IEICE TRANS. INF. & SYST., VOL.E96–D, NO.1 JANUARY 2013

PAPER

Resco: Automatic Collection of Leaked Resources

Ziying DAI[†a)], ***Student Member***, **Xiaoguang MAO**[†b)], ***Nonmember***, **Yan LEI**[†], ***Student Member***, **Xiaomin WAN**[†], ***and*** **Kerong BEN**[††], ***Nonmembers***

SUMMARY A garbage collector relieves programmers from manual memory management and improves productivity and program reliability. However, there are many other finite system resources that programmers must manage by themselves, such as sockets and database connections. Growing resource leaks can lead to performance degradation and even program crashes. This paper presents the automatic resource collection approach called Resco (RESource COllector) to tolerate non-memory resource leaks. Resco prevents performance degradation and crashes due to resource leaks by two steps. First, it utilizes monitors to count resource consumption and request resource collections independently of memory usage when resource limits are about to be violated. Second, it responds to a resource collection request by safely releasing leaked resources. We implement Resco based on a Java Virtual Machine for Java programs. The performance evaluation against standard benchmarks shows that Resco has a very low overhead, around 1% or 3%. Experiments on resource leak bugs show that Resco successfully prevents most of these programs from crashing with little increase in execution time.

key words: *resource leaks, resource collection, fault tolerance, monitoring*

1. Introduction

Automatic garbage collection has gained considerable success in many mainstream programming languages, such as Java and C#. Garbage collector relieves programmers from manual memory management and improves productivity and program reliability [1]. However, there are many other non-memory, finite system resources, such as file descriptors and database connections, whose management garbage collector does not help with. In programs written in Java-like languages, once acquired, a resource instance must be released by explicitly calling a cleanup method. A *resource leak* is a software bug that occurs when the cleanup method of the resource is not invoked after the last use of the resource. Growing resource leaks can hurt application performance and even result in system crashes due to resource exhaustion.

Resource leaks are common in Java programs [2]. It is difficult for programmers to correctly release all resources along all possible exceptional paths. Implicit control flows introduced by exceptions also impose difficulties on traditional testing and analysis techniques [3]. Current approaches either strive to detect and fix these leaks or provide new language features to simplify resource management [2], [3], [5]–[8]. Although these approaches can achieve success to some extent, there are some escaped resource leaks that manifest themselves after programs have been deployed. We also note that similar to memory leaks, leaks of other resources are not necessarily a problem. A few or a small amount of leaked resources will neither affect program behavior nor performance. Only when accumulated leaks exhaust all available resources or lead to large extra computation overhead, a system failure occurs. Based on these two observations, we propose the automatic resource collection approach to enforce resource limits and tolerate resource leaks. Resource collections are triggered just when there are so many leaked resources that the system is about to crash or its performance is about to degrade. Let us take database connections as an example. We can collect leaked connections at the limit of the connection pool size to avoid performance degradation due to establishing extra physical connections, and also at the limit of the maximum number of concurrent connections to prevent database crashes. Since opening physical connections is time-consuming, connections should be closed as soon as possible. In such cases, current solutions such as leak detection and fixing can benefit and are complementary to ours.

This paper presents the Resco approach to automatically collect leaked non-memory resources in response to abnormal resource consumption. When the program approaches a resource limit, the resource collection process is triggered. First, Resco identifies leaked resources. Among un-released resources, unreachable ones are definitely leaked. However, according to the leak definition above, there may be leaked resources that are still reachable. Because determining whether an object is live (will be used later) or not is un-decidable in general, most resource leak detection approaches including garbage collectors target unreachable resources [2], [3], [8]. Resco also employs this approach to identify leaked resources as unreleased and unreachable resources. Second, corresponding cleanup methods such as `close` are invoked to safely release these leaked resources. Resco is analogous to the finalization mechanism since both aim at reclaiming unreachable resources. However, the execution of `finalize` methods may be arbitrarily delayed in an indeterminate way [9], which makes it well known that finalization is unqualified to perform resource collections. There are two main rea-

Manuscript received May 2, 2012.
Manuscript revised October 4, 2012.

[†]The authors are with School of Computer, National University of Defense Technology, 410073, Changsha, China.
[††]The author is with the Department of Computer Engineering, Naval University of Engineering, 430033, Wuhan, China.
a) E-mail: ziyingdai@nudt.edu.cn
b) E-mail: xgmao@nudt.edu.cn (Corresponding author)

DOI: 10.1587/transinf.E96.D.28

sons: (1) `finalize` methods are bound to garbage collector that may not run until the application is about to run out of memory. However, the application may be about to exhaust some non-memory resources or may suffer performance degradation due to huge resource consumption while there is still a large amount of available memory; (2) Various finalization implementations do not always execute `finalize` methods immediately when they are ready to be called [9]. Asynchronous finalization is a necessary feature for correct implementation, but the situation becomes worse because of delayed invocations of ready `finalize` methods. Resco improves the situation based on two design decisions. First, Resco separates non-memory resource collections from memory collections. Resource collections are triggered in response to abnormal non-memory resource consumption independently of memory usage. For the flexibility to enforce various limits and to handle new application-specific resources, we instrument monitors into application code and system libraries to count the resource consumption and request resource collections if necessary. Second, the separate thread to release leaked resources is given the privilege to run immediately after the liveness analysis. We ensure the safety by releasing only leaked resource instances that are not depended on by any object with actions (e.g. `close` or `finalize`) to perform in future and that do not reference any reachable instances of the collection-triggering resource or its wrappers.

We implement Resco based on Jikes RVM [11] for Java programs. Users provide Resco with resource releasing specifications through a configuration file that includes resource types, resource acquiring and releasing methods, and resource limits. Resco can handle new application-specific resources that have a limited amount available to programs. We evaluate Resco's performance through standard benchmarks. The experimental results show that the runtime overhead of Resco is very low, around 1% or 3%. To evaluate Resco's ability to collect leaked resources, we conduct experiments on four leaks. Resco successfully tolerates three of these four leaks and reclaims all concerned leaked resources. Resco performs stably in the long term with variations when resource collections are triggered. Resco cannot tolerate the fourth leak because its leaked resources are still reachable.

The rest of this paper is organized as follows. Section 2 presents several concepts about resources to facilitate our discussions and to drive the Resco approach. In Sect. 3, we give the details of our Resco approach. The implementation of Resco for Java programs is presented in Sect. 4. Section 5 presents the experimental evaluation. Related work is presented in Sect. 6. Finally, we conclude in Sect. 7.

2. Concepts

Resco reclaims leaked resources in a demand-driven way. So the first question is to count resource consumption. This is not as straightforward as what we may imagine. First, we can look at many resources from different viewpoints and thus impose different limits on them. These limits can not necessarily be mapped to low-level system resources. For example, database connections typically have two limits: the size of the connection pool and the maximum number of concurrently available connections. These two limits usually have small numbers and are set for performance considerations. Second, the common resource we refer to may not be the one on which the system directly imposes limits. Moreover, one limited resource may be required by several other resources and one resource may require several different limited resources. Let us take the file descriptor as an example. All examples in this paper are provided in the context of Java except explicitly stated. We know that `FileInputStream` is a finite resource whose instances should be closed after use. However, its background limited resource is the file descriptor that the operating system usually imposes a constraint on its maximum available number. One instance of `FileInputStream` occupies one file descriptor. Knowing just these is not enough, because the file descriptor is also the background resource of `FileOutputSteam`, `Socket`, etc.

