A Detection and Measurement Approach for Memory Leaked Objects in Java Programs

Qiao YU†, Member, Shujuan JIANG†a), and Yingqi LIU†, Nonmembers

SUMMARY Memory leak occurs when useless objects cannot be released for a long time during program execution. Memory leaked objects may cause memory overflow, system performance degradation and even cause the system to crash when they become serious. This paper presents a dynamic approach for detecting and measuring memory leaked objects in Java programs. First, our approach tracks the program by JDI and records heap information to find out the potentially leaked objects. Second, we present memory leaking confidence to measure the influence of these objects on the program. Finally, we select three open-source programs to evaluate the efficiency of our approach. Furthermore, we choose ten programs from DaCapo 9.12 benchmark suite to reveal the time overhead of our approach. The experimental results show that our approach is able to detect and measure memory leaked objects efficiently.

key words: memory leak, heap information, leaked objects, memory leaking confidence

1. Introduction

Memory leak detection plays an important role in the process of software testing. Memory leak is one of the runtime faults of memory management, which refers to a situation that the allocated memory cannot be released promptly for negligence or errors when it is no longer used during program execution. However, it may be associated with the coding of programmers.

For C/C++ programs, the allocation and deallocation of dynamic memory are achieved by function calls, such as new/delete, malloc/free. However, the virtual memory in Java programs is managed by the built-in garbage collector (GC) automatically. The GC adopts a collection algorithm to reclaim the waste objects produced during program execution. But the recovery efficiency of memory may be various with different garbage collection algorithms. Therefore, many researchers focus on the improvement of garbage collection algorithms [1]–[4], but these algorithms cannot be able to reclaim the stale objects, which are useless but referenced by other objects. That is to say, garbage collection algorithms cannot avoid memory leak in practice.

The commercial profilers, such as JProfiler [5] and YourKit [6], can achieve heap snapshots, memory distribution, and even memory leak detection in Java programs. But they are not able to locate the real operations of leakage, which is essential for solving memory leaks.

In recent years, memory leak detection has drawn much attention from researchers. Generally, memory leak detection methods can be divided into static detection and dynamic detection. Static detection methods, such as context-sensitive, path-sensitive and flow-sensitive [7]–[10], are usually applied to detect memory leaks. But static detection could not reflect the run-time heap information accurately. Moreover, dynamic detection methods often focus on objects or heap growth [11]–[13], object staleness [14]–[17] and leak tolerance [18]–[21] to detect the potentially leaked objects. In fact, the efficiency of dynamic detection may be higher than that of static detection, but the overhead of dynamic detection would be increased correspondingly. Therefore, it is imperative to put forward an efficient and reasonable approach for improving the performance of memory leak detection.

In this paper, we present a dynamic approach for detecting and measuring memory leaked objects in Java programs. First, our approach focuses on such operations which may cause memory leak, then we perform these operations repeatedly to accelerate the leakage. At the same time, the execution process is tracked by Java Debug Interface (JDI) [22]. Second, we record heap information periodically and regard the growing number of objects as potentially leaked objects. Finally, we present memory leak detection and measuring the leaking influence of these objects on the program.

The main contributions of this paper are as follows:

- A new technique to obtain heap information periodically with low time overhead.
- An efficient algorithm for counting the quantity of active objects in heap.
- An approach, based on object growth and staleness, to measuring the leaking influence of different objects on the program.
- The results show that our approach is able to detect and measure memory leaked objects efficiently.

The rest of this paper is organized as follows. Section 2 gives an overview of memory leaked objects in Java programs. Section 3 introduces our approach in detail, including the detection and measurement of memory leaked objects. Section 4 discusses the empirical studies with three open-source programs and ten benchmarks to show the validity of our approach. Section 5 summarizes the related work.
work of memory leak detection in recent years, and Sect. 6 draws conclusions and discusses the future work.

2. Overview of Memory Leaked Objects

Memory leaked objects in Java programs should meet the following conditions: the first condition is that there are references pointing to this object; the second one is that this object will never be used, which is called a stale object. Existing references will avoid stale objects from being reclaimed by GC, so memory leak occurs.

**Definition 1** (Active Object): If there are references pointing to an object in heap, then the object is called active object, which cannot be reclaimed by GC automatically. In other words, it is reachable (directly or indirectly) from a GC root.

