LeakSurvivor: Towards Safely Tolerating Memory Leaks for Garbage-Collected Languages

Yan Tang, Qi Gao, and Feng Qin
The Ohio State University
{*tangya, gaoq, qin*}*@cse.ohio-state.edu*

Abstract

Continuous memory leaks severely hurt program performance and software availability for garbage-collected programs. This paper presents a *safe* method, called LeakSurvivor, to tolerate continuous memory leaks at runtime for garbage-collected programs. Our main idea is to periodically swap out the "Potentially Leaked" (PL) memory objects identified by leak detectors from the virtual memory to disks. As a result, the virtual memory space occupied by the PL objects can be reclaimed by garbage collectors and available for future uses. If a swapped-out PL object is accesses later, LeakSurvivor will restore it from disks to the memory for correct program execution. Furthermore, LeakSurvivor helps developers to prune false positives.

We have built the prototype of LeakSurvivor on top of Jikes RVM 2.4.2, a high performance Java-in-Java virtual machine developed by IBM. We conduct the experiments with three Java applications including Eclipse, SPECjbb2000 and Jigsaw. Among them, Eclipse and Jigsaw contain memory leaks introduced by their developers, while SPECjbb2000 contain a memory leak injected by us. Our results show that LeakSurvivor effectively tolerates memory leaks for two applications (Eclipse and SPECjbb2000), i.e., no cumulative performance degradation and no software failures when facing continuous memory leaks at runtime. For Jigsaw, LeakSurvivor extends the program lifetime by two times and improves the performance by 46% compared with native runs. Furthermore, when there are no memory leaks, LeakSurvivor imposes small runtime overhead, i.e., 2.5% over the leak detector and 23.7% over the native runs.

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 [42]. 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 [3, 7].

Unfortunately, memory leaks still exist and severely affect performance and availability for garbage-collected programs, even though garbage collectors can reclaim unreachable memory objects [23]. 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 [12, 27]. Even worse, memory leaks may exhaust system resources, eventually causing program crashes [18, 27].

The performance degradation problem due to memory leaks cannot be completely solved by the paging mechanism used in OS memory management or by infinitely increasing the heap size (e.g., in 64-bit machines). The OS paging mechanism swaps out physical memory to disks when physical memory is under pressure, which leaves the virtual memory space un-reclaimed. Therefore, heap pressure (i.e., little available heap space) in the virtual memory due to memory leaks remains the same, leading to cumulative performance degradation. Alternatively, infinitely increasing the heap size, e.g., in 64-bit machines, reduces the heap pressure and thus eliminates the increasing GC frequency. However, once the working set of garbage collectors, i.e., the whole heap, becomes too large to fit into the available physical memory, the program performance will be drastically degraded due to increased memory paging during each GC phase [44].

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 perfor-*

mance and survive software failures caused by continuous memory leaks.

There are only a few studies [13, 35, 24, 34] on tolerating memory leaks. They are either unsafe or unable to prevent performance degradation and out-of-memory errors. Cyclic memory allocation [34] tolerates memory leaks by restricting each allocation site to a fixed number of live objects. While this method may work for applications that have predictable memory requirements, it is in general unsafe since the live objects could be overwritten. Melt [13] and Plug [35] isolate leaked objects from non-leaked objects and rely on the OS paging for releasing the physical memory occupied by leaked objects. While they are effective in alleviating the physical memory shortage caused by memory leaks, these methods cannot reduce heap usage in the virtual memory and will eventually trigger the out-of-memory errors. Goldstein et al. proposed to dump large unused objects to disks [24]. However, their approach does not handle small leaked objects, which can insidiously cause performance and availability problems. Furthermore, it lacks an automatic mechanism to detect leaked objects.

Our Contributions. In this paper, we propose a *safe* method for tolerating continuous memory leaks for garbage-collected programs at runtime. It can avoid cumulative performance degradation and software failures due to continuous memory leaks. In addition, it helps developers to prune false positives.

Our idea is to periodically move the "Potentially Leaked" (PL) memory objects identified by leak detectors from the virtual memory to disks. As a result, the heap space occupied by PL objects can be reclaimed by garbage collectors and thus are available for future uses. If a swapped-out PL object is accesses later, LeakSurvivor will restore it from disks to the memory for correct program execution.

Based on the above idea, we build a tool called LeakSurvivor in Jikes RVM [7], a high performance Java-in-Java virtual machine developed by IBM, to tolerate memory leaks at runtime. We evaluate LeakSurvivor with three applications, including Eclipse, a popular integrated development environment, SPECjbb2000, a simulator of an order-processing server system, and Jigsaw, a web server. Among them, Eclipse and Jigsaw contain memory leaks introduced by their developers, while SPECjbb2000 contain a memory leak injected by us. Unlike previous approaches, LeakSurvivor possesses the following advantages:

- LeakSurvivor can safely and effectively tolerate memory leaks for garbage-collected program at runtime. Our evaluation shows that it can tolerate leaks for two of the three applications (Eclipse and SPECjbb2000). For Jigsaw, LeakSurvivor extends its lifetime by two times and improves performance by 46% compared with native runs.

- LeakSurvivor incurs low overhead for programs during normal execution without memory leaks. This is because it only adds one extra check during GC phase. Our evaluation with DaCapo benchmarks shows that LeakSurvivor incurs small runtime overhead, i.e., 2.5% over the leak detector and 23.7% over the native runs, when there are no leaks.

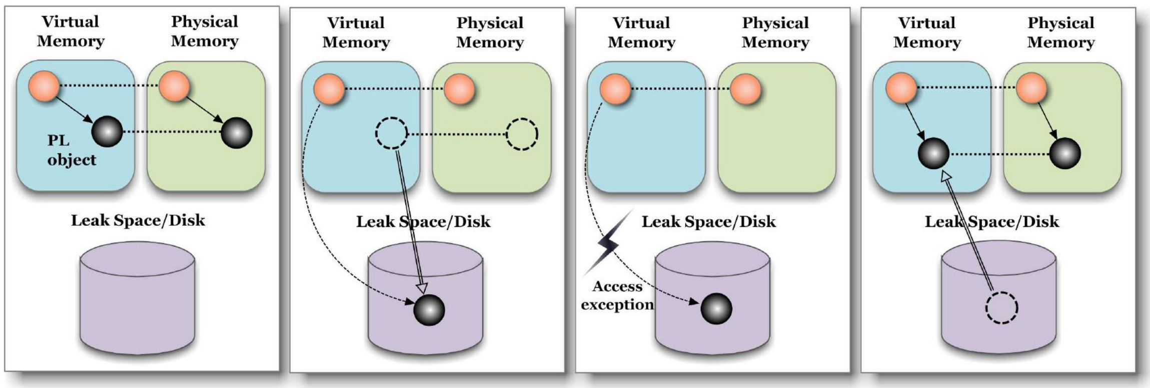
- LeakSurvivor requires no modification in the applications' source codes. Therefore, it can be easily applied to legacy garbage-collected programs.

- LeakSurvivor provides false positive information on memory leaks, if any, to developers for pruning purposes. Once a PL object is restored to the memory, LeakSurvivor marks it as a false positive and passes this information to the leak detectors.

2 Main Idea of LeakSurvivor

The main idea of LeakSurvivor is to periodically swap out "Potentially Leaked" (PL) objects from the virtual memory to disks so that garbage collectors can reclaim the heap space occupied by these objects. If all leaked memory objects are identified as PL objects and swapped out, the heap pressure due to memory leaks is eliminated. As a result, LeakSurvivor can avoid cumulative program performance degradation and future program crashes due to continuous memory leaks. Upon accesses to some swapped-out PL objects during subsequent program execution, i.e., PL objects are falsely identified, LeakSurvivor swaps them from disks to memory so that the programs can continue execution correctly. In this way, LeakSurvivor guarantees the safety of leak tolerance. Additionally, LeakSurvivor helps developers to prune false positives.