To facilitate discussions and to drive our Resco approach, we give several concepts below. The concrete resource introduced below is to denote resources on which limits are directly imposed, and the concept of abstract resource is to characterize resources at the programming level. Resource-related concepts can also be found in [6], [23], but we give our own ones here for the specific purpose of collecting leaked resources.

DEFINITION 1. *(**Concrete Resource**) A concrete resource is a hardware or software entity whose amount available to a program is limited. A concrete resource CR is formally specified as a pair ⟨N, L⟩, where N is its unique identifier and L is its limit.* Examples of concrete resources include the operating system managed hardware resources such as memory and disk space, and software entities such as file descriptors and database connections. We focus on non-memory finite resources in this paper due to the fact that garbage collectors are widely used and there is so much good research work on memory leaks [4], [20]–[22]. The goal of the concrete resource concept is to count resource consumption and enforce resource limits. The notable characteristic of a concrete resource is that from some viewpoint, the system/application directly imposes limits on it, that is, it is limited not because they require other limited resources. For example, the file descriptor is a concrete resource, while at the same time `FileInputStream` is not a concrete resource. The limit of a concrete resource is naturally set in the system configuration and/or application administration and is imposed by the application or its hosting environment such as the operating system, the virtual machine or other applications for some functionality or performance reasons. If one such limit is violated, the application will perform poorly and even crash. The goal of Resco is to enforce limits of concrete resources via automatic resource collections.

DEFINITION 2. *(**Abstract Resource**) An abstract resource is a data type that wraps some concrete resources. Its*

instances are allocated to programs by some API methods called acquiring methods and de-allocated by some other API methods called releasing methods. An acquiring (releasing) method acquires (releases) some amount v_a (v_r) of a wrapped concrete resource CR. We specify an acquiring (releasing) method M_{CRa} (M_{CRr}) as a pair $\langle m_{CRa}, v_a \rangle$ ($\langle m_{CRr}, v_r \rangle$), where m is this methods fully-qualified signature and CR is the concrete resource acquired (released) by this method. M_{CRa} and its corresponding M_{CRr} form a pair $\langle M_{CRa}, M_{CRr} \rangle$ with $v_a = v_r$. Let RA denote the set of all acquiring methods of an abstract resource and RR denote the set of all its releasing methods. Define the total function SPEC: RA → RR as the set of all such method pairs. SPEC formalizes the specification of this abstract resources management. Finally, an abstract resource AR is formally specified as a tuple (N, S, SPEC), where N is its unique identifier, and S is the set of its wrapped concrete resources, and SPEC is the resource management specification. An abstract resource may wrap more than one concrete resources and an acquiring (releasing) method may acquire (release) more than one concrete resources. In these cases, the acquiring (releasing) method is treated per concrete resource. An abstract resource can also be a concrete resource. For example, database connections are their own concrete resources as they have specific limits to themselves. Abstract resources are finite resources that programmers should manage carefully because an abstract resource's over-consumption will lead to the over-consumption of underlying concrete resources. Typically, programmers manage resources at the abstract resource level. As there are well understood API methods of programming-level resources and potential new application-specific resources may be added, we define abstract resources here to gain a straightforward and flexible solution of our Resco approach and to facilitate discussions of resource monitoring and collecting.

Definition 3. *(**Resource Collection Configuration**) A resource collection configuration C is a set of abstract resources AR_i that are provided for Resco to collect their leaked instances.* Users do not need to provide the complete resource management specification for abstract resources. Users can provide specifications related to only some of all concrete resources, but all abstract resources that wrap an interesting concrete resource should be specified. This requirement is to guarantee the correct counting of resource usage and enforcing of resource limits. An example of the resource collection configuration is provided in Fig. 1. The first three abstract resources have the value *bottom* of the attribute *hierarchy* which represents the wrapping hierarchy of all abstract resources wrapping a concrete resource. Other abstract resources in the `java.io` package are wrappers of these three ones. Only abstract resources at the bottom of the wrapping hierarchy need to be monitored for resource consumption. Although there are several acquiring methods for each of these three abstract resources (e.g., there are 3 public constructors of `FileInputStream`), only one is provided here for each of them as other acquiring methods simply invoke it. The omission of redundant acquiring methods (methods that acquire resources by invoking other acquiring methods) from the resource collection configuration facilitates the instrumentation of resource monitors (see Sect. 3 for detail).

```
<resource_collection_configuration>
  <concrete_resource name="file descriptor">
    <abstract_resource name="java.io.FileInputStream"
                       hierarchy="buttom" >
      <specification acquire="init(File)"
                     release="void close()" value=1 />
    </abstract_resource>
    <abstract_resource name="java.io.FileOutputStream"
                       hierarchy="buttom">
      <specification acquire="init(File, boolean)"
                     release="void close()" value=1 />
    </abstract_resource>
    <abstract_resource name="java.io.RandomAccessFile"
                       hierarchy="buttom">
      <specification acquire="init(File, String)"
                     release="void close()" value=1 />
    </abstract_resource>
    <abstract_resource name="java.io.FileReader"
                       hierarchy="high">
      <specification acquire="all constructors"
                     release="void close()" value=1 />
    </abstract_resource>
    ...
  </concrete_resource>
</resource_collection_configuration>
```

Fig. 1 An example of the resource collection configuration in the XML format. The concrete resource is the file descriptor, and we only provide the wrapping abstract resources in the `java.io` package. There are other 20 abstract resources omitted from this figure for space limit. This example is generated according to the source code of Oracle jdk6.27 [37] API.

3. Resco Approach

Programmers acquire some resource by calling the appropriate API method, and they release the resource by calling other API method. These resource management API method pairs for acquiring (releasing) resources are sometimes called resource-releasing specifications [10]. The problem of how to gain resource-releasing specifications has be studied by many researchers [10], [24], and we consider it orthogonal to our work. The only requirement of Resco for users is to provide their interesting resources and corresponding resource management API methods. Specifying resources at the abstract resource level facilitates users and makes Resco consistent with existing work. Besides common system-level resources, there are also other application-specific resources that have a limited amount available to a program for their own purposes. We expect that Resco can not only manage common system resources but also application-specific ones. We decide to deal with the problem of leaked resource collections at the abstract resource level, which has two main advantages: (1) Monitors are instrumented into the application code and/or libraries, which