According to the usefulness of active object, they can be divided into live object and stale object. Live object shows that an object is still needed for the application, but stale object indicates that an object has not been used for a long time [23]. Therefore, active but stale objects may be the potentially leaked objects.

Figure 1 shows three different states of objects in Java programs. The symbol A represents the live objects, and they are reachable from the GC root as marked in Fig. 1. The symbol B represents the stale objects, which will become memory leaked objects along with the execution of program. The symbol C represents the unreachable objects from the GC root, and they will be reclaimed by GC automatically.

Furthermore, we choose a sample program to explain the root cause of memory leaked objects. This sample is simplified from ArgoUML [24], which is one of our experimental programs. As described in Fig. 2, there are three classes: `ProjectManager`, `Project` and `ProjectMember`. It shows the process of “New Project” operation. Whenever we perform this operation, a `Project` object is created (Line 3). Then method `makeUntitledProject()` is called (Line 4) and executed (Line 10). Finally, method `addMember()` is executed to add `Project` object to `Vector` (Line 17). Therefore, even if method `makeEmptyProject()` has finished execution, the `Project` object is not allowed to be reclaimed by GC. Because `Vector` is keeping reference to it. Accordingly, if this `Project` object would not be used during the follow execution, it will become stale object. In other words, it may be the potentially leaked object, which is the same object as symbol B described in Fig. 1.

In fact, this situation occurs frequently in real-world applications. Moreover, they would probably be ignored by the programmers, especially when the references between objects are more complicated.

Besides, memory leaks are invisible and they will not make significant impact. This also increases the difficulty of memory leak detection. This paper attempts to find the operations which could probably cause memory leaked objects, such as file open/close, window open/close and continuous “New” operation, etc. Then we only perform such operations repeatedly without any other operations, including save operations. In fact, many temporary objects will be created during this process. However, if the quantity of some objects is proportional to the number of repeated operations, we will regard them as potentially leaked objects. If one operation could cause one leaked object, then the quantity of leaked objects will gradually increase along with the repeated operations. Therefore, repeated operations could accelerate the accumulation of memory leaked objects. Thus, they will become easier to be detected. From this point of view, we propose the following approach.

3. Our Approach

First of all, we give the framework of our approach as shown in Fig. 3, and it can be classified into two phases: detection and measurement.

During the detection phase, our approach tracks the program dynamically by JDI, and it records heap information periodically (Sect. 3.1). Then, we count the active objects and find out the potentially leaked objects according to the heap information (Sect. 3.2). During the measurement phase, we present MLC, combining object growth and staleness, to measure the leaking influence of these objects on the program (Sect. 3.3). Finally, based on the results above,
we give a leak report to determining memory leaked objects.

3.1 Recording Heap Information

**Definition 2 (Execution Trace):** The statements set of program \( P: S = \langle s_1, s_2, s_3 \ldots, s_m \rangle \). We give a unique timestamp \( (ts) \) for each statement of \( S \), marked as \( s^i_\tau \), which stands for the chronological sequence of statements produced along with the program execution, then the execution trace \( (ET) \) of \( P: ET = \langle s^1_\tau, s^2_\tau, s^3_\tau \ldots, s^m_\tau \rangle \).

In this step, we find out the potentially leaked operations and perform them repeatedly. It is tracked by JDI all the time, then we record heap information periodically, including class name, method name, execution time, memory usage, etc. Moreover, we can also obtain the object ID, object size and object quantity of each type according to the heap information.

In addition, periodically recording refers to the heap information that we record with a periodic length of \( ET \) each time, thereby reducing the frequency of information collection. However, the periodic length of \( ET \) is depended on the program itself. Therefore, it may vary on different programs.

3.2 Analyzing Potentially Leaked Objects

During this process, the most critical part is to count the active objects of each type in heap. Its detailed process is as shown in Algorithm 1.

We can obtain the real-time thread reference tracked by JDI during program execution. In fact, \( ThreadReference \) represents a thread object on the virtual machine, which contains the real-time frames in heap. Actually, each frame represents a dynamic method call. At the same time, we can obtain the variables or objects of each frame. Therefore, thread reference is the key to obtain objects information in heap.