Figure 1 shows the process by which LeakSurvivor tolerates memory leaks at runtime and guarantees the correctness of subsequent program execution. During program execution, LeakSurvivor periodically checks whether there are PL objects marked by the integrated leak detectors. If so, it swaps out the PL objects by copying the object contents from memory to the leak space (See Section 3.2.1), which is mainly on disks. Additionally, LeakSurvivor modifies all the PL objects' incoming references, i.e., references from other objects to the PL objects, and outgoing references, i.e., references from the PL objects to other objects (See Section 3.2). After this, the virtual memory space occupied by the PL objects can be safely reclaimed by garbage collectors and are available for future uses. Figure 1(a)(b) shows the



(a) Mark a PL object (b) Swap out a PL object (c) Access a PL object (d) Swap in a PL object

Figure 1: LeakSurvivor: The main idea (PL means "Potentially Leaked").

swap-out process, which reduces heap pressure due to memory leaks.

Unfortunately, even the state-of-the-art leak detectors report false positives, i.e., PL objects that are not really leaked. Once the program accesses a swapped-out PL object through an incoming reference, which was modified to a unique reserved address during a swapping out phase, an OS exception (segmentation fault) will be triggered. In the exception handling routine, LeakSurvivor identifies the PL object based on the unique reserved address and swaps in the object by copying its content from the leak space to memory and restoring the corresponding incoming and outgoing references. After returning from the exception, the program continues execution correctly by retrying the "faulty" instruction. The swap-in process is shown in Figure 1(c)(d).

3 Design and Implementation

LeakSurvivor consists of three major components that detect and tolerate memory leaks during program execution. As shown in Figure 2, the three components in LeakSurvivor are: (1) leak detectors for detecting "Potentially Leaked" (PL) memory objects; (2) a swap-out component for copying PL objects from the memory to the leak space, which is mainly on disks; (3) a swap-in component for restoring PL objects from the leak space to the memory upon future accesses to them. We implement LeakSurvivor on top of Jikes RVM 2.4.2, a high-performance Java-in-Java virtual machine [7]. We do not see any particular difficulties to implement the ideas of LeakSurvivor in other Java Virtual Machines (JVM).

3.1 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*, i.e., references in global, static, and stack variables, are automatically reclaimed by garbage collectors. Therefore, leak detectors for garbage-collected programs only report potentially leaked objects that are still reachable from *roots* as PL objects. There are two types of such leak detectors. The first type is to exploit heap growth and heap difference for identifying leaked objects or data types that cause heap growth. This type of tools include JRockit [43], LeakBot [33], and Cork [31], just to name a few. The second type is to exploit the lifetime or staleness of each object for identifying leaked objects that are not used for a long time. In the prototype of LeakSurvivor, we use Sleigh [12], a lightweight, low space overhead leak detector of the second type. The idea of our LeakSurvivor, however, is independent of leak detection methods.

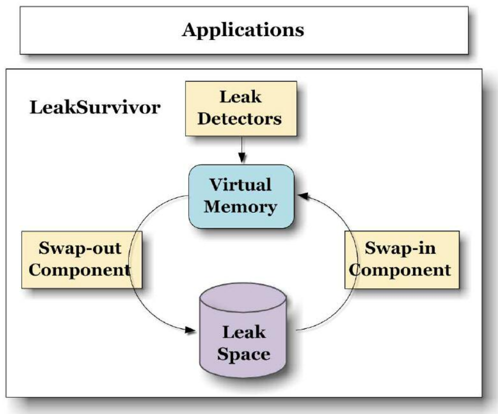


Figure 2: LeakSurvivor architecture

Sleigh tracks accesses to each object at runtime and marks objects that are not accessed for a long time as *stale* objects. More specifically, Sleigh designates two bits as the stale counter for each object to record how long the object has not been accessed. It resets the stale counter to zero upon object allocation and accesses, and increases the stale counter in logarithmic scale at garbage collection (GC) time. Once both stale bits become ones, the object is deemed as stale. To minimize the space overhead, Sleigh leverages a statistical approach, called BELL, to encode object allocation and last-use site information into a single bit per object. Periodically, it decodes the information based on a large number of stale objects and reports class names, the allocation and last-use sites of stale objects. In its implementation, Sleigh borrows four unused bits in the object header and thus incurs no per-object space overhead. For more details about Sleigh, please refer to the paper [12].

LeakSurvivor considers the stale objects whose class names are reported by Sleigh's decoding phases as PL objects. Sleigh does not report class names for the stale objects with small occurrence since they are most unlikely continuous leaks and much less harmful. Furthermore, LeakSurvivor focuses on application classes instead of primitive types such as char array, classes with the prefix "java.*", etc. Since Sleigh's decoding phase is expensive, we currently run it offline and pass the results to LeakSurvivor. In the future, we plan to extend Sleigh to run the decoding phase online in a separate machine or different cores of the same machine. Additionally, we modified Sleigh to record false positives reported by the swap-in component so that they can assist developers for further bug diagnosis and the falsely identified objects will no longer be marked as PL objects.

3.2 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 specifically, after PL objects are identified by leak detectors, the swap-out component copies their content from the virtual memory to the leak space (Section 3.2.1). In addition, the swap-out component assigns all the incoming references to each PL object with a unique reserved address (Section 3.2.2) 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 (Section 3.2.3) to record such information for facilitating future garbage collections. During subsequent program execution after a swap-out process (Section 3.2.4), if an instruction has nothing to do with any reserved address, it is normally executed. Otherwise, it triggers an OS exception, which will be handled by the swap-in component (Section 3.3) so that the program can correctly continue execution. We summarize the swap-out process in Section 3.2.5.

3.2.1 Leak Space

The leak space mainly uses disks for storing PL objects so that the memory space for these objects can be released for future uses and the PL objects can be restored if needed. Due to slow disk operations, it is essential to manage the leak space for efficient object write and read operations in the leak space. In LeakSurvivor, write operations are dominant since continuous memory leaks make PL objects continuously to be moved to the disks. In the meantime, LeakSurvivor reads PL objects from disks only when they are falsely identified by Sleigh, which has low false positive rates [12].

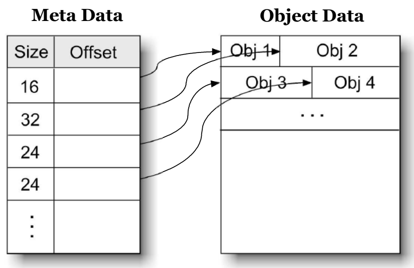


Figure 3: Leak space structure

LeakSurvivor organizes the leak space in a way similar to the Log-structured File System [38] for efficient write accesses of various-sized PL objects and random read accesses. More specifically, a PL object is sequentially appended to the last PL object in the leak space for a write operation. The disk space occupied by the PL objects will not be reclaimed even when the objects are brought back to the memory. This design is to trade disk capacity for slow write operations given the ever-increasing disk capacity per dollar [26]. Once the disk capacity becomes a real problem, we can use compaction techniques [17] to clean up disk space occupied by useless PL objects.

To provide efficient random read accesses to PL objects in the leak space, LeakSurvivor uses fix-sized meta data for each PL object. As shown in Figure 3, the meta data for each PL object consists of its size, its disk address, and other related information specific for Jikes RVM. The index of each entry is the unique id associated with a PL object. With the index number, LeakSurvivor can easily identify the meta data of the PL object and read the object from the leak space.

