SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

KAL: Kernel-assisted Non-invasive Memory Leak Tolerance with a General-purpose Memory Allocator

Jinkyu Jeong[1,†], Euiseong Seo[2], Jeonghwan Choi[3]
Hwanju Kim[1], Heeseung Jo[1], Joonwon Lee[3,*]

CA Lab., CS Dept., Korea Advanced Institute of Science and Technology, Korea[1]
School of ECE, Ulsan National Institute of Science and Technology, Korea[2]
School of ICE, Sungkunkwan University, Korea[3]

SUMMARY

Memory leaks are a continuing problem in software developed with programming languages such as C and C++. A recent approach adopted by some researchers is to tolerate leaks in the software application and to reclaim the leaked memory by use of specially constructed memory allocation routines. However such routines replace the usual general purpose memory allocator and tend to be less efficient in speed and in memory utilization.

We propose a new scheme which coexists with the existing memory allocation routines and which reclaims memory leaks. Our scheme identifies and reclaims leaked memory at the kernel level. There are some major advantages to our approach: (1) the application software does not need to be modified; (2) the application does not need to be suspended while leaked memory is reclaimed; (3) a remote host can be used to identify the leaked memory, thus minimizing impact on the application program's performance; (4) our scheme does not degrade service availability of the application while detecting and reclaiming memory leaks.

We have implemented a prototype that works with the GNU C library and with the Linux kernel. Our prototype has been tested and evaluated with various real world applications. Our results show that the computational overhead of our approach is around 2% of that incurred by the conventional memory allocator in terms of throughput

*Correspondence to: Joonwon Lee, joonwon@skku.edu, School of ISC, Sungkyunkwan University, Suwon, Rep. of Korea
†E-mail: jinkyu@calab.kaist.ac.kr
Contract/grant sponsor: This work was supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government(MEST); contract/grant number: 2010-0080381
Contract/grant sponsor: This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology; contract/grant number: 2009-0089491

Received 13 April 1999
Revised 23 September 2002

and average response time. We also verified that the prototype successfully suppressed address space expansion caused by memory leaks when the applications are run on synthetic workloads.

KEY WORDS: memory leak toleration, memory leak, garbage collection, Lea memory allocator, operating systems

1. INTRODUCTION

A memory leak is a dynamic memory object that is unnecessary but continues occupying memory space during the execution of software. Although from the programmer's view there are diverse types of mistakes causing memory leaks, the result is the same; an application fails to maintain control of dynamic memory objects because it loses the pointers to the objects while the application is still running. Memory leaks in iterative codes, even those of small sizes, will result in catastrophic consequences for the whole system. In spite of a great deal of research aimed at resolving memory leaks, they continue to be significant threats to the performance, security and availability of computer systems [35].

The existing research on resolving memory leaks can be classified into three categories; debugging tools [32, 18, 19], conservative garbage collectors [9], and leak tolerant memory allocators [29, 28].

The debugging tools help developers to identify and remove the memory leaks in source codes. Although the majority of memory leaks can be resolved by the proactive use of the debugging tools at the development stage, because there is no ideal algorithm that can distinguish memory leaks from normal memory object manipulation, some memory leaks may remain in the final products.

A conservative garbage collector ensures that systems are tolerant of memory leaks by detecting and reclaiming unreachable objects while the systems are running. It marks reachable memory objects from the *rootset* of an application and collects unmarked memory objects in the heap. Modification of the existing memory allocator is required to achieve efficient garbage collection [7].

While the garbage collector focuses on reclaiming memory leaks, the leak tolerant memory allocators totally mitigate the adverses effects of memory leaks. The leak tolerant memory allocators reorganize the placement of dynamic memory objects in order to ensure that memory leaks are automatically swapped out or to be eliminated by *cyclic memory allocation* [25, 29, 28].

The two latter approaches commonly require the modification of the existing general-purpose memory allocator such as the widely used Lea allocator [24]. Generally, the general-purpose allocator is effective in terms of both performance and space [4]. Although those new memory allocators are tolerant of memory leaks in runtime, this approach could compromises either performance or space, or both.

In this paper, we propose a novel memory leak tolerance scheme that is non-invasive with respect to the memory allocators. To this end, we separate the memory leak detection and resolving mechanism from the memory allocator by placing them on the kernel level. In our scheme, the general-purpose allocator services memory allocation for applications while our solution on the kernel level provides leak toleration.

The memory leak tolerance is carried out in two phases; 1) detecting leaks and 2) reclaiming the memory space allocated by the leaks. The primary role of the kernel level mechanism is to provide the leak detection phase with a memory snapshot of the application software. Note that the detected memory leaks must be reclaimed using user level function *free()*. Since our scheme works in the kernel level, we use up-calling of *free()* to reclaim leaked memory.

This suggested scheme includes several enhancements from the existing memory leak tolerant scheme by adopting following techniques;

- Taking a snapshot in a kernel extension: Note that modern commodity operating systems provide a kernel extension mechanism, which enables a procedure to be dynamically inserted into the kernel. Our solution can be implemented as a kernel extension and be applied to running applications without modifying, suspending and rebooting of the applications.
- Separation of leak detection: Leak detection, a part of our scheme, can be run on a physically separated machine. This separation minimizes performance impact on the target application's performance. Leak detection is conducted upon a memory snapshot sent to the remote host.
- Live service of an application: Taking a memory snapshot of an application is making a clone of application's memory using a copy operation. During taking a snapshot, an application could change its memory contents, since the application is still running. Accordingly it is necessary to suspend application to preserve consistency of the snapshot. A long suspention time, however, compromises the service availability of the application software. To minimize this impact, we adopted the *precopy* [34] approach, which is a well-known method to minimize a downtime of the application during process migration. Since migrating memory content of a process is similar to taking a memory snapshot, the precopy approach could be applied in our scheme.

While a kernel provides multiple sets of contiguously addressed memory pages to a user-level memory allocator, the memory allocator divides the given pages into small memory objects and provides them to an application. Leak detection requires the memory allocator level information, which the kernel cannot be aware of. This semantic gap ensures that it is difficult for a kernel to know where the small memory objects reside.

To eliminate this semantic gap, we suggests a technique, *heap dissection*, that identifies the small memory objects from the page allocation information on the kernel level. Although the actual design and implementation of the heap dissection depends on the type of memory allocator used, this requires little work. We discuss this issue in detail based on the results of our prototype implementation.

We built a prototype called KAL (Kernel-Assisted Leak toleration) in Linux and GNU C library (Glibc) based system. KAL consists of a Linux kernel extension and a leak detector application. The kernel module takes a snapshot and up-calls to the user level allocator for memory leak reclamation. The leak detector application analyzes the application's memory to find leaks. Heap dissection in the leak detector is implemented based on a general memory allocator Lea memory allocator in Glibc. Implementation of heap dissection has a low engineering cost, from analyzing heap construction to making dissection code. The code is only about 200 lines, and the engineering costs are reasonable. We evaluated KAL with a synthetic application and three real-world server applications. While the address space of an application grows by leak injection, our prototype successfully suppresses the address space expansion in runtime. The performance overhead is around 2% that of the normal case.

The rest of this paper is organized as follows. The next section describes the background for our work. Section 3 shows the design and implementation issues of our scheme. Section 4 evaluates the effectiveness of our scheme and demonstrates the overhead incurred by our scheme to various applications. Section 5 reviews the previous leak detection approaches in comparison with our suggested scheme. Finally, we discuss future works and conclude this paper in Section 6.

2. BACKGROUND

This section describes the background for our work. The first subsection presents a representative general-purpose memory allocator, viz., the Lea allocator. The next subsection describes the advantage of the kernel extension mechanism and task's address space management in the Linux kernel. Finally, we discuss our leak detection along with some limitations in Section 2.3.

2.1. The Lea memory allocator

The Lea memory allocator [24] provides both high speed and low memory consumption in comparison to the other memory allocation schemes [3]. It is used in the Linux-based systems as the user-level memory allocator by implementing it in the GNU C library, which is currently the de-facto standard C library.

While the Linux kernel provides a heap space using a `brk` system call, the allocator splits the heap space into small objects that are assigned to user tasks using `malloc`-like functions. The approach employed by this allocator depends on the requested object. *Bins* in the Lea allocator is a data structure that manages free objects based on their size. The overall structure of the bins are shown in Figure 1

Tiny objects with sizes less than 80 bytes are managed in fast bins for quick object allocation. Free objects of equal sizes are stored in singly linked in a fast bin. Freeing a tiny object does not require coalescing fast bins for quick object deallocation. Based on the heuristics of the Lea allocator, fast bins outperforms for repeated small requests such as allocating tree nodes or linked list nodes.

Small objects with sizes less than 512 bytes are managed in bins. A bin is a linked list of free memory objects of equal size. After receiving a request for a memory allocation of a given size, the lea memory allocator finds a free object from the bin with the smallest size that exceeds the requested size. It uses the best-fit heuristic algorithm to find the proper bin for a given request.