First, our algorithm obtains the frames of \( ThreadReference \) and saves them to \( FrameList \) (Line 1). After that, it traverses \( FrameList \) to get the variables of each frame (Line 3). Second, it obtains the value of each variable (Line 5) to determine whether a variable is primitive value or reference type. If the type of a variable is reference type, it saves this type into \( ReferTypeList \) without repetition (Lines 6-8).

Third, it also traverses \( ReferTypeList \) to get all instances of each reference type (Lines 12-14). In fact, these instances are exactly the active objects in heap. Finally, it returns \( ObjectsList \) (Line 15), which saves all active objects of each type in heap.

We can count all active objects of each type in heap by Algorithm 1. While, the growing number of active objects are probably the stale objects. Therefore, we regard these objects as potentially leaked objects. Next, we calculate \( MLC \), combining object growth and staleness, to measure the leaking influence of these objects on the program.

3.3 Calculating \( MLC \)

\( MLC \) is proposed to measure the leaking influence of potentially leaked objects on the program. Its calculation is divided into two steps: (1) calculating the staleness rate \( (R_s) \) of potentially leaked objects; (2) calculating \( MLC \) of these objects.

Firstly, we assume that \( Obj \) represents an object allocated in memory. Then, we give the following explanations: \( TS_1 \) stands for the execution sequence when \( Obj \) has...
The definition of $R_s$ defined in [17]. The difference is that we add the $n$ parameter to improve the leaking influence of object growth.

$$MLC = R_s \times \left(\frac{N \times MS}{CM}\right)^{1-R_s}$$

The value of $MLC$ is also between 0 and 1. It combines object growth and staleness to measure the leaking influence on the program. Generally, the higher $R_s$ and the larger of $n$ are, the more obvious memory leaking influence is.

4. Experimental Evaluation

4.1 Experiments

To evaluate the efficiency of our approach, we choose three open-source programs and ten programs from DaCapo 9.12 benchmark suite [26]. Our experiments are conducted on Intel Core2 Duo processor, with 2.94GHz and 2GB of RAM, running with Windows 7 - 64bits and Sun JDK 1.7.

4.1.1 ArgoUML

ArgoUML is an open-source UML modeling tool [24]. We choose 0.14 version in our experiment, and we perform the common operation of “New Project” for 100 times repeatedly without any other operations. We find out that the quantity of five kind of objects increases in proportion with the number of “New Project” operations. Because we have not performed any save operations, it should not be increased in the quantity of such objects. Thus, we regard these growing number of objects as potentially leaked objects.

The heap information statistics of ArgoUML are shown in Table 1. The first column represents the execution sequence of program. It indicates that we record the heap information whenever it executes 350000 lines. In other words, the periodic length of $ET$ is 350000. The second column shows current execution time, which will depend on the periodic length of $ET$ and our actual operations. The third column is the same meaning as CM in formula (2). The rest of columns show the quantity of five potentially leaked objects. They (Obj1, Obj2, Obj3, Obj4 and Obj5) represent the object of Project, ToolBar, UMLClassDiagram, EventListenerList and PopupToolBoxButton respectively.

Especially, we conduct several experiments with different periodic length of $ET$ (e.g. 35000, 350000, 3500000). We all perform the same “New Project” operation for 100 times during these experiments. When the periodic length of $ET$ is equal to 35000, we track 100 sets of heap information. But it takes more than six hours. When it is equal to 350000, we track 100 sets of heap information. Similarly, when it is equal to 3500000, we only track one set of heap information, which is not enough to detect leaked objects. Therefore, we choose 350000 as the periodic length of $ET$ eventually. However, the periodic length of $ET$ is not unique.

As is shown in Table 1, the quantity of each object increases obviously along with 100 repeated operations. The
quantity of Obj5 is about three times as many as the number of operations. But the other four is basically the same as the number of operations. Particularly, our approach is designed to record heap information periodically, and we execute the same operations repeatedly in order to track more objects. However, the tracked time of different objects may be various. When one object is tracked, there may have existed many instances of this object in heap. Therefore, we can see that several initial values of columns Obj1-Obj5 are zero. In fact, it means that they have not been tracked yet rather than there does not exist such objects in heap.