To further improve the performance of the leak space write and read operations, LeakSurvivor dedicates two chunks of memory as buffers for storing PL objects and their meta data respectively. More specifically, LeakSurvivor first stores a PL object and its meta data in the buffer. Once the buffer is full, the chunk of PL objects and their meta data will be flushed to disks. We currently implement synchronous flushes in LeakSurvivor. For better performance, we plan to support asynchronous flushes. During our experiments, 8 MB buffer is large enough to provide efficient PL object writes to the leak space. Furthermore, when LeakSurvivor reads a PL object from the leak space, it first checks the buffer to see whether the object is still there. If yes, LeakSurvivor avoids the expensive disk read (See Section 5.2.5).

3.2.2 Incoming References and Reserved Addresses

After a PL object is copied from the memory to the leak space, LeakSurvivor assigns all the incoming references, i.e. references to the PL object, with a unique reserved address to guarantee correct program execution upon future accesses to the PL object. To modify all the incoming references, LeakSurvivor leverages forwarding pointer techniques [17, 21] and integrates the reference modification into the object traversal process at GC phases. More specifically, LeakSurvivor stores the unique reserved address for each PL object in its object header as the forwarding pointer. During the GC phase, when the traversed object has a reference to a PL object, LeakSurvivor modifies its reference to the reserved address stored in the forwarding pointer. As a result, LeakSurvivor avoids one extra live object traversal for modifying incoming references of each PL object.

The reserved addresses are within the address range reserved exclusively for the OS kernel use. Consequently, any access to a PL object via de-referencing the associated reserved address at the user level triggers an OS exception. In the exception handling routine, the swap-in component brings the PL object from the leak space to the memory so that the program can continue execution correctly.

Such use of reserved addresses in LeakSurvivor has no conflict with the use of reserved addresses by the kernel itself. This is because the virtual memory pointed to by reserved addresses can be normally accessed at the kernel level without raising any exception. However, it may cause problems if the reserved addresses for some PL objects are passed from applications to the kernel. To address it, LeakSurvivor intercepts system calls in Java system call wrappers and swaps in the objects before invoking the system calls.

LeakSurvivor maintains a one-to-one mapping between the reserved addresses and the PL objects by sequentially assigning a different reserved address to each new PL object. Equation 1 shows this mapping, where $Addr(obj)$ is the reserved address of a PL object, $Index(obj)$ is the meta data index of a PL object, and $OFFSET$ is the base address of the reserved address range. Therefore, given a reserved address, LeakSurvivor can easily locate the corresponding PL object by calculating the meta data index, and vice versa.

$$Addr(obj) = Index(obj) + OFFSET \qquad (1)$$

The reserved address range is large enough for tolerating many leaked memory objects. In a 32-bit machine, OSes such as Linux usually reserve 1GB address range exclusively for the kernel use, which means LeakSurvivor can move around 1G PL objects to the leak space. According to previous study [10], the size of most objects in Java is from 40 to 80 bytes, which includes both the object content and header information. Therefore, LeakSurvivor can tolerate continuous memory leaks with total size of 40-80 GB, which is 10-20 times of the entire virtual memory space, in a 32-bit machine. With emerging 64-bit machines, LeakSurvivor can leverage much more reserved addresses for tolerating memory leaks.

It is obvious that garbage collectors should not follow reserved addresses. Otherwise, it will trigger unnecessary swapping of PL objects from the leak space to the memory. We modify the garbage collector so that it checks whether the currently-being-traversed reference is within the reserved address range or not. If yes, the garbage collector does not de-reference it and continue with other references and objects. Otherwise, the garbage collector executes as usual.

3.2.3 Outgoing References and Swap-Out Table

After a PL object is moved from the memory to the leak space, LeakSurvivor must properly handle its outgoing references for two reasons. First, the outgoing references need to be updated if the pointed-to objects are moved to different memory locations, which can happen for copying garbage collectors [17, 21]. Second, garbage collectors need to traverse the outgoing references during GC phases. Otherwise, objects that are pointed to only by these references will be incorrectly reclaimed.

LeakSurvivor uses a in-memory Swap-Out Table (SOT) to record all the outgoing references for each PL object. More specifically, each SOT entry represents one outgoing reference from a PL object. We modify the garbage collector to add all the outgoing references in the SOT to the $roots$ before it starts the object traversal process. Consequently, the garbage collector can efficiently and correctly reclaim unreachable objects without reading PL objects from disks.

As shown in Figure 4, each SOT entry consists of two fields: one is the outgoing reference from the PL object to another object in the memory and the other is the reference counter for recording the number of outgoing references sharing the entry. For handling each outgoing reference from a PL object in the swap-out process, LeakSurvivor first looks it up in the SOT. If any matching entry is found, LeakSurvivor simply increments the reference counter of that entry by one. Otherwise, a new

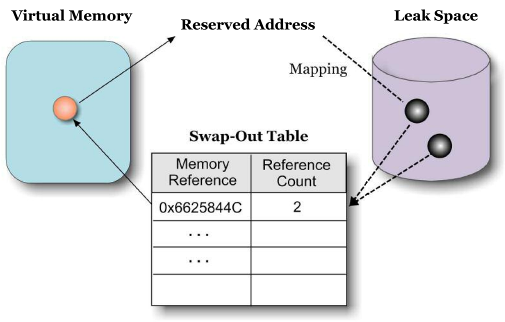


Figure 4: Swap-out table and outgoing references

entry is created for the outgoing reference. After this, LeakSurvivor modifies the field in the PL object from the outgoing reference to the index of the SOT entry.

Since the SOT resides in memory, we must limit its memory consumption from overshadowing the benefits of swapping out PL objects. Fortunately, the memory consumed by the SOT is much smaller than the memory saved by moving PL objects to the leak space. This is because many objects in garbage collected languages, such as Java, only have 1 to 2 incoming reference [10]. In this situation, The children objects of a PL object are also likely leaked since they cannot be reached without accessing the only parent PL object first. Therefore, LeakSurvivor does not need to create a SOT entry for many PL objects upon moving them to the leak space. Additionally, LeakSurvivor reduces the SOT size by reference counters if multiple PL objects have outgoing references to the same object in the memory. Furthermore, LeakSurvivor reclaims a SOT entry if the reference counter becomes zero, which occurs when all the related PL objects are moved back to the memory. Another chance to reclaim a SOT entry is before flushing PL objects from the memory to disks. During this period, LeakSurvivor checks the SOT entries used by the PL objects in the buffers to see whether their outgoing references are modified to some reserved addresses. If any, LeakSurvivor reclaims the SOT entry and modifies the field of the PL object to the reserved address. Our experiments show that a SOT with 1 MB is sufficient to tolerate continuous memory leaks for the evaluated applications (See Section 5.2.4).

3.2.4 *Instruction Execution*

LeakSurvivor guarantees correct program execution after swapping out PL objects into the leak space. We can verify it by examining all the machine instructions that are possibly executed since Jikes RVM compiles class methods into native code for better performance.

We classify all the machine instructions into three types based on how the instructions are related to the references (i.e., reserved addresses) to PL objects. The first and simplest type is instructions whose operands have nothing to do with the references to any PL object, i.e., not performing any operations over the references. This type of instructions can be correctly executed without being affected by LeakSurvivor.

The second type of instructions perform non-dereferencing operations such as the equality operation "==" and the assignment "=" over the references to some PL objects. This type of instructions can be correctly executed since LeakSurvivor associates each PL object with a unique reserved address so that the program distinguishes references to different PL objects from each other. For example, after a PL object A is moved to the leak space, its incoming reference value will be changed from A_{old} to some reserved address A_{resv}. In this scenario, if A_{old} equals to B_{old}, i.e., both references pointing to the same PL object, then LeakSurvivor ensures that A_{resv} equals to B_{resv} as described in Section 3.2.2, and vice versa. Note, type checking systems in garbage-collected programs prohibit arithmetic operations such as $+$ and $-$ over references [25].

