Tolerating Memory Leaks AT Runtime

JITENDER SINGH YADAV, MOHIT YADAV* ANKIT JAIN*

Email-jitender1191m@gmail.com
mohit12yadav@yahoo.com
ankitjain199256@gmail.com

ABSTRACT
Type safety and garbage collection in driven languages wipes out memory errors such as dangling pointers, double frees, and leaks of unreachable objects. Unfortunately, a program still leaks memory if it continues references to objects it will not use again. Leaked objects decrease program locale and increase exhaust memory and crash the program. This paper introduces leak tolerance approaches called Melt and LeakSurviver, that safely eliminates performance degradations and crashes due to leaks of dead but reachable objects in managed languages, given efficient disk space to hold leaking objects. Melt (1) identifies stale objects that the program is not accessing; (2) segregates in use and stale objects by storing stale objects to disk; and (3) preserves safety by activating stale objects if the program subsequently accesses them. We design and build a prototype implementation of Melt in a Java VM and show it adds overhead low enough for production systems. Whereas existing VMs grind to a halt and then crash on programs with leaks, Melt keeps many of these programs running much longer without significantly degrading performance. Melt provides users the illusion of a fixed leak and gives developers more time to fix leaky programs. Leak-Surviver is handled in three ways which are discussed in the context.


1. INTRODUCTION:-
Garbage-collected languages such as Java and C# become increasingly popular. This is partly because programs written in these languages are free of many types of memory errors, which are notorious for compromising system availability and security. For example, by forbidding explicit pointers, Java programs do not encounter memory corruptions due to incorrect pointer-arithmetics. With tremendous advances of hardware and Just-In-Time (JIT) compiler techniques, garbage collected languages are now applied even in...
enterprise server environments. Unfortunately, memory leaks still exist and severely affect performance and availability for garbage collected programs, even though garbage collectors can reclaim unreachable memory objects. Memory leaks occur in a garbage-collected program when the program keeps references to memory objects that are no longer needed. As a result, the heap space occupied by leaked memory objects cannot be reclaimed by garbage collectors. For long-running garbage-collected programs, continuously-leaked memory objects take up more and more space in the heap, leading to more frequent invocation of garbage collections (GC) at runtime. Additionally, more leaked objects in the heap space increase the object traversal time during each GC phase. Therefore, continuous memory leaks cumulatively degrade overall program performance. Even worse, memory leaks may exhaust system resources, eventually causing program crashes.

Therefore, it is imperative to devise mechanisms for tolerating continuous memory leaks at runtime for garbage-collected programs.

The mechanisms must enable programs to maintain relatively stable performance and survive software failures caused by continuous memory leaks.

This paper presents two leak tolerance approaches called Leak-Survivor and Melt.

2. LEAK TOLERANCE APPROACHES.

Leaks are especially hard to reproduce, find, and fix since they have no immediate symptoms.

2.1. LEAK TOLERANCE WITH MELT.

Melt’s primary objective is to give the illusion there is no leak: performance does not degrade as the leak grows, the program does not crash, and it runs correctly. Stale objects are isolated from the in-use objects in a separate stale space, which resides on disk. The collector never accesses objects in the stale space, except when moving objects to the stale space. The application never accesses objects in the stale space, except when activating objects from the stale space. We classify reachable objects that the program has not referenced in a while as stale. If the program never accesses them again, they are true leaks. To identify stale objects, Melt requires modifications to both the garbage collector and the dynamic compiler.

At a high level, the modified collector marks objects as stale on each collection, and the modified compiler adds instrumentation to the application to unmark objects at each use.

The compiler adds instrumentation called a conditional read barrier [8] to every load of an object reference. The barrier checks whether the reference is stale. If it is stale, the barrier unmarks the referenced object and the reference. The following pseudo code shows the barrier:

```plaintext
b = a.f; // Application code
if (b & 0x1)
{
    // Conditional barrier
    t = b; // Backup ref
    b &= ~0x1; // Unmark ref
```
This conditional barrier reduces overhead since it performs stores only the first time the application loads each reference in each mutator epoch (the mutator is the application alone not the collector). Checking for a marked reference, rather than a marked object, reduces overhead since it avoids an extra memory load.

### 2.1.2. THE STALE SPACE.

When the garbage collection traces the heap, it now also moves stale objects to the stale space, which resides on disk. Stub-scion pairs. References from stale objects to in-use objects are problematic because moving collectors such as copying and compacting collectors move in-use objects.