Then, we give the leak report of ArgoUML in detail. First of all, Table 2 shows the locations of potentially leaked objects in ArgoUML. It shows object name, class name, method name (Columns 2-4) and instantiated line of object (Column 5). In addition, there are two instantiated locations method name (Columns 2-4) and instantiated line of object (Column 5). In Table 2, the first four objects (Obj1, Obj2, Obj3, Obj4) are also obvious. We can conclude that these five kind of objects are all leaked object under repeated “New Project” operations. Moreover, we also find that this leakage will not occur in ArgoUML 0.15 version and above.

| Table 1 | The heap information statistics of ArgoUML. |
|---------|------------------------------------------|
| TS      | CT(s)         | CM(MB) | Obj1 | Obj2 | Obj3 | Obj4 | Obj5 |
|---------|--------------|--------|------|------|------|------|------|
| 350000  | 220.84       | 4.5    | 0    | 0    | 0    | 0    | 0    |
| 700000  | 380.28       | 4.23   | 0    | 0    | 0    | 0    | 48   |
| 1050000 | 544.34       | 6.03   | 27   | 28   | 27   | 0    | 81   |
| 1400000 | 747.98       | 6.43   | 39   | 40   | 39   | 40   | 116  |
| 1750000 | 890.25       | 7.93   | 50   | 50   | 49   | 49   | 147  |
| 2100000 | 1088.63      | 6.72   | 60   | 60   | 59   | 60   | 177  |
| 2450000 | 1281.28      | 7.69   | 69   | 70   | 69   | 69   | 207  |
| 2800000 | 1484.69      | 9.67   | 79   | 79   | 78   | 79   | 234  |
| 3150000 | 1678.20      | 7.81   | 89   | 89   | 88   | 88   | 264  |
| 3500000 | 1879.69      | 11.84  | 100  | 100  | 99   | 100  | 297  |

| Table 2 | The locations of potentially leaked objects in ArgoUML. |
|---------|-----------------------------------------------------|
| Object Name | Class Name | Method Name | Line |
| Obj1 | Project | org.argouml.kernel.ProjectManager | makeEmptyProject() | 200 |
| Obj2 | ToolBar | org.argouml.uml.diagram.ui.UMLDiagram | initToolBar() | 194 |
| Obj3 | UMLClassDiagram | org.argouml.kernel.Project | makeUntitledProject() | 214 |
| Obj4 | EventListenerList | org.argouml.model.uml.umlModelEventPump | put() | 900 |
| Obj5 | PopupToolBoxButton | org.argouml.uml.diagram.ui.UMLDiagram | buildPopupToolBoxButton() | 254&264 |

| Table 3 | The leakage of ArgoUML. |
|---------|-------------------------|
| Object Name | TS1 | TS2 | TS3 | Rr | N | n | MLC |
| Obj1 | Project | 279849 | 344328 | 3803468 | 0.9817 | 100 | 1 | 0.8565 |
| Obj2 | ToolBar | 303843 | 356691 | 3803468 | 0.9849 | 100 | 1 | 0.8801 |
| Obj3 | UMLClassDiagram | 286193 | 351005 | 3803468 | 0.9816 | 100 | 1 | 0.8558 |
| Obj4 | EventListenerList | 283437 | 348051 | 3803468 | 0.9849 | 100 | 1 | 0.9539 |
| Obj5 | PopupToolBoxButton | 304009 | 356857 | 3803468 | 0.9849 | 100 | 1 | 0.9539 |

Because the value of MLC is between 0 and 1, judging from its values in Table 3, the leaking influence of Obj5 is the most serious. While, the leaking influence of other four objects (Obj1, Obj2, Obj3, Obj4) is also obvious. We can conclude that these five kind of objects are all leaked object under repeated “New Project” operations. Moreover, we also find that this leakage will not occur in ArgoUML 0.15 version and above.

4.1.2 JFlex

JFlex is a scanner generator for Java [27]. We choose the version of 1.4.3. In our experiment, we perform “Browse” operation repeatedly to select the file path for 100 times. The results show that the quantity of object FileDialog keeps increasing all the time.

The heap information statistics of JFlex are shown in Table 4, column TS means that the periodic length of ET is 350. Similarly, we conduct several experiments with different periodic length of ET. Columns CT and CM are remain the same meaning as shown in Table 1. The last column represents the quantity of object FileDialog, which is consistent with the number of operations. Additionally, we find out that object FileDialog is created in class JFlex.gui.MainFrame, method MainFrame() of line 261 according to the heap information.