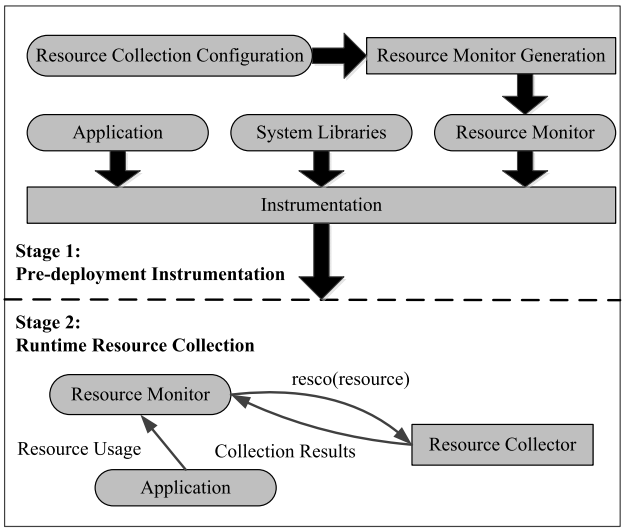


Fig. 2 Resco architecture.

```
1   public class MonitorCR {
2       //concrete resource
3       private ConResource cr = new ConResource();
4       private long limit = 1024; //limit of RC
5       private long count = 0;     //currently occupied RC
6       private long released = 0; //released RC in last collection
7       private long INFINITE = 16 * 1024 * 1024;
8       private long trigger = (long)(limit * 0.9); //resco's trigger
9
10      public synchronized void resourceAcquire(int v){
11          count = count + v;
12          if(count >= trigger){
13              long startCount = count;
14              System.resco(cr); //perform resource collection
15              long endCount = count;
16              released = startCount - endCount;
17              if(released == 0) adjust(limit);
18          }
19      }
20
21      public synchronized void resourceRelease(int v){
22          count = count - v; }
23
24      private void adjust(long currentLimit){
25          if(trigger < currentLimit)
26              trigger = currentLimit;  //set trigger to limit
27          else trigger = INFINITE; //forbid further collections
28      }
29  }
```

Fig. 3 The prototype monitor for the abstract resource *CR*. Only important fields and methods are presented here.

makes it easy to scale to new application-specific resources without modifications to the underlying runtime system; (2) As resources are specified by users at the abstract resource level, we can achieve a straightforward and flexible solution of counting resource usage and collecting leaked resources.

Figure 2 depicts the architecture of Resco that consists of two stages. The first stage is pre-deployment instrumentation. In this stage, resource monitors are first generated according to user-provided resource collection configuration. Then, these monitors are instrumented into target application and system libraries. The second stage is runtime resource collection. Instrumented monitors count the application's resource usage and make a request for resource collections if the resource limit is about to be violated. Then resource collector reclaims leaked resources and the reclamation result is returned to the requesting monitor. This monitor uses this feedback to adjust its request for next resource collection. Resco comprises three main components: (1) Resource monitor generation that generates monitors for each concrete resource (Sect. 3.1); (2) Instrumentation that instruments resource monitors into target application and system libraries (Sect. 3.2); and (3) Resource collector that performs resource collections (Sect. 3.3).

3.1 Resource Monitor Generation

There are two steps toward the resource monitor generation: (1) Identify all concrete resources *SCR* in the resource collection configuration, $SCR = \{CR \mid CR \in S \wedge \langle N, S, SPEC\rangle \in C\}$; (2) Generate a monitor for each concrete resource $CR \in SCR$ according to the monitor template. Core functionalities of a resource monitor include counting resource acquisition and releases, requesting resource collections if necessary, and adjusting future requests for resource collections based on the history of collection results. A prototype monitor for concrete resource *CR* in Java is presented in Fig. 3. This piece of code serves as an example of our discussion. The invariant $count \leq limit$ of class `MonitorCR` should not be violated. The method `resourceAcquire` is responsible to count resource acquisition and check the class invariant before the resource is allocated by acquiring methods. If the limit is about to be violated, it send a request for collecting leaked resources. The method `resourceRelease` simply subtracts from *count* the amount of resource released by releasing methods. The method `adjust` changes the trigger of next resource collection according to previous resource collection results. This adjustment is necessary for multi-level limits and serves to avoid too frequent collections. The resource collector begins to work when the program is close to resource exhaustion. We choose the threshold as 90% of the resource limit. If none of leaked resources is reclaimed in one collection, the trigger of next resource collection is adjusted to the limit itself. If the resource has more than one limit, the trigger is moved from the smaller one to the bigger one in order if necessary. A generated monitor is a singleton class. Only its one instance is created to monitor all instances of all abstract resources wrapping corresponding concrete resource. So, accesses to this instance should be synchronized among different threads. Rescos singleton monitors are one important reason for its low runtime overhead. We require that monitors do not use abstract resources that monitored by other monitors to avoid possible dead locks.

The resource collection configuration is provided for common system resources, such as file descriptors and database connections. As limits of these resources are system-specific and/or application-specific, our tool searches the system and application configurations for these limits. Some application-specific resources can be defined

```
Input  : C, MonitorCR for every CR,
         resource management code
Output: instrumented resource management code

1  foreach CR ∈ SCR do
2  |  SAR = {AR | AR = <N, S, SPEC> ∧ CR ∈ S ∧ AR ∈ C}
3  |  foreach AR = <N, S, SPEC> ∈ SAR do
4  |  |  foreach <M_CRa, M_CRr> ∈ SPEC with
   |  |  |        M_CRa = <m_CRa, v_a> and M_CRr = <m_CRr, v_r> do
5  |  |  |  instrument at the beginning of m_CRa
   |  |  |        with MonitorCR.resourceAcquire(v_a)
6  |  |  |  instrument at the end of m_CRr
   |  |  |        with MonitorCR.resourceRelease(v_r)
```

Fig. 4 Algorithm for resource management code instrumentation.

and provided through a configuration file. Specifications of default system resources can also be modified through this configuration file.

3.2 Instrumentation

The instrumentation process is to instrument generated resource monitors into resource management code that resides in the application or system libraries. This process includes two steps for each concrete resource $CR \in SCR$: (1) Identify all abstract resources *SAR* that wrap *CR*; (2) Instrument the monitor `MonitorCR` into all $AR \in SAR$. The instrumentation algorithm is presented in Fig. 4. The `resourceAcquire` method is inserted at the beginning of an acquiring method to check the limit and request resource collections in advance if necessary. The `resourceRelease` method is inserted at the end of a releasing method to ensure that we do not falsely count resource de-allocation in exceptional situations that may occur in the releasing method and lead to failed resource cleanup. It is common that one abstract resource wraps another abstract resource and that one acquiring method acquires resources by invoking another resource acquiring method. Resco requires that redundant acquiring methods (methods that acquire resources by invoking other acquiring methods) are omitted from the resource collection configuration and that the abstract resources that should be monitored are distinguished from the abstract resources that are just wrappers of them (e.g. using the *hierarchy* attribute in Fig. 1). This guarantees that resource consumption can be correctly counted by monitors. If v_a (v_r) is not statically decidable, some method call that dynamically gets this value should be passed into `resourceAcquire` (`resourceRelease`) as the parameter. The source code of the target application and system libraries is not necessarily needed, and this is implementation-specific.

3.3 Resource Collector

The third component of Resco is the resource collector to which monitors send requests for collecting leaked resources during runtime. The request carries as a parameter the concrete resource *CR* whose limit triggers this request.

```
        Input: S_uar, S_rr, CR
        Output: S_s

1  foreach object o in the heap do
2  |  mark[o] := unvisited
3  |  reach[o] := unreachable
4  |  ref[o] := 0
5  foreach o ∈ S_uar do
6  |  if mark[o] = unvisited then
7  |  |  DFS(o)
8  S_s := ∅
9  foreach o ∈ S_uar do
10 |  if reach[o] = unreachable ∧ ref[o] = 0 then
11 |  |  add o to S_s

DFS(o):
1  mark[o] := visited
2  if o is an instance of CR ∧ o ∈ S_rr then
3  |  ref[o] := 1
4  foreach object o' adjacent to o do
5  |  reach[o'] := reachable
6  |  if mark[o'] = unvisited then
7  |  |  DFS(o')
8  |  if ref[o'] := 1 then
9  |  |  ref[o] := 1
```

Fig. 5 Algorithm to compute leaked abstract resources that can be safely released.