### 2.1.3. ACTIVATING STALE OBJECTS.

Melt prevents the application from directly accessing the stale space since (1) these accesses would violate the invariant that the stale space is not part of the application’s working set, and (2) object references in the stale space may refer to stubs and scions. Melt intercepts application access to stale objects by modifying the read barrier to check for references to the stale space

```java
b = a.f; // Application code
if (b & 0x1) {
    // Read barrier
    t = b;
    b &= ~0x1;
    // Check if in stale space
    if (inStaleSpace(b)) {
        b = activateStaleObject(b);
    }
}
```

### 2.2. LEAK TOLERANCE WITH LEAKSURVIVOR.

It can avoid cumulative performance degradation and software failures due to continuous memory leaks. In addition, it helps developers to prune false positives.

#### 2.2.1. COMPONENTS

LeakSurvivor consists of three major components that detect and tolerate memory leaks during program execution.

#### 2.2.1.1. LEAK DETECTORS.
Leak detectors identify PL memory objects and pass this information to the swap-out component. For garbage collected programs, memory objects that are unreachable from roots.

2.2.1.2. SWAP-OUT COMPONENT.

The swap-out component saves the PL objects at runtime so that the virtual memory space occupied by them can be safely reclaimed by garbage collectors and are available for future memory requests. More specially, after PL objects are identified by leak detectors, the swap-out component copies their content from the virtual memory to the leak space.

Figure 2: Swap-out table and outgoing references

In addition, the swap-out component assigns all the incoming references to each PL object with a unique reserved address that is not allowed to be dereferenced at the user level. For all the outgoing references from each PL object, the swap-out component uses a swap-out table to record such information for facilitating future garbage collections.

2.2.1.3. SWAP-IN COMPONENT.
To handle an OS exception due to de-referencing a reserved address, the swap-in component first identifies the target PL object, allocates virtual memory space for the PL object, copies its content from leak space to the memory, and restores its incoming and outgoing references.

Additionally, the swap-in component notifies the leak detectors of the falsely-identified PL object. After this, the program automatically retries the faulty instruction and continues the execution correctly.

3. FUTURE ASPECTS.

We plan to extend our work in several dimensions in the future. First, we will evaluate LeakSurvivor using more applications with memory leaks. We currently only have three applications since it is difficult to find real-world applications with well-documented memory leaks. Second, we plan to support LeakSurvivor with asynchronous disk flushes and derived pointers. Third, we will investigate the idea of LeakSurvivor for tolerating memory leaks in C/C++ programs, which is difficult since there is no type information at the binary level.

4. CONCLUSION.

Garbage collection and type safety save programmers from many memory bugs, but dead, reachable objects hurt performance and crash programs. Given enough disk space, our leak tolerance approach keeps programs from slowing down and running out of memory. Melt requires only time and space proportional to in-use memory, rather than leaked memory, and preserves safety by activating stale objects on disk that the program later references. Our Melt implementation adds low enough overhead for deployed use. For growing leaks in real programs, Melt substantially delays crashes due to out-of-memory errors. With plenty of disk space, Melt has the potential to improve the user experience. It buys developers more time to fix leaks and keeps users happy by providing the illusion there is no leak. These attributes make Melt a compelling feature for future production VMs.

Figure 3: Swap-in table and incoming references
In summary, **LeakSurvivor** is a safe and non-invasive method to tolerate continuous memory leaks at runtime for garbage-collected programs. It helps programs to avoid cumulative performance degradation and software failures due to continuous memory leaks. It does so by swapping out potentially-leaked memory objects to disks and reclaiming virtual memory space occupied by them. **LeakSurvivor** is safe because it swaps in the swapped-out objects to the memory upon future accesses to them if they are falsely identified as leaks. Additionally, **LeakSurvivor** assists developers to diagnose memory leaks by providing false positives information due to swapped-in objects. Furthermore, it requires no modification to applications' source code. This indicates that safely reclaiming virtual memory space occupied by potentially-leaked objects is an effective way to eliminate leaks. In addition, **LeakSurvivor** improves the performance of programs with continuous memory leaks by 24%–46%.

### 5. REFERENCES.

1. Yan Tang, Qi Gao, and Feng Qin
2. The Ohio State University ftangya, gaoq, qing@cse.ohio-state.edu
3. Department of Computer Sciences, the University of Texas at Austin
5. Sun Developer Network Forum. Reflections & Reference
8. https://forums.oracle.com/thread/1157499?start=0&tstart=0
9. create/avoid memory leak in Java by Venkat Subramaniam (www.agiledeveloper.com/articles/MemoryLeak092002.pdf)