In addition, we choose different value of N for comparison, but the value of n is 1, which is not changed along with the value of N. The results are as shown in Table 5, and the value of MS is also 72 byte. Column CM represents...
the current memory usage, and other columns are the same meaning as shown in Table 3. When \( N \) is larger, the value of \( R_s \) and \( MLC \) will be larger. In terms of the execution process, the longer the execution time of a certain program is, the larger the value of \( R_s \) is. Therefore, the actual leakage results are consistent with our measurement.

4.1.3 Portecle

Portecle is a GUI application for creating, managing and examining key stores, keys, certificates, etc. [28]. We choose the version of 1.8 in our experiments. By tracking its execution, we detect that continuous “New Keystore Type” operations could lead to certain objects increased. Similarly, we conduct this operation for 100 times continuously without any other operations. The results show that the quantity of five kinds of objects keep increasing, which will be regarded as potentially leaked objects. However, the quantity of them would increase first and then fluctuate, rather than increasing gradually as we have expected.

In order to more accurately describe the increasing of each object, we choose 1000 as the periodic length of \( ET \). At last, we record 48 sets of heap information after 100 operations. From the statistics of heap information, we can conclude that the quantity of \( Obj1, Obj2 \) and \( Obj3 \) remain the same all the time. However, the quantity of \( Obj4 \) and \( Obj5 \) increase more significantly.

Figure 5 describes the quantity of five objects. The x-axis represents the timestamp of execution sequence, and the y-axis represents the quantity of objects. As can be seen from Fig. 5, the quantity of all objects is increasing during the first part of the process (\( TS=[0,24000] \)), and fluctuating during the second part (\( TS=[24000,48000] \)).

Table 6 shows the locations of these objects in Portecle, and Table 7 shows the results of \( MLC \), which are the same meaning of each column in Table 2 and Table 3 respectively. The value of \( MS \) is also 72 byte, and the value of \( CM \) is 25.97MB when \( TS \) is 24000. It should be noted that the value of \( n \) is uncertain during the second half, so we only calculate \( MLC \) of the first part as described in Table 7. However, there are multiple locations of \( Obj4 \) and \( Obj5 \), we only choose one of them for calculation which will not affect the value of \( MLC \) obviously.

According to the values of \( MLC \) in Table 7, the leaking influence of these objects are all significant during the first part of the process, especially \( Obj1 \) and \( Obj5 \). But we can predict that the leaking influence of second part could not be various because of the fluctuation in the quantity of objects.

4.2 Discussion

To illustrate the efficiency of our approach, we make comparison with specific related works.

For ArgoUML0.14, Chen [29] also found that repeated “New Project” operations would cause memory leak. But it only detected Project as leaked object, as described in Fig. 2. However, our approach detects five kinds of leaked objects, which contain the Project. Moreover, we prove that memory leak will also occur in JFlex and Portecle under certain operations, which have not been detected by other methods.

Table 8 describes the execution time of ArgoUML, JFlex and Portecle. Column \( OT \) shows the original execution time of them for 100 repeated operations as mentioned.
YU et al.: A DETECTION AND MEASUREMENT APPROACH FOR MEMORY LEAKED OBJECTS IN JAVA PROGRAMS

Table 7 The leakage of Portecle.

| Object Name          | TS1 | TS2 | TS3   | RM   | N | n | MLC   |
|----------------------|-----|-----|-------|------|---|---|-------|
| Obj1 DNewKeyStoreType| 2295| 2461| 48325 | 0.9964 | 50 | 1 | 0.9649 |
| Obj2 JLabel          | 2301| 3202| 48325 | 0.9804 | 50 | 1 | 0.8229 |
| Obj3 ButtonGroup     | 2396| 3251| 48325 | 0.9814 | 50 | 1 | 0.8312 |
| Obj4 JButton         | 2432| 3285| 48325 | 0.9814 | 100| 2 | 0.9000 |
| Obj5 JPanel          | 2408| 3263| 48325 | 0.9814 | 200| 4 | 0.9476 |

Table 8 Execution time of programs.

| Program   | OT (s) | JT (s) |
|-----------|--------|--------|
| ArgoUML   | 25.95  | 2217.72|
| JFlex     | 73.78  | 142.94 |
| Portecle  | 71.38  | 1289.15|

Table 9 Time overhead of benchmarks.