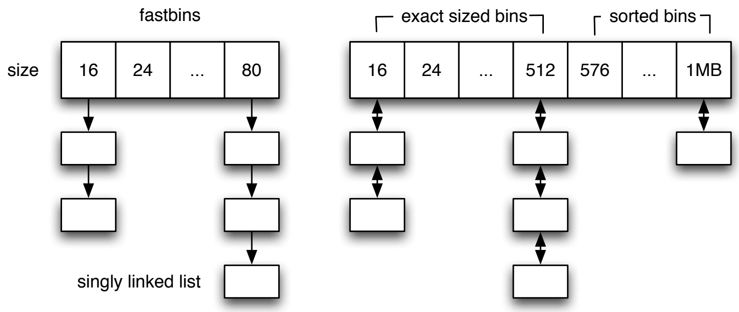


Figure 1. Bins of the Lea allocator for freed object management. Fastbins for small free objects of sizes less than 80 bytes have a single linked list for quick allocation. Exact-sized bins of sizes less than 512 bytes have a double linked list with the same size of objects. Free objects with sizes less than 1Mbytes are stored in a sorted linked list in sorted bins.

Requests for a medium-sized object (less than 1MB) or certain predefined events trigger the Lea memory allocator to coalesce all adjacent objects in the bins. For medium-sized objects, the Lea allocator performs immediate coalescing and splitting, and matches a suitable free object according to the best-fit. In case of splitting a freed object, the remaining space is inserted into the free list of a suitably sized bin, for further reuse.

Large objects are allocated and freed using `mmap`. Mmap is a POSIX-compliant Unix system call that maps files or devices into memory. Linux provides anonymous mappings that maps a certain physical memory space to a virtual memory space in a user task. To allocate a large-sized memory object, the Lea allocator invokes a mmap system call to map anonymous pages of the requested size. When the mmapped object is freed, the mmapped anonymous pages are unmapped via a `munmap` system call.

All free and used objects are enumerated in the heap space of a task as shown in Figure 2. Both types of objects have the same header which contains the size of the object, the size of the previous adjacent object and a *inuse* flag. An inuse flag denotes whether the object is free or used. When the object is free, it also contains two additional headers, a forward pointer and a backward pointer, which is used in the best-fit algorithm as a free list in the bins. Two free adjacent objects are coalesced into a single larger free object. Since every object is adjacent to the previous object and the next object, our heap dissection can be easily processed using header information. From the size header and the inuse flag, the heap dissection can distinguish all objects, including whether the object is free or used. The last object in the heap is specially treated in the allocator. The object, viz., *top*, is a free, but it is not linked in any bins. When no free object is in the bins, top is splitted into two objects: one is for a new request and the other remains as top. When there is no available space in top for a new request, a `brk` system call is invoked in order to enlarge top.

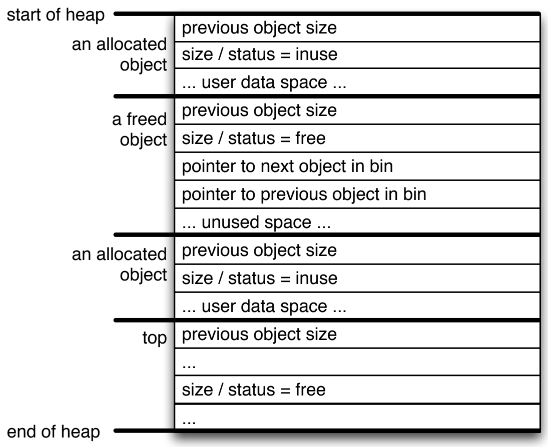


Figure 2. Object placement in heap space. All objects are adjacent to objects in the previous size field, which stores the size of the adjacent object. The status flag (*inuse* flag) denotes whether the object is free or used.

2.2. Kernel extension and address space management in kernel

Most commodity operating systems support a kernel extension mechanism as a loadable kernel module, which is similar to a dynamic loadable library (dll) on the user level. The greatest advantage of a kernel module, which indicates a kernel extension file in Linux, is that a kernel extension mechanism enables a monolithic kernel to be extensible, as the name suggests. The required functionality can be dynamically inserted into the system using a kernel module. When the functionality provided by a kernel module is no longer required, it can be unloaded dynamically.

The functionality in a kernel extension works on the most privileged level on which the kernel is operating. Accordingly, this mechanism enhances a monolithic kernel in terms of both flexibility and performance. A device driver code of the new device can be developed independently by the device vendor. New file system features can be supported on demand by compiling the file system source code for the kernel module. The *balloon* driver [36] in a virtualized environment is a representative application of a kernel extension.

The modern processor provides a virtual memory facility for protection and isolation between user tasks. The kernel manages the physical memory installed in the system, and allows user tasks to use the memory. It authenticates a user task's access control of a memory page and establishes mappings from the physical memory to the user task's linear address (virtual address). A user task accesses the physical memory by accessing the provided virtual memory address.

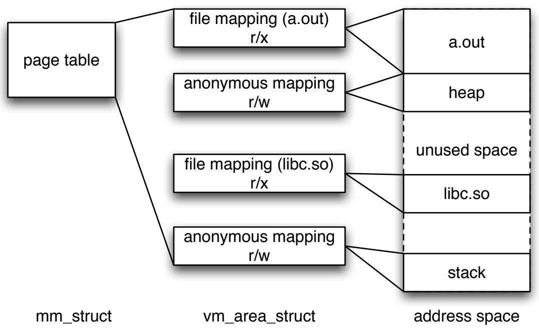


Figure 3. Address space structure in Kernel. Each `vm_area_struct` indicates the type of mapping (file, device, or anonymouse page) and the permission of the address space (Read, Write, or eXecute). All `vm_area_struct`s are stored in `mm_struct`.

Address space can be dynamically expanded and reduced by a `mmap` and `munmap` system call, respectively. `mmap` maps memory, device or file into the address space of a task. A task can access a file or a device by accessing a mapped region of the file in its memory space. By mapping zero-filled pages (anonymous pages in Linux) to a task's memory space, the pages can be used as heap space, stack or data sections. The heap is actually managed by a `brk` system call, but the detailed implementation of `brk` involves a `mmap` system call.

Memory regions, each of which consists of contiguous address space, are managed by the descriptor `vm_area_struct` (vma). The vma includes the first and last address of the region, the mapping type (file mapping, device mapping, and anonymous mapping), and the permission of the memory region (readable, writable and executable). All vmas are linked in the `mm_struct` structure variable, which is the per-task memory management descriptor in Linux. The `mm_struct` also contains the task's page table, which is used for address translation from a virtual address to a physical address by the processor. Figure 3 shows an example of address space management in the Linux kernel.

Although a `mmap` system call provides address space management of a task, two types of mapping are possible. An application actively makes the mapped pages resident by locking the pages using a `mlock` system call, or the page is made resident by demand paging caused by the first fault on the mapped page. The kernel initially makes the page table entry of the mapped page as a dummy; the page table entry is valid but no actual page is present. The first access to the page causes a page fault, inducing the kernel to fill the content of the faulted page.

2.3. Leak detection and limitation

In memory leak detection, we first decide what is regarded as a memory leak. Currently, both unreachable and stale objects are considered as memory leaks. Unlike unreachable objects, stale objects cannot be reclaimed for uncertainty. We also assume that it is impossible to isolate a stale object from the memory consumption of the application. Treating a stale object in C or C++ requires moving the object in the heap area. In C or C++, moving a memory object is actually impossible, because pointer misidentification is possible, which can cause unexpected behavior of the application. Additionally, stale objects cannot be isolated to the swap area because we cannot control the object placement in the heap area. Accordingly, our leak detection approach focuses only on unreachable objects.

Detecting an unreachable object is simple with the well known algorithm *mark and sweep* [26]. It finds pointer links from *rootset* of the process to the heap area, and marks accessible objects; there are several pointers from rootset to the object, either direct or indirect. After the marking phase, the entire heap area is swept and unmarked objects are reported, which have no pointer to them. The unmarked objects are unreachable objects and candidates of reclamation.

Pointer misidentification can also occur while using the mark and sweep algorithm. However, it does not generate false positives regarding whether a memory object is leaked or not. If there is a value that can be misidentified and the value points to a truly leaked object, misidentification can occur by reporting that the leaked object is not a leak. Then, the misidentified leaked object is not reclaimed, but the application continues to be safe. If there is a false positive, the misidentified leak is not in fact a leak, and reclamation will affect the application. However, pointer misidentification only generates false negatives in our scheme. When the application is executed, the value inducing misidentification is changed, the leaked object is reported by the mark and sweep algorithm in the next leak detection, and the leak is reclaimed. Otherwise, the value remains unchanged, application continues to execute with slight extra memory consumption. Although our scheme cannot adapt an advanced treatment for pointer misidentification, such as black listing, pointer misidentification is rare in practice [9].