To perform resource collections, the resource collector first determines leaked instances of abstract resources that wrap *CR*. Any analysis that provides object liveness information can be integrated into our resource collector. We choose to identify leaked resources as unreachable ones here like most other approaches to facilitate users and for reliable implementation. A full-heap tracing from root objects is performed, and we get a set S_{uar} that includes all unreachable instances of abstract resources wrapping *CR* and a set S_{rr} of reachable instances of abstract resources wrapping *CR* and reachable instances of *CR* itself. Then we perform a second full-heap tracing beginning from objects in S_{uar}, and get a subset S_s of S_{uar} that can be safely released. If the program's runtime system has a garbage collector that includes finalization, we should incorporate unreachable objects with the `finalize` method identified during the first tracing into roots for the second tracing. The algorithm to compute S_s is presented in Fig. 5. The array *reach* is used to determine whether objects can be reached in the second tracing and the array *ref* is used to indicate whether objects reference a reachable resource instance in S_{rr}. The procedure DFS is a classic Depth-First Search algorithm to perform the traversal of directed graphs. We adapt it to compute *reach* and *ref*. Please note the complexity of this algorithm keeps unchanged before and after the adaptation. There are two reasons that guarantee the safety of releasing resources in S_s: (1) Objects in S_s are not depended on by any object with actions (e.g. `close` or `finalize`) to perform in future; and (2) Objects in S_s do not reference any reachable instances of *CR* or wrappers of *CR*. We assume that corresponding releasing methods only release acquired *CR* and do nothing else. So, invoking these releasing methods will not cause any unexpected side effects. Directly releasing resources

in S_{uar} is not safe. As the first example, consider that if you build a `BufferedWriter` based on a `FileWriter`, the `FileWriter` should not be closed first. Otherwise, you cannot successfully close the `BufferedWriter` because its `close` method first flushes the buffer, which requires that the wrapped `FileWriter` should be open. The first reason of the safety of Resco guarantees that the `BufferedWriter` instead of the `FileWriter` wil be released. As a second example, consider that two abstract resources wrap the same concrete resource and one of them is reachable but the other is unreachable. If we close the unreachable abstract resource, the wrapped concrete resource will be closed. Then future operations of the reachable abstract resource may fail. The second reason of the safety of Resco guarantees that the unreachable abstract resource will not be closed.

All leaked resource instances in S_s are reclaimed by invoking corresponding releasing methods. To avoid dead locks, a separate thread is used to perform these invocations. To timely release leaked resources, this thread is given the privilege to run immediately after the second tracing. All application threads are blocked until this thread terminates or it is blocked by some locks. There is a possibility that some or even all of leaked resources cannot be released immediately and their releases are delayed to some later time due to locks. However, it is worthy to have a try.

To perform collections, appropriate releasing methods should be called dynamically. This is not easy in general considering for example how to decide at runtime the methods' receiver and its parameters' values. We make the following three assumptions to simplify this problem: (1) Releasing methods are public interfaces of corresponding abstract resources; (2) There is only one releasing method for each concrete resource wrapped by the same abstract resource; and (3) Releasing methods have no parameters. These assumptions hold for most non-memory system resources in Java and we believe that these are good rules that resource designs should obey. For example, the interface `Closeable` since Java 1.5 is implemented by all resources in the `java.io` package and some other resources. It includes only one non-parameter method `close` that releases resources the object is holding. `Socket`, `ServerSocket` and `Connection` also have a similar releasing method `close`. All these resources satisfy the above three assumptions. The newly introduced interface `AutoClosable` [28] since Java 1.7 also satisfies these assumptions. To guarantee that releasing an already released resource causes no problem, we require that an abstract resource instance can be repeatedly released, that is, a releasing method can be safely called more than once to close the same resource instance. This requirement is also necessary to make Resco to co-exist with existing finalization. This is not generally true for all abstract resources, but we can easily achieve this by instrumenting the abstract resource *AR* whose releasing method *M* do not meet this requirement. First, add a private field *closed* to *AR* with the initial Boolean value *false*; Second, add a statement *closed* = *true* at the end of *M*; Finally, impose the condition *closed* == *false* on the whole body of *M* by the `if` statement. There is one notable point that the method `resourceRelease` should not be repeatedly called for one resource instance when the instrumented releasing method is called more than once. This is easy to implement by testing *closed* before the execution of `resourceRelease`. Under above assumptions, releasing resource is simple: the only releasing method is invoked directly on the unreachable resource instance to reclaim it.

3.3.1 Discussions

We do not claim that Resco can collect all leaked instances of the triggering concrete resource. First, S_s is a conservative subset of S_{uar} that may exclude some leaked resources. However, we consider here the safety as the most important because we also expect Resco to cope with performance degradation due to resource leaks. Second, Resco can not reclaim resources that have already been collected by the garbage collector. However, since Resco and the garbage collector are triggered separately (by non-memory concrete resource consumption and by memory consumption, respectively), it is not necessary that some garbage collections have occurred before a resource collection. Even if part of the leaked resources are garbage collected by some previous garbage collections, it is better to reclaim the remaining leaked resources than to do nothing and leave the program to crash. These reclaimed resources can possibly enable the program to successfully complete its task. Otherwise, the program crash is unavoidable. One possible solution to this problem is to trigger the resource collector of Resco every time the garbage collection is requested. However, we do not adopt this approach and keep Resco separate from the garbage collector to avoid additional overhead on the process of garbage collections. As garbage collections are common and possible frequent depending on the available memory during the execution of programs written by garbage-collected languages, the performance of the garbage collector is critical. When a garbage collection is requested, it is possible that there are not any leaked resources. Even if there are some leaked resources, we do not have to reclaim them now if the consumption of the resource does not touch its limit. For example, many garbage collections are performed during the execution of benchmarks of our performance evaluation experiments, but no resource collection needs to be triggered. Bloch [27] reports that using finalizers has a severe performance penalty. Binding Resco to the garbage collector will cause unnecessary and non-trivial overhead on the process of garbage collections. Despite of the conservations of Resco, it is practical since it can successfully tolerate typical resource leaks during our experiments.

4. Implementation

We have implemented our Resco approach for Java based on Jikes RVM 3.1.1, a production-level, open-source Java-in-Java virtual machine. We call this implementation also

as Resco for simplicity. The resource collector is implemented as a class `RC` based on the MMTK memory manager toolkit [12] that provides the memory management functionality for Jikes RVM. Resco utilizes the the Mark-Sweep collector provided in Jikes RVM. However, `RC` is separately implemented and is independent of the underlying garbage collector. So, it can work with any garbage collectors of Jikes RVM. The full-heap tracing functionality of MMTK is mainly employed by Resco to get unreachable resource instances that can be safely released. To close leaked resources, Java's reflection (mainly the class `java.lang.Class` and classes in the package `java.lang.reflect`) is employed. We add a new method `resco` to `java.lang.System` that requests `RC` to perform resource collections. Resource monitors call this method to send resource collection requests to the resource collector. We implement the instrumentation tool based on Javassist [29] to instrument the Java bytecode. For system resources such as the file descriptor, resource monitors are statically inserted into Java core classes before the Java Virtual Machine (JVM) starts up because the default JVM security policies prevent Java core libraries from being modified or reloaded once the JVM is running. For other resources, the instrumentation is performed during load-time by our specialized classloader. So, source code of the application and system libraries is not needed.