| Program | OT (s) | JT (s) | %OH |
|---------|--------|--------|-----|
| avorra  | 4.43   | 4.90   | 11% |
| eclipse | 58.67  | 106.91 | 82% |
| fop     | 3.87   | 5.71   | 48% |
| jython  | 12.50  | 16.94  | 36% |
| luindex | 2.07   | 2.19   | 6%  |
| lusearch| 5.05   | 8.80   | 74% |
| pmd     | 6.74   | 8.68   | 29% |
| sunflow | 6.91   | 9.75   | 41% |
| tomcat  | 7.14   | 8.93   | 25% |
| xalan   | 6.04   | 8.36   | 38% |

above. Column JT shows the tracked execution time by JDI, which depends on the efficiency of our approach and the number of our operations. The time overhead of our programs is reasonable, which cannot affect the normal execution of programs.

Especially, our approach records heap information with different periodic length of ET, while [17] detects leaked container objects with different GC rate. Additionally, the typical commercial tool - JProfiler, achieves memory leak detection by taking heap snapshots periodically. That is to say, different approaches would take various measures to collect information. Therefore, there is no uniform measurement methodology across different approaches at present, which will increase the difficulty of comparison work [23].

To further reveal the time overhead of our approach, we choose ten benchmarks for experiments. The results are shown in Table 9, Column OT shows the original execution time of ten benchmarks, and Column JT shows the tracked execution time by JDI, which may be various under different hardware environment. During our experiments on benchmarks, we only track their executions by JDI, but not recording any heap information. However, the values of JT would be increased with different periodic length of ET to record heap information. Column %OH represents the time overhead of benchmarks under our approach, and they may be various depending on the program. Generally, the results of time overhead are basically reasonable to some extent.

Besides, we also conclude that memory leak is related to the execution path of programs. In other words, it is related to the selection of test cases. Actually, the “New Project”, “Browse” and “New Keystore Type” operations are equivalent to the test cases of our experiments. Because, these operations will determine the execution path of programs. Therefore, only by choosing appropriate test cases can we achieve efficient memory leak detection.

Additionally, the value of periodic length will cause a range of effects. First, it determines the execution time of program directly. The longer the periodic length is, the shorter the execution time is, and vice versa. Second, it will affect the leak detection rate. We set the periodic length of ET to control the collected frequency of heap information. The shorter the periodic length of ET is, the more accurate the heap information we will get, so we can detect more memory leaked objects. In this case, it will reduce the execution performance of programs. Conversely, the leak detection rate will be reduced when the periodic length is longer enough.

4.3 Threats to Validity

We recognize that there are several threats to the validity of our experiments described as follows:

(1) The validity of heap information

We obtain the heap information periodically by JDI. In our experiment, we execute the same operations repeatedly to track more memory leaked objects, but we cannot guarantee that all leaked objects would be tracked.

(2) The periodic length of ET

It will affect the leak detection rate and performance of programs. So we set different periodic length of ET for many experiments, and finally we select the more reasonable results as defined in Sect. 4.1.

(3) The rationality of our measurement

When the quantity of a certain object is large, we only choose the first instantiated object to calculate RM in our experiments. Actually, the result will be slightly larger than its actual value. But it has no significant effect on the value of MLC.

(4) The selection of experimental programs

In our experiments, we perform the same operation for 100 times to accelerate the leakage. This technique is also used in [29], [30]. Therefore, our experimental programs are all GUI applications for the convenience of repeated operations. Above all, the applications should be finished in a limited time. Conversely, if applications could not be finished in a limited time, we can select one or more periods of execution process for detection and measurement.

5. Related Work

Memory leak has drawn much attention from researchers in recent years. How to accurately and efficiently identify
memory leaked objects is the emphasis and difficulty of current research. The main methods can be divided into static detection and dynamic detection.

5.1 Static Detection

Static analysis methods are widely applied in memory leak detection. Xie and Aiken [7] present a leak detection algorithm based on escape analysis, which considers context sensitive and path sensitive using boolean constraints. Xu and Zhang [8] present a static inter-procedural analysis method based on path sensitive and context sensitive, and they introduce escape analysis and symbolic execution to building memory object model. Xu et al. [9] present a static inter-procedural detection algorithm, which adopts path-sensitive symbolic execution to capture the memory actions and path conditions. Additionally, they propose an analysis model, which is called Memory State Transition Graph (MSTG), to describe the tracking process and results. An MSTG consists of nodes and edges. Nodes represent the state of objects, and edges represent the path conditions.

