak Mckoi is an open-source SQL database system written in Java (`http://www.mckoi.com`). It contains a leak that previous work reproduced to evaluate leak survival techniques [9, 10]. However, none of the existing work has investigated the root cause of the leak. We reproduced the leak by writing a client that repeatedly (100 times) establishes a connection, executes a few SQL queries, and closes the connection. We started with our tier H approach and created a transaction that crosses each iteration of this process in the client. Our tool reported 10 warnings, all regarding objects that are reachable from a thread object of type `DatabaseDispatcher`. In all the warnings reported, this thread object references a `DatabaseSystem` object, which transitively caches many other never-used objects. By inspecting only the constructors of `DatabaseSystem` and `DatabaseDispatcher`, we found that they are mutually referenced. The creation of each `DatabaseSystem` object explicitly creates a `DatabaseDispatcher` object and runs this thread in the background. There are two major problems:

(1) `DatabaseDispatcher` runs in a `while(true)` loop, which means no `DatabaseSystem` object can ever be garbage collected even though its `dispose` method is invoked. To address this problem, we broke this reference cycle when `dispose` is invoked on `DatabaseSystem`. Next, in `DatabaseDispatcher`, we modified the `while(true)` loop to terminate if its referenced `DatabaseSystem` object becomes null. This modification resulted in a very slight improvement in performance, leading us to believe there must be a bigger problem.

(2) A `DatabaseSystem` object can be created in two situations. (a) A new database needs to be started (e.g., a connection is established); (b) a method `dbExists` (that checks whether a database instance has already been there) is called. In every iteration, two `DatabaseSystem` objects are created but only one of them gets disposed. The one created by `dbExists` is never reclaimed, because `dispose` is not explicitly invoked on this object, which we quickly discovered by using a call graph generated by an IDE tool. The developer took it for granted that this object would be garbage collected, but it is referenced by a live thread. We added a call that invokes `dispose` at the end of method `dbExists`.

For this case, all warnings generated by Sleigh are about objects cached (transitively) by `DatabaseSystem`, and they are far away from the leaking thread. It would be difficult to understand why these objects are not garbage collected without appropriate reference paths, as reported by LeakChaser. The memory footprint of the database before and after the leak is fixed is shown in Figure 7(b). The original version of the program ran out of memory at the 106^{th} iteration, while the modified version ran indefinitely (as far as we could tell).

These six case studies demonstrate that developers do not need to have much implementation knowledge in order to diagnose leaks with LeakChaser. LeakChaser generates more relevant reports than Sleigh: it is easier to find the root cause from reference chains that cache a violating object, than from the allocation or last-use site of the object.

6. Overhead

We evaluated the performance of our technique using 19 programs from the SPECjvm98 [34] and DaCapo [7] benchmarks, on a dual-core machine with an Intel Xeon 2.83GHZ processor, running Linux 2.6.18. We ran each program using the large workload.

Table 2 reports the time and space overheads that our tracking infrastructure incurs on normal executions (without assertions or transactions written). The overhead measurements that we show in this section are obtained based on a (high-performance) FastAdaptiveImmix configuration that uses the optimizing compiler and the state-of-the-art Immix garbage collection algorithm [6].

The LeakChaser infrastructure slows programs by less than 10% on average using the optimizing compiler and an advanced garbage collector. Much of the overhead comes from extra operations during GC and the barrier inserted at each object reference read to check the use of the object. Column (b) of Table 2 reports the detailed GC slowdown caused by our tool, which is 2.3×, averaged across the 19 programs. The space overheads, reported in column (c) of Table 2, are less than 10%, primarily due to the extra word added to the header of each object. In some cases, the peak memory consumption for LeakChaser is even lower than that for the original run (i.e., *OS* is smaller than 1), presumably because GC is triggered at a different set of program points that happens to have a lower maximum reachable memory size.