Some pointers to the middle part of memory objects may exist since many data structures in C/C++ programs use those pointers as links to point other instances (e.g., linked list or tree). This case could make it hard to detect memory leaks since those pointers also cause false negative in leak detection. This false negative case, however, is a fundamental limitation of conservative garbage collection in C/C++ based programs. Though static analysis approaches may reduce the false negative detections, they are currently not considered in our work.

3. DESIGN AND IMPLEMENTATION

KAL consists of two major components as shown in Figure 4. One is the KAL kernel extension (KAL-ext) that is responsible for taking the memory snapshot of a target application. The other is the KAL leak detector (KAL-d) that detects memory leaks from the snapshot. We implemented KAL-ext in Linux kernel version 2.6.21, and KAL-d suited to the Lea memory allocator.

The following subsections describes the details of KAL along with implementation issues. The subsections are ordered according to the KAL processing sequence. This is started by taking a snapshot of the application's memory using KAL-ext in Section 3.1. Heap dissection

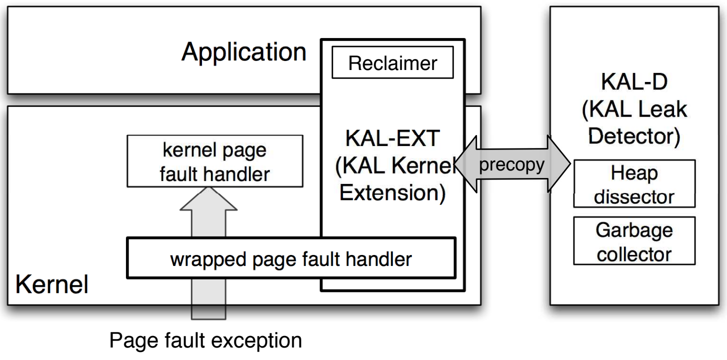


Figure 4. Overall design structure of KAL (Kernel Assisted Leak toleration

and detecting memory leaks using KAL-d are mentioned in Section 3.2. Finally, Section 3.3 deals with reclamation of detected memory leaks.

3.1. KAL kernel extension

The role of a KAL-ext is to export an application's memory for leak detection. There are various approaches to taking an application's memory snapshot. However, it is important to minimize the downtime of an application. The downtime is the time which an application service is unavailable for; downtime is inevitable, because an application changes its memory content while it is running. The well known systemic approach *precopy* [34, 14] is a proper method for taking the memory contents of an application while minimizing the downtime, thereby an application seems to continue working while taking a snapshot. Although, precopy is less effective than a stop and copy approach when the total size of an application's memory is small (on the order of kilobytes), we only assume that the memory size of application is sufficiently large (on the order of tens of megabytes).

Algorithm 1 is the pseudo-code of precopy in KAL-ext. It consists of three phases:

1. **Prepare phase:** initiates taking a snapshot. This phase disables all write permissions of pages belonging to a target application, and exports all pages to KAL-d in line 24.
2. **Precopy phase:** sends pages that is modified in prior round to KAL-d. If the difference of the number of pages written in adjacent rounds is less than a predefined threshold, it jumps to the finalize phase, otherwise, it continues the precopy phase. If pages are modified, a page fault occurs and the page fault handler enables KAL-ext to recognize which page is modified during the round in line 13.
3. **Finalize phase:** stops application and finally sends the last written pages. Then, it also sends detailed information about the target task. For example, the address information of the stack, heap and global data is sent to KAL-d. This phase also exports hardware

Algorithm 1 Pseudo-code of precopying in KAL-ext

```
Require: $\tau$ : task_struct                  ▷ denotes target task's PCB (process control block)
Require: $P$                                                  ▷ set of write disabled pages of $\tau$
Require: $W_R$                                          ▷ set of pages of $\tau$ written in round $R$
Require: $W_\delta$                                 ▷ predefined threshold for exting precopy phase
 1: procedure __WRAP_HANDLE_MM_FAULT($fault\_addr$: uint, $write\_access$)
 2:     if $current = \tau$ then
 3:         if $\neg write\_access$ then
 4:             $\_\_handle\_mm\_fault(fault\_addr, write\_access)$
 5:             $P \cup \{page(fault\_addr)\}$
 6:             disable write permission of $page(fault\_addr)$
 7:         else if $\neg(page(fault\_addr) \in P)$ then
 8:             $\_\_handle\_mm\_fault(fault\_addr, write\_access)$
 9:             $P \cup \{page(fault\_addr)\}$
10:         else
11:             enable write permission of $page(fault\_addr)$
12:         end if
13:         $W_R \cup \{page(fault\_addr)\}$
14:     end if
15: end procedure
16: procedure PRECOPY
17:     replace $\_\_handle\_mm\_fault()$ as $\_\_wrap\_handle\_mm\_fault()$
18:     $R \Leftarrow 0$                                                    ▷ prepare phase
19:     for all $vma$: vm_area_struct in $\tau$ do
20:         for all $page$: uint in $vma$ do
21:             disable write permission of $page$
22:             $P \cup \{page\}$
23:             send content of $page$ to KAL-d
24:         end for
25:     end for
26:     repeat                                                          ▷ precopy phase
27:         $R \Leftarrow R + 1$                               ▷ $R$ denotes round during precopy
28:         $W_R \Leftarrow \phi$
29:         for all $page \in W_{R-1}$ do
30:             disable write permission of $page$
31:             send content of $page$ to KAL-d
32:         end for
33:     until $|W_R| - |W_{R-1}| \leq W_\delta$
34:     stop $\tau$
35:     send metadata of $\tau$ to KAL-d                                ▷ finalize phase
36:     replace $\_\_wrap\_handle\_mm\_fault()$ as $\_\_handle\_mm\_fault()$
37:     resume $\tau$
38: end procedure
```

register states to the leak detector in line 35. Finally, KAL-ext allows the application to resume its execution in line 37.

The KAL-ext is started by accessing an application's memory address space. Modern computer architecture provides memory space isolation between user-level tasks using the virtual memory management unit. Accordingly, the kernel module is first required to access the target application's page table.

In KAL-ext, accessing the memory of an application is simple by exploiting a property of a kernel thread. In Linux, a kernel thread actually has no page table. Because all applications share the page table translating kernel area, the kernel thread gets the page table when they are scheduled. We generate a kernel thread that is attached to a target application's page table, and assign the page table of the created kernel thread as the page table of the target application. Thereby the kernel thread can not only safely access the kernel but also the target application's address space.

To accomplish precopy on the kernel extension level, we must catch the page faults for a target application, since write page faults during precopy can indicate modification of an application's memory. The *kprobe* [22] supports dynamic instrumentation for almost all functions in Linux, which is a proper method used to catch page faults in the kernel extension. The kprobe, however, can degrade the throughput, because of increased amount of exception handling. In our prototype, we dynamically write the jump instruction for the page fault handler to our page fault handler, instead of using kprobe for catching page faults of the target application in line 17 of Algorithm 1.

The prepare phase disables the write permissions of the application's memory pages, which can possess any pointers in the application. The size of these pages is generally less than the size of the entire address space.

In line 34 and 37, In order to suspend and to resume a target application, KAL-ext sends signals `SIGSTOP` and `SIGCONT` respectively. The signal mechanism is a general means of notifying various events from the kernel. If the running state of an application on the kernel level is changed, other external events can induce the application to resume. Sending signals can ensure that the application is no longer updating their memory contents. Actually, kernel codes can write an application's memory using write-like system calls. However, this does not change pointers in the application, because write-like system calls only fill data to the application's buffer. The application cannot be returned to the user level, because `SIGSTOP` is delivered during the application's execution.

Our implementation of KAL-ext must also be effective with two general facilities: demand paging and copy on write. The address space of the application can be expanded by the two facilities in runtime, but we cannot know exactly when the two facilities are used. Since these facilities obviously involve write operations to the page table, we can hook the page table writing operation by disabling write permissions to the page table, but this method usually incurs extra overheads. Accordingly, we carefully consider page faults to the target application, and treat proper actions in each event. These facilities all enable proper page fault exceptions, and set proper flags in the virtual memory data structure in the kernel. For example, when the heap is expanded by demand paging, we insert the expanded page into our write-disabled page and monitor page updating during the precopy phase, as shown in line 4-9. If a page is

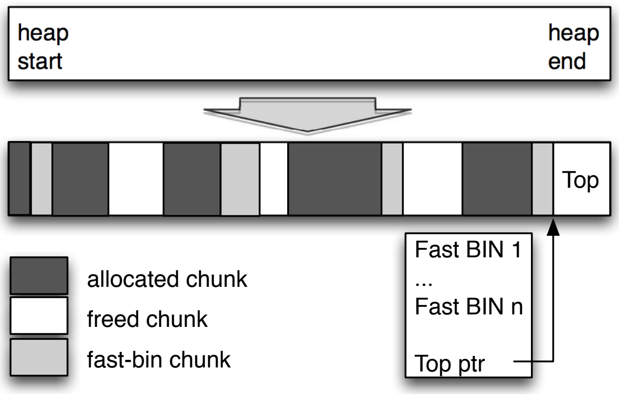


Figure 5. Heap dissection of Lea memory allocator

shrunk, the address space information sent in the finalize phase prevents the old page from disrupting leak detection; the address space information does not contain the fact that the shrunk page no longer belongs, and the shrunk page is not used in leak detection.