5. Experimental Results

To evaluate Resco, we collect its performance measurement from standard benchmarks and apply it to tolerate several resource leaks.

5.1 Experimental Setup

Resco uses the default production configuration of Jikes RVM that is the highest performance configuration. We use the DaCapo benchmarks of version 2006-10-MR2 and version 9.12-bach [13], and SPECjvm98 [14]. We run each benchmark program with a single medium heap size fixed at two times the minimum in which it is possible for it to execute. Each benchmark runs ten times and the geometric mean of these results is presented as the final result. We run DaCapo benchmarks with their default workloads, and run SPECjvm98 benchmarks with the large input size (-s100). All experiments run on a machine of a 3.0 GHz $\times$ 4 Intel Core i5-2320 CPU and 4 GB of RAM, running Linux 2.6.38.6.

We used the same resource collection configuration for all experiments in this paper. Two common concrete resources and corresponding abstract resources are considered: (1) the file descriptor with its per-process limit as 1024 (this is the default value on our experimental machine). Corresponding abstract resources include file I/O streams in the `java.io` package such as `FileInputStream` and `FileOutputStream`, and sockets in the `java.net` package; and (2) the database connection of JDBC with its limit being application-specific. These connections themselves are their own abstract resources.

5.2 Performance

The performance of our resource monitor generation is quite reasonable as monitors for these two concrete resources are generated in less than 0.1 second. Classes of concerned abstract resources in the Java core Library `rt.jar` are instrumented with the trivial cost of less than 50 milliseconds.

Figure 6 presents the runtime overhead of Resco that includes overhead of the load-time instrumentation, the resource monitoring and resource collections. The Jikes RVM configuration denotes running the benchmark on the unmodified version of Jikes RVM 3.1.1. The Resco configuration denotes running the benchmark on our Resco tool. Each bar is normalized to its corresponding Jikes RVM configuration. The error bars show the range of the ten runs for each benchmark. As a large part of benchmarks within DaCapo 9.12-bach cant run on unmodified Jikes RVM 3.1.1, we only present here performance results of these ones that can run. To avoid benchmark name collisions between DaCapo 2006-MR2 and DaCapo 9.12-bach, we precede names of benchmarks in DaCapo 9.12-bach with _.

The graph shows that the runtime overhead of Resco is very low. The overall overhead increases by 1.32% as the geometric mean and in the worst case of `mtrt` the increase is 3.95%. During this empirical study, we find that no resource collection has been triggered. Benchmarks used here have been intensively examined and used by many researchers, so it is reasonable that they have few resource leak bugs. Moreover, even if there is a resource leak activated during the benchmark run, accumulated leaked resources may be not enough to trigger the resource collection. The runtime overhead of the resource collection phase of Resco will be further evaluated in the next section against known resource leak bugs.

5.3 Collecting Leaked Resources

To evaluate the capability of Resco to collect leaked resources, we apply Resco to four benchmark programs with known resource leak bugs. Table 1 gives overviews of these four resource leaks that we could reproduce: one from Ant [30], one from BIRT [31] one from Derby [32] and the last is a micro-benchmark leak from the Open Web Application Security Project [33]. To investigate the overhead of the phase of resource collection, we compute the ratio of the time for resource collections to the time for the whole run. The resource collection overhead is presented in the last column of Table 1.

These four experiments run with the median 256 M memory under original Jikes RVM and Resco respectively. Under Jikes RVM, the first three programs quickly crash due to resource exhaustion, while Resco collects leaked resources before resources are exhausted, and enables them to continuously run to successfully complete tasks. Resco

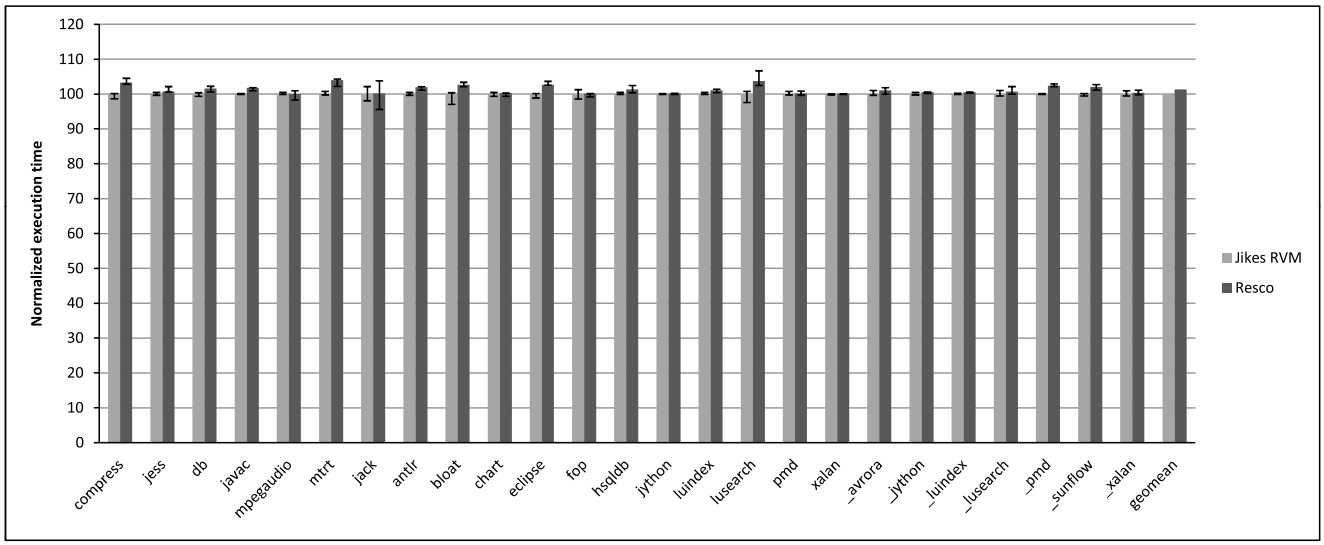


Fig. 6 Runtime overhead of Resco. The overall overhead increases by 1.32% as the geometric mean and in the worst case of `mtrt` the increase is 3.95%.

Table 1 Results of applying Resco on four resource leaks.

Benchmark Programs	LOC	Leak	Results	Overhead
Ant	32 K	File descriptor leak	All leaked file descriptors are collected	3.58%
BIRT + Tomcat + MySQL	1.7 M	BIRT's connection leak	All leaked connections are collected	0.72%
File processing	75	File descriptor leak	All leaked file descriptors are collected	2.21%
Derby	426 K	Connection leak	None of leaked connections is collected	4.02%

cannot reclaim Derby's leaked connections because Derby still keeps references to these open connections, but this experiment reveals the fact that Resco exhibits low resource collecting overhead even for these leaks that it cannot help with.

Ant leak. Ant is a famous Java project build tool. A file descriptor leak bug #4008 resides in ant v1.4. As no reproducing test case is provided in the bug report, we write one by ourselves. We define two Ant copy tasks: the first one copies all 515 files within the src directory of Ant v1.4 source distribution to another place; the second one copies ten copies of above files to another place, coping with ten times as much as the workload of the first one. To reproduce this bug, we employ one pattern file for each file to be copied.