The overhead of assertion checking and inference is reported in Table 3. The overall running time measurements are available only for *Jbb*, *MySQL*, and *Mckoi*, as the other three cases are all based on Eclipse IDE operations. Note that in a run with 4,346 transactions and 442,988 objects (shown in Table 1), our tool slows the program 6.6× and 5.5× overall for the two configurations. Similarly to Table 2, the overhead can be larger if we consider only GC time. For example, for *Diff*, a 2.8× slowdown can be seen for GC time for FastAdaptiveImmix. Note that these overheads have not prevented us from collecting data from any real-world application. In a production setting, run-time overhead can be effectively reduced by sampling (i.e., the current sampling rate is 100%). Future work could also define a tradeoff framework between the quality of the reported information and the frequency of assertion table scanning (similar to QVM [5]), and find an appropriate sampling rate that can enable the reporting of sufficient information at acceptably low cost.

7. Related Work

While there exists a large body of work on detecting Java memory leaks, ours is the first semantics-aware approach that uses multiple

Bench	(a) Overall time			(b) GC time			(c) Space		
	T_1	T_2	*OT*	G_1	G_2	*OG*	S_1	S_2	*OS*
check	0.033	0.068	2.1	0.016	0.033	2.0	11.4	10.3	0.9
compr	9.4	9.5	1.0	0.016	0.033	2.0	17.9	18.0	1.0
jess	2.2	2.4	1.1	0.022	0.049	2.2	48.9	58.4	1.2
db	4.9	5.0	1.0	0.024	0.062	2.6	22.2	24.9	1.1
javac	2.1	2.4	1.1	0.030	0.064	2.1	98.1	99.0	1.0
mpeg	6.2	6.2	1.0	0.020	0.039	2.0	19.4	19.8	1.0
mtrt	1.6	1.6	1.0	0.031	0.081	2.6	46.5	43.6	0.9
jack	1.7	1.8	1.0	0.025	0.050	2.0	79.6	76.8	1.0
antlr	38.7	43.2	1.1	0.032	0.074	2.3	53.5	57.3	1.1
bloat	98.3	105.9	1.1	0.073	0.169	2.3	515.1	520.0	1.0
chart	19.7	21.4	1.1	0.084	0.232	2.8	367.4	479.4	1.3
eclipse	150.8	157.5	1.0	0.207	0.465	2.2	512.4	532.7	1.0
fop	1.4	1.8	1.3	0.064	0.15	2.3	107.6	102.2	0.9
hsqldb	9.3	16.4	1.8	0.331	1.21	3.7	420.5	432.6	1.0
jython	45.6	48.6	1.1	0.077	0.172	2.2	119.7	137.8	1.2
luindex	35.3	37.7	1.1	0.031	0.070	2.3	45.5	49.8	1.1
lusearch	7.8	8.1	1.0	0.038	0.088	2.3	105.0	129.5	1.2
pmd	22.9	23.6	1.0	0.055	0.127	2.3	115.3	157.5	1.4
xalan	32.3	39.5	1.2	0.066	0.145	2.2	214.2	209.0	1.0
GeoMean	-	-	1.1	-	-	2.3	-	-	1.1

Table 2. Time and space overheads incurred by our infrastructure for the FastAdaptiveImmix configuration. Shown in column (a) are the original overall execution times (T_1) in seconds, the execution times for our modified JVM (T_2), and the overheads (*OT*) in times ($\times$). Column (b) reports the GC times and overheads: G_1 and G_2 show the average times for each GC run in the original RVM and our modified RVM, respectively. *OG* reports the GC overheads as G_2/G_1. Memory consumption and overheads are shown in column (c): S_1 and S_2 are the maximum amounts of memory used during the executions in the original RVM and our RVM, respectively. *OS* reports the space overheads as S_1/S_2.