3.2. KAL leak detector

KAL-d receives the application's memory snapshot, which is analyzed to detect leaks. Separating the leak detector from the application can gain two benefits. One is that it is executed independently of the target application, ensuring that it does not affect the behavior of the target application. The other benefit is that a separated analyzer can be executed in a different machine. In case of embedded systems, the limited CPU and memory resources limit the leak detection in the systems. However, the separation enables leak detection by executing the leak detector performed in a physically separated machine.

The role of KAL-d is straight forward. It receives contents required for the mark and sweep algorithm. All pages are classified into two categories, heap pages and other pages. Heap pages contain memory objects that may or may not be memory leaks. The other pages are usually rootsets used by the mark and sweep algorithm.

Since a snapshot has only page level information, heap dissection is required for small memory object level information as shown in Figure 5. As mentioned in Section 2.1, small objects contains all required information to dissect the heap into small objects. Using the header information of the first object in the heap, we can construct object level information as follows. The first object is found at the front of the heap in line 2, Algorithm 2. The following objects are discovered by simple address calculation in line 10. To know which objects are free, we also investigate the inuse flag of each object's header in line 6. In particular, the inuse flags

Algorithm 2 Pseudo-code of heap dissection of Lea memory allocator

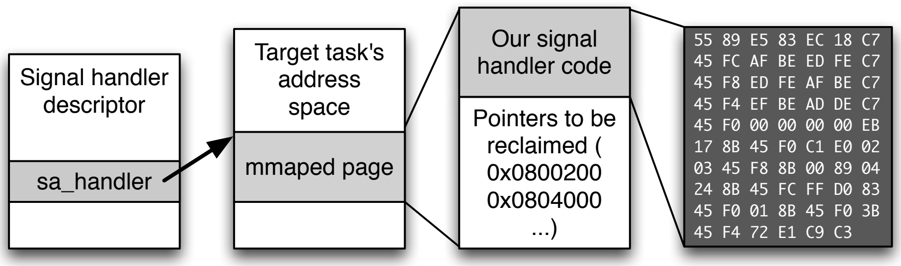
```
Require: $H : \{h | h = (page, content)\}$    ▷ heap area of target task, received from precopy
 1: procedure DISSECT_HEAP($H$: $(page, content)$, $Arena$: malloc_state )
 2:     $ptr \Leftarrow h.page$ $(h.page \leq h'.page$ $where$ $h' \in H)$
 3:     while $ptr \leq h.page+$ PAGE_SIZE $(h.page \geq h'.page$ $where$ $h' \in H)$ do
 4:         $size \Leftarrow *(ptr + 4)$
 5:         $chunk.next \Leftarrow (ptr, size, flag \Leftarrow \phi)$
 6:         if $\neg$ PREV_INUSE $\in size[0:2]$ then
 7:             $chunk.flag \cup \{$FREED_CHUNK$\}$
 8:         end if
 9:         $chunk \Leftarrow chunk.next$
10:         $ptr \Leftarrow ptr + size$
11:     end while
12:     $top \Leftarrow chunk$
13:     assert($Arena.top\_ptr = top.ptr$)
14:     for all $fast\_bin \in top$ do
15:         while $fast\_bin.ptr! =$NULL do
16:             $chunk \Leftarrow find\_chunk\_by\_ptr(fast\_bin.ptr)$
17:             $chunk.flag \cup \{$FREED_CHUNK$\}$
18:             $fast\_bin.ptr \Leftarrow chunk.ptr + 8$
19:         end while
20:     end for
21: end procedure
```

for small objects are meaningless, since the small objects are specially managed using a *fast bin* linked list, which was denoted as a free list of small objects in Section 2.1. As shown in the figure, *arena*, the metadata structure of the Lea allocator, contains the headers of the fast bin linked list. To find these free fast bin objects, we found the arena data structure in the global data section of Glibc. In our implementation, we used a trick to find arena. Since arena contains the address of a top object, the address is used as a key to find arena in the Glibc data section. The address of a top object is determined at the forepart of heap dissection.

Based on the objects generated by heap dissection, we applied the mark and sweep algorithm. In the finalize phase, KAL-ext exports all necessary information for the mark and sweep algorithm. We used a general mark and sweep algorithm, so we omitted a detailed description of our procedure.

3.2.1. Separation

Garbage collection on a snapshot is slower than native garbage collection, because dereferencing of pointers is fully emulated by software on the snapshot. For example, the pointer 0x800000 is directly dereferenced in the original machine, but the pointer dereferencing must be emulated in KAL-d, since it works on the snapshot. This emulated dereferencing consumes a great deal of CPU time and can affect the system performance. An alternative is separating the


Signal handler descriptor
sa_handler
Target task's address space
mmaped page
Our signal handler code
Pointers to be reclaimed (0x0800200 0x0804000 ...)

```
55 89 E5 83 EC 18 C7
45 FC AF BE ED FE C7
45 F8 ED FE AF BE C7
45 F4 EF BE AD DE C7
45 F0 00 00 00 00 EB
17 8B 45 F0 C1 E0 02
03 45 F8 8B 00 89 04
24 8B 45 FC FF D0 83
45 F0 01 8B 45 F0 3B
45 F4 72 E1 C9 C3
```

Figure 6. Reclaming memory leaks using signal handler in kernel.

leak detector to a different physical machine. KAL-ext sends pages to a different machine via the network instead of page copy. Because the kernel thread directly accesses pages of the application's space, it can reduce the copy for network exporting.

Separating not only the execution of the application but also the processing resources can benefit several cases. First, execution separation guarantees that the target application continues operating with a real-world workload [13]. Second, performance separation can be adapted in resource constrained embedded systems. When the system has marginal CPU computation power for the original functions, unwanted extra CPU consumption from leak detection can degrade the quality of the system. By separating leak analysis from the system, this unexpected situation becomes rare. Especially, network-based exporting requires a small CPU time, but this has little adverse effect on the original functions.

3.3. Leak reclamation

In this section, we describe the process of reclamation on the kernel level. Because the memory allocator library entirely manages memory objects, we must make the library call `free()` in order to reclaim memory leaks. This is a simple up-call from the kernel space to the user space. The *Signal* mechanism is a popular up-call mechanism in most operating systems, since this is how the kernel sends an event to a user level application. Because the signal handler code is on the user level, the implementation of signal involves calling a user level function on the kernel level by changing to user-mode. We use the signal mechanism to make the library call `free()`.

The address of `free()` is simply calculated from the mapping address of the Glibc shared object file plus an offset of a `free()` symbol in Glibc.

Algorithm 3 is the pseudo-code of calling `free()` in KAL-ext. We first allocate memory pages in the target task's address space; the memory pages are filled for upcall as shown in Figure 6. Because our kernel thread has the same memory management structure (*mm_struct*) as the target application, we can directly call the `do_mmap_pgoff()` function while avoiding

Algorithm 3 pseudo code of library `free()` upcall in kernel extension

```
1:  procedure OUR_SIGNAL_HANDLER($sig$: integer)
2:      void* $(lib\_free)(void * ptr) \Leftarrow (void*)$0xMAGIC0
3:      void** $ptrs \Leftarrow (void * *)$0xMAGIC1
4:      unsigned long $n \Leftarrow$0xMAGIC2
5:      for all $ptr \in ptrs$ do
6:          $lib\_free(ptr)$
7:      end for
8:  end procedure
9:  procedure RECLAIM($ptrs$)
10:     $mmaped\_ptr \Leftarrow do\_mmap\_pgoff($sizeof$(ptrs)+$sizeof$(our\_signal\_handler),$
11:     PROT_READ|PROT_WRITE|PROT_EXEC,
12:     MAP_ANONYMOUS|MAP_EXECUTABLE|MAP_LOCKED)
13:     $copy\_to\_user(mmaped\_ptr, our\_signal\_handler)$
14:     $copy\_to\_user(mmaped\_ptr+$sizeof$(our\_signal\_handler), ptrs)$
15:     0xMAGIC0 $\Leftarrow address\ of$ free()
16:     0xMAGIC1 $\Leftarrow mmaped\_ptr+$ sizeof$(our\_signal\_handler)$
17:     0xMAGIC2 $\Leftarrow |ptrs|$
18:     $sig \Leftarrow$ empty signal number
19:     $set\_signal\_handler(sig, mmaped\_ptr)$
20:     send signal $sig$
21: end procedure
```

problems; it is not allowed to call the `do_mmap_pgoff()` function in a kernel thread context because the function works with a user level task's *mm_struct*. Then, we fill the newly allocated pages with two components: one is a signal handler code that actually calls the library `free()` function, in line 2-4. The other are the pointers that will be passed to `free()`. After filling the pages, it chooses an unused signal to register our signal handler, which will the reclaim memory leaks. Finally, the kernel module sends the chosen signal to the application in line 16. In the kernel, the signal is automatically delivered to the application when the application is returning to user-mode. Then, the registered signal handler is invoked and the handler finally calls the `free()` library function with the given parameters. The chosen signal is delivered only when the target application is in running state. When the application is blocked for waiting I/O, signal delivery to the application could result in unpredictable behavior of the application. Accordingly, the signal delivery is delayed until the application becomes runnable. Although this delay postponds the reclamation of memory leaks, any additional memory leaks do not occur while the application is blocked.