We run these two tasks with Jikes RVM and Resco respectively. Both tools successfully complete the first task. For task two, Jikes RVM crashes with an error saying "Too many open files", and none of these files is copied. Resco successfully copies all these 5150 files. During this task, resource collections are triggered 5 times, and in total 4585 open file descriptors are reclaimed. This experiment has the resource collection overhead of 3.58%.

BIRT leak. BIRT is an open source Eclipse-based reporting system. We manage to reproduce its one database connection leak bug #237190. This is a very serious bug that can soon cause service loss as it is stated in the bug report that "we have consumed all available connections in the database (170) and crashed Oracle just by running a single production report every minute". We setup this experiment by deploying BIRT v2.3.1 into Tomcat v5.5.26 [34] with MySQL v5.0.67 [35] as the database system. We remain all default configurations of Tomcat and default 100 as the allowed maximum number of concurrent sessions of MySQL. We reproduce this bug with the test case provided in the bug report that selects a column from a table in MySQL and prints a string "Hello world.". We run this report using Firefox v11.0 [36] at the local machine. We develop a plugin for Firefox that repeatedly opens the same web page (for this experiment, it repeatedly run the test report) and logs the page-loading time of each open. We find that each run of the test report leaks a connection.

We repeatedly run the test report under Jikes RVM until we get the exception message within the 101st run "... rejected establishment of connection ... Too many connections" and no database selection results are displayed. Next, we repeatedly run the test report under Resco and every run successfully presents the right report contents. We present the page-loading time of each of the first 1000 iterations of Resco and that of the first 100 iterations of Jikes RVM here in Fig. 7. Resource collections are triggered 11 times and in total 979 leaked connections are collected during Resco's first 1000 iterations. The performance of Resco remains stable in the long term. There are obvious increases in time during iterations that resource collections are triggered. The resource collection overhead of this experiment is 0.72%. For the 11 iterations within which resource collections occur, the geometric mean of resource collecting overhead of

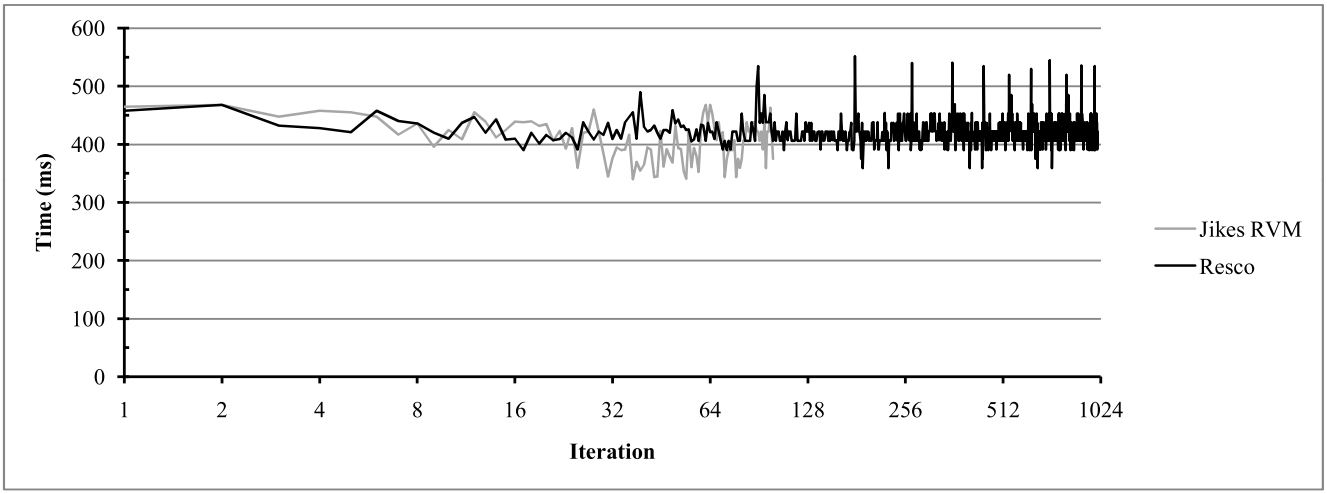


Fig. 7 Page-loading time of the BIRT leak experiment with logarithmic x-axis.

```
1  private void processFile (String fName) throws
2  FileNotFoundException, IOException{
3      FileInputStream fis = new FileInputStream (fName);
4      int sz;
5      byte[] byteArray = new byte[BLOCK_SIZE];
6      while ((sz = fis.read (byteArray)) != -1) {
7          processBytes (byteArray, sz);
8      }
9  }
```

Fig. 8 The source code of the micro-benchmark from the OWASP.

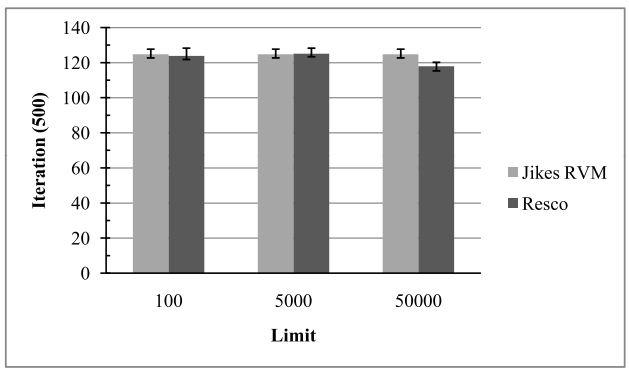


Fig. 9 The number of iterations completed within 30 seconds for different limits of connections of Derby leak experiment.

each iteration is 28.80%.

File processing leak. This leak is from the Open Web Application Security Project (OWASP) as an example of the Unreleased Resource vulnerability. This example (Fig. 8) is a simple Java method that does not close the file descriptor it opens as shown below. We write a class of in total several ten lines of Java code to run this example. The test task is as follows: one empty file is created and written into with a short string, then this example method is called with this file as input and the string is read from this input file and printed onto the standard output terminal. Clearly, one run of this task will leak an open file descriptor.

Running under Jikes RVM, the program quickly crashes with an exception saying "Too many open files" after 1017 iterations. In contrast, Resco can successfully prevent it from crashing and keep it continuously running 10000 iterations before we force it to exit. During these 10000 iterations, resource collections are triggered 10 times, and in total 9170 leaked open file descriptors are collected. The overhead of resource collections during these 10000 iterations is 2.21%.

Derby leak. Derby is an open source relational database implemented entirely in Java. We succeed to reproduce one database connection leak bug #3596 in Derby v10.4.1.3. This bug is reproduced by the attached test case that iteratively requests a connection and then executes a simple query within 30 seconds. The number of entries in the transaction table is printed after every 2500 iterations. We add several lines to print the exact number of open connections within derby just before the test exits. This test case is correct with respect to resource management. We find that each iteration leaks one connection. But Derby still maintains references to these leaked connections. Just like most other database systems, Derby have extra references to connections for no-functional purpose, such as monitoring and clean-up for abnormal situations. As these leaked connections are not unreachable, Resco cannot reclaim them.