Bench	(a) Overall time			(b) GC time			(c) Space		
	T_1	T_2	*OT*	G_1	G_2	*OG*	S_1	S_2	*OS*
Diff	-	-	-	0.27	0.75	2.8	534.2	534.4	1.0
Jbb	50.7*	9.2*	5.5	0.008	0.056	7.0	313.7	315.3	1.0
Editor	-	-	-	0.15	0.41	2.7	300.5	367.2	1.2
WTP	-	-	-	0.22	0.58	2.6	52.7	52.9	1.0
MySQL	6.8	14.0	2.1	0.045	0.108	2.4	17.1	17.7	1.0
Mckoi	165.6	172.2	1.0	0.046	0.154	3.3	129.2	144.2	1.1
GeoMean	-	-	2.3	-	-	3.2	-	-	1.0

Table 3. Overheads for the cases that we have studied under FastAdaptiveImmix. * indicates that we measure throughput (#operations per second) instead of running time. For both MySQL and Mckoi, a fixed number of iterations (100) was run for this measurement.

layers to help programmers at different levels of code familarities to specify, check, and infer unnecessary references. Related work falls into four major categories: leak detection, bloat analysis, checking of unconventional properties, and GC assertions.

Dynamic analysis for memory leak detection Dynamic analysis [8, 13, 14, 16, 17, 18, 20, 23, 28, 29] has typically been used to detect memory problems. As described in Sections 1 and 5, existing techniques have a number of deficiencies. Commercial leak detectors such as [16, 28] enable visualization of heap objects of different types, but do not provide the ability to pinpoint the cause of a memory leak. Existing research detectors use growing types [20, 23] (i.e., types whose number of instances continues to grow) or object staleness [8, 18] to identify suspicious data structures that may contribute to a memory leak. However, in general, a memory leak caused by redundant references is due to a complex interplay of memory growth and staleness and possibly other factors. In addition, these techniques are often unaware of high-level semantics, and perform whole-program profiling; this leads to less relevant information in the generated reports. Xu and Rountev [36] introduce a technique that profiles container behaviors to detect containers that cache large numbers of unused objects. Novark *et al.*. [27] find memory leaks and bloat in C/C++ programs by segregating objects based on their allocation contexts and staleness. Recent work [14] uses a tainting framework to propagate relevant information in order to identify leaking objects for C/C++ programs.

Fundamentally different from these techniques, our work presents an assertion framework that allows programmers to explicitly express their interests (i.e., related to high-level semantics), based on the insight that developers' knowledge is essential for a leak detector to produce highly relevant reports.

Software bloat analysis As a more general problem [26, 40], software bloat analysis [4, 21, 22, 25, 32, 36, 37, 38, 39] attempts to find, remove, and prevent performance problems due to inefficiencies in the code execution and the use of memory. Prior work [22, 24] proposes metrics to provide performance assessment of use of data structures. Mitchell *et al.* [24] propose a manual approach that detects bloat by structuring behavior according to the flow of information, and their later work [22] introduces a way to find data structures that consume excessive amounts of memory. Work by Dufour *et al.* [15] uses a blended escape analysis to characterize and find excessive use of temporary data structures. By approximating object lifetimes, the analysis has been shown to be useful in classifying the usage of newly created objects in the problematic areas. Shankar *et al.* propose Jolt [32], an approach that makes aggressive method inlining decisions based on the identification of regions that make extensive use of temporary objects. Work by Xu *et al.* [38] detects memory bloat by profiling copy chains

and copy graphs. Other work [31] dynamically identifies inappropriately used Java collections and recommends to the user those that should really be used. The technique presented in our work can also be considered as a bloat analysis, and it can be used to find a specific type of bloat caused by unnecessary references.

Asserting and inferring unconventional properties The Java Modeling Language [2] contains a few expressions (such as `\duration` and `\space`) that allow programmers to specify performance-related properties of a program. Burnim and Sen propose an assertion framework [11] to specify that regions of a parallel program behave deterministically despite non-deterministic thread interleavings. They later propose to dynamically infer deterministic specifications for parallel programs from the execution, given a set of inputs and schedules [12].

GC Assertions, QVM, and Merlin Closest to our work are heap property assertion frameworks such as GC Assertions [3, 30] and QVM [5, 35]. For example, one can explicitly specify at a certain program point that an object should be dead soon (i.e, *assertDead*), or that an object must be owned by another object in the object graph (i.e., *assertOwns*). While such assertions can be quite useful in helping diagnosis, they are limited in the following three important aspects related to leak detection.