Compared with static detection methods, our approach focuses on the run-time heap information which is more accurate for memory leak detection. Through our experiments, we also find that memory leak is quite related to the execution path of programs.

5.2 Dynamic Detection

Dynamic detection is a method of analyzing the execution behavior by running program itself, which is also the main approach for memory leak detection.

Generally, memory leaks will cause the growth of objects and heap. Jump and McKinley [11], [12] implement a dynamic heap-summarization technique in Cork. It presents a type points-from graph (TPFG) to describe types and references. Then, the growth of data structures will be detected by comparing TPFGs. LeakBot [13] is achieved to detect potentially leaked data structures, and they will be ranked by their likelihood of containing memory leaks.

The state of objects is the key to reflect memory management errors. Therefore, many researchers focus on the staleness of objects to detect memory leaks. At the same time, these approaches also focus on the allocation and deallocation of objects. Hauswirth and Chilimbi [14] implement a detection tool-SWAT, which traces allocations/frees to build a heap model and adopts adaptive profiling model to monitor loads/stores of objects. Moreover, it reports stale objects which have not been accessed recently as leaked objects. Bond and McKinley [15] also focus on stale objects which are not used for a long time, and they will get the allocation and last-use sites of objects. Xu and Rountev [17] present a heap-tracking technique based on container. It tracks the containers instead of objects, and then it computes the confidence for each container to measure the staleness and memory contribution to system memory leaks. However, our approach focuses on both the growth of objects quantity and object staleness, which makes it more efficient to detect and measure memory leaked objects in Java programs.

Additionally, leak tolerance is also proposed to tolerate the occurrence of memory leaks. Dai et al. [18] present an automatic resource collection approach to tolerate non-memory resource leaks. Nguyen and Rinard [19] present a technique of cyclic memory allocation to find m-bounded allocation sites. If the difference between allocated and deallocated objects from one site grows above m, then there is a memory leak at this site. Tang et al. [20] present a safe method to swap out potentially leaked objects periodically from the virtual memory to disks. Then, more memory space can be reclaimed by GC. But when a swapped-out object is accessed later, it will restore the object from disks to memory to keep the normal execution of program. However, these methods will cause additional overhead of detection to a certain extent.

Above all, the main idea of dynamic detection is to obtain the allocation and usage information of objects. Then we can find memory leaked objects or data structures according to these information. But dynamic execution may increase the performance overhead of program. In our experiment, we set the periodic length of ET to coordinate the performance overhead and memory leak detection rate.

6. Conclusions and Future Work

This paper presents an effective approach for detecting and measuring memory leaked objects in Java programs. It tracks the execution of programs by JDI and records heap information periodically to find out the potentially leaked objects. After that, we present MLC to measure the leaking influence of these objects on the program. Furthermore, we choose three open-source programs and ten benchmarks to evaluate the efficiency and time overhead of our approach respectively. The results show that our approach is able to detect and measure memory leaked objects efficiently.

In addition, our approach could not be fully applied to multi-threaded programs. We will carry out new programs to discover more problems and continuously improve the performance of our approach in the future.

Acknowledgments

This work was supported partly by the National Natural Science Foundation of China [No.60970032] and the 333 Project.

References

[1] D.F. Bacon, P. Cheng, and V. Rajan, “A real-time garbage collector with low overhead and consistent utilization,” ACM SIGPLAN Notices, vol.38, no.1, pp.285–298, 2003.
[2] M. Hertz, Y. Feng, and E.D. Berger, “Garbage collection without paging,” ACM SIGPLAN Notices, vol.40, no.6, pp.143–153, 2005.
[3] U. Ramachandran, N. Harel, A. Mandriwala, and K. Knobe, “Distributed garbage collection algorithms for timestamped data,” IEEE
YU et al.: A DETECTION AND MEASUREMENT APPROACH FOR MEMORY LEAKED OBJECTS IN JAVA PROGRAMS

[4] S.M. Blackburn and K.S. McKinley, "Inmix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance," ACM SIGPLAN Notices, vol.43, no.6, pp.22–32, 2008.