As the test case is a Java program that directly interacts with Derby, we cannot find how to set the connection pool size or the maximum number of concurrent connections. So, we select three different limits for Resco to measure performance, and they are 100, 5000, and 50000. For each limit, resource collections are triggered two times and none of leaked connections is reclaimed. Figure 9 presents the performance of Resco compared with Jikes RVM in terms of iterations completed within 30 seconds for these three limits. The error bars show the range of the ten runs for each limit. It can be seen that when the limit is normal such as 100 or 5000, Resco's performance overhead is trivial; when

the limit is extremely large such as 50000, the overhead is obvious since the liveness analysis of the resource collector is very costly now. We also measure the time cost of resource collections: for the limit of 100, the overhead of resource collections is 0.18%; for the limit of 5000, the overhead is 0.53%; for the limit of 50000, the overhead is 4.02%. So, even for the extremely large limit, Resco's resoruce collection overhead is acceptably low. We conclude from this experiment that the resource collection overhead of Resco is low even for these leaks that it cannot help with.

6. Related Work

Current approaches to resource leaks either strive to detect and fix these leaks or provide new language features to simplify the resource management. There are few researchers that try to tolerate non-memory resource leaks. We present a new automatic resource collection approach and a tool that safely collects leaked non-memory resources to enforce resource limits.

6.1 Static Resource Leak Analysis

There are several tools and techniques that statically detect resources leaks [2], [3], [6] and insert releasing method calls at appropriate code points [6]. Most of these approaches adopt the reachability to conservatively approximate resources' liveness as Resco does. The CLOSURE [6] uses the interest reachability to better approximate resources' liveness, but it requires programmer-provided annotations.

Torlak et al. [2] present a tool Tracker that finds resource leaks in Java programs through an inter-procedural static analysis. Tracker takes resource-releasing specifications as input and symbolically tracks every resource along paths in the program's CFG. If a resource is not released and becomes unreachable, a resource leak is found. Weimer et al. [3] presents a path-sensitive and intra-procedural static data-flow analysis to find a kind of resource leak defects caused by the implicit control flow resulting from a checked language-level exception. The analysis considers each method body in turn by symbolically executing all its paths. If a resource is not in the closed state at the end of a path, a resource leak is found. This analysis may report false leaks or miss real leaks. The CLOSER [6] automatically inserts resource release calls into programs based on a modular, flow-sensitive analysis to determine the liveness of system resources at each program point. The interest reachability is proposed as an alternative to common reachability to better approximate resource liveness. As the notion of interest is application-specific and depends on program semantics, it is required that programmers should provide relevant annotations.

6.2 Dynamic Resource Leak Detection

Resco's monitors are singleton classes, which is a notable feature that differs from other dynamic leak detection techniques such as QVM [8] and PQL [7]. These approaches employ one monitor instance for each resource object. This can lead to enormous monitor instances in practice, which causes performance issues and is a very challenging problem in runtime verification [25]. Resco's singleton monitors are one important reason for its low runtime overhead. There are several other techniques that explore the staleness of objects to aggressively collect leaked memory [20], [26]. However, as cleanup of non-memory resources is not reversible, the object staleness cannot be easily applied to non-memory resource collections.

The QVM [8] is based on a Java Virtual Machine that detects and helps diagnose defects as violations of specified correctness properties. The garbage collector is instrumented to provide object death events when an object is decided to be unreachable during garbage collections. To check resource-releasing specifications, every resource object is instrumented and its states are tracked separately. If a resource object dies before it is released, a resource leak is detected. The PQL [7] is a pattern language that allows programmers to express common program error patterns. It is shown that PQL queries can effectively find mismatched method pairs which typically include resource leak bugs. PQL queries are first translated to state machines against which each resource instance are tracked. Although it is claimed that programmer-specified recovery actions can be performed as response to runtime query matches, releasing leaked resources are not discussed in this paper. As mismatched method pairs are liveness queries that depend on the absence of the second action, their matches are found at the end of an execution. Performing resource releasing then is too late and makes no sense. Other approaches based on aspects (e.g., [16], [17]) cannot precisely capture object death due to the lack of direct support from garbage collector. So, they are not suitable to detect resource leaks.

6.3 Language Features

Most garbage collectors allow a `finalize` method to be associated with an object. The `finalize` method is intended to perform some cleanup work that will be executed before its associating object is garbage collected. As the execution of `finalize` methods may be arbitrarily delayed in indeterminate way [9], it is generally agreed that the Finalization mechanism is not competent to reclaim finite system resource. Besides this delayed execution, another main drawback of Java's finalization is that ordering of invocations of different finalize methods cannot be guaranteed. As dependencies between resources are common, Java's finalization is not safe. In contrast, Resco performs resource collections in a safe manner. Resco is conservative because directly released resource instances are a subset of all unreachable ones. However, as it is a common design that releasing methods of one abstract resource call those of wrapped resources [10], Resco works well in practice. Modula-3 style finalization can guarantee ordering of `finalize` methods, however, the finalization need be performed in consecu-

tive multi-cycles [9]. Resco choose a more conservative approach here to achieve low overhead while still maintain practicability.

Many languages provide the mechanism of automatic releases of scoped resources that when a resource is out of its lexical scope, its releasing method is automatically invoked. Examples include destructors of C ++ and the `using` statement of C# [18]. Java 7 introduces the `try-with-resource` statement called Automatic Resource Management (ARM). Resources declared in this statement will be automatically closed once the program runs out of the try block. The declared resource should implement the `java.lang.AutoCloseable` interface. When resources are used in local scope, these mechanisms work well. However, there are situations in which the heap holds references to resources that are not confined to a convenient lexical scope.

To cope with resource leaks, Weimer et al. [3] propose a language extension called compensation stack that allows annotating resource acquiring methods with compensations such as resource releasing method invocations. These compensations are put in stacks that guarantee included compensations to execute in the last-in-first-out order. Compensations can be executed by the programmer, but they are often executed automatically when a heap-allocated compensation stack is finalized or when a stack-allocated compensation stack goes out of scope. The Furm [5] groups resources into a resource tree on which a single `release` call can close all these resources in deterministic order. The resource tree can be closed by the programmer, or when a thread dies, resource trees used by this thread will be closed automatically. One requirement of Furm is that each type of resource must be wrapped in a corresponding new class and interfaces of resource acquiring methods have to be changed. The type system of the Vault programming language [19] allows function post-conditions to be specified to guarantee that annotated functions cannot allocate and leak resources.

7. Conclusion and Future Work

This paper presents the Resco approach that reclaims leaked non-memory resources in response to abnormal resource consumption. Resco is a resource leak tolerance approach that complements existing techniques such as leak detection and fixing. For some concrete resource, there is a limit throughout the system. Our monitors now can only monitor application-level limits, which is adequate for our per-application resource collections. To enforce system-level limits is a future research direction. In addition, we plan to conduct further experiments on resource leak bugs to evaluate Resco's ability to collect leaked resources, especially on these leaks that cause performance degradation.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under Grant No.90818024 and 60803042, the National High Technology Research and Development Program of China (863 program) under Grant No. 2011AA010106 and 2012AA011201, and Program for New Century Excellent Talents in University.

References

[1] R.K. Dybvig, C. Bruggeman, and D. Eby, "Guardians in a generation-based garbage collector," Proc. ACM SIGPLAN 1993 conference on Programming language design and implementation, pp.207–216, Albuquerque, New Mexico, United States, 1993.

[2] E. Torlak and S. Chandra, "Effective interprocedural resource leak detection," Proc. 32nd ACM/IEEE International Conference on Software Engineering, pp.535–544, Cape Town, South Africa, 2010.

[3] W. Weimer and G.C. Necula, "Exceptional situations and program reliability," ACM Trans. Program. Lang. Syst, vol.30, no.2, pp.1–51, 2008.