First, reachability information is used to approximate the liveness of objects, which may result in false positives. For example, *assertOwns* (a, b) asserts a reachability relationship between objects a and b, and it fails when b can be reached in a path that does not contain a in a certain GC run. However, it is possible that object b becomes unreachable from object b in one GC, while later b is owned by a again. Second, because a GC assertion predicts a future heap state (i.e., the state at the closest GC run) and the global reachability information evolves all the time, whether or not this assertion will fail depends significantly on when and where the next GC occurs, which may in turn be affected by many factors, such as the initial and maximum heap sizes and specific GC implementation strategies. Third, these approaches are intended for programmers who have sufficiently deep program knowledge and insights. In real-world software development, only a handful of programmers can have such knowledge, especially when a performance problem occurs in program code that is not written by themselves.

Our work solves the first and second problems by allowing programmers to specify objects' lifetime relationships instead of reachability properties. In order to tackle the third problem, we use a combination of assertion checking and transaction property inference to allow programmers with little application-specific knowledge to quickly identify the cause of the problem.

The GC Assertions [3] framework includes a block-structured *assert-alldead* assertion, which asserts that all objects allocated in the block must be dead by the end of the block. While it is related to the transaction abstractions proposed in this paper, there are two important distinctions between them. First, *assert-alldead* does not allow objects in the specified structure to escape the structure, while our approach allows checking and inferring shared objects, providing more flexibility for diagnosing problems. Second, our transaction abstraction separates temporal and spatial scopes of the structure, while these scopes are combined in this earlier work.

Merlin [19] is an efficient algorithm that can provide precise time-of-death statistics for heap objects by computing when objects die using collected timestamps. LeakChaser could potentially exploit this technique in the future to capture assertion failures between GCs, as we currently report an assertion failure only when it is actually witnessed during a GC run.

8. Conclusions and Future Work

This paper presents the first framework that allows programmers with little program knowledge to quickly find the root cause of a memory leak, by bringing high-level program semantics into low-level leak detection. The most significant advantages of this work over existing Java leak detection tools are that (1) the approach uses lightweight user annotations to improve the relevance of the generated reports, and can provide semantics-related diagnostic information (e.g., which object escapes which transaction), and (2) it is designed in a multi-tier way so that programmers at different levels of skill and code familiarity can use it to identify performance problems. Our experience shows that the three-tier technique is quite effective: the tool can help a programmer identify the root cause of a leak after an iterative process of specifying, inferring, and checking object lifetime properties. For the case studies we conducted, the tool provided more relevant information than an existing memory leak detector Sleigh. For example, using the tool, we quickly found the root causes of memory issues in the analyzed programs, including both known memory leaks and problems that have not been reported before. While LeakChaser incurs relatively large overhead (2.9× on average), we found it acceptable for debugging and performance tuning.

The assertions and transaction constructs may be easily incorporated into the Java language so that they can be used in a production JVM to help detect memory problems. Similarly to functional specifications that are used widely in program testing and debugging, writing performance specifications can have significant advantages. For example, it could enable *unit performance testing* that can identify performance violations even before degradation is observed. In addition, performance specifications (similar to the ones described in this paper) may help bridge the gap between performance analysis and the large body of work on model checking and verification, so that one may be able to prove (statically) a program is "bloat-free" or "bloat-bounded" with respect to the performance specifications provided.

A possible direction of future work would be to investigate such relationships in order to prevent small performance issues in the early stage of software development, before they pile up and become significant.

Acknowledgments

We thank the PLDI reviewers for their valuable and thorough comments. The initial idea for this work was inspired by a conversation that Guoqing Xu had with Nick Mitchell during an internship with IBM T. J. Watson Research Center. This material is based upon work supported by the National Science Foundation under CAREER grants CCF-0546040 and CCF-0953759, grant CCF-1017204, and by an IBM Software Quality Innovation Faculty Award. Guoqing Xu was supported in part by an IBM Ph.D. Fellowship Award.