4. EVALUATION

4.1. Overview of Evaluation

The main advantage of KAL is the separation of the leak detection phase from the application's execution. This advantage stands against previous conservative garbage collection in terms of trade-off between performance effect and separation overhead. Conservative garbage collection imposes performance overhead by invoking a garbage collection in runtime. In KAL, a garbage collection overhead is, however, completely separated from the application; instead, KAL induces network and computing overheads for sending memory snapshots to a separated machine through network. Accordingly, (1) we need to compare the trade-off between conservative garbage collection and KAL.

The separation scheme is aimed to separate leak detection cost from a machine executing a target application. When the target application is CPU-bound, leak detection cost could affect the execution of the target application or other concurrent applications. In order to figure out those effects, (2) we evaluate each experiment both with separation and without separation.

For long-running applications such as Web servers, proxy servers and DB servers, KAL needs to be periodically applied to the target application. We define a *KAL invocation* (or an *invocation of KAL*) as one process using KAL from taking a memory snapshot to reclaiming leaks. We define the *detection period* as a time between two consecutive KAL invocations. The detection period could influence the number of written pages during the period, and it could also affect the heap size of the application being increased. Accordingly, (3) we varied the detection period to figure out its impact on the performance.

In our evaluation, we compared KAL with Conservative Garbage Collector [5] (we denote this as CGC). The basic algorithm of CGC and KAL is similar; they both inspect memory leaks using the mark and sweep algorithm and reclaim those detected leaks. Since the main difference of KAL from the CGC is the separation of the mark and sweep algorithm from application execution, besides performance, we need to compare those approaches for additional overheads such as downtime, network cost, and garbage collection time in KAL-d.

4.2. Environment

Our prototype KAL is implemented in Linux 2.6.23 with Glibc 2.7. The memory allocator in Glibc is the Lea memory allocator [24]. The evaluation is conducted in a machine installed with Pentium 4 3.0GHz CPU and 1GB main memory. For evaluating the separation scheme, a remote host is equipped with 2.6GHz i7 CPU and 8GB main memory. We used higher performance hardware for the separated machine in order to model embedded system development environment.

We built a synthetic application named *Leaktest* to measure the overheads. It manages a linked list with a sorted order and handles 10,000 operations that insert or remove random values. The storage of each list node is taken by corresponding memory allocator libraries: (1) the Lea memory allocator for KAL and (2) Conservative Garbage Collector 7.0 by Boehm [5] for CGC. Since the execution sequence and the sequence of random values are deterministic, the execution time of Leaktest can be a metric to evaluate our scheme with comparing CGC

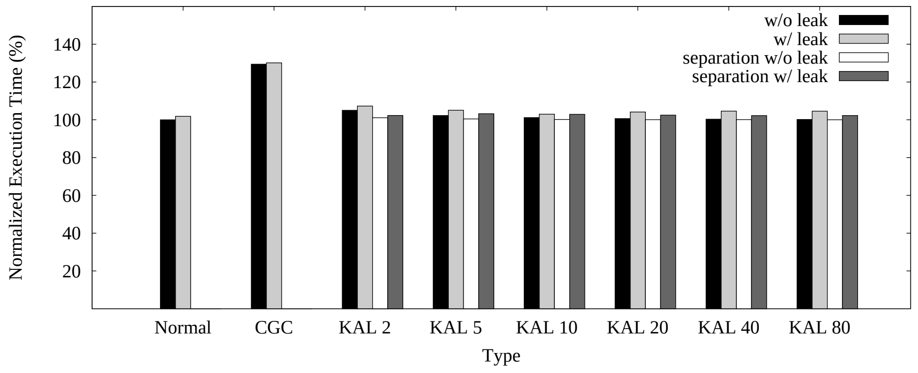


Figure 7. Normalized execution time of Leaktest with and without separation. Lower is better.

and a normal case. A normal case or a baseline denotes that a target application runs without any leak tolerance approach. The memory allocator in every normal case is the Lea memory allocator.

In order to show the leak tolerance of KAL, we evaluated two real-world applications used in a previous work [28]. *Xined-2.3.10*, which is an internet daemon server, generates memory leaks when it rejects an invalid connection request. *Squid-2.4.STABLE1*, which is a proxy server, produces memory leaks when it serves an invalid SNMP request.

We measured the performance impact of KAL upon three additional benchmark suites. They are *OSDB* [30], which is an open source database benchmark, *httpload* [21], which is a multiple http client benchmark that is used to evaluate Squid proxy server performance, and *httperf* [27], which is a Web server benchmark.

4.3. Performance overhead

In this subsection, we measured the performance overheads caused by KAL in runtime. First, we measured the execution time of Leaktest. The detection frequencies are varied from 2 seconds to 80 seconds, while CGC collects memory leaks at every 2.26 seconds. KAL N denotes that the detection period is N seconds; KAL is applied at every N seconds. We conducted all evaluations with separation and without separation. We also varied executions of Leaktest without memory leaks and with memory leaks in order to figure out the effect of leaks.

Figure 7 shows the normalized execution time of Leaktest. The execution time of Leaktest without memory leaks is used as a baseline and other values are normalized to this baseline. In CGC, the average garbage collection time is around 0.8 milliseconds, and the sum of each garbage collection time is 0.1 seconds, 0.003% of the execution time of Leaktest. Even though garbage collections block the execution of Leaktest, the garbage collection overhead itself is, even aggregated, negligible. As we described earlier, CGC replaces memory allocator with its

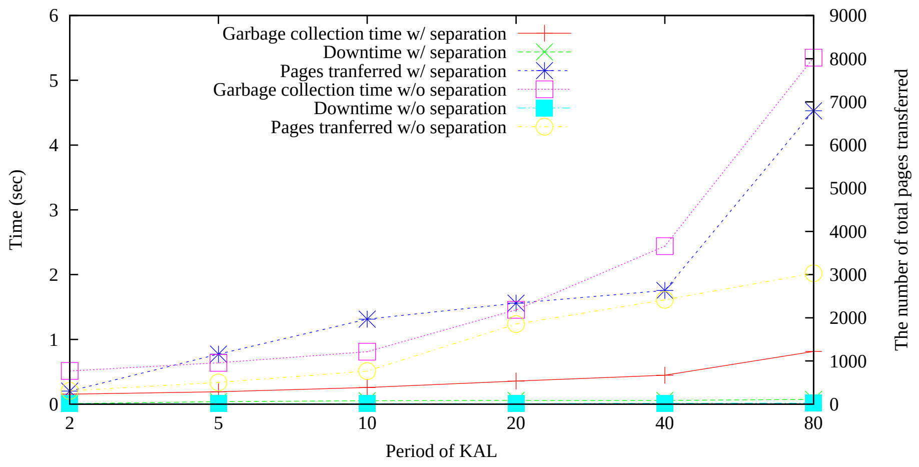


Figure 8. Extra performance overheads which are not directly comparable to Conservative Garbage Collector.

own one, and the new memory allocator performs slower than the Lea memory allocator. The bar labeled CGC shows about 30% of increased execution time compared to the normal case. However, our scheme performs similar execution time compared to the normal case because our scheme keeps in use of the standard memory allocator. Note that Leaktest application runs for 200 seconds; KAL 2 applies leak detections for 100 times. As shown in the figure, the leak detection period in KAL shows a negligible impact on the performance (the execution time) of Leaktest. As the detection period is increasing, the unit overhead is also increasing (this will be shown in Figure 8), but the total overhead is almost same. This result is caused from the fact that the total overhead is N times of a unit overhead where the N is decreased when the detection period increases.

Obviously, Leaktest with leaks requires more execution time compared to the normal case without leaks. This slowdown is due to the increased working set size caused by memory leaks. As we further show the heap size during this experiment, each memory leak makes a hole in the heap area and it eventually enlarges the heap size of the application; if the size of new memory allocation does not fit in the hole, the memory allocator expands the heap space. Since accessible memory objects place sparsely in heap area, the enlarged heap space adds the hidden costs such as more page faults and a larger cache miss ratio, and those effects may result in a slower execution time.