The last type of instructions de-reference the references to some PL objects, which triggers OS exceptions due to the reserved addresses. Such instructions are compiled from operators such as field access, method invocation, *instanceof*, *type cast*, etc. [25]. They attempt to access either the content or the header information of the PL objects. Therefore, LeakSurvivor needs to bring the corresponding PL objects to the memory for continuing program execution correctly. Array bounds checking, for example, will trigger the swapping in of the array object itself, but leave the objects referenced by array elements untouched.

3.2.5 *Swap-out Process*

There are three steps for moving a PL object to the leak space and modifying the outgoing references. First, LeakSurvivor attempts to move the children objects to the leak space if they are stale. Otherwise, LeakSurvivor skips them since they are being actively accessed. Second, the content of the PL object is copied from the memory to the leak space. We use the depth-first order to avoid unnecessary SOT entries if children objects are also swapped out. The last step is to create a SOT entry for each outgoing reference that is still pointing to an object in memory.

LeakSurvivor integrates the swap-out process with GC's object traversal process to achieve better performance. More specifically, when each object is being traversed by the garbage collector, LeakSurvivor checks whether it is stale and its class name reported by Sleigh. If yes, it is a PL object and LeakSurvivor swaps it out via the above three steps. Additionally, if a stale object is pointed to by the reference of any SOT entry, LeakSurvivor swaps it out since it is likely to be a PL object due to its staleness and its ancient PL object. After an object is swapped out, the garbage collector will assign all the future incoming references to this object with its associated reserved address using forwarding pointer techniques. Note, we modify the garbage collector to add the references in all the SOT entries to the *roots* before the object traversal.

When there are circular references among some swapped-out objects, the above swap-out process creates one SOT entry recording the outgoing reference from the last PL object to the first PL object. This is incorrect because after the swap-out process the first PL object is moved to the leak space, which makes this outgoing reference in the SOT entry obsolete. To address it, LeakSurvivor inspects all the SOT entries to see whether the outgoing references are pointing to some objects in the leak space. If so, LeakSurvivor modifies the fields in the corresponding PL objects to the reserved addresses.

.

3.3 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 (Section 3.3.1), allocates virtual memory space for the PL object, copies its content from leak space to the memory, and restores its incoming and outgoing references (Section 3.3.2). 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. We discuss the multi-threading issue in Section 3.3.3 and summarize the swap-in process in Section 3.3.4.

3.3.1 PL Object Identification

In an OS exception, LeakSurvivor checks whether it is a segmentation fault. If yes, LeakSurvivor retrieves the de-referenced reserved address from the base register in the faulty instruction. Otherwise, it passes the exception to the default handlers. Given a reserved address $Addr(obj)$, LeakSurvivor can calculate the meta data index of the PL object, i.e., $Addr(obj) - OFFSET$, based on Equation 1. Then, LeakSurvivor fetches the information of the PL object and reads it from the leak space.

The above scheme works in most cases since the compiler often generates code with $base\_address + offset$ addressing mode, where $base\_address$ is the starting address of an PL object. However, the compiler occasionally generates code that first calculates the result of $base\_address + offset$, then stores the result in the base register, and finally de-references the address stored in

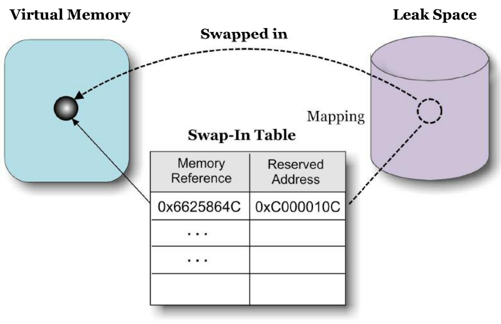


Figure 5: Swap-in table and incoming references

the base register. In this case, LeakSurvivor incorrectly considers the value of $base\_address + offset$ stored in the base register as another PL object. To solve this problem, we currently modify the compiler to only generate code with $base\_address + offset$ addressing mode. We plan to extend LeakSurvivor to solve this problem more elegantly based on ideas of derived pointers [20].

3.3.2 Incoming and Outgoing References

After copying the PL object from the leak space to the memory at a new location, LeakSurvivor needs to restore its incoming and outgoing references. The incoming references come from three sources: registers, the memory, and the leak space. For incoming references from registers, LeakSurvivor scans the general registers and modifies their values to the new memory address if they are the reserved address.

For incoming references from the memory, one simple way is to traverse all the live objects from *roots* and modify the incoming references from the reserved address to the new memory address of a swapped-in PL object. For better performance, LeakSurvivor delays this modification process to the next GC phase with the help of the swap-in table described as below.

For incoming references from other PL objects in the leak space, it is prohibitively expensive to scan all the PL objects and make modifications. LeakSurvivor introduces a Swap-In Table (SIT) to address this issue. As shown in Figure 5, each SIT entry records the mapping from the reserved address of a PL object to its new memory address after being swapped in. With the SIT, LeakSurvivor can leave the incoming references from the leak space untouched until the PL object is swapped back to the memory. In such situation, LeakSurvivor looks up the incoming reference in the SIT and modifies the reference to the new memory address.

In summary, LeakSurvivor only modifies the incoming references from registers at the time of handling the exception and retries the "faulty" instruction. This method may severely hurt overall program performance in some scenarios, although we have not experienced such situation in the experiments. For example, if a program de-references a reserved address in a loop, it may raise exceptions repetitively by the same faulty instruction since LeakSurvivor does not modify the incoming reference from the memory. To alleviate this problem, we can modify local variables that contain incoming references to the new memory address by scanning the stack frames. Also we can modify the incoming references in the memory by performing live object traversal to avoid such repetitive exceptions.

The SIT is small due to the low false positive rate of the leak detector [12]. Our experimental results in Section 5 show that 0.5 MB SIT size is big enough for all the three evaluated applications. If the SIT becomes too large, it indicates that the false positive of the leak detector is high and we should try other detectors.

For the outgoing references in the PL object that is being swapped into the memory, LeakSurvivor resolves them by looking up the SOT. More specifically, for each outgoing reference, LeakSurvivor reads the corresponding SOT entry, and modifies the outgoing reference to the reference in that entry. A more sophisticated alternative is to first check whether that SOT entry is pointing to the memory. If so, it is up-to-date and LeakSurvivor will update outgoing reference with it; otherwise, it will search SIT table until either there is no matching result or the reference in the matching entry points to an object in the memory. We implement the lazy method (the first one) because of its simplicity.

3.3.3 Multi-Threading

When an application has multiple threads, more than one thread can access some PL objects by de-referencing the reserved addresses concurrently. This triggers multiple OS exceptions at the same time. Therefore, LeakSurvivor has to be thread-safe for the swap-in process. Similarly, the swap-out process also needs to be thread-safe. The prototype of LeakSurvivor serializes the swap-in and swap-out processes in multiple threads by disabling thread switching once it enters any of the processes. It is equivalent to use a big lock to synchronize the entire swap-in and swap-out processes among multiple threads. For better performance, we plan to extend LeakSurvivor with fine-grained locks to enable concurrent execution of the swap-in and swap-out processes in multiple threads.

3.3.4 Swap-in Process

During the swap-in process, LeakSurvivor first identify the reserved address of the accessed PL object based on the faulty instruction. Then it looks up the reserved address in the SIT to see whether the PL object has already been swapped into the memory. If hit, LeakSurvivor modifies the registers that contain the reserved address to the new memory address found in the SIT. Otherwise, LeakSurvivor reads the PL object from the leak space to the memory and inserts the mapping from the reserved address to the new memory address into the SIT. After this, LeakSurvivor updates the PL object's outgoing references based on the result of SOT look-up. Finally, it notifies the leak detector of the false positive.