References

[1] *The Jikes Research Virtual Machine*, 2011. `http://jikesrvm.org`.

[2] *JML Reference Manual*, 2010. `http://www.jmlspecs.org`.

[3] E. E. Aftandilian and S. Z. Guyer. GC Assertions: Using the garbage collector to check heap properties. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 235–244, 2009.

[4] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 739–753, 2010.

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

[6] S. M. Blackburn and K. S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 22–32, 2008.

[7] 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 *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, 2006.

[8] M. D. Bond and K. S. McKinley. Bell: Bit-encoding online memory leak detection. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 61–72, 2006.

[9] M. D. Bond and K. S. McKinley. Leak pruning. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 277–288, 2009.

[10] M. D. Bond and K. S. McKinley. Tolerating memory leaks. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 109–126, 2008.

[11] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 3–12, 2009.

[12] J. Burnim and K. Sen. Determin: Inferring likely deterministic specifications of multithreaded programs. In *International Conference on Software Engineering (ICSE)*, pages 415–424, 2010.

[13] CA Technologies. CA Wily Introscope LeakHunter. `www.ca.com/us/application-management.aspx`.

[14] J. Clause and A. Orso. Leakpoint: pinpointing the causes of memory leaks. In *International Conference on Software Engineering (ICSE)*, pages 515–224, 2010.

[15] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 59–70, 2008.

[16] ej-technologies GmbH. JProfiler. `www.ej-technologies.com`.

[17] R. Hastings and B. Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *Winter 1992 USENIX Conference*, pages 125–138, 1992.

[18] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 156–164, 2004.

[19] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Generating object lifetime traces with Merlin. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28 (3):476–516, 2006.

[20] M. Jump and K. S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 31–38, 2007.

[21] N. Mitchell. The runtime structure of object ownership. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 74–98, 2006.

[22] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 245–260, 2007.

[23] N. Mitchell and G. Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 351–377, 2003.

[24] N. Mitchell, G. Sevitsky, and H. Srinivasan. Modeling runtime behavior in framework-based applications. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 429–451, 2006.

[25] N. Mitchell, E. Schonberg, and G. Sevitsky. Making sense of large heaps. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 77–97, 2009.

[26] N. Mitchell, E. Schonberg, and G. Sevitsky. Four trends leading to Java runtime bloat. *IEEE Software*, 27(1):56–63, 2010.

[27] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 397–407, 2009.

[28] Quest Software. JProbe. `www.quest.com/jprobe`.

[29] D. Rayside and L. Mendel. Object ownership profiling: A technique for finding and fixing memory leaks. In *International Conference on Automated Software Engineering (ASE)*, pages 194–203, 2007.

[30] C. Reichenbach, N. Immerman, Y. Smaragdakis, E. Aftandilian, and S. Z. Guyer. What can the GC compute efficiently? A language for heap assertions at GC time. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 256–269, 2010.

[31] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 408–418, 2009.

[32] A. Shankar, M. Arnold, and R. Bodik. Jolt: Lightweight dynamic analysis and removal of object churn. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 127–142, 2008.

[33] Y. Tang, Q. Gao, and F. Qin. LeakSurvivor: Towards safely tolerating memory leaks for garbage-collected languages. In *The USENIX Anual Technical Conference (USENIX)*, pages 307–320, 2008.

[34] The Standard Performance Evaluation Corporation. SPECjvm98 Benchmark Set. http://www.spec.org/jvm98/.

[35] M. Vechev, E. Yahav, and G. Yorsh. PHALANX: Parallel checking of expressive heap assertions. In *International Symposium on Memory Management (ISMM)*, pages 41–50, 2010.

[36] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *International Conference on Software Engineering (ICSE)*, pages 151–160, 2008.

[37] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 160–173, 2010.

[38] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: Profiling copies to find runtime bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 419–430, 2009.

[39] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 174–186, 2010.

[40] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *ACM SIGSOFT FSE/SDP Workshop on the Future of Software Engineering Research*, 2010.