Since Figure 7 only shows the total execution time, we need to provide more detailed overheads of KAL. While CGC performs a garbage collection upon applications memory directly, KAL exports memory to other process or machine and conducts a garbage collection upon them. The cost of CGC is total garbage collection time plus the allocator performance,

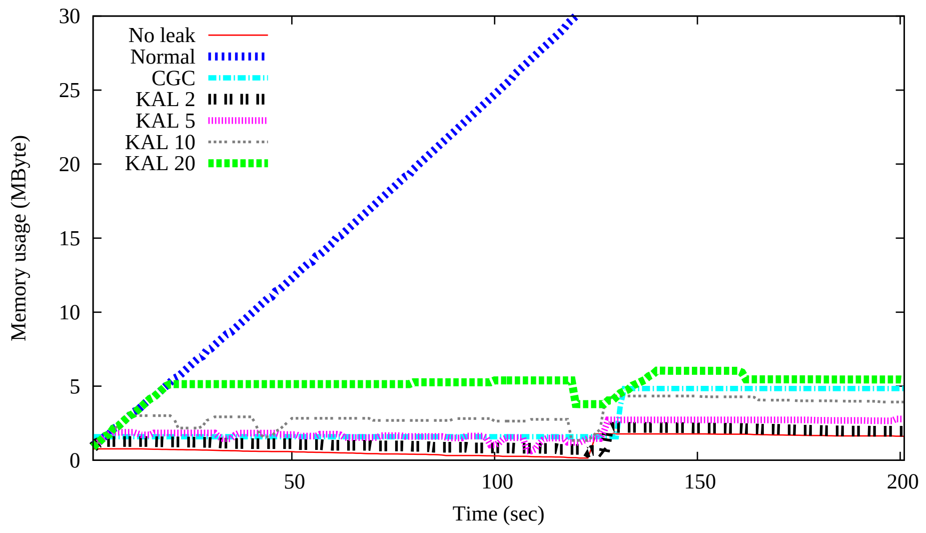


Figure 9. Heap size of Leaktest in Kbytes. No leak is an ideal case that does not generate memory leaks. Normal or CGC with leaks is cases where memory leaks are generated. All KAL N run with memory leaks.

and that of KAL is transferring memory via network plus precopying overhead including downtime. These costs are summarized in Figure 8.

As shown in Figure 8, the downtimes caused by KAL are about few milliseconds, and they are negligible to the execution time of a target application. The figure also reveals that the separation scheme has no relevance for affecting the downtime. Garbage collection time is proportional to detection period, since a longer detection period makes the heap larger if memory leak exists. Before the first leak toleration, existing memory leaks occupy memory space and other new memory allocations should require expanding the heap. As a result of this observation, garbage collection time is also increased from longer detection period. The number of page transferred during precopy is also increased when the detection period becomes longer. During the period between two consecutive KAL invocations, the longer detection period has more chances to make more pages being written. In a precopy phase, larger memory transferring may increase the total rounds of the precopy phase, and it results in the increased amount of transferring pages. The garbage collection time in separation is larger than that in no-separation since the performance of the remote host is higher than that of the machine that runs the target application.

Since the detection period can also affect the leak tolerance of Leaktest, we measured heap size of Leaktest for each detection period as shown in Figure 9. As we expected, the longer detection period leads to larger heap size due to the memory holes in the heap area. However,

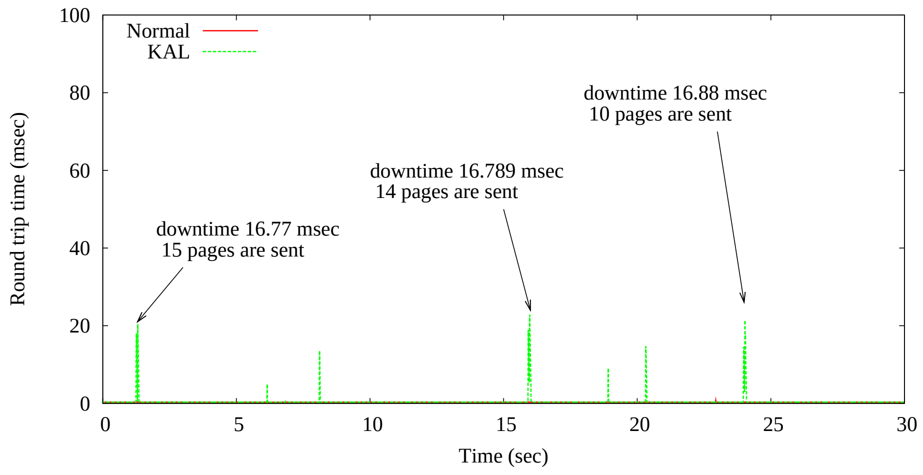


Figure 10. Response time of a web server using KAL three times. Three downtimes lead to three increased response times of each about 17 milliseconds.

the increased heap size is not always a serious problem, since each de-allocated memory holes can be reused for other memory allocation.

When KAL performs a garbage collection with the same detection period as CGC (average 2 seconds), the heap size of KAL 2 is a half of that of CGC case. As we argued, the general-purpose memory allocator does not only show better performance but also consume a smaller amount memory when providing the same memory allocations. When the detection period is 10 seconds, KAL suppresses the memory expansion with almost same level as the heap size of CGC case.

We also measured the round trip time of each http request using *httperf* in order to figure out the overhead caused by precopy itself. We disabled leak detection and reclaiming in KAL-d and KAL-ext, respectively. Httperf requests 100 64Kbytes files per second during the 30 seconds of the test. During this experiment, we applied KAL three times to figure out the effect of our scheme including precopy. In Figure 10, the x-axis denotes the time when a request is issued, and the y-axis denotes the response time of each request. Since httperf ran on a 100Mbps Ethernet, two consecutive requests have no disturbance to each other; a next http request is issued after a previous http response is arrived. The three peaks of response times are caused by the three downtimes when KAL tries to take memory snapshots for three times. While almost all response times are around 0.3 milliseconds, the response times of the three peaks hit up to 20 milliseconds. Although transferring 10MB memory snapshot of the Web server requires 100 milliseconds of downtime in the stop and copy approach, precopy requires only 17 milliseconds of downtime.

During the downtime, only about ten pages are transferred to the leak detector. Because KAL stops a Web server task during the finalizing phase, it induces peak response times, but the downtime does not affect the response time of successive requests.

KAL accompanies spatial overhead in addition to the performance overhead. KAL requires sufficient memory space to store the application's memory snapshot. The size of the snapshot, however, is not identical to the total address space of the target application. Because our scheme requires only pointer-containable pages, which include a global data section, stack and heap, the sizes of these pages are smaller than those of the whole address space of an application in our experiments, KAL consumed about 20% of the spatial overhead for each server daemon. If physical separation of KAL-d is possible, the memory overhead is also isolated from the original machine. Additionally, the spatial overhead of our scheme does not last for the entire lifetime of the application. The overhead is only imposed during leak toleration from the prepare phase of precopy to the end of leak detection in KAL-d.

4.4. Real world application

In this section, we provide experimental results upon real-world applications and benchmark suites. The first evaluation is Xinetd internet daemon. Since Xinetd ran without cycle conserving, we have no way to see performance of Xinetd directly. As an alternative, we ran SPECCPU as a background job with Xinetd, and we measured the execution time of SPECCPU. If our scheme requires more CPU cycles, the execution time of SPECCPU will be increased. Otherwise, the execution time of SPECCPU will be the same as a normal case. Figure 11 shows the execution time of SPECCPU for each case (baseline, CGC and KAL with varying detection periods). The value is normalized to the execution time of baseline. As shown in the figure, Xinetd consumes a little amount of CPU time, and the overhead of both CGC and KAL is relatively small. Without separation scheme, KAL consumes more CPU cycles (SPECCPU shows longer execution time) than that with separation scheme.

The next evaluation is Squid proxy server. We used Httpload benchmark, which requests URLs via Squid proxy server. The proxy server caches URLs and responds to Httpload. We places web server hosting about 20,000 web pages with an average size of 25Kbytes. Httpload requests 100 URLs per second to the web server via Squid proxy server. Figure 12 shows average response time of HTTP requests. As shown in the figure, KAL shows minimal performance degradation to Squid as compared to normal case.