[4] H. Fujio, H. Okamura, and T. Dohi, "Fine-grained shock models to rejuvenate software systems," IEICE Trans. Inf. & Syst., vol.E86-D, no.10, pp.2165–2171, Oct. 2003.

[5] D.A. Park and S.V. Rice, "A framework for unified resource management in Java," Proc. 4th International Symposium on Principles and Practice of Programming in Java, pp.113–122, Mannheim, Germany, 2006.

[6] I. Dillig, T. Dillig, E. Yahav, and S. Chandra, "The CLOSER: automating resource management in java," Proc. 7th International Symposium on Memory Management, pp.1–10, Tucson, AZ, USA, 2008.

[7] M. Martin, B. Livshits, and M.S. Lam, "Finding application errors and security flaws using PQL: a program query language," Proc. 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, pp.365–383, San Diego, CA, USA, 2005.

[8] M. Arnold, M. Vechev, and E. Yahav, "QVM: An efficient runtime for detecting defects in deployed systems," ACM Trans. Softw. Eng. Methodol, vol.21, no.1, pp.1–35, 2011.

[9] H.J. Boehm, "Destructors, finalizers, and synchronization," Proc. 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp.262–272, New Orleans, Louisiana, USA, 2003.

[10] Q. Wu, G. Liang, Q. Wang, T. Xie, and H. Mei, "Iterative mining of resource-releasing specifications," Proc. 26th IEEE/ACM International Conference on Automated Software Engineering, pp.233–242, Oread, Lawrence, Kan., 2011.

[11] B. Alpern, C.R. Attanasio, J.J. Barton, M.G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S.J. Fink, D. Grove, M. Hind, S.F. Hummel, D. Lieber, V. Litvinov, M.F. Mergen, T. Ngo, J.R. Russell, V. Sarkar, M.J. Serrano, J.C. Shepherd, S.E. Smith, V.C. Sreedhar, H. Srinivasan, and J. Whaley, "The Jalapeño Virtual Machine," IBM Systems Journal, vol.39, no.1, pp.211–238, 2000.

[12] S.M. Blackburn, P. Cheng, and K.S. McKinley, "Oil and Water? High Performance Garbage Collection in Java with MMTk," Proc. 26th International Conference on Software Engineering, pp.137–146, Edinburgh, United Kingdom, 2004.

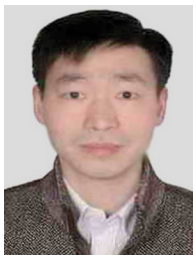
[13] S.M. Blackburn, R. Garner, C. Hoffmann, A.M. Khang, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J.B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D.V. Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," Proc. 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, pp.169–190, Portland, Oregon, USA, 2006.

[14] Standard Performance Evaluation Corporation, "SPECjvm98 Documentation," release 1.03 edition, 1999.

[15] L. Bauer, A.W. Appel, and E.W. Felten, "Mechanisms for secure modular programming in Java," Software – Practice and Experience,

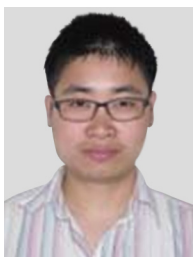
vol.33, no.5, pp.461–480, 2003.
[16] C. Allan, P. Avgustinov, A.S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Adding trace matching with free variables to AspectJ," Proc. 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp.345–364, San Diego, CA, USA, 2005.
[17] F. Chen and G. Rosu, "Mop: An efficient and generic runtime verification framework," Proc. 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, pp.569–588, Montreal, Quebec, Canad, 2007.
[18] A. Hejlsberg, P. Golde, and S. Wiltamuth, C# Language Specification, Addison Wesley, Oct. 2003.
[19] R. DeLine and M. Fähndrich, "Enforcing high-level protocols in low-level software," Proc. ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, pp.59–69, Snowbird, Utah, United States, 2001.
[20] M.D. Bond and K.S. McKinley, "Tolerating memory leaks," Proc. 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, pp.109–126, Nashville, TN, USA, 2008.
[21] M.D. Bond and K.S. McKinley, "Leak pruning," Proc. 14th International Conference on Architectural Support for Programming Languages and Operating Systems, pp.277–288, Washington, DC, USA, 2009.
[22] S.Z. Guyer, K.S. McKinley, and D. Frampton, "Free-Me: A static analysis for automatic individual object reclamation," Proc. of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp.364–375, Ottawa, Ontario, Canada, 2006.
[23] L. Moreau and C. Queinnec, "Resource aware programming," ACM Trans. Program. Lang. Syst, vol.27, no.3 pp.441–476, 2005.
[24] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring Resource Specifications from Natural Language API Documentation," Proc. 2009 IEEE/ACM International Conference on Automated Software Engineering, pp.307–318, Auckland, New Zealand, 2009.
[25] D. Jin, P. O'Neil Meredith, D. Griffith, and G. Rosu, "Garbage collection for monitoring parametric properties," Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp.415–424, San Jose, California, USA, 2011.
[26] Y. Tang, Y. Tang, Q. Gao, and F. Qin, "LeakSurvivor: Towards safely tolerating memory leaks for garbage-collected languages," USENIX 2008 Annual Technical Conference on Annual Technical Conference, pp.307–320, Boston, MA, 2008.
[27] Joshua Bloch, Effective Java, 2rd ed., Pearson Education Inc, 2008.
[28] Java 7, http://jdk7.java.net/, accessed Feb. 12. 2012.
[29] Javassist, http://www.jboss.org/javassist, accessed Feb. 1. 2012.
[30] Apache Ant, http://ant.apache.org/, accessed Feb. 15. 2012.
[31] BIRT, http://www.eclipse.org/birt/phoenix/, accessed Feb. 20. 2012.
[32] Derby, http://db.apache.org/derby/, accessed Feb. 25. 2012.
[33] The Open Web Application Security Project, https://www.owasp.org/index.php/Main_Page, accessed March 3. 2012.
[34] Apache Tomcat, http://tomcat.apache.org/, accessed March 10. 2012.
[35] MySQL, http://www.mysql.com/, accessed March 16. 2012.
[36] Firefox, http://www.getfirefox.net/, accessed March 16. 2012.
[37] Oracle Java SDK and JRE, http://www.oracle.com/technetwork/java/archive-139210.html. accessed Jan. 10. 2012.

Ziying Dai is currently a Ph.D. candidate in computer science and technology at School of Computer, National University of Defense Technology, 410073, Changsha, China. His research interests include fault tolerance and software debugging.



Xiaoguang Mao is currently a full professor at School of Computer, National University of Defense Technology, 410073, Changsha, China. He received his Ph.D. degree in computer science from National University of Defense Technology in 1997. His research interests include high confidence software, software development methodology, software assurance, software service engineering, etc.

Yan Lei is currently a Ph.D. candidate in computer science and technology at School of Computer, National University of Defense Technology, 410073, Changsha, China. His research interests include software debugging.



Xiaomin Wan received his B.S. degree in Information and Computing Science from Central South University, Changsha, China in 2004, and his M.S. degree in Computer Science from National University of Defense Technology, China in 2007. His research interests include software behavior analysis, program comprehension and software trustworthiness.

Kerong Ben is currently a full professor at Department of Computer Engineering, Naval University of Engineering, 430033, Wuhan, China. He received his Ph.D. degree in computer science and technology from National University of Defense Technology in 1994. His research interests include software testing and software maintenance.