4 Evaluation Methodology

Our experiments are conducted on two machines, each has a 2.8 GHz Intel Xeon processor, 512 KB L2 cache, 1 GB of memory, and 120 GB hard drive, running with Linux 2.4.27-no-SMP. They are connected by a 100 Mbps Ethernet connection. We run the evaluated applications on one machine. For server programs, we run their clients on the other machine.

We implement LeakSurvivor on top of Jikes RVM 2.4.2, a high performance Java-in-Java virtual machine developed by IBM [7]. The leak detector deployed in LeakSurvivor is Sleigh, a space-efficient, low false positive leak detection tool for Java applications [12]. LeakSurvivor uses a mark-sweep garbage collector because currently Sleigh does not support moving objects. However, we do not see any difficulty in porting LeakSurvivor to copying garbage collectors. For all the experiments, we use the fast adaptive optimizing compiler, and default heap size (100 MB) in Jikes RVM. Additionally, we set the base of the stale counter to 4, the default value in Sleigh.

Application	Version	#LOC	Description
Eclipse	3.1.2	2813358	an integrated development environment
SPECjbb-2000	1.02	30486	a server simulator
Jigsaw	2.0.2	121776	a web server
SPECjbb-2000-fp	1.02	30586	SPECjbb2000 with controlled false positives
DaCapo	2006-10 MR2	N/A	A standard Java benchmark suite

Table 1: Applications used in evaluation

We evaluate LeakSurvivor with four different applications shown in Table 1, including a popular integrated development environment (Eclipse [2]), a simulator of an order-processing server (SPECjbb2000 [6]), a web server (Jigsaw [4]), and a standard Java benchmark suite consisting of 11 benchmarks(DaCapo [10]). Among them, Eclipse and Jigsaw contain memory leaks originally introduced by their developers, while SPECjbb2000 contain a memory leak injected by us. In addition, to evaluate the performance of LeakSurvivor under different false positive rates, we modify

SPECjbb2000 by accessing leaked objects with different controlled rates and rename it as SPECjbb2000-fp. We use DaCapo benchmarks to measure the performance overhead of LeakSurvivor when there is no leaks.

In this paper, we design three sets of experiments to evaluate the key aspects of LeakSurvivor:

- The first set evaluates the functionality of LeakSurvivor in tolerating memory leaks at runtime. In this set of experiments, we run the applications with memory leaks being triggered as fast as possible in two different configurations: one with LeakSurvivor, i.e., running programs in Jikes RVM with Sleigh and LeakSurvivor, and the other without LeakSurvivor, i.e., natively running programs in JikesRVM without Sleigh and LeakSurvivor. We collect the average response time for client programs (Eclipse) and average throughput for server programs (SPECjbb2000 and Jigsaw) and the detailed statistics, e.g., GC time, memory usage, etc.

- The second set evaluates the performance overhead of LeakSurvivor when there is no memory leaks, which tells us how LeakSurvivor performs during normal program execution. We measure the runtime and space overhead.

- The third set evaluates the sensitivity of LeakSurvivor to different false positive rates of leak detectors. We test SPECjbb2000-fp with LeakSurvivor using different controlled false positive rates.

For Eclipse, each comparison between two directories triggers memory leaks. We adopt the script developed at the University of Texas at Austin [12] to repeatedly compare the source code of two versions of Jikes RVM: 2.4.0 and 2.4.1 (109 of 898 files differ; textual diff is 874 lines). We measure the time for each round of such comparison and consider it as the response time.

For SPECjbb2000, we inject a memory leak by modifying the object access order of a transaction queue from First-in First-out to Last-in First-out. We run it with these leak-triggering transactions in an infinite loop and measure the average transactions per second during last 60 seconds as the throughput.

For Jigsaw, the leak occurs during a client disconnection. We write a client with 10 threads, each sends 20 connect and disconnect requests per second concurrently for triggering the memory leaks. In the meantime, we use ab [1] in the Apache Web server suite as a normal client and measure the throughput of Jigsaw. The client ab creates 10 concurrent connections. For each connection, ab continuously sends out requests to fetch different files whose sizes range in 1 KB, 2 KB, 4 KB, ..., 256 KB with uniform distribution.

5 Experimental Results

5.1 Overall Results

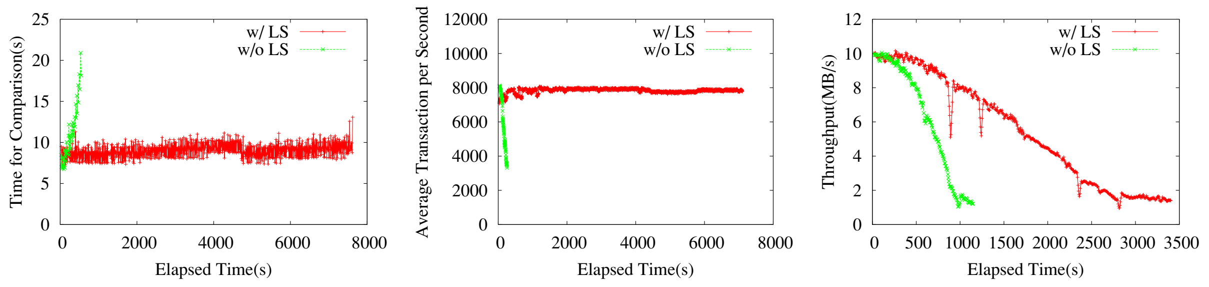
Table 2 shows the overall effectiveness of LeakSurvivor in tolerating memory leaks. For each buggy application, the table shows whether the program crashes and how long it executes before crash, if any, in two different configurations: one with LeakSurvivor and the other without LeakSurvivor. The table also shows the ratio of the program performance with LeakSurvivor to the performance without LeakSurvivor. The performance is averaged over the period when both of the programs are alive. The last three columns in the table show the total sizes of swapped-out objects and live objects, i.e., objects reachable from $roots$, when programs are executing with LeakSurvivor and without LeakSurvivor.

As shown in Table 2, LeakSurvivor successfully tolerates memory leaks for two of the three applications, including Eclipse and SPECjbb2000. Without LeakSurvivor, they crash after 555 seconds' and 328 seconds' execution respectively. This is because continuously-leaked memory objects gradually occupy the heap space and quickly push the total size of live objects to reach their upper bounds, 54.7 MB and 55.3 MB respectively. Note, the live object region is part of the whole heap in Jikes RVM, whose default maximal size, 100 MB, is used in our experiments. In contrast, with LeakSurvivor, the two applications do not crash even after around 2 hours and still maintain relatively stable performance. In this situation, we believe LeakSurvivor successfully tolerate memory leaks and forcefully terminate the execution. LeakSurvivor can tolerate memory leaks because it swaps out leaked memory objects and releases virtual space occupied by them for future uses. Therefore, the total size of the live objects can be kept under a certain percentage of the maximal heap size. For example, LeakSurvivor swaps out 673 MB memory objects and maintains its live object size under 34 MB for Eclipse before we terminate it.

LeakSurvivor does not survive Jigsaw since it has "semantic" memory leaks, which are not detected by the integrated leak detector. The class name of leaked objects in Jigsaw is $HashTableEntry$, which is a $< key, value >$ pair. However, the leak detector can only report the $value$ part as PL objects. This is because the key part is still accessed from time to time for hash table lookup and rehashing. Therefore, LeakSurvivor only swaps out the $value$ part of the leaked entry and the key part in the heap cumulatively degrades the program performance and eventually crashes the program. Nonetheless, LeakSurvivor still helps extend the program lifetime by two times — the program crashes at 3402 seconds when running with LeakSurvivor and crashes at

Apps	Tolerable?		Live Time (sec)		Performance Ratio*	Total Size of Objects (MB)		
	w/ LS	w/o LS	w/ LS	w/o LS	(w/ LS : w/o LS)	SO (w/ LS)	Live (w/ LS)	Live (w/o LS)
Eclipse	Yes	No	> 7626	555	1.27:1	> 673	34	55
SPECjbb2000	Yes	No	> 7135	328	1.24:1	> 857	28	55
Jigsaw	No	No	3402	1147	1.46:1	82	45	52

Table 2: Overall results of LeakSurvivor. w/ LS means with LeakSurvivor, w/o LS means without LeakSurvivor, SO means Swapped-Out. Live objects are objects reachable from roots. *Performance ratio is the ratio of averaged program performance (response time for Eclipse and throughput for SPECjbb2000 and Jigsaw) with LeakSurvivor to that without LeakSurvivor during the period when both are still executing.



(a) Eclipse (b) SPECjbb2000 (c) Jigsaw

Figure 6: Overall performance for applications w/ and w/o LeakSurvivor

1147 seconds when running without LeakSurvivor. This is because LeakSurvivor swaps out part of the leaked objects, 82 MB out of 120 MB total leaked objects.

Table 2 also shows that LeakSurvivor improves the overall program performance when memory leaks occur continuously. For example, Jigsaw with LeakSurvivor provides 46% more average throughput than that without LeakSurvivor when both are alive. This is mainly because continuously leaked objects occupy more and more heap space, which in turn invokes GC more often, when executing programs without LeakSurvivor. Furthermore, the object traversal time in GC phases also increases due to more live objects. On the contrary, executing programs with LeakSurvivor does not suffer from these two problems because LeakSurvivor keeps swapping out leaked objects and maintains low heap pressure.

5.2 LeakSurvivor Performance with Application Leaks

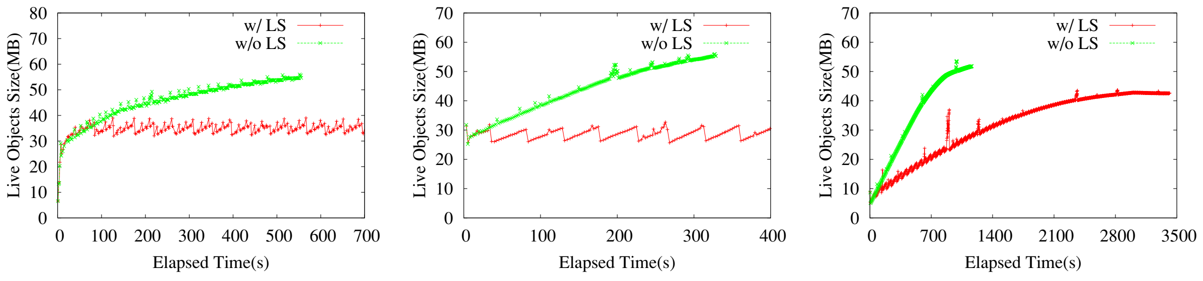
5.2.1 LeakSurvivor Performance

We measure the performance for the three applications with and without LeakSurvivor when memory leaks occur continuously. As shown in Figure 6, LeakSurvivor helps avoid cumulative performance degradation and software failures for Eclipse and SPECjbb2000. For example, with LeakSurvivor, the comparison time for Eclipse is within the range from 7.2 to 11.3 seconds without increasing trends before we manually terminate the program at around 2 hours. In contrast, without LeakSurvivor, the comparison time for Eclipse drastically increases from 6.7 to 20.9 seconds within around 9.5 minutes before its crash. Although LeakSurvivor cannot fully tolerate leaks in Jigsaw, it alleviates the degree of cumulative performance degradation caused by continuously leaked memory objects. With LeakSurvivor, Jigsaw takes around 2580 seconds for the throughput degraded to below 2 MB per second, while it only takes about 920 seconds for the same performance degradation without LeakSurvivor.

Figure 6 also shows that the performance of programs without LeakSurvivor is slightly better than the performance of programs with LeakSurvivor at the initial period of program execution. For example, Eclipse without LeakSurvivor outperforms Eclipse with LeakSurvivor by 9.7% on average during the execution period from 0 to 200 seconds. This is because, at the initial period of program execution, the runtime overhead incurred by LeakSurvivor and the leak detector is larger than the performance degradation due to continuous memory leaks.

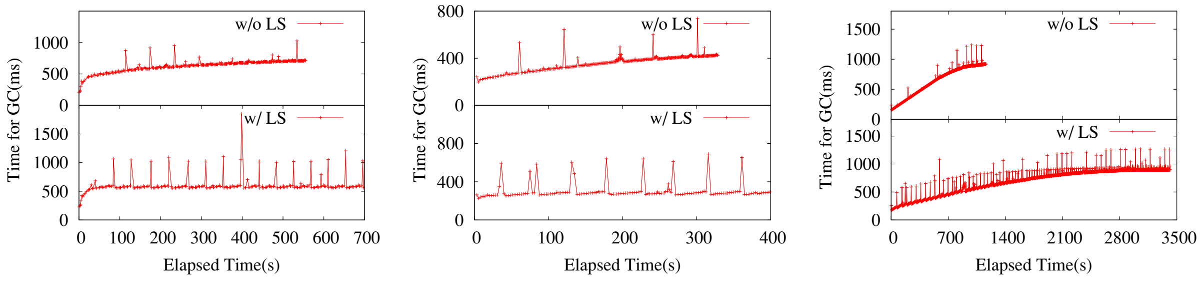
5.2.2 Live Objects Size

Figure 7 shows that LeakSurvivor can effectively control the growth of live object sizes when memory leaks continuously occur, which contributes significantly to the relatively stable overall program performance. For example, without LeakSurvivor the live object size in Eclipse increases sharply from 30 MB to 55 MB before the program crashes. In contrast, with LeakSurvivor the live object size in Eclipse is bounded between 33 MB and 41 MB. This is because LeakSurvivor swaps out PL objects periodically, which in most cases brings down the live object size from 38 to 34 MB when the live object size reaches 38 MB due to continuously-leaked



(a) Eclipse (b) SPECjbb2000 (c) Jigsaw

Figure 7: Live object sizes for w/ and w/o LeakSurvivor



(a) Eclipse (b) SPECjbb2000 (c) Jigsaw

Figure 8: GC performance for w/ and w/o LeakSurvivor

memory objects. When Eclipse is forcefully terminated, LeakSurvivor swaps out 673 MB PL objects, while its live object size is still under 37 MB. For better readability, Figure 7(a) and (b) show shorter time ranges.

5.2.3 *GC Performance*

Figure 8 shows the time for performing one GC phase, which is directly related to the performance in garbage-collected programs. With LeakSurvivor, the GC time is mostly stable except for some spikes, which are caused by swapping out PL objects at certain GC phases synchronously. However, the overhead of the swap-out process in LeakSurvivor is amortized among all the GC phases and thus not reflected in the overall program performance. For example, the average GC time for Eclipse is 0.607 seconds, while the average non-spike GC time for Eclipse is 0.585 seconds. In other words, the amortized overhead caused by the swap-out process in LeakSurvivor is 3.7 % for GC time.

In contrast, without LeakSurvivor, the GC time steadily increases. For example, the GC time for Eclipse increases from 0.414 to 0.711 seconds right before it crashes. The increased GC time partially contributes to the performance degradation shown in Figure 6.

More importantly, without LeakSurvivor, the GC frequency dramatically increases (not readable in the figure), which severely degrades the program performance. For example, at the beginning of Eclipse's execution, the GC frequency is 23 GCs per minute, and it becomes 51 GCs per minute right before it crashes. In contrast, the GC frequency remains the stable for programs with LeakSurvivor. For example, the GC frequency for Eclipse with LeakSurvivor falls within the range of 21 to 30 GCs per minute.

Apps	SOT Size (MB)	SIT Size (MB)	LS Buf Size (MB)
Eclipse	0.56	0.02	8
SPECjbb2000	0.66	0.20	8
Jigsaw	0.53	0.00008	8

Table 3: LeakSurvivor space overhead. LS means Leak Space, Buf means Buffer.

5.2.4 *LeakSurvivor Space Overhead*

Table 3 shows the space overhead of LeakSurvivor for the three applications before their termination or crash. The space overhead of LeakSurvivor is relatively small (8.02 MB – 8.86 MB). It consists of three parts: SOT, SIT, and the leak space buffer. The space overhead comes predominantly from the leak space buffer, which has fixed size, i.e., 8 MB, for better swap-out performance. The SIT size depends on false positive rates, which are 0.001%, 0.17%, and 0.0004% for Eclipse, SPECjbb2000, and Jigsaw respectively. The SOT size is small and increases very slowly compared with the size of swapped out objects. For example, the SOT size for Eclipse increases from 0.51 MB (0 entries) to 0.56 MB (35270 entries) before it is forcefully terminated at

Apps	SOT Entry #	SwapOut Obj #	SIT Entry #	SIT Hit #	Buf Hit #	Disk Hit #	Exp #
Eclipse	35270	19293061	200	155816	189	11	78163
SPECjbb2000	41419	15020902	25277	59285	18967	6310	46219
Jigsaw	108	2831100	10	10	10	0	20

Table 4: Swap-in & swap-out statistics. Buf means Buffer, Exp means Exception. SIT Hit #, Buffer Hit #, and Disk Hit # are the number of hits when LeakSurvivor looks up reserved addresses in the swap-in table, the leak space buffer, and disks, respectively, during swap-in operations.

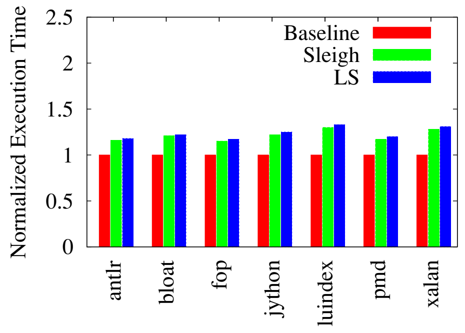


Figure 9: Performance of DaCapo benchmarks. We run the benchmarks in three different configurations: (1) Baseline, i.e., without Sleigh and LeakSurvivor; (2) Sleigh, i.e., only enabling Sleigh; (3) LS, i.e., enabling both Sleigh and LeakSurvivor.

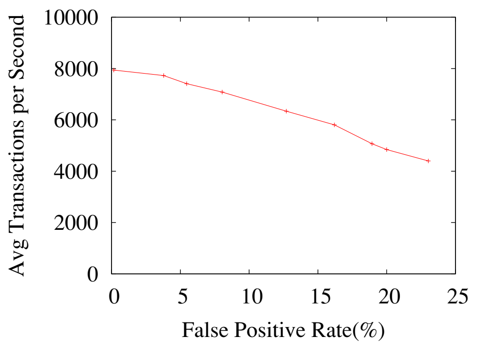


Figure 10: LeakSurvivor performance for SPECjbb2000 with different false positive rates

7626 seconds, after swapping out 19293061 PL objects with total size of 673 MB.

There are two reasons for the small SOT size. First, LeakSurvivor swaps out the children PL objects before swapping the parent so that it does not need one SOT entry to record the outgoing reference from the parent PL object. Second, the SOT uses reference counters to share entries if possible.

5.2.5 *Exceptions and False Positives*

Table 4 shows that SIT and the leak space buffer help reduce the swap-in overhead. First, once an object has been swapped in, all the subsequent exceptions incurred at the same reserved address only need to search SIT and update new references. More than 70% of swap-in operations belong to this category. Second, more than 75% of the first-time swap-in operations hit in the leak space buffer. For this case, we only need a *memcpy* to move the PL object back from the in-memory buffer without disk reads. In addition, Table 4 shows the false positive rates of the leak detector (SIT Entry # / SwapOut Obj #), i.e., 0.001%, 0.17%, and 0.0004% for Eclipse, SPECjbb2000, and Jigsaw respectively, are low.

5.3 LeakSurvivor Overhead without Application Leaks

We measure the runtime overhead incurred by LeakSurvivor when running with programs that have no memory leaks. Figure 9 shows the performance of DaCapo benchmarks in three different configurations: baseline without Sleigh and LeakSurvivor, with Sleigh, and with LeakSurvivor (including Sleigh). The results show that LeakSurvivor incurs small runtime overhead, i.e., 2.5%, on top of Sleigh. The overhead mainly comes from one extra PL object checking for each object traversal during GC phases and slightly increased boot image size (51.30MB for LeakSurvivor v.s. 50.61MB for Sleigh).

Compared with the baseline, LeakSurvivor (including Sleigh) incurs 23.7% runtime overhead, which is mainly caused by the instrumentation code for tracking memory object accesses. Additionally, LeakSurvivor's internal data structure imposes no space overhead when there is no leaks because the SOT, SIT, and the leak space buffer are created on demand.

5.4 Sensitivity Study

We conduct experiments to examine how sensitive the LeakSurvivor is to different false positive rates. As shown in Figure 10, the performance of LeakSurvivor can be severely hurt by the large false positive rates. For example, the average throughput decreases from 7920 to 4400 transactions per second when the false positive rates increases from 0.16% to 23.03%. This is because the overhead incurred by the swap-in process increases dramatically once the false positives increase. Figure 10 also shows that LeakSurvivor's performance is acceptable if the false positive rates are within 5%, which serves as a guide for selecting leak detectors.

6 Related Work

Our work builds on much previous work. Due to the space limit, this section briefly describes the related work that is not discussed in previous sections.

General Fault Tolerance. A considerable amount of research, such as software rejuvenation [30, 22, 11], micro-rebooting [15, 14], and checkpointing-based re-execution methods [8, 45, 16, 32], has been conducted in surviving general software failures. These approaches can deal with failures caused by memory leaks, but programs still suffer cumulative performance degradation before restart and may repetitively fail even after re-execution given that most memory leaks are deterministic. Recently proposed approaches including Rx [37] and DieHard [9] exploit data diversity for tolerating many types of deterministic bugs such as buffer overflows, dangling pointers, and double free, but they do not address memory leaks.

Memory Leak Detection. Memory leaks can be detected statically or dynamically. While static methods [28] can detect some memory leaks without incurring runtime overhead, they may report many false positives due to lack of runtime information. To dynamically detect memory leaks, Purify [27] and Valgrind [39] track object references to identify unreachable objects for C/C++ programs. For garbage-collected programs, unreachable objects are implicitly reclaimed by GCs [23] and thereby only useless objects threaten system availability. JRockit [43], .NET Memory Profiler [40], JProbe [5], LeakBot [33], and Cork [31], track heap updates to identify objects that cause the heap to grow. Other methods, such as SWAT [18], SafeMem [36], and Sleigh [12], leverage object lifetime or staleness (time since last use) to identify leaked objects.

Other Related Work. Bookmarking collection [29] can be used to save physical memory from memory leaks at page granularity although its primary goal is to reduce garbage collection overhead. It has the same problem as Melt [13] because it fails to reclaim the heap space.

LeakSurvivor reclaims the PL objects and thereby the space can be reused by the application, which may result in "hot" objects clustered together. This may improve program performance as done in various prior work on data layout optimizations [19]. The object recovery mechanism exploited by the Swap-In component is related to OS page faults [41]. The SIT is related to forwarding pointer techniques used by Copying GC [17, 21].

7 Conclusions and Future Work

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.

We evaluate LeakSurvivor with three applications that contain continuous memory leaks. The results show that LeakSurvivor can effectively tolerate memory leaks for two of the applications (Eclipse and JBB2000) and extend the lifetime of one application (Jigsaw) by two times. Without LeakSurvivor, all the three applications severely suffer cumulative performance degradation and eventually crash within 20 minutes. This indicates that safely reclaiming virtual memory space occupied by potentially-leaked objects is an effective way to tolerate memory leaks. In addition, LeakSurvivor improves the performance of programs with continuous memory leaks by 24%–46%.

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.

8 Acknowledgments

The authors would like to thank the anonymous reviewers for their invaluable feedback on this paper. We appreciate that Yu Chen and Ming Wu in Microsoft Research Asia provided insightful comments at the very early stage of this project. We are indebted to Michael Bond and Dr. Kathryn McKinley in generous sharing and answering questions about Sleigh and the script in addition to their invaluable comments. We thank Enhua Tan for setting up the experiment platform, as well as Shuang Liang, Feng Chen and Xiaoning Ding for discussion. We appreciate enormous support for this project from Dr. Xiaodong Zhang.

References

[1] ab - Apache HTTP Server Benchmarking Tool. http://httpd.apache.org/docs/2.0/programs/ab.html.

[2] Eclipse - An Open Development Platform. http://www.eclipse.org.

[3] Java 2 Platform, Enterprise Edition. http://java.sun.com/j2ee/reference/whitepapers/j2ee_guide.pdf.

[4] Jigsaw - W3C's Server. http://www.w3.org/Jigsaw/.

[5] JProbe. http://www.javaperformancetuning.com/tools/jprobe/.

[6] SPECjbb2000, A Java Business Benchmark. http://www.spec.org/osg/jbb2000.

[7] B. Alpern, S. Augart, S. M. Blackburn, and et al. The Jikes Research Virtual Machine Project: Building An Open-source Research Community. *IBM Syst. J.*, 44(2):399–417, 2005.

[8] C. Amza, Armando Cox, and W. Zwaenepoel. Data Replication Strategies for Fault Tolerance and Availability on Commodity Clusters. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks*, Jun 2000.

[9] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *ACM conf. on Programming language design and implementation(PLDI'06)*, 2006.

[10] S. M. Blackburn, R. Garner, and et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *ACM conf. on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'06)*, New York, NY, USA, October 2006.

[11] Andrea Bobbio and Matteo Sereno. Fine Grained Software Rejuvenation Models. In *International Computer Performance and Dependability Symposium*, Sep 1998.

[12] Michael D. Bond and Kathryn S. McKinley. Bell: Bit-encoding Online Memory Leak Detection. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, Oct 2006.

[13] Michael D. Bond and Kathryn S. McKinley. Tolerating Memory Leaks. Technical report, UT Austin Technical Report TR-07-64, Dec 2007.

[14] George Candea, James Cutler, Armando Fox, Rushabh Doshi, Priyank Garg, and Rakesh Gowda. Reducing Recovery Time in A Small Recursively Restartable System. In *Intl. Conf. on Dependable Systems and Networks*, Jun 2002.

[15] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A Technique for Cheap Recovery. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, Dec 2004.

[16] Y. Chen, James S. Plank, and Kai Li. CLIP: A Checkpointing Tool for Message-passing Parallel Programs. In *ACM/IEEE Supercomputing Conference (SC'97)*, Nov 1997.

[17] C. J. Cheney. A Non-recursive List Compacting Algorithm. *Communications of the ACM*, 13(11):677, November 1970.

[18] Trishul M. Chilimbi and Matthias Hauswirth. Low-overhead Memory Leak Detection Using Adaptive Statistical Profiling. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, Oct 2004.

[19] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious Structure Layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[20] A. Diwan, E. Moss, and R. Hudson. Compiler Support for Garbage Collection in A Statically Typed Language. In *ACM Conf. on Programming Language Design and Implementation (PLDI '92)*, June 1992.

[21] Robert R. Fenichel and Jerome C. Yochelson. A Lisp Garbage Collector for Virtual Memory Computer Systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[22] S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi. On The Analysis of Software Rejuvenation Policies. In *Proceedings of the Annual Conference on Computer Assurance*, Jun 1997.

[23] Richard Gillam. An Introduction to Garbage Collection. http://oss.software.ibm.com/icu/docs/papers/cpp_report/an_introduction_to_garbage_collection_part_i.html.

[24] Maayan Goldstein, Onn Shehory, and Yaron Weinsberg. Can Self-healing Software Cope with Loitering? In *SOQUA '07: Fourth international workshop on Software quality assurance*, pages 1–8, New York, NY, USA, 2007. ACM.

[25] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[26] E. Grochowski and R. D. Halem. Technological Impact of Magnetic Hard Disk Drives on Storage Systems. *IBM Syst. J.*, 42(2):338–346, 2003.

[27] R. Hasting and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the USENIX Winter 1992 Technical Conference*, Dec 1992.

[28] David L. Heine and Monica S. Lam. A Practical Flow-sensitive and Context-sensitive C And C++ Memory Leak Detector. In *ACM conf. on Programming language design and implementation (PLDI'03)*, 2003.

[29] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage Collection without Paging. *SIGPLAN Not.*, 40(6):143–153, 2005.

[30] Yennun Huang, Chandra Kintala, Nick Kolettis, and N. Dudley Fulton. Software Rejuvenation: Analysis, Module and Applications. In *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing*, Jun 1995.

[31] Maria Jump and Kathryn S. McKinley. Cork: Dynamic Memory Leak Detection for Java. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan 2007.

[32] David E. Lowell and Peter M. Chen. Discount Checking: Transparent, Low-Overhead Recovery for General Applications. Technical report, CSE-TR-410-99, Univ. of Michigan, 1998.

[33] Nick Mitchell and Gary Sevitsky. LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. In *European Conference on Object-Oriented Programming, (ECOOP '03)*, 2003.

[34] Huu Hai Nguyen and Martin Rinard. Detecting and Eliminating Memory Leaks Using Cyclic Memory Allocation. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 15–30, New York, NY, USA, 2007. ACM.

[35] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Plug: Automatically Tolerating Memory Leaks in C and C++ Applications. Technical report, UMass CS Technical Report 08-09, April 2008.

[36] Feng Qin, Shan Lu, and Yuanyuan Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *Intl. Symposium on High-Performance Computer Architecture*, Feb 2005.

[37] Feng Qin, Joe Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating Bugs as Allergies – A Safe Method to Survive Software Failure. In *Proceedings of the 20th ACM Symposium on Operating System Principles*, Oct 2005.

[38] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of A Log-structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[39] J. Seward. Valgrind, An Open-source Memory Debugger for x86-GNU/Linux. available at URL http://developer.kde.org/ sewardj/.

[40] SciTech Software. .NET Memory Profiler. http://www.scitech.se/-memprofiler/.

[41] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[42] US-CERT. US-CERT Vulnerability Notes Database. http://www.kb.cert.org/vuls.

[43] BEA WebLogic. JRockit: Java for The Enterprise. http://www.bea.com/content/news_events/white_papers/BEA_Rockit_wp.pdf.

[44] Ting Yang, Emery D. Berger, Scott F. Kaplan, J. Eliot B. Moss, and B. Moss. CRAMM: virtual Memory Support for Garbage-collected Applications. In *USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[45] Yuanyuan Zhou, Peter M. Chen, and Kai Li. Fast Cluster Failover Using Virtual Memory-Mapped Communication. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, Jun 1999.