Although above two applications contain memory leaks, the results have shown that the overhead of KAL is negligible. However, in order to argue that KAL achieves performance isolation about leak tolerance, we need to evaluate KAL in more intensive workloads; Xinetd and Squid only consume a little amount of CPU resources. Accordingly, we evaluated KAL with OSDB benchmark suite as a more intensive workload. OSDB consists of an online transaction program and a database server. Including clients, OSDB consumes as many CPU resources as possible. Note that because conservative garbage collector cannot be integrated with the OSDB benchmark, we omitted the CGC result. As shown in Figure 13, when the detection period is small, KAL degrades performance about 8% (2% with separation scheme). Note that we adjusted the priority of KAL-d for task scheduler to favor OSDB tasks more than KAL-d in order to minimize performance impact from KAL-d's garbage collection. Without

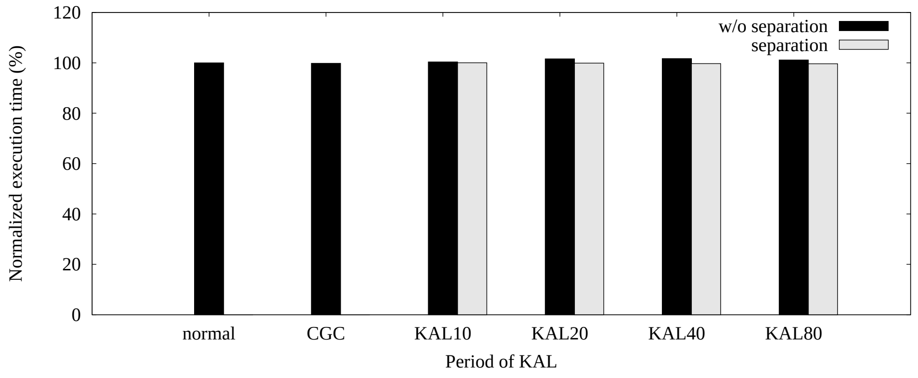


Figure 11. Normalized execution time of SPECCPU2000 as a background job with Xinetd daemon. Lower is better.

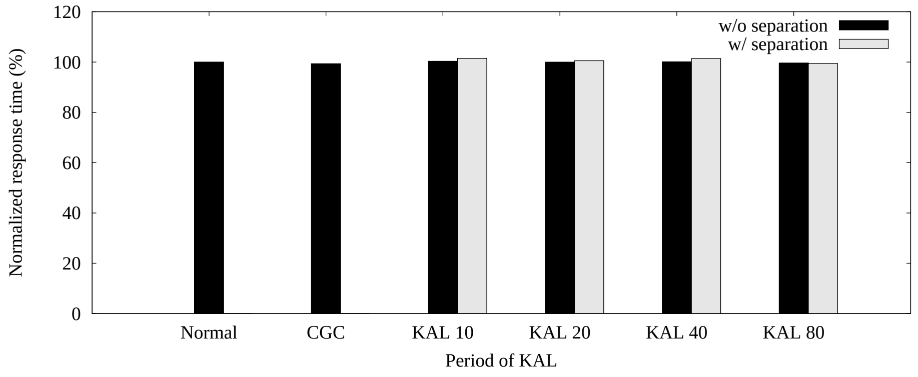


Figure 12. Normalized response time of httpload benchmark requesting HTTP requests to Squid proxy server. Lower is better.

separation scheme, KAL-d disturbs OSDB benchmark application since OSDB tries to consume CPU resources as many as possible, but KAL-d consumes a certain amount of CPU resources for garbage collection. However, when the detection period becomes longer, the performance overhead is minimal to the normal case.

4.5. Leak toleration

Although the performance overhead is negligible, KAL is useless if it cannot safely tolerate memory leaks. In order to figure out the leak toleration of our scheme, we measured the heap

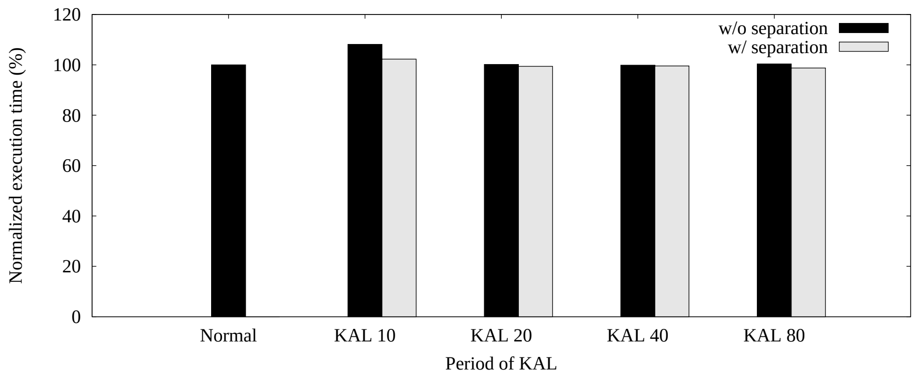


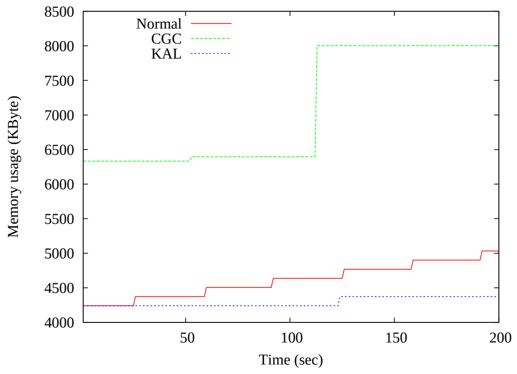
Figure 13. Normalized execution time of OSDB benchmark suit. Lower is better.

size of the target application, such as Xinetd, Squid and Leaktest. Note that we evaluated leak toleration with the separation scheme; with or without separation scheme the result of leak tolerance is the same.

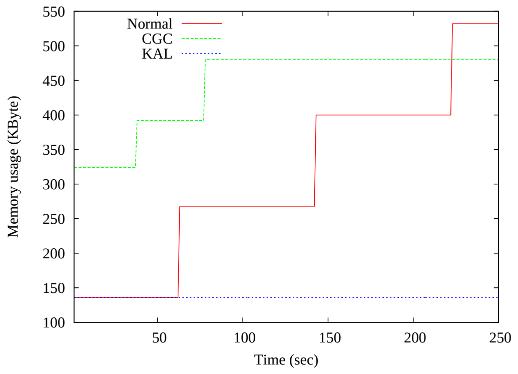
100 invalid telnet connection requests are tried to Xinetd server at every second. 50 bytes of memory leaks occur for each invalid request. In Squid, we use Httpload benchmark to request valid services while we concurrently generates invalid SNMP requests. Squid server generates about 1Kbyte of memory leaks when an invalid request is sent to the server. For about 200 seconds of evaluation, we measured the heap sizes of server daemons. We set the detection period of KAL as 20 seconds for each experiment. Figure 14 shows heap size of each experiment. The figure shows that KAL successfully suppresses the expansion of the heap space while normal case increases the heap size due to memory leaks. As we already argued, the conservative garbage collector also suppresses the memory leaks, but it does not utilize the heap spaces in comparison with the general-purpose memory allocator.

The last experiment was about an Apache Web server that serves PHP-based web pages. Although a Web server only requires a small amount memory, the PHP and MySQL modules in a Web server require significant amounts of memory in the heap. As an approximation, one web page request, which requires both a PHP computation and a database transaction, makes 3,000 function calls for `malloc()` and `free()`. We skipped the deallocation operation every 3,000th invocation of `free()` in order to intentionally inject leaks.

In Figure 15, Ideal denotes the case where no memory leak is injected, normal indicates that memory leaks are injected, and KAL means that KAL is applied every 120 seconds with memory leak injection. The total evaluation time is 1,200 seconds; 10 leak detections and reclamations are done. From 0-120 seconds, the address space of KAL is increased similarly to that of the normal case. At 120 seconds, KAL is first applied to a Web server and it reclaims memory leaks generated during the 120 seconds. During the next 120 seconds, the heap space is not expanded, because the Lea memory allocator provides free objects, reclaimed and making



(a) Heap size of Squid



(b) Heap size of Xinetd

Figure 14. Heap size of real-world application

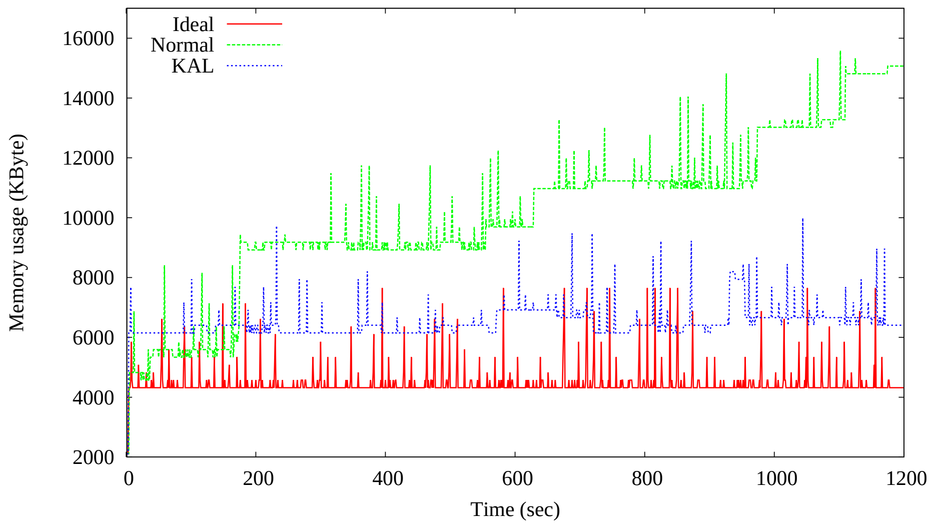


Figure 15. Memory usage (heap size) of web server in three scenarios: ideal case without memory leaks, normal case with synthetic memory leaks and KAL with synthetic memory leaks.

holes in heap space, for new memory object allocations. At 600 seconds, the size of the heap is slightly increased, but the size is recovered in the next application of KAL. As shown in the figure, KAL successfully suppresses address space expansion caused by injected memory leaks.

5. RELATED WORK

This section first discusses memory leaks in software and then describes various approaches used to detect and tolerate memory leaks.

Memory leaks in type unsafe languages (C and C++) are usually caused by failure to deallocate dynamic memory objects. In these languages, programmers are responsible for allocating or deallocating dynamic objects using `malloc()` or `free()` like functions explicitly. As software is becoming more complex, programmers are finding it hard to completely manage these functions; software can suffer from memory leaks even after it is released to the market.

Type safe languages (e.g., Java, C#, et al.) allow programmers to ignore concerns about dynamic memory management. They implicitly allocate memory objects and a garbage collector automatically deallocates them. Accordingly, unreachable objects from the rootset are eliminated. However stale objects retaining useless pointers produce similar effects to those of memory leaks. Although we focus on type unsafe languages, we review previous work for detecting or tolerating stale objects.

Most work on memory reliability is generally focused on making reliable software. Previous works usually not only detect memory leaks but also find memory access errors such as memory access overflow, buffer overflow or unauthorized access to memory. In this section, we also review all previous work that includes detecting memory leaks explicitly or tolerating memory leaks from applications.

Memory leak detection approaches are classified into three classes: debugging tools, conservative garbage collector, and leak tolerating allocator.

Debugging tools: generally instrument `malloc()` like functions to collect dynamic memory management data in the application. Mprof [37] checks the call-chain, which not only monitors caller of the `malloc()` caller but also considers the caller of the caller, to detect memory leaks and report in *gprof* style. Valgrind [32] and Purify [18] are representative tools used to detect memory leaks in the system. They not only detect memory leaks but also find array out of indexing, and various memory access errors by emulating the application. But, they are not applicable in embedded systems because significant runtime overhead (3-10x) slows down the target application. Emulation also makes it hard to apply in embedded system for porting issues. SafeMem [31] uses ECC-memory to detect memory leaks with little runtime overhead. It also reports memory access errors, but the limitation is that it depends on special hardware ECC-memory.

Conservative garbage collectors: Garbage collection reclaims unreachable objects in the program [15]. A conservative garbage collector is a collector working with type unsafe languages like C or C++ [5]. Boehm et al. first announced a conservative garbage collector using the mark and sweep algorithm [9]. They adapt various techniques to enhance performance [8, 7] and increase the garbage collection accuracy in type unsafe languages[6]. Various optimizations of a conservative garbage collector are also implemented [16]. Kuechlin studied a fork based parallel garbage collection in SAC-2 based system [23], but it has no separation or no similar scheme for embedded systems. In managed languages, *Bookmarking collection* reduces paging by bookmarking swapped pages during garbage collection [20]. The cost of the mark and sweep algorithm is that garbage collection disrupts some applications with real-time constraints.

Replacing memory allocator: Replacing memory allocator approaches have focused on improving the performance of the memory allocator [1, 12], recently, various studies have focused on reconstructing the memory allocator for memory safety including memory leak toleration. DieHard [2] and its subset Archipelago [25] proposed probabilistic memory safety by approximating an infinite-sized heap and one object per page memory allocation respectively. Although Archipelago is targeted to increase general memory reliability, leaked objects are naturally swapped out in cold storage, because they are never accessed [25]. *Plug* proposed a context sensitive allocation strategy. By segregating heap allocation sites, leaks can swapped out to disk; leaks are typically generated in the same allocation site [29]. *Cyclic memory allocation* [28] tolerates leaks by limiting m objects in a allocation site. In managed languages, other approaches [10, 33, 17, 11] also tolerate memory leaks by modifying memory allocation and garbage collection.

6. CONCLUSION

In spite of the significant progress in memory leak detection and treatment studies, few of those research results are currently in use for commodity products. Because the general-purpose memory allocator outperforms in terms of performance and space efficiency, many user applications still use the general-purpose memory allocator.

In this paper, we proposed a novel memory leak tolerance scheme that is able to co-operate with a general-purpose memory allocator. From the kernel level assist, we separate memory allocation and leak toleration. Based on the separation, the general-purpose memory allocator serves high performance memory allocation while our scheme detects and reclaims memory leaks of the application software. As a result, the application software gains two advantages which are high performance memory allocation and safety from memory leaks. Additionally, since our scheme can be implemented using the kernel extension mechanism, the application software does not require any modification, suspension and re-invocation.

Our evaluation demonstrates that our prototype successfully tolerates memory leaks in a synthetic workload and two real-world applications. The overall throughput of our scheme is close to that of general-purpose memory allocator without leak tolerance. The experimental results reveal that KAL outperforms the conservative garbage collector in terms of execution time and memory space usage. When the benchmark is a CPU-bound workload, the performance of benchmark suites is degraded. Separation scheme, which enhances the functionality of KAL, however, minimizes the overhead of KAL even in a CPU-bound workload.

REFERENCES

1. D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 187–196, New York, NY, USA, 1993. ACM.
2. E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 158–168, New York, NY, USA, 2006. ACM.
3. E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 114–124, New York, NY, USA, 2001. ACM.
4. E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–12, New York, NY, USA, 2002. ACM.
5. H.-J. Boehm. A garbage collector for c and c++. http://www.hpl.hp.com/personal/Hans_Boehm/gc/.
6. H.-J. Boehm. Space efficient conservative garbage collection. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 197–206, New York, NY, USA, 1993. ACM.
7. H.-J. Boehm. Bounding space usage of conservative garbage collectors. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–100, New York, NY, USA, 2002. ACM.
8. H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 157–164, New York, NY, USA, 1991. ACM.
9. H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice Experience*, 18(9):807–820, 1988.
10. M. D. Bond and K. S. McKinley. Tolerating memory leaks. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 109–126, New York, NY, USA, 2008. ACM.
11. M. D. Bond and K. S. McKinley. Leak pruning. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 277–288, New York, NY, USA, 2009. ACM.
12. J. Bonwick. The slab allocator: an object-caching kernel memory allocator. In *USTC'94: Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference*, pages 6–6, Berkeley, CA, USA, 1994. USENIX Association.
13. J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.
14. C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.
15. E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.
16. T. Endo and K. Taura. Reducing pause time of conservative collectors. In *ISMM '02: Proceedings of the 3rd international symposium on Memory management*, pages 119–131, New York, NY, USA, 2002. ACM.
17. M. Goldstein, O. Shehory, and Y. 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.
18. R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter 1992 USENIX Conference*, pages 125–138, Berkeley, CA, USA, 1991. USENIX Association.
19. M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 156–164, New York, NY, USA, 2004. ACM.
20. M. Hertz, Y. Feng, and E. D. Berger. Garbage collection without paging. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 143–153, New York, NY, USA, 2005. ACM.
21. Http_load. Multiprocessing http test client. http://acme.com/software/http_load/.
22. R. Krishnakumar. Kernel korner: kprobes-a kernel debugger. *Linux J.*, 2005(133):11, 2005.
23. W. Kuechlin. Parsac-2: A parallel sac-2 based on threads. *Lecture Nodes in Computer Science*, 508:0302–9743, 1991.

24. D. Lea. A memory allocator.
25. V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: trading address space for reliability and security. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 115–124, New York, NY, USA, 2008. ACM.
26. J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960.
27. D. Mosberger and T. Jin. httperf—a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, 1998.
28. H. H. Nguyen and M. 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.
29. G. Novark, E. D. Berger, and B. G. Zorn. Plug: Automatically tolerating memory leaks in c and c++ applications. Technical Report 08-09, University of Massachusetts Amherst, 2008.
30. OSDB. The open source database benchmark. http://osdb.sourceforge.net/.
31. F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 291–302, Washington, DC, USA, 2005. IEEE Computer Society.
32. J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.
33. Y. Tang, Q. Gao, and F. Qin. Leaksurvivor: Towards saftely tolerating memory leaks for garbage-collected languages. In *Proceedings of the 2008 USENIX annual Technical Conference*, pages 307–320. USENIX Association, 2008.
34. M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemptable remote execution facilities for the v-system. In *SOSP '85: Proceedings of the tenth ACM symposium on Operating systems principles*, pages 2–12, New York, NY, USA, 1985. ACM.
35. US-CERT. Us-cert, 2008.
36. C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.
37. B. G. Zorn and P. N. Hilfinger. A memory allocation profiler for c and lisp programs. In *Proceedings of the Summer Usenix Conference*, pages 223–237, Berkeley, CA, USA, 1988. USENIX Association.