[5] JProfiler. http://www.ej-technologies.com/products/jprofiler/overview.html

[6] Yourkit. http://www.yourkit.com/overview/

[7] Y. Xie and A. Aiken, "Context-and-path-sensitive memory leak detection," ACM SIGSOFT Software Engineering Notes, vol.30, no.5, pp.115–125, 2005.

[8] Z. Xu and J. Zhang, "Path and context sensitive inter-procedural memory leak detection," Proc. International Conference on Quality Software, Oxford, UK, pp.412–420, Aug. 2008.

[9] Z. Xu, J. Zhang, and Z. Xu, "Memory leak detection based on memory state transition graph," Proc. Asia Pacific in Software Engineering Conference, Ho Chi Minh city, Vietnam, pp.33–40, Dec. 2011.

[10] Y. Sui, D. Ye, and J. Xue, "Static memory leak detection using full-sparse value-flow analysis," Proc. International Symposium on Software Testing and Analysis, Minneapolis, MN, pp.254–264, July 2012.

[11] M. Jump and K.S. McKinley, "Cork: Dynamic memory leak detection for garbage-collected languages," ACM SIGPLAN Notices, vol.42, no.1, pp.31–38, 2007.

[12] M. Jump and K.S. McKinley, "Detecting memory leaks in managed languages with Cork," Software: Practice and Experience, vol.40, no.1, pp.1–22, 2010.

[13] N. Mitchell and G. Sevitsky, " LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications," ECOOP 2003 Object-Oriented Programming, pp.351–377, 2003.

[14] M. Hauswirth and T.M. Chilimbi, "Low-overhead memory leak detection using adaptive statistical profiling," ACM SIGPLAN Notices, vol.39, no.11, pp.156–164, 2004.

[15] M.D. Bond and K.S. McKinley, "Bell: bit-encoding online memory leak detection," ACM Sigplan Notices, vol.41, no.11, pp.61–72, 2006.

[16] M.D. Bond and K.S. McKinley, "Tolerating memory leaks," ACM Sigplan Notices, vol.43, no.10, pp.109–126, 2008.

[17] G. Xu and A. Rountev, "Precise memory leak detection for java software using container profiling," Proc. International Conference on Software Engineering, Leipzig, Germany, pp.151–160, May 2008.

[18] Z. Dai, X. Mao, Y. Lei, X. Wan, and K. Ben, "Resco: Automatic collection of leaked resources," IEICE Trans. Inf. & Syst., vol.E96-D, no.1, pp.28–39, Jan. 2013.

[19] H.H. Nguyen and M. Rinard, "Detecting and eliminating memory leaks using cyclic memory allocation," Proc. International Symposium on Memory Management, Montreal, Canada, pp.15–30, Oct. 2007.

[20] Y. Tang, Q. Gao, and F. Qin, "LeakSurvivor: Towards safely tolerating memory leaks for garbage-collected languages," Proc. USENIX Annual Technical Conference, pp.307–320, Boston, Massachusetts, June 2008.

[21] G. Novark, E.D. Berger, and B.G. Zorn, "Plug: automatically tolerating memory leaks in C and C++ applications," University of Massachusetts, 2008.

[22] Java Debug Interface. http://www.ibm.com/developerworks/cn/java/j-lo-jpda4/

[23] V. Šor and S.N. Srirama, "Memory leak detection in Java: Taxonomy and classification of approaches," J. Systems and Software, 2014.

[24] ArgoUML. http://argouml.tigris.org/

[25] EcIEmma. http://eclemma.org/

[26] S.M. Blackburn, R. Garner, C. Hoffmann, A.M. Khang, K.S. McKinley, R. Bentzur, et al., "The DaCaPo benchmarks: Java benchmarking development and analysis," ACM SIGPLAN Notices, vol.41, no.1, pp.169–190, 2006.

Qiao Yu was born in 1989. She is a Ph.D. candidate at School of Computer Science and Technology, China University of Mining and Technology. Her research interest is software analysis and testing.

Shujuan Jiang was born in 1966. She is a professor and Ph.D. supervisor at School of Computer Science and Technology, China University of Mining and Technology. Her research interests include compilation techniques and software engineering, etc.

Yingqi Liu was born in 1992. She is an undergraduate student at School of Computer Science and Technology, China University of Mining